Common Lisp 学习

作者：kenshinl

时间：Last Update: September 5, 2022

版本：0.1

封面图片作者：Elina Bernpaintner

Learn Common Lisp

目录

第 1 章 Common Lisp 语法

1.1 语法

```
;; 退出 sbcl 解释器
>:abort

;; 加载脚本
>: (load "test.lsp")
```

```
;; 定义全局变量
* (defparameter *glob* 99)

;; 定义全局常量
* (defconstant limit (+ *glob* 1))

;; 检查一个符号是否是全局变量或者全局常量
* (boundp '*glob*)
T

;; setf 给全局变量或者局部变量赋值
;; 如果 setf 的第一个实参是符号，且符号不是某个局部变量的名字，则 setf 把这个符号设为全局变量
* (setf *glob* 98)
98

;; 传入 setf 的第一个实参，还可以是表达式或变量名。在这种情况下，
;; 第二个实参的值被插入至第一个实参所引用的位置
* (setf x '(a b c))
(A B C)
* (setf (car x) 'n)
N
* x
(N B C)

;; 判断一个变量是不是列表
* (defun my-listp (x)
    (or (null x) (consp x)))

;; 判断一个变量是不是原子
* (defun my-atom (x)
    (or (null x) (not (consp x))))


;; nil 既是一个原子，又是一个列表
* (atom nil)
T
* (listp nil)
T

;; eql 比较两个对象的地址，equal 比较两个对象的值
* (eql (cons 'a nil) (cons 'a nil))
NIL
* (equal (cons 'a nil) (cons 'a nil))
T

;; 获取列表的第 n 个元素
```

```
* (nth 0 '(a b c))
A
;; 获取列表的第 n 个 cdr
* (nthcdr 2 '(a b c d))
(C D)

;; 返回列表的最后一个 cons 对象
* (last '(a b c d))
(D)
;; 获取列表的最后一个元素
* (car (last '(a b c d)))
D

;; mapcar 接受一个函数以及一个或多个列表，并返回把函数应用
;; 至每个列表的元素的结果，直到有的列表没有元素为止
* (mapcar #'(lambda (x) (+ x 10))
     '(1 2 3))
(11 12 13)

* (mapcar #'list
    '(a b c)
    '(1 2 3 4))
((A 1) (B 2) (C 3))


;; maplist 接受同样的参数，将列表的渐进的下一个 cdr 传入函数
* (maplist #'(lambda (x) x)
    '(a b c))
((A B C) (B C) (C))

;; member 返回从元素出现的位置开始的剩余列表元素，如果列表中不存在这个元素返回 nil
* (member 'b '(a b c))
(B C)

* (member 'd '(a b c))
NIL

;; 关键字参数
;; Common Lisp 函数接受一个或多个关键字参数。这些关键字参数不同的地方是，
他们不是把对应的参数放在特定的位置作匹配，而是在函数调用中用特殊标签，
称为关键字，来作匹配。一个关键字是一个前面有冒号的符号。关键字参数不关心顺序。

;; member 有一个 :test 关键字参数给使用者自定义比较函数
* (member '(a) '((a) (b)) :test #'equal)
((A) (B))

;; 函数 adjoin 像是条件式的 cons 。它接受一个对象及一个列表，
;; 如果对象还不是列表的成员，才构造对象至列表上。
* (adjoin 'b '(a b c))
(A B C)
* (adjoin 'd '(a b c))
(D A B C)

;; 并集、交集、补集
* (union '(a b c) '(b c d))
(A B C D)
* (intersection '(a b c) '(a c e))
(C A)
* (set-difference '(a b c d e) '(e d c))
(B A)

;; length 获取列表的长度
* (length '(a b c))
```

```
3

;; subseq 复制列表的一个子列表
* (subseq '(a b c d e) 1 3)
(B C)
* (subseq '(a b c d e) 1)
(B C D E)

;; reverse 返回一个反转过的列表
* (reverse '(a b c))
(C B A)

;; sort 接受一个序列及一个比较两个参数的函数，返回一个有同样元素的序列，
;; 根据比较函数来排序， sort 会修改传入的序列。
* (sort '(3 2 1 6 5 4) #'>)
(6 5 4 3 2 1)

;; 函数 every 和 some 接受一个比较函数及一个或多个序列。
;; 当我们仅输入一个序列时，它们测试序列元素是否满足判断式
* (every #'oddp '(1 2 3))
NIL
* (some #'oddp '(1 2 3))
T

;; 如果它们输入多于一个序列时，比较函数必须接受与序列一样多的元素作为参数，
;; every/some 从所有序列中提取一个元素作为比较函数的参数
* (every #'> '(4 5 6) '(1 2 3))
T

;; push/pop 在列表头部插入/删除元素
* (defparameter *y* '(1 2 3))
*Y*
* *y*
(1 2 3)
* (push 4 *y*)
(4 1 2 3)
* (pop *y*)
4
* *y*
(1 2 3)

;; pushnew 宏是 push 的变种，使用了 adjoin 而不是 cons
;; pushnew 只会往列表中插入不存在的元素
* (defparameter *y* '(1 2 3))
*Y*
* (pushnew 4 *y*)
(4 1 2 3)
* (pushnew 2 *y*)
(4 1 2 3)

;; 一个非正规列表的 Cons 对象称之为点状列表 (dotted list)
;; 在点状表示法，每个 Cons 对象的 car 与 cdr 由一个句点隔开来表示。
* (cons 'a 'b)
(A . B)

;; 用 Cons 对象来表示映射 (mapping)也是很自然的。一个由 Cons 对象组成
;; 的列表称之为关联列表(assoc-listor alist)。
