第12章

12 LISP名字的由来：列表处理

无论是出于历史还是实践，列表在Lisp中都扮演着重要的角色。在历史上，列表曾经是Lisp最初的复合数据类型，尽管很多年以来它是这方面唯一的数据类型。现在，Lisp程序员可能会使用向量、哈希表、用户自定义的类或者结构体来代替列表。

从实践上来讲，由于列表对特定问题提供了极佳的解决方案，因此它们仍然能留在语言之中。比如有这样一个问题：如何将代码表示成数据，从而支持代码转换和生成代码的宏。它就是特定于Lisp的，这就可以解释为什么其他语言没有因缺少Lisp式列表所带来的不便。更一般地讲，列表是用于表达任何异构和层次数据的极佳数据结构。另外，它们相当轻量并且支持函数式的编程风格，而这种编程风格也是Lisp传统的另一个重要方面。

因此，你需要更加深入地理解列表。一旦对列表的工作方式有了更加深刻的理解，你将会对如何适时地使用它们有更好的认识。

12.1 “没有列表”

Spoon Boy：不要试图弯曲列表，那是不可能的。你要试着看清真相。

Neo：什么真相？

Spoon Boy：没有列表。

Neo：没有列表？

Spoon Boy：你会发现，弯曲的不是列表，而只是你自己。[1]

理解列表的关键在于，要理解它们在很大程度上是一种构建在更基本数据类型实例对象之上的描述。那些更简单的对象是称为点对单元（cons cell）的成对的值，使用函数CONS可以创建它们。

CONS接受两个实参并返回一个含有两个值的新点对单元。[2]这些值可以是对任何类型对象的引用。除非第二个值是NIL或是另一个点对单元，否则点对都将打印成在括号中并用一个点分隔两个值的形式，即所谓的“点对”。

① 改编自《黑客帝国》（http://us.imdb.com/Quotes?0133093）。

② CONS最初是动词construct（构造）的简称。

```
(cons 1 2) → (1 . 2)
```

点对单元中的两个值分别称为CAR和CDR，它们同时也是用来访问这两个值的函数名。在它们刚出现的年代，这些名字是有意义的，至少对于那些在IBM 704计算机上最早实现Lisp的人们来说是这样的。但即便在那时，它们也只被看作是用来实现这些操作的汇编助记符。然而，这些名字稍显缺乏意义也并不是件很坏的事情。当考虑单独的点对单元时，最好将它们想象成是简单的没有任何特别语义的任意数对。因此：

```
(car (cons 1 2)) → 1
(cdr (cons 1 2)) → 2
```

CAR和CDR也都能够支持SETF的位置，即给定一个已有的点对单元，有可能将新的值赋给它的任何一个值。[①]

```
(defparameter *cons* (cons 1 2))
*cons*                  → (1 . 2)
(setf (car *cons*) 10) → 10
*cons*                  → (10 . 2)
(setf (cdr *cons*) 20) → 20
*cons*                  → (10 . 20)
```

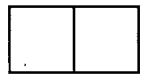
由于点对中的值可以是对任何类型对象的引用，因此可以通过将点对连接在一起，用它们构造出更大型的结构。列表是通过将点对以链状连接在一起而构成的。列表的元素被保存在点对的CAR中，而对后续点对的链接则被保存在CDR中。链上最后一个单元的CDR为NIL，正如第4章所提到的那样，它同时代表空列表和布尔值false。

这一安排毫无疑问是Lisp所独有的，它被称为*单链表*。不过，很少有Lisp家族之外的语言会对这种低微的数据类型提供如此广泛的支持。

因此当我讲一个特定的值是一个列表时，其实真正的意思是说它要么是NIL要么是对一个点对单元的引用。该点对单元的CAR就是该列表的第一个元素，而CDR则包含着其余元素，它引用着其他的列表，这有可能是另一个点对单元或NIL。Lisp打印器可以理解这种约定并能将这种链状的点对单元打印成括号列表而不是用点分隔的数对。

```
(cons 1 nil)                    → (1)
(cons 1 (cons 2 nil))           → (1 2)
(cons 1 (cons 2 (cons 3 nil))) → (1 2 3)
```

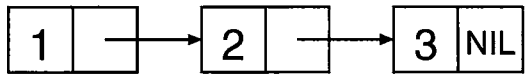
当谈论构建在点对单元之上的结构时，一些图例可以很好地帮助我们理解它们。方框和箭头所组成的图例可以像下面这样将点对单元表示成一对方框。



