第三章　基本数据类型之一—数字

elisp 里的对象都是有类型的，而且每一个对象它们知道自己是什么类型。你得到一个变量名之后可以用一系列检测方法来测试这个变量是什么类型（好像没有什么方法来让它说出自己是什么类型的）。内建的 emacs 数据类型称为 primitive types，包括整数、浮点数、cons、符号（symbol）、字符串、向量（vector）、散列表（hash-table）、subr（内建函数，比如 cons, if, and 之类）、byte-code function，和其它特殊类型，例如缓冲区（ buffer）。

在开始前有必要先了解一下读入语法和输出形式。所谓读入语法是让 elisp 解释器明白输入字符所代表的对象，你不可能让 elisp 读入.#@!? 这样奇怪的东西还能好好工作吧（ perl 好像经常要受这样的折磨:)）。简单的来说，一种数据类型有（也可能没有，比如散列表）对应的规则来让解释器产生这种数据类型，比如 123 产生整数 123，(a . b) 产生一个 cons。所谓输出形式是解释器用产生一个字符串来表示一个数据对象。比如整数 123 的输出形式就是 123，cons cell (a . b) 的输出形式是(a . b)。与读入语法不同的是，数据对象都有输出形式。比如散列表的输出可能是这样的：

```
#<hash-table 'eql nil 0/65 0xa7344c8>
```

通常一个对象的数据对象的输出形式和它的读入形式都是相同的。现在就先从简单的数据类型—数字开始吧。

emacs 的数字分为整数和浮点数（和 C 比没有双精度数 double）。1，1.，+1, -1, 536870913, 0, -0 这些都是整数。整数的范围是和机器是有关的，一般来最小范围是在-268435456 to 268435455（29 位，-2**28　2**28-1）。可以从 most-positive-fixnum 和 most-negative-fixnum 两个变量得到整数的范围。

你可以用多种进制来输入一个整数。比如：

```
#b101100 => 44       ; 二进制
#o54 => 44           ; 八进制
#x2c => 44           ; 十进制
```

最神奇的是你可以用 2 到 36 之间任意一个数作为基数，比如：

```
#24r1k => 44         ; 二十四进制
```

之所以最大是 36，是因为只有 0-9 和 a-z 36 个字符来表示数字。但是我想基本上不会有人会用到 emacs 的这个特性。

1500.0, 15e2, 15.0e2, 1.5e3, 和.15e4 都可以用来表示一个浮点数 1500.。遵循 IEEE 标准，elisp 也有一个特殊类型的值称为 NaN (not-a-number)。你可以用=(/ 0.0 0.0)= 产生这个数。

3.1　测试函数

整数类型测试函数是 integerp，浮点数类型测试函数是 floatp。数字类型测试用 numberp。你可以分别运行这几个例子来试验一下：

```
(integerp 1.)                           ; => t
```

```
(integerp 1.0)                          ; => nil
(floatp 1.)                             ; => nil
(floatp -0.0e+NaN)                      ; => t
(numberp 1)                             ; => t
```

还提供一些特殊测试，比如测试是否是零的 zerop，还有非负整数测试的 wholenump。

注：elisp 测试函数一般都是用 p 来结尾，p 是 predicate 的第一个字母。如果函数名是一个单词，通常只是在这个单词后加一个 p，如果是多个单词，一般是加 -p。

3.2 数的比较

常用的比较操作符号是我们在其它言中都很熟悉的，比如<，>，>=，<=，不一样的是，由于赋值是使用 set 函数，所以 = 不再是一个赋值运算符了，而是测试数字相等符号。和其它语言类似，对于浮点数的相等测试都是不可靠的。比如：

```
(setq foo (- (+ 1.0 1.0e-3) 1.0))       ; => 0.0009999999999998899
(setq bar 1.0e-3)                       ; => 0.001
(= foo bar)                             ; => nil
```

所以一定要确定两个浮点数是否相同，是要在一定误差内进行比较。这里给出一个函数：

```
(defvar fuzz-factor 1.0e-6)
(defun approx-equal (x y)
  (or (and (= x 0) (= y 0))
      (< (/ (abs (- x y))
            (max (abs x) (abs y)))
         fuzz-factor)))
(approx-equal foo bar)                  ; => t
```

还有一个测试数字是否相等的函数 eql，这是函数不仅测试数字的值是否相等，还测试数字类型是否一致，比如：

```
(= 1.0 1)                               ; => t
(eql 1.0 1)                             ; => nil
```

elisp 没有+=，-=，/=，*= 这样的命令式语言里常见符号，如果你想实现类似功能的语句，只能用赋值函数 setq 来实现了。/= 符号被用来作为不等于的测试了。

