第13章 超越列表：点对单元的其他用法 13

如同你在前面章节里看到的，列表数据类型是由一组操作点对单元的函数描述的。另外，Common Lisp还提供了一些函数，它们允许你把点对单元构建出的数据结构看作树、集合及查询表。本章将简要介绍这其中的一些数据结构及其处理函数。和列表处理函数一样，在开始编写更复杂的宏以及需要将Lisp代码作为数据处理时，这其中有很多函数会很有用。

13.1 树

由点对单元构建的数据结构既然可看作成列表，自然也可看成是树。毕竟，换另一种思考方式，树不就是一种列表的列表吗？将一组点对单元作为列表来看待的函数与将同样的点对单元作为树来看待的函数，其区别就在于函数将到哪些点对单元里去寻找该列表或树的值。由一个列表函数所查找的点对单元称为列表结构，其查找方式是以第一个点对单元开始，然后跟着CDR引用直到遇到NIL。列表元素就是由列表结构点对单元的CAR所引用的对象。如果列表结构中的一个点对单元带有一个引用到其他点对单元的CAR，那么被引用的点对单元将被视为作为外部列表元素的一个列表的头部。[1]而另一方面，树结构则是同时跟随CAR和CDR引用，只要它们指向其他点对单元。因此，树中的值就是该树结构中所有点对单元引用的非点对单元的值。

例如，下面的方框和箭头图例显示了构成列表的列表((1 2) (3 4) (5 6))的点对单元。列表结构仅包括虚线框之内的三个点对单元，而树结构则包含全部的点对单元。

为了搞清列表函数和树函数之间的区别，你可以考查一下函数COPY-LIST和COPY-TREE复制这些点对单元的方式。作为一个列表函数COPY-LIST只复制那些构成列表结构的点对单元。也就是说，它根据虚线框之内的每个点对单元生成一个对应的新点对单元。每一个这些新点对单元的CAR均指向与原来列表结构中的点对单元的CAR相同的对象。这样，COPY-LIST就不会复制子

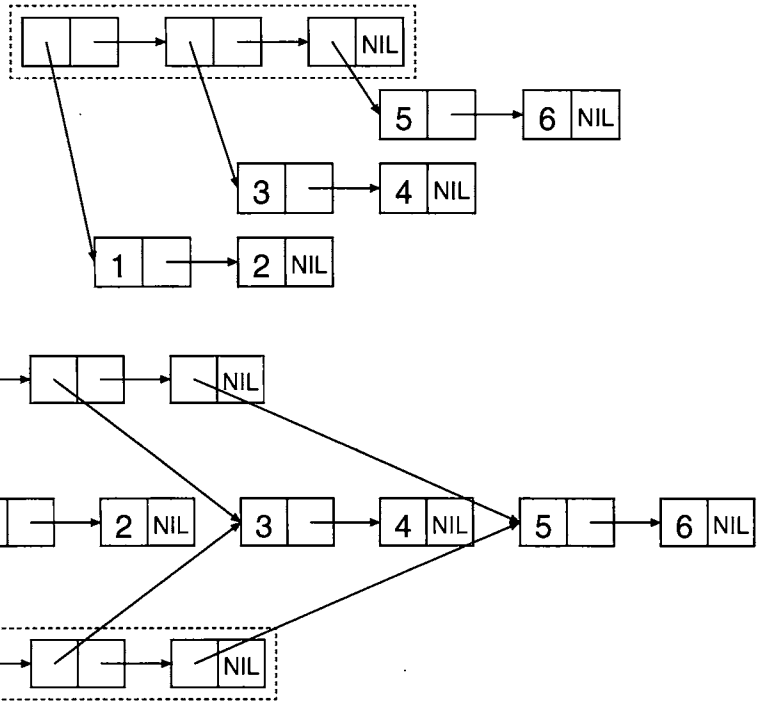
---

① 有可能构造出一串点对单元，其中最后一个点对单元的CDR不为NIL而是一些其他的原子。这被称为点列表，因为其最后一个元素前带有一个点。

```
(cons 1 (cons 2 (cons 3 4))) → (1 2 3 . 4)
```