左边的方框代表CAR，而右边的则代表CDR。保存在一个特定点对单元中的值要么画在适当

① 当给定SETF的位置是CAR或CDR时，它将展开成一个对函数RPLACA或RPLACD的调用。和那些仍然使用SETQ的一样，一些守旧的Lisp程序员仍然直接使用RPLACA和RPLACD，但现代风格是使用CAR或CDR的SETF。

的方框之内，要么通过一个从方框指向其所引用值的箭头来表示。[1]例如，列表(1 2 3)是由三个点对单元通过它们的CDR链接在一起所构成的，如下所示：



尽管如此，一般在使用列表时并不需要处理单独的点对单元——创建和管理列表的函数将为你做这些事。例如，LIST函数可以在背后为你构建一些点对单元并将它们链接在一起。下面的LIST表达式等价于前面的CONS表达式：

```
(list 1)     → (1)
(list 1 2)   → (1 2)
(list 1 2 3) → (1 2 3)
```

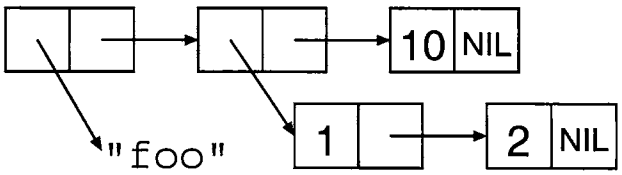
类似地，当从列表的角度考虑问题时，并不需要使用没有意义的名字CAR和CDR，FIRST和REST分别是CAR和CDR的同义词，当处理作为列表的点对时应该使用它们。

```
(defparameter *list* (list 1 2 3 4))
(first *list*)         → 1
(rest *list*)          → (2 3 4)
(first (rest *list*)) → 2
```

因为点对单元可以保存任何类型的值，所以它也可以保存列表。并且单一列表可以保存不同类型的对象。

```
(list "foo" (list 1 2) 10) → ("foo" (1 2) 10)
```

该列表的结构如下所示：



由于列表可以将其他列表作为元素，因此可以用它们来表示任意深度与复杂度的树。由此它们可以成为任何异构和层次数据的极佳表示方式。例如，基于Lisp的XML处理器通常在内部将XML文档表示成列表。另一个明显的树型结构数据的例子就是Lisp代码本身。第30章和第31章将编写一个HTML生成库，其中使用列表的列表来表示被生成的HTML。第13章将介绍如何用点对来表示其他数据结构。

Common Lisp为处理列表提供了一个相当大的函数库。在12.5节和12.6节里将介绍一些更重要的这类函数。但利用取自函数式编程的一些观点来考虑，这些函数将更容易被理解。

① 在一般情况下，诸如数字这类简单对象画在相应方框的内部，而更复杂的对象画在方框的外部并带有一个来自方框的箭头以指示该引用。这实际上很好地反映了许多Common Lisp实现的工作方式。从概念上来讲，尽管所有对象都是按引用保存的，但特定的简单不可修改的对象可以被直接保存在点对单元里。

12.2　函数式编程和列表

函数式编程的本质在于，程序完全由没有副作用的函数组成，也就是说，函数完全基于其参数的值来计算结果。函数式风格的好处在于它使得程序更易于理解。在消除副作用的同时也消除了所有超距作用的可能。并且由于函数的结果仅取决于其参数的值，因此它的行为更容易被理解和测试。例如，当看到像(+ 3 4)这样的表达式时，你知道其结果完全取决于“+”函数的定义以及值3和4。你不需要担心程序执行以前发生的事，因为没有什么可以改变该表达式的求值结果。