3.3 数的转换

整数向浮点数转换是通过 float 函数进行的。而浮点数转换成整数有这样几个函数：

- truncate 转换成靠近 0 的整数
- floor 转换成最接近的不比本身大的整数

- ceiling 转换成最接近的不比本身小的整数
- round 四舍五入后的整数，换句话说和它的差绝对值最小的整数

很晕是吧。自己用 1.2, 1.7, -1.2, -1.7 对这四个函数操作一遍就知道区别了（可以直接看 info。按键顺序是 C-h i m elisp RET m Numeric Conversions RET。以后简写成 info elisp - Numeric Conversions）。

这里提一个问题，浮点数的范围是无穷大的，而整数是有范围的，如果用前面的函数转换 1e20 成一个整数会出现什么情况呢？试试就知道了。

3.4 数的运算

四则运算没有什么好说的，就是 `+ - * /`。值得注意的是，和 C 语言类似，如果参数都是整数，作除法时要记住(/ 5 6) 是会等于 0 的。如果参数中有浮点数，整数会自动转换成浮点数进行运算，所以(/ 5 6.0) 的值才会是 5/6。

没有++ 和-- 操作了，类似的两个函数是 1+ 和 1-。可以用 setq 赋值来代替++ 和--：

```
(setq foo 10)                          ; => 10
(setq foo (1+ foo))                    ; => 11
(setq foo (1- foo))                    ; => 10
```

注：可能有人看过有 incf 和 decf 两个实现 ++ 和 -- 操作。这两个宏是可以用的。这两个宏是 Common Lisp 里的，emacs 有模拟的 Common Lisp 的库 cl。但是 RMS 认为最好不要使用这个库。但是你可以在你的 elisp 包中使用这两个宏，只要在文件头写上：

```
(eval-when-compile
  (require 'cl))
```

由于 incf 和 decf 是两个宏，所以这样写不会在运行里导入 cl 库。有点离题是，总之一句话，教主说不好的东西，我们最好不要用它。其它无所谓，只可惜了两个我最常用的函数 remove-if 和 remove-if-not。不过如果你也用 emms 的话，可以在 emms-compat 里找到这两个函数的替代品。

abs 取数的绝对值。

有两个取整的函数，一个是符号 `%`，一个是函数 mod。这两个函数有什么差别呢？一是 `%` 的第个参数必须是整数，而 mod 的第一个参数可以是整数也可以是浮点数。二是即使对相同的参数，两个函数也不一定有相同的返回值：

```
(+ (% DIVIDEND DIVISOR)
   (* (/ DIVIDEND DIVISOR) DIVISOR))
```

和 DIVIDEND 是相同的。而：

```
(+ (mod DIVIDEND DIVISOR)
   (* (floor DIVIDEND DIVISOR) DIVISOR))
```

和 DIVIDEND 是相同的。

三角运算有函数：sin, cos, tan, asin, acos, atan。开方函数是 sqrt。

exp 是以 e 为底的指数运算，expt 可以指定底数的指数运算。log 默认底数是 e，但是也可以指定底数。log10 就是(log x 10)。logb 是以 2 为底数运算，但是返回的是一个整数。这个函数是用来计算数的位。

random 可以产生随机数。可以用(random t) 来产生一个新种子。虽然 emacs 每次启动后调用 random 总是产生相同的随机数，但是运行过程中，你不知道调用了多少次，所以使用时还是不需要再调用一次(random t) 来产生新的种子。

位运算这样高级的操作我就不说了，自己看 info elisp - Bitwise Operations on Integers 吧。

3.5 函数列表

```
;; 测试函数
(integerp OBJECT)
(floatp OBJECT)
(numberp OBJECT)
(zerop NUMBER)
(wholenump OBJECT)
;; 比较函数
(> NUM1 NUM2)
(< NUM1 NUM2)
(>= NUM1 NUM2)
(<= NUM1 NUM2)
(= NUM1 NUM2)
(eql OBJ1 OBJ2)
(/= NUM1 NUM2)
;; 转换函数
(float ARG)
(truncate ARG &optional DIVISOR)
(floor ARG &optional DIVISOR)
(ceiling ARG &optional DIVISOR)
(round ARG &optional DIVISOR)
;; 运算
(+ &rest NUMBERS-OR-MARKERS)
(- &optional NUMBER-OR-MARKER &rest MORE-NUMBERS-OR-MARKERS)
(* &rest NUMBERS-OR-MARKERS)
(/ DIVIDEND DIVISOR &rest DIVISORS)
(1+ NUMBER)
(1- NUMBER)
(abs ARG)
(% X Y)
(mod X Y)
(sin ARG)
(cos ARG)
```

```
(tan ARG)
(asin ARG)
(acos ARG)
(atan Y &optional X)
(sqrt ARG)
(exp ARG)
(expt ARG1 ARG2)
(log ARG &optional BASE)
(log10 ARG)
(logb ARG)
;; 随机数
(random &optional N)
```

3.6 变量列表

```
most-positive-fixnum
most-negative-fixnum
```