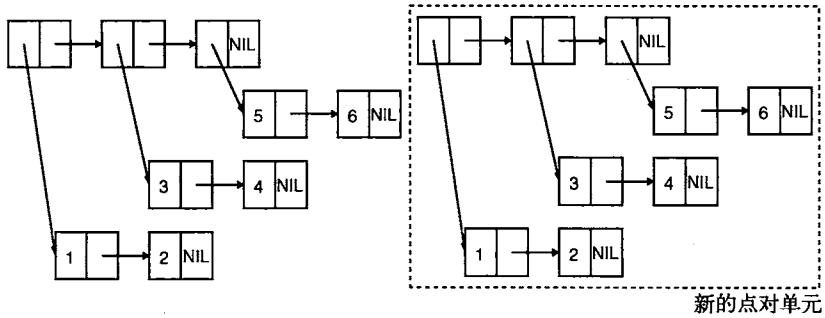
没有点的列表（其最后一个CDR为NIL）称为正则列表。

列表(1 2)、(3 4)或(5 6)，如下图所示：



新的点对单元

另一方面，**COPY-TREE**将会为图中的每个点对单元都生成一个新的点对单元，并将它们以相同的结构连接在一起，如下图所示：



新的点对单元

当原先的点对单元中引用了一个原子值时，复制中的相应点对单元也将指向相同的值。这样，由原先的树和**COPY-TREE**所产生的复制共同引用的唯一对象就是数字1～6以及符号**NIL**。

另一个在树的点对单元的**CAR**和**CDR**上都进行遍历的函数是**TREE-EQUAL**。它会比较两棵树，当这两棵树具有相同的形状以及它们对应的叶子是**EQL**等价时（或者如果它们满足带有:test关键字参数的测试），函数就认为它们相等。

其他一些以树为中心处理对象的函数包括**SUBSTITUTE**和**NSUBSTITUTE**这两个序列函数用于树的类似版本及其-IF和-IF-NOT变体。函数**SUBST**会像**SUBSTITUTE**一样接受一个新项、一个旧项和一棵树（跟序列的情况刚好相反），以及:key和:test关键字参数，然后返回一棵与原先的树具有相同形状的新树，只不过其中所有旧项的实例都被替换成新项。例如：

```
CL-USER> (subst 10 1 '(1 2 (3 2 1) ((1 1) (2 2))))
(10 2 (3 2 10) ((10 10) (2 2)))
```

SUBST-IF与**SUBSTITUTE-IF**相似。它接受一个单参数函数而不是一个旧项，该函数在树的每一个原子值上都会被调用，并且当它返回真时，新树中的对应位置将被填充成新值。**SUBST-IF-NOT**也是一样，只不过那些测试返回**NIL**的值才会被替换。**NSUBST**、**NSUBST-IF**和**NSUBST-IF-NOT**是**SUBST**系列函数的回收性版本。和其他大多数回收性函数一样，只有在明确知道不存在修改共享结构的危险时，才可以将这些函数作为它们非破坏性同伴的原位替代品来使用。特别的是，你必须总是保存这些函数的返回值，因为无法保证其结果与原先的树是**EQ**等价的。[①]

13.2　集合

集合也可以用点对单元来实现。事实上，你可以将任何列表都看作是集合，Common Lisp提供的几个函数可用于对列表进行集合论意义上的操作。但你应当在头脑中牢记，由于列表结构的组织方式，当集合变得更大时，这些操作将会越来越低效。

也就是说，使用内置的集合函数可以轻松地写出集合操作的代码，并且对于小型的集合而言，它们可能会比其他替代实现更为高效。如果性能评估显示这些函数成为代码的性能瓶颈，那么你也总能将列表替换成构建在哈希表或位向量之上的集合。

可以使用函数**ADJOIN**来构造集合。**ADJOIN**接受一个项和一个代表集合的列表并返回另一个代表集合的列表，其中含有该项和原先集合中的所有项。为了检测该项是否存在，它必须扫描该列表。如果该项没有被找到，那么**ADJOIN**就会创建一个保存该项的新点对单元，并让其指向原先的列表并返回它。否则，它返回原先的列表。