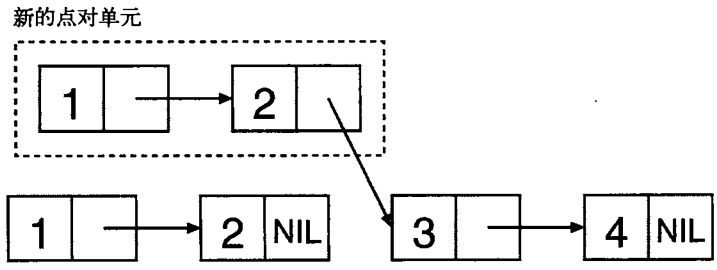
处理数字的函数天生就是函数式的，因为数字都是不可改变的对象。另一方面，如同刚刚看到的那样，通过**SETF**构成点对单元的**CAR**和**CDR**，列表是可改变的。但列表可以当作函数式数据类型来对待，只要将其值视为是由它们包含的元素所决定的即可。这样，形式(1 2 3 4)表示的任何列表在函数式意义上就将等价于任何其他含有这四个值的列表，无论实际表示该列表的是什么点对单元。并且，任何接受一个列表作为实参且其返回值完全依赖于列表内容的函数，也同样可以认为是函数式的。例如，当给定列表(1 2 3 4)时，序列函数**REVERSE**总是返回列表(4 3 2 1)。但由函数式等价的列表作为实参的不同**REVERSE**调用将返回函数式等价的结果列表。我将在12.6节里讨论的函数式编程的另一个方面即是对高阶函数的使用：函数将其他函数作为数据来对待，接受它们作为实参或是返回它们作为结果。

多数Common Lisp的列表操作函数都以函数式风格写成的。我将在后面讨论如何将函数式风格和其他编码风格混合在一起使用，但首先应当理解函数式风格应用在列表上的一些微妙之处。

多数列表函数之所以会用函数式编写，是因为这能使它们返回与其实参共享点对单元的结果。举一个具体的例子，函数**APPEND**可接受任意数量的列表实参并返回一个含有其参数列表的所有元素的新列表。例如：

```
(append (list 1 2) (list 3 4)) → (1 2 3 4)
```

从函数式观点来看，**APPEND**的工作是返回列表(1 2 3 4)而无需修改列表(1 2)和(3 4)中的任何点对单元。显然，为了实现该目标，可以创建由四个新的点对单元组成的新列表。但这样做并无必要。相反，**APPEND**实际上只用两个新的点对单元来持有值1或2，然后将它们连接在一起，并把第2个点对单元的**CDR**指向最后一个实参——列表(3 4)——的头部。然后它返回含有1的那个新生成的点对单元。原先的点对单元都未被修改过，并且结果确实是列表(1 2 3 4)。唯一美中不足的是，**APPEND**返回的列表与列表(3 4)共享了一些点对单元。产生的结构如下所示：



一般而言，**APPEND**必须复制除最后一个实参以外的所有其他实参，但它的返回结果却总是会与其最后一个实参共享结构。

其他一些函数也相似地利用了列表共享结构的能力。一些像**APPEND**这样的函数被指定总是返回以特定方式共享结构的结果。其他函数则被简单地允许根据具体实现来返回共享的结构。

12.3 “破坏性”操作

如果Common Lisp只是一门纯函数式语言，那么故事就应该到此为止了。不过，因为在一个点对单元被创建之后有可能通过对其**CAR**或**CDR**进行**SETF**操作来修改它，所以你需要想一想副作用是如何跟结构共享混合的。

由于Lisp的函数式传统，修改已有对象的操作被称作是*破坏性的*（destructive）。在函数式编程中，改变一个对象的状态相当于“破坏”了它，因为它不再代表相同的那个值了。尽管如此，使用同样的术语来描述所有的状态修改操作会在一定程度上产生误解，因为存在两种相当不同破坏性操作，即*副作用性*（for-side-effect）操作和*回收性*（recycling）操作。[①]

副作用性操作是那些专门利用其副作用的操作。就此而言，所有对**SETF**的使用都是破坏性的，此外还包括诸如**VECTOR-PUSH**或**VECTOR-POP**这类在底层使用**SETF**来修改已有对象状态的函数。但是将这些操作描述成是破坏性的有一点不公平——它们没打算被用于以函数式风格编写的代码中，因此你不该用函数式术语来描述它们。但如果将非函数式的副作用性操作和那些返回结构共享结果的函数混合使用，那么就需要小心不要疏忽地修改了共享的结构。例如，考虑下面三个定义：

```
(defparameter *list-1* (list 1 2))
(defparameter *list-2* (list 3 4))
(defparameter *list-3* (append *list-1* *list-2*))
```

在对这些形式求值之后，你有了三个列表，但是`*list-3*`和`*list-2*`就像前面的图示中的列表那样共享了一些结构。

```
*list-1*                → (1 2)
*list-2*                → (3 4)
*list-3*                → (1 2 3 4)
```

现在看看当修改了`*list-2*`时会发生什么：

```
(setf (first *list-2*) 0) → 0
*list-2*                → (0 4)      ; 如你所愿
*list-3*                → (1 2 0 4) ; 你可能并不想要这种结果
```

