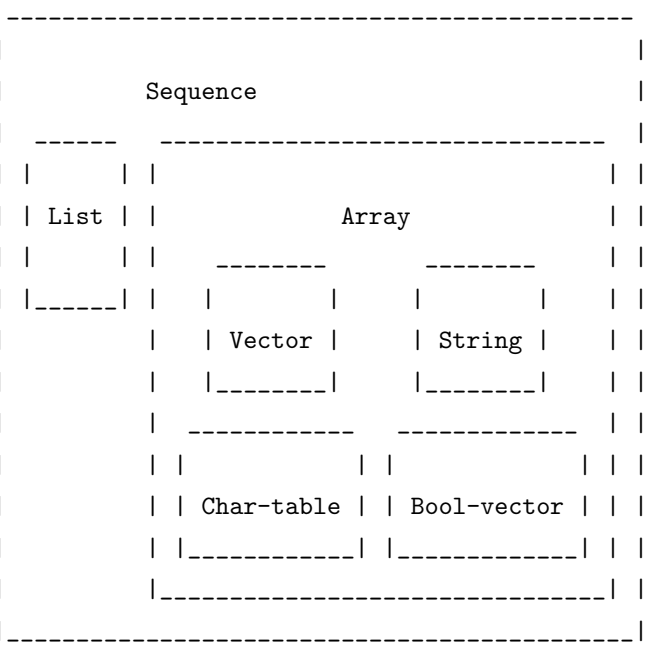
# 第六章　基本数据类型之四——序列和数组

序列是列表和数组的统称，也就是说列表和数组都是序列。它们的共性是内部的元素都是有序的。elisp 里的数组包括字符串、向量、char-table 和布尔向量。它们的关系可以用下面图表示：




数组有这样一些特性：

- 数组内的元素都对应一个下标，第一个元素下标为 0，接下来是 1。数组内的元素可以在常数时间内访问。
- 数组在创建之后就无法改变它的长度。
- 数组是自求值的。
- 数组里的元素都可以用 aref 来访问，用 aset 来设置。

向量可以看成是一种通用的数组，它的元素可以是任意的对象。而字符串是一种特殊的数组，它的元素只能是字符。如果元素是字符时，使用字符串相比向量更好，因为字符串需要的空间更少（只需要向量的 1/4），输出更直观，能用文本属性（ text property），能使用 emacs 的 IO 操作。但是有时必须使用向量，比如存储按键序列。

由于 char-table 和 bool-vector 使用较少，而且较难理解，这里就不介绍了。

## 6.1　测试函数

sequencep 用来测试一个对象是否是一个序列。arrayp 测试对象是否是数组。vectorp、char-table-p 和 bool-vector-p 分别测试对象是否是向量、 char-table、bool-vector。

## 6.2 序列的通用函数

一直没有提到一个重要的函数 length，它可以得到序列的长度。但是这个函数只对真列表有效。对于一个点列表和环形列表这个函数就不适用了。点列表会出参数类型不对的错误，而环形列表就更危险，会陷入死循环。如果不确定参数类型，不妨用 safe-length。比如：

```
(safe-length '(a . b))                  ; => 1
(safe-length '#1=(1 2 . #1#))           ; => 3
```

> 思考题
> 写一个函数来检测列表是否是一个环形列表。由于现在还没有介绍 let 绑定和循环，不过如果会函数定义，还是可以用递归来实现的。

取得序列里第 n 个元素可以用 elt 函数。但是我建议，对于已知类型的序列，还是用对应的函数比较好。也就是说，如果是列表就用 nth，如果是数组就用 aref。这样一方面是省去 elt 内部的判断，另一方面读代码时能很清楚知道序列的类型。

copy-sequence 在前面已经提到了。不过同样 copy-sequence 不能用于点列表和环形列表。对于点列表可以用 copy-tree 函数。环形列表就没有办法复制了。好在这样的数据结构很少用到。

## 6.3 数组操作

创建字符串已经说过了。创建向量可以用 vector 函数：

```
(vector 'foo 23 [bar baz] "rats")
```

当然也可以直接用向量的读入语法创建向量，但是由于数组是自求值的，所以这样得到的向量和原来是一样的，也就是说参数不进行求值，看下面的例子就明白了：

```
foo                                     ; => (a b)
[foo]                                   ; => [foo]
(vector foo)                            ; => [(a b)]
```

用 make-vector 可以生成元素相同的向量。

```
(make-vector 9 'Z)                      ; => [Z Z Z Z Z Z Z Z Z]
```

fillarray 可以把整个数组用某个元素填充。

```
(fillarray (make-vector 3 'Z) 5)        ; => [5 5 5]
```

aref 和 aset 可以用于访问和修改数组的元素。如果使用下标超出数组长度的话，会产生一个错误。所以要先确定数组的长度才能用这两个函数。

vconcat 可以把多个序列用 vconcat 连接成一个向量。但是这个序列必须是真列表。这也是把列表转换成向量的方法。

```
(vconcat [A B C] "aa" '(foo (6 7)))     ; => [A B C 97 97 foo (6 7)]
```

把向量转换成列表可以用 append 函数，这在前一节中已经提到。

思考题
如果知道 elisp 的 let 绑定和循环的使用方法，不妨试试实现一个 elisp 的 tr 函数，它接受三个参数，一是要操作的字符串，另外两个分别是要替换的字符集，和对应的替换后的字符集（当它是空集时，删除字符串中所有对应的字符）。

## 6.4 函数列表

```
;; 测试函数
(sequencep OBJECT)
(arrayp OBJECT)
(vectorp OBJECT)
(char-table-p OBJECT)
(bool-vector-p OBJECT)
;; 序列函数
(length SEQUENCE)
(safe-length LIST)
(elt SEQUENCE N)
(copy-sequence ARG)
(copy-tree TREE &optional VECP)
;; 数组函数
(vector &rest OBJECTS)
(make-vector LENGTH INIT)
(aref ARRAY IDX)
(aset ARRAY IDX NEWELT)
(vconcat &rest SEQUENCES)
(append &rest SEQUENCES)
```

## 6.5 问题解答

### 6.5.1 测试列表是否是环形列表

这个算法是从 safe-length 定义中得到的。你可以直接看它的源码。下面是我写的函数。

```
(defun circular-list-p (list)
  (and (consp list)
       (circular-list-p-1 (cdr list) list 0)))

(defun circular-list-p-1 (tail halftail len)
  (if (eq tail halftail)
      t
    (if (consp tail)
        (circular-list-p-1 (cdr tail)
```

```
                          (if (= (% len 2) 0)
                              (cdr halftail)
                            halftail)
                          (1+ len))
      nil)))
```

### 6.5.2 转换字符的 tr 函数

```
(defun my-tr (str from to)
  (if (= (length to) 0)                ; 空字符串
      (progn
        (setq from (append from nil))
        (concat
         (delq nil
               (mapcar (lambda (c)
                         (if (member c from)
                             nil c))
                       (append str nil)))))
    (let (table newstr pair)
      ;; 构建转换表
      (dotimes (i (length from))
        (push (cons (aref from i) (aref to i)) table))
      (dotimes (i (length str))
        (push
         (if (setq pair (assoc (aref str i) table))
             (cdr pair)
           (aref str i))
         newstr))
      (concat (nreverse newstr) nil))))
```

这里用到的 dotimes 函数相当于一个 C 里的 for 循环。如果改写成 while 循环，相当于：

```
(let (var)
  (while (< var count)
    body
    (setq var (1+ var)))
  result)
```

从这个例子也可以看出，由于列表具有丰富的函数和可变长度，使列表比数组使用更方便，而且效率往往更高。