ADJOIN也接受:key和:test关键字参数，它们被用于检测该项是否存在于原先的列表中。和**CONS**一样，**ADJOIN**不会影响原先的列表——如果打算修改一个特定的列表，则需要将**ADJOIN**返回的值赋值到该列表所来自的位置上。**PUSHNEW**修改宏可以自动做到这点。

```
CL-USER> (defparameter *set* ())
*SET*
CL-USER> (adjoin 1 *set*)
(1)
CL-USER> *set*
NIL
CL-USER> (setf *set* (adjoin 1 *set*))
(1)
```

① **NSUBST**家族的函数确实可以就地修改树。不过，这里有一种边界情况：当被传递的“树”事实上是一个原子时，它不可能被就地修改，因此**NSUBST**的结果是将与其参数不同的对象：(nsubst 'x 'y 'y) -> X。

```
CL-USER> (pushnew 2 *set*)
(2 1)
CL-USER> *set*
(2 1)
CL-USER> (pushnew 2 *set*)
(2 1)
```

你可以使用**MEMBER**和相关的函数**MEMBER-IF**以及**MEMBER-IF-NOT**来测试一个给定项是否在一个集合中。这些函数与序列函数**FIND**、**FIND-IF**以及**FIND-IF-NOT**相似，不过它们只能用于列表。当指定项存在时，它们并不返回该项，而是返回含有该项的那个点对单元，即以指定项开始的子列表。当指定项不在列表中时，所有三个函数均返回**NIL**。

其余的集合论函数提供了批量操作：**INTERSECTION**、**UNION**、**SET-DIFFERENCE**以及**SET-EXCLUSIVE-OR**。这些函数中的每一个都接受两个列表以及`:key`和`:test`关键字参数，并返回一个新列表，其代表了在两个列表上进行适当的集合论操作所得到的结果：**INTERSECTION**返回一个由两个参数中可找到的所有元素组成的列表。**UNION**返回一个列表，其含有来自两个参数的每个唯一元素的一个实例。[①]**SET-DIFFERENCE**返回一个列表，其含有来自第一个参数但并不出现在第二个参数中的所有元素。而**SET-EXCLUSIVE-OR**则返回一个列表，其含有仅来自两个参数列表中的一个而不是两者的那些元素。这些函数中的每一个也都有一个相应的回收性函数，唯一区别在于后者的名字带有一个前缀*N*。

最后，函数**SUBSETP**接受两个列表以及通常的`:key`和`:test`关键字参数，并在第一个列表是第二个列表的一个子集时返回真，也就是说，第一个列表中的每一个元素也都存在于第二个列表中。列表中元素的顺序无关紧要。

```
CL-USER> (subsetp '(3 2 1) '(1 2 3 4))
T
CL-USER> (subsetp '(1 2 3 4) '(3 2 1))
NIL
```

13.3 查询表：alist和plist

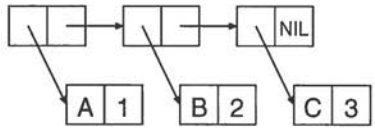
除了树和集合以外，你还可以用点对单元来构建表将键映射到值上。有两类基于点对的查询表会经常用到，这两者都是前面章节里提到过的。它们是*关联表*，也称为alist、*属性表*以及plist。尽管alist与plist都不能用于大型表（那种情况下你可以使用哈希表），但是值得去了解其使用方式，这既是因为对于小型的表而言，它们可以比哈希表更加高效，同时也是因为它们的一些专有属性十分有用。

alist是一种数据结构，它能将一些键映射到值上，同时也支持反向查询，并且当给定一个值时，它还能找出一个对应的键。alist也支持添加键/值映射来掩盖已有的映射，并且当这个映射以后被移除时原先的映射还可以再次暴露出来。

从底层来看，alist本质上是一个列表，其每一个元素本身都是一个点对单元。每个元素可以

① **UNION**从每个列表中只接受一个元素，但如果任何一个列表含有重复的元素，那么结果可能也含有重复的元素。

被想象成是一个键值对，其中键保存在点对单元的CAR中而值保存在CDR中。例如，下面是一个将符号A映射到数字1、B映射到2，C映射到3的alist的方框和箭头图例：