在共享的结构中，由于`*list-2*`中的第一个点对单元也是`*list-3*`中的第三个点对单元，对`*list-2*`的改变也改变了`*list-3*`，对`*list-2*`的**FIRST**进行**SETF**改变了该点对单元中**CAR**部分的值，从而影响了两个列表。

而另一种破坏性操作，即回收性操作，其本来就是用于函数式代码中的。它们的副作用仅是

① for-side-effect是被语言标准所采用的短语，而recycling则是我自己的发明。多数Lisp著作简单地将术语“破坏性”统用在这两类操作上，从而产生了我正试图消除的误解。

一种优化手段。特别地，它们在构造结果时会重用来自它们实参的特定点对单元。尽管如此，和诸如APPEND这种在返回列表中包含未经修改的点对单元的函数有所不同的是，回收性函数将点对单元作为原材料来重用，如有必要它将修改其CAR和CDR来构造想要的结果。这样，只有当调用回收性函数之后不再需要原先列表的情况下，回收性函数才可以被安全地使用。

为了观察回收性函数是怎样工作的，让我们将REVERSE，即返回一个序列的逆序版本的非破坏性函数，与它的回收性版本NREVERSE进行比较。由于REVERSE不修改其参数，它必须为将要逆序的列表的每个元素分配一个新的点对单元。但假如写出了类似下面的代码：

```
(setf *list* (reverse *list*))
```

通过将REVERSE的结果赋值回*list*，你就删除了对*list*原先的值的引用。假设原先列表中的点对单元不被任何其他位置引用，它们现在可以被作为垃圾收集了。不过，在许多Lisp实现中，立即重用已有的点对单元会比分配新的点对单元并让老的变成垃圾更加高效。

NREVERSE就能让你这么做。函数名字中*N*的含义是non-consing，意思是它不需要分配任何新的点对单元。虽然故意没有说明NREVERSE的明确的副作用（它可以修改列表中任何点对单元的任何CAR或CDR），但典型的实现可能会沿着列表依次改变每个点对单元的CDR，使其指向前一个点对单元。最终返回的点对单元曾经是旧列表的最后一个点对单元，而现在则成为逆序后的列表的头节点。不需要分配新的点对单元，也没有产生垃圾。

像NREVERSE这样的回收性函数大多都带有对应的能产生相同结果但没有破坏性的同伴函数。一般来说，回收性函数和它们的非破坏性同伴带有相同的名字，除了一个起始的字母N。尽管如此，并非所有函数都是这样，其中包括几个更常用的回收性函数。例如NCONC，它是APPEND的回收性版本；而DELETE、DELETE-IF、DELETE-IF-NOT和DELETE-DUPLICATES则是序列函数的REMOVE家族的回收性版本。

总之，你可以用与回收性函数的非破坏性同伴相同的方式来使用它们，但只有当你知道参数在函数返回之后不再被使用时，才能安全地使用它们。多数回收性函数的副作用说明并未严格到足以信赖的程度。

但有一组回收性函数带有可靠的明确指定的副作用，这使得事情变得更复杂了。它们是NCONC，即APPEND的回收性版本，以及NSUBSTITUTE和它的-IF和-IF-NOT变体，这些是序列函数SUBSTITUTE及其变体的回收性版本。

和APPEND一样，NCONC返回其列表实参的连结体，但是它以下面的方式构造其结果：对于传递给它的每一个非空列表，NCONC会将该列表的最后一个点对单元的CDR设置成指向下一个非空列表的第一个点对单元。然后它返回第一个列表，后者现在是拼接在一起的结果的开始部分。结果如下所示：

```
(defparameter *x* (list 1 2 3))

(nconc *x* (list 4 5 6)) → (1 2 3 4 5 6)

*x* → (1 2 3 4 5 6)
```

NSUBSTITUTE及其变体可靠地沿着列表实参的列表结构向下遍历，将任何带有旧值的点对单

元的CAR部分SETF到新的值上，否则保持列表原封不动。然后它返回最初的列表，其带有与SUBSTITUTE计算得到的结果相同的值。[1]

关于NCONC和NSUBSTITUTE，关键是需要记住，它们是不能依赖于回收性函数的副作用这一规则的例外。忽视它们副作用的可靠性，而像任何其他回收性函数一样来使用它们，只用来产生返回值，这种做法不但完全可以接受，甚至还是一种好的编程风格。