除非CDR中的值是一个列表，否则代表键值对的点对单元在表示成S-表达式时将是一个点对(dotted pair)。例如上图所表示的alist将被打印成下面的样子：

```
((A . 1) (B . 2) (C . 3))
```

alist的主查询函数是ASSOC，其接受一个键和一个alist并返回第一个CAR匹配该键的点对单元，或是在没有找到匹配时返回NIL。

```
CL-USER> (assoc 'a '((a . 1) (b . 2) (c . 3)))
(A . 1)
CL-USER> (assoc 'c '((a . 1) (b . 2) (c . 3)))
(C . 3)
CL-USER> (assoc 'd '((a . 1) (b . 2) (c . 3)))
NIL
```

为了得到一个给定键的对应值，可以简单地将ASSOC的结果传给CDR。

```
CL-USER> (cdr (assoc 'a '((a . 1) (b . 2) (c . 3))))
1
```

在默认情况下，指定的键使用EQL与alist中的键进行比较，但你可以通过使用:key和:test关键字参数的标准组合来改变这一行为。例如，如果想要用字符串的键，则可以这样写。

```
CL-USER> (assoc "a" '(("a" . 1) ("b" . 2) ("c" . 3)) :test #'string=)
("a" . 1)
```

如果没有指定:test为STRING=，ASSOC将可能返回NIL，因为带有相同内容的两个字符串不一定EQL等价。

```
CL-USER> (assoc "a" '(("a" . 1) ("b" . 2) ("c" . 3)))
NIL
```

由于ASSOC搜索列表时会从列表的前面开始扫描，因此alist中的一个键值对可以遮盖列表中后面带有相同键的其他键值对。

```
CL-USER> (assoc 'a '((a . 10) (a . 1) (b . 2) (c . 3)))
(A . 10)
```

可以像下面这样使用CONS向一个alist的前面添加键值对。

```
(cons (cons 'new-key 'new-value) alist)
```

但为方便起见，Common Lisp提供了函数ACONS，它可以让你这样写：

```
(acons 'new-key 'new-value alist)
```

和CONS一样，ACONS是一个函数，因此它不能修改用来保存所传递的alist的位置。如果你想要修改alist，你需要这样写成。

```
(setf alist (acons 'new-key 'new-value alist))
```

或

```
(push (cons 'new-key 'new-value) alist)
```

很明显，使用ASSOC搜索一个alist所花的时间是当匹配对被发现时当前列表深度的函数。在最坏情况下，检测到没有匹配的对将需要ASSOC扫描alist的每一个元素。但由于alist的基本机制是如此轻量，故而对于小型的表来说，alist可以在性能上超过哈希表。另外，alist在如何做查询方面也提供了更大的灵活性。我已经提到了ASSOC接受:key和:test关键字参数。当这些还不能满足你的需要时，可以使用ASSOC-IF和ASSOC-IF-NOT函数，其返回CAR部分满足（或不满足，在ASSOC-IF-NOT的情况下）传递到指定项上的测试函数的第一个键值对。并且还有另外3个函数，即RASSOC、RASSOC-IF和RASSOC-IF-NOT，和对应的ASSOC系列函数相似，只是它们使用每个元素的CDR中的值作为键，从而进行反向查询。

函数COPY-ALIST与COPY-TREE相似，除了代替复制整个树结构，它只复制那些构成列表结构的点对单元，外加那些单元的CAR部分直接引用的点对单元。换句话说，原先的alist和它的副本将同时含有相同的对象作为键和值，哪怕这些键或值刚好也由点对单元构成也是如此。

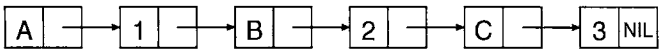
最后，你可以从两个分开的键和值的列表中用函数PAIRLIS构造出一个alist。返回的alist可能含有与原先列表相同或相反顺序的键值对。例如，你可能得到下面这样的结果：

```
CL-USER> (pairlis '(a b c) '(1 2 3))
((C . 3) (B . 2) (A . 1))
```