12.4 组合回收性函数和共享结构

尽管可以在函数实参在函数调用之后不会被使用的情况下使用回收性函数，但值得注意的是，如果不小心将一个回收性函数用在了以后会用到的参数上，你肯定会遭遇搬起石头砸自己脚的境遇。

使事情变得更糟的是，共享结构和回收性函数会用于不同的目的。非破坏性列表函数在点对单元永远不会被修改的假设下返回带有共享结构的列表，但是回收性函数却通过违反这一假设得以正常工作。或者换另一种说法，使用共享结构是基于不在乎究竟由哪些点对单元构成列表这一前提的，而使用回收性函数则要求精确地知道哪些点对单元会在哪里被引用到。

在实践中，回收性函数会有一些习惯用法。其中最常见的一种是构造一个列表，它是由一个在列表前端不断做点对分配操作的函数返回，通常是将元素PUSH进一个保存在局部变量中的列表里，然后返回对其NREVERSE的结果。[2]

这是一种构造列表的有效方式，因为每次PUSH都只创建一个点对单元并修改一个局部变量，而NREVERSE只需穿过列表并重新赋值每个元素的CDR。由于列表完全是在函数之内创建，所以完全不存在任何函数之外的代码会引用列表的任何点对单元的风险。下面是一个函数使用该习惯用法来构造一个由从0开始的前n个数字组成的列表：[3]

```
(defun upto (max)
  (let ((result nil))
    (dotimes (i max)
      (push i result))
    (nreverse result)))

(upto 10) → (0 1 2 3 4 5 6 7 8 9)
```

12

① 字符串函数NSTRING-CAPITALIZE、NSTRING-DOWNCASE和NSTRING-UPCASE也具有相似的行为——它们返回与其不带N的同伴相同的结果，但被指定在原位修改其字符串参数。

② 例如，在一次对Common Lisp Open Code Collection（CLOCC，即一个由许多开发者所写的功能丰富的库集合）中所有回收性函数的使用情况进行评测中，PUSH/NREVERSE习惯用法在所有的回收性函数使用中占据了将近一半。

③ 当然，还有其他方法来做到相同的事。例如，扩展的LOOP宏尤其方便做到这一点，并且很可能会生成比PUSH/NREVERSE版本更高效的代码。

```
(defun upto (max)
  (loop for i below max collect i))
```

无论如何，重要的是能够识别PUSH/NREVERSE习惯用法，因为其相当普遍。

还有一个最常见的回收性习惯用法，[1]是将回收性函数的返回值立即重新赋值到含有可能会被回收的值的位置上。例如，你经常看到像下面这样的表达式，它使用了**DELETE**，即**REMOVE**的回收性版本：

```
(setf foo (delete nil foo))
```

这将foo的值设置到了它的旧值上，只是所有的**NIL**都被移除了。不过，即便是这种习惯用法，你在使用时也需小心一些。如果foo和在其他位置上引用的列表共享了一些结构，那么使用**DELETE**来代替**REMOVE**可能会破坏其他那些列表的结构。例如早先那两个共享了它们最后两个点对单元的列表*list-2*和*list-3*：

```
*list-2* → (0 4)
*list-3* → (1 2 0 4)
```

可以像下面这样将4从*list-3*中删除：

```
(setf *list-3* (delete 4 *list-3*)) → (1 2 0)
```

不过，**DELETE**将很可能进行必要的删除，通过将第三个点对单元的**CDR**设置为**NIL**，从而从列表中断开了第四个保存了数字4的点对单元。由于*list-3*的第三个点对单元同时也是*list-2*的第一个点对单元，所以上述操作也改变了*list-2*：

```
*list-2* → (0)
```

如果使用**REMOVE**来代替**DELETE**，它将会构造一个含有值1、2和0的列表，在必要时创建新的点对单元而不会修改*list-3*中的任何点对单元。在这种情况下，*list-2*将不会受到影响。

PUSH/NREVERSE和**SETF/DELETE**的习惯用法很可能占据了80%的回收性函数使用。其他的使用是可能的，但需要小心地跟踪哪些函数返回共享的结构而哪些没有。

总之，当操作列表时，最好是以函数式风格来编写自己的代码——函数应当只依赖于它们的列表实参的内容而不应该修改它们。当然，按照这样的规则将会排除对任何破坏性函数的使用，无论是回收性的还是其他。一旦运行了代码，如果性能评估显示需要进行优化，你可以将非破坏性列表操作替换成相应的回收性操作，但只有当你确定其他任何位置不会引用实参列表时才可以这样做。

最后需要注意的是，当第11章里提到的排序函数**SORT**、**STABLE-SORT**和**MERGE**应用于列表时，它们也是回收性函数。[2]不过，这些函数并没有非破坏性的同伴，因此当需要对列表排序而又不破坏它时，你需要传给排序函数一个由**COPY-LIST**生成的列表副本。无论哪种情况，你都需要确保可以保存排序函数的结果，因为原先的实参很可能已经一团糟了。例如：

```
CL-USER> (defparameter *list* (list 4 3 2 1))
*LIST*
CL-USER> (sort *list* #'<)
(1 2 3 4)                          ; looks good
CL-USER> *list*
(4)                                ; whoops!
```

① 这一习惯用法在CLOCC代码库里占据了30%的回收性使用。

② **SORT**和**STABLE-SORT**在向量上可被用作副作用性操作，但由于它们仍然返回排序了的向量，你应当忽略这一事实，并出于一致性的目的仅使用它们的返回值。

12.5 列表处理函数

有了前面这些背景知识，现在就可以开始学习Common Lisp为处理列表而提供的函数库了。

前面已经介绍了获取列表中元素的基本函数：**FIRST**和**REST**。尽管可以通过将足够多的**REST**调用（用于深入列表）和一个**FIRST**调用（用于抽取元素）组合起来，以获得一个列表中的任意元素，但这样可能有点冗长。因此，Common Lisp提供了以从**SECOND**到**TENTH**的由其他序数命名的函数来返回相应的元素。而函数**NTH**则更为普遍，它接受两个参数，一个索引和一个列表，并返回列表中第*n*个（从0开始）元素。类似地，**NTHCDR**接受一个索引和一个列表，并返回调用**CDR** *n*次的结果。（这样，`(nthcdr 0 ...)`简单地返回最初的列表，而`(nthcdr 1 ...)`等价于**REST**。）但要注意的是，就计算机完成的工作而言，这些函数都不会比等价的**FIRST**和**REST**组合更高效，因此无法在没有跟随*n*个**CDR**引用的情况下得到一个列表的第*n*个元素。[1]

28个复合**CAR/CDR**函数则是另一个不时会用到的函数家族。每个函数都是通过将由最多四个A和D组成的序列放在C和R之间来命名的，其中每个A代表对**CAR**的调用而每个D代表对**CDR**的调用。因此我们可以得到：

```
(caar list) ≡ (car (car list))
(cadr list) ≡ (car (cdr list))
(cadadr list) ≡ (car (cdr (car (cdr list))))
```

但要注意，这其中许多函数仅当应用于含有其他列表的列表时才有意义。例如，**CAAR**抽取出给定列表的**CAR**的**CAR**，因此传递给它的列表必须含有另一个列表，并将该列表用作其第一个元素。换句话说，这些函数其实是用于树而不是列表的：

```
(caar (list 1 2 3))                  → error
(caar (list (list 1 2) 3))           → 1
(cadr (list (list 1 2) (list 3 4)))  → (3 4)
(caadr (list (list 1 2) (list 3 4))) → 3
```

现在这些函数不像以前那样常用了，并且即便是最顽固的守旧Lisp黑客也倾向于避免使用过长的组合。尽管如此，它们还是被用在很多古老的Lisp代码上，因此至少应当去理解它们的工作方式。[2]

① **NTH**基本上等价于序列函数**ELT**，但只工作在列表上。另外容易使人困惑的是，**NTH**接受其索引作为第一个参数，而**ELT**正相反。另一个区别是，如果你试图在一个大于或等于列表长度的索引上访问一个元素，**ELT**将报错，而**NTH**将返回**NIL**。

② 特别地，在发明解构参数列表之前，它们通常被用来解出传递给宏的表达式的不同部分。例如，你可以将下列表达式：

```
(when (> x 10) (print x))
```

像下面这样拆开：

```
;; the condition
(cadr '(when (> x 10) (print x))) → (> X 10)
;; the body, as a list
(cddr '(when (> x 10) (print x))) → ((PRINT X))
```