或者你也可能刚好得到下面这样的效果：

```
CL-USER> (pairlis '(a b c) '(1 2 3))
((A . 1) (B . 2) (C . 3))
```

另一类查询表是属性表或plist，你曾经在第3章里用它来表示数据库中的行。从结构上来讲，plist只是一个正常的列表，其中带有交替出现的键和值作为列表中的值。例如，一个将A、B和C分别映射到1、2和3的plist就是一个简单的列表(A 1 B 2 C 3)。用方框和箭头的形式来表示，它看起来像这样：



不过，plist不像alist那样灵活。事实上，plist仅支持一种基本查询操作，即函数GETF，其接受一个plist和一个键，返回所关联的值或是在键没有被找到时返回NIL。GETF也接受一个可选的第三个参数，它将在键没有被找到时代替NIL作为返回值。

与ASSOC不同，其使用EQL作为默认测试并允许通过:test参数提供一个不同的测试函数，GETF总是使用EQ来测试所提供的键是否匹配plist中的键。因此，你一定不能用数字和字符作为plist中的键。正如你在第4章里看到的那样，EQ对于这些类型的行为在本质上是未定义的。从实

践上来讲，一个plist中的键差不多总是符号，这是合理的，因为plist最初被发明用于实现符号“属性”，即名字和值之间的任意映射。

你可以将**SETF**与**GETF**一起使用来设置与给定键关联的值。**SETF**也会稍微特别地对待**GETF**，**GETF**的第一个参数被视为将要修改的位置。这样，你可以使用**GETF**的**SETF**来向一个已有的plist中添加新的键值对。

```
CL-USER> (defparameter *plist* ())
*PLIST*
CL-USER> *plist*
NIL
CL-USER> (setf (getf *plist* :a) 1)
1
CL-USER> *plist*
(:A 1)
CL-USER> (setf (getf *plist* :a) 2)
2
CL-USER> *plist*
(:A 2)
```

为了从plist中移除一个键/值对，你可以使用宏**REMF**，它将作为其第一个参数给定的位置设置成含有除了指定的那一个以外的所有键值对的plist。当给定的键被实际找到时，它返回真。

```
CL-USER> (remf *plist* :a)
T
CL-USER> *plist*
NIL
```

和**GETF**一样，**REMF**总是使用**EQ**来比较给定的键和plist中的键。

由于plist经常被用于想从同一个plist中抽取出几个属性的场合，所以Common Lisp还提供了一个函数**GET-PROPERTIES**，它能更高效地从单一plist中抽取出多个值。它接受一个plist和一个需要被搜索的键的列表，并返回多个值：第一个被找到的键、其对应的值，以及一个以被找到的键开始的列表的头部。这可以允许你处理一个属性表，抽取出想要的属性，而无需持续地从列表的开始处重新扫描。例如，下面的函数使用假想的函数process-property有效地处理用于指定键列表的Plist中的所有键/值对。

```
(defun process-properties (plist keys)
  (loop while plist do
       (multiple-value-bind (key value tail) (get-properties plist keys)
         (when key (process-property key value))
         (setf plist (cddr tail)))))
```

关于plist，最后特别要指出的是它们与符号之间的关系：每一个符号对象都有一个相关联的plist，以便用来保存关于该符号的信息。这个plist可以通过函数**SYMBOL-PLIST**获取到。但你很少需要关心整个plist，更常见的情况是使用函数GET，其接受一个符号和一个键，功能相当于在符号的**SYMBOL-PLIST**上对同一个键使用**GETF**。

```
(get 'symbol 'key) ≡ (getf (symbol-plist 'symbol) 'key)
```

和**GETF**一样，**GET**也可以用**SETF**来操作，因此你可以像下面这样将任意信息附加到一个符号上：

```
(setf (get 'some-symbol 'my-key) "information")
```

为了从一个符号的plist中移除属性，你可以使用**SYMBOL-PLIST**上的**REMF**或是更便捷的函数**REMPROP**。[1]

```
(remprop 'symbol 'key) ≡ (remf (symbol-plist 'symbol key))
```

向名字中附加任意信息对于任何类型的符号编程来说都是很有用的。例如，第24章将编写一个宏，它将向名字中附加信息，以便同一个宏的其他实例能将其抽取出并用于生成它们的展开式。

13.4 DESTRUCTURING-BIND

最后一个我需要介绍的，同时也是你将在后续章节里使用的一个用于拆分列表的工具是**DESTRUCTURING-BIND**宏。这个宏提供了一种*解构*（destructure）任意列表的方式，这类似于宏形参列表分拆它们的参数列表的方式。**DESTRUCTURING-BIND**的基本骨架如下所示：

```
(destructuring-bind (parameter*) list
  body-form*)
```

该参数列表可以包含宏参数列表中支持的任何参数类型，比如**&optional**、**&rest**和**&key**参数。[2]并且，如同在宏参数列表中一样，任何参数都可以被替换成一个嵌套的解构参数列表，从而将一个原本绑定在单个参数上的列表拆开。其中的*list*形式被求值一次并且应当返回一个列表，其随后被解构并且适当的值会被绑定到形参列表的对应变量中，然后那些*body-form*将在这些绑定的作用下被求值。一些简单的例子如下所示：

```
(destructuring-bind (x y z) (list 1 2 3)
  (list :x x :y y :z z)) → (:X 1 :Y 2 :Z 3)

(destructuring-bind (x y z) (list 1 (list 2 20) 3)
  (list :x x :y y :z z)) → (:X 1 :Y (2 20) :Z 3)

(destructuring-bind (x (y1 y2) z) (list 1 (list 2 20) 3)
  (list :x x :y1 y1 :y2 y2 :z z)) → (:X 1 :Y1 2 :Y2 20 :Z 3)

(destructuring-bind (x (y1 &optional y2) z) (list 1 (list 2 20) 3)
  (list :x x :y1 y1 :y2 y2 :z z)) → (:X 1 :Y1 2 :Y2 20 :Z 3)

(destructuring-bind (x (y1 &optional y2) z) (list 1 (list 2) 3)
  (list :x x :y1 y1 :y2 y2 :z z)) → (:X 1 :Y1 2 :Y2 NIL :Z 3)
```

① 直接**SETF SYMBOL-PLIST**也是有可能的。不过这是一个坏主意，因为不同的代码可能出于不同的原因添加了不同的属性到符号的plist上。如果一段代码清除了该符号的整个plist，它可能干扰其他向plist中添加自己的属性的代码。

② 宏参数列表确实支持一种参数类型，即**&environment**参数，而**DESTRUCTURING-BIND**不支持。不过，我没有在第8章里讨论这种参数类型，并且你现在也不需要考虑它。

```
(destructuring-bind (&key x y z) (list :x 1 :y 2 :z 3)
  (list :x x :y y :z z)) → (:X 1 :Y 2 :Z 3)

(destructuring-bind (&key x y z) (list :z 1 :y 2 :x 3)
  (list :x x :y y :z z)) → (:X 3 :Y 2 :Z 1)
```

另外还有一种参数（尽管第8章并未介绍），它既可以用在**DESTRUCTURING-BIND**中，也可以用在宏参数列表中，这就是**&whole**。如果被指定，它必须是参数列表中的第一个参数，并且它会绑定到整个列表形式上。[①]在一个**&whole**参数之后，其他参数可以像通常那样出现并且将像没有**&whole**参数存在那样抽取出列表中的指定部分。一个将**&whole**与**DESTRUCTURING-BIND**一起使用的例子如下所示：

```
(destructuring-bind (&whole whole &key x y z) (list :z 1 :y 2 :x 3)
  (list :x x :y y :z z :whole whole))
→ (:X 3 :Y 2 :Z 1 :WHOLE (:Z 1 :Y 2 :X 3))
```

你将在一个宏里使用**&whole**参数，它是将在第31章里开发的HTML生成库的一部分。不过，在那之前，我还要谈及更多的主题。在关于点对单元的两章相当Lisp化的主题之后，下面将介绍如何处理文件和文件名这种相对乏味的问题。

① 当一个**&whole**参数被用在宏参数列表中时，它所绑定的形式是整个宏形式，包括该宏的名字。