如果你正在非函数式地使用列表，这些FIRST-TENTH和CAR、CADR等函数也可用作SETF的位置。

表12-1　其他列表处理函数描述

函　数	描　述
LAST	返回列表的最后一个点对单元。带有一个整数参数时，返回最后*n*个点对单元
BUTLAST	返回列表的一个副本，最后一个点对单元除外。带有一个整数参数时，排除最后*n*个单元
NBUTLAST	BUTLAST的回收性版本。可能修改并返回其参数列表但缺少可靠的副作用
LDIFF	返回列表直到某个给定点对单元的副本
TAILP	返回真，如果给定对象是作为列表一部分的点对单元
LIST*	构造一个列表来保存除最后一个参数外的所有参数，然后让最后一个参数成为这个列表最后一个节点的CDR。换句话说，它组合了 LIST和 APPEND
MAKE-LIST	构造一个*n*项的列表。该列表的初始元素是NIL或者通过:initial-element关键字参数所指定的值
REVAPPEND	REVERSE和APPEND的组合。像REVERSE那样求逆第一个参数，再将其追加到第二个参数上
NRECONC	REVAPPEND的回收性版本。像 NREVERSE那样求逆第一个参数，再将其追加到第二个参数上。没有可靠的副作用
CONSP	用来测试一个对象是否为点对单元的谓词
ATOM	用来测试一个对象是否不是点对单元的谓词
LISTP	用来测试一个对象是否为点对单元或NIL的谓词
NULL	用来测试一个对象是否为NIL的谓词。功能上等价于 NOT但在测试空列表而非布尔假时文体上推荐使用

12.6　映射

函数式风格的另一个重要方面是对高阶函数的使用，即那些接受其他函数作为参数或将函数作为返回值的函数。前面章节里出现过几个高阶函数，例如MAP。尽管MAP可被同时用于列表和向量（也就是说，任何类型的序列），但Common Lisp另外还提供了6个特定用于列表的映射函数。这6个函数之间的区别在于它们构造结果的方式，以及它们究竟会将函数应用到列表的元素还是列表结构的点对单元上。

MAPCAR是最接近MAP的函数。因为它总是返回一个列表，所以它并不要求MAP所要求的结果类型实参。作为替代，它的第一个参数是想要应用的函数，而后续参数是其元素将为该函数提供实参的列表。除此之外，它和MAP的行为相同：函数被应用在列表实参的相继元素上，每次函数的应用会从每个列表中各接受一个元素。每次函数调用的结果都被收集到一个新列表中。例如：

```
(mapcar #'(lambda (x) (* 2 x)) (list 1 2 3)) → (2 4 6)
(mapcar #'+ (list 1 2 3) (list 10 20 30)) → (11 22 33)
```

MAPLIST也和MAPCAR较为相似，它们之间的区别在于MAPLIST传递给函数的不是列表元素而是实际的点对单元。[1]这样，该函数不仅可以访问到列表中每个元素的值（通过点对单元的CAR），还可以访问到列表的其余部分（通过CDR）。

除构造结果的方式不同，MAPCAN和MAPCON与MAPCAR和MAPLIST的工作方式很相似。MAPCAR和MAPLIST会构造一个全新的列表来保存函数调用的结果，而MAPCAN和MAPCON则通过将结果（必须是列表）用NCONC拼接在一起来产生它们的结果。这样，每次函数调用都可以向结果中提供任意数量的元素。[2]MAPCAN像MAPCAR那样把列表的元素传递到映射函数中，而MAPCON则像MAPLIST那样来传递点对单元。

最后，函数MAPC和MAPL是伪装成函数的控制构造，它们只返回第一个列表实参，因此只有当映射函数的副作用有用时，它们才是有用的。MAPC是MAPCAR和MAPCAN的近亲，而MAPL属于MAPLIST/MAPCON家族。

12.7 其他结构

尽管点对单元和列表通常被视作同义词，但这并不很准确。正如我早先提到的，你可以使用列表的列表来表示树。正如本章所讨论的函数允许你将构建在点对单元上的结构视为列表那样，其他函数也允许你使用点对单元来表示树、集合以及两类键/值映射表。我将在第13章讨论一些这类函数。

① 因此，MAPLIST是两个函数中更基本的。如果你只有MAPLIST，你可以在它的基础上构造出MAPCAR，但你不可能在MAPCAR的基础上构造MAPLIST。

② 在没有像REMOVE这样的过滤函数的Lisp方言中，过滤一个列表的惯用方法是使用MAPCAN：

```
(mapcan #'(lambda (x) (if (= x 10) nil (list x))) list) ≡ (remove 10 list)
```