* (defparameter *trans* '((+ . "add") (- . "sub")))
*TRANS*
* *trans*
((+ . "add") (- . "sub"))

;; assoc 用来取出在关联列表中，与给定的键值有关联的 Cons 对
```

```
;; 如果 assoc 没有找到对应的键时返回 nil
* (assoc '+ *trans*)
(+ . "add")


;; 可以调用 make-array 来构造一个数组，第一个实参为一个指定数组维度的列表。
;; 关键字参数 :initial-element 用来指定默认值
* (defparameter arr (make-array '(2 3) :initial-element nil))
ARR
* arr
#2A((NIL NIL NIL) (NIL NIL NIL))

;; 用 aref 取出数组内的元素
(aref arr 0 0)
NIL

;; 用 setf 和 aref 来设置数组的某个元素
(setf (aref arr 0 0) 'b)
B
arr
#2A((B NIL NIL) (NIL NIL NIL))

;; 我们只想要一维的数组，可以给 make-array 第一个实参传一个整数，而不是一个列表
* (defparameter vec (make-array '3 :initial-element 10))
VEC
* vec
#(10 10 10)

;; 一维数组又称为向量（vector）。可以通过 vector 函数来创建向量，向量的元素可以是任何类型
* (vector "a" 'b 10)
#("a" B 10)

;; 可以用 aref 来存取向量，但有一个更快的函数叫做 svref ，专门用来存取向量
* (svref vec 1)
10

;; 字符串是字符组成的向量。我们用一系列由双引号包住的字符，来表示一个字符串常量，而字符 c 用 #\c 表示。
;; 每个字符都有一个相关的整数   通常是 ASCII 码，但不一定是。在多数的 Lisp 实现里，函数 char-code
;; 返回与字符相关的数字，而 code-char 返回与数字相关的字符。
;; 字符比较函数 char<, char<=, char=, char>=, char>, 以及 char/=。
* (sort "hello" #'char<)
"ehllo"

;; 由于字符串是字符向量，序列与数组的函数都可以用在字符串。可以用 aref 来取出元素。
(aref "abc" 1)
#\b
;; 也可以使用针对字符串设计的 char 函数来取出元素。
(char "abc" 1)
#\b

;; 使用比较两个字符串
(equal "abc" "abc")
T
;; 使用 string-equal 忽略大小写比较两个字符串
(string-equal "abc" "ABC")
T

;; 使用 concatenate 连接字符串
(concatenate 'string "a " "b c")
"a b c"

;; 可以使用通用的 elt 函数来取序列的元素
(elt '(1 2 3) 1)
```

```
2
(elt "abc" 1)
#\b

;; 定义结构体
* (defstruct point
    x
    y)
POINT

;; 上面定义了一个 point 结构，具有两个字段 x 与 y 。
;; 同时隐式地定义了 make-point, point-p, copy-point, point-x, point-y 函数。
;; 每一个 make-point 的调用，会返回一个新的 point 。可以通过给予对应的关键字参数，来指定单一字段的值。
* (setf p (make-point :x 1 :y 2))
#S(POINT :X 1 :Y 2)

;; 使用生成的 point-x 函数访问 x 字段
* (point-x p)
1
;; 使用 setf 设置 x 字段
* (setf (point-x p) 3)
3
* p
#S(POINT :X 3 :Y 2)


;; 创建哈希表
* (setf ht (make-hash-table))
#<HASH-TABLE :TEST EQL :COUNT 0 {1001DEA573}>

;; 要取出与给定键值有关的数值，我们调用 gethash 并传入一个键值与哈希表。
;; 预设情况下，如果没有与这个键值相关的数值， gethash 会返回 nil
;; 函数 gethash 返回两个数值。第一个值是与键值有关的数值，第二个值说明了
;; 哈希表是否含有任何用此键值来储存的数值。由于第二个值是 nil ，我们知道
;; 第一个 nil是缺省的返回值，而不是因为 nil 是与 a 有关的数值。
* (gethash 'a ht)
NIL
NIL

;; 要把数值与键值作关联，使用 gethash 搭配 setf
* (setf (gethash 'a ht) 'hello)
HELLO
* (gethash 'a ht)
HELLO
T

;; 要从集合中移除一个对象，你可以调用 remhash ，它从一个哈希表中移除一个词条
* (remhash 'a ht)
T
* (gethash 'a ht)
NIL
NIL


;; 哈希表有一个迭代函数 maphash ，它接受两个实参，接受两个参数的函数以及哈希表。
;; 该函数会被每个键值对调用，没有特定的顺序。maphash 总是返回 nil。
* (setf (gethash 'a ht) 1
       (gethash 'b ht) 2)
2
* (maphash #'(lambda (k v)
    (format t "~A = ~A~%" k v))
    ht)
A = 1
```

```
B = 2
NIL

;; 和任何牵涉到查询的结构一样，哈希表一定有某种比较键值的概念。
;; 预设是使用 eql ，但你可以提供一个额外的关键字参数 :test 来告诉哈希表
;; 要使用 eq，equal，还是 equalp。
* (setf ht (make-hash-table :test #'equal))
#<HASH-TABLE :TEST EQUAL :COUNT 0 {1001E9F823}>
```

section 控制流

```
;; Common Lisp 有三个构造区块（block）的基本操作符： progn，block，tagbody。
;; 在 progn 主体中的表达式会依序求值，并返回最后一个表达式的值。
* (progn
    (format t "a")
    (format t "b")
    ( + 11 12))
ab
23

;; block 像是带有名字及紧急出口的 progn 。第一个实参应为符号，
;; 表示区块的名字。在主体中的任何地方，可以停止求值，并通过使
;; 用 return-from 指定区块的名字，来立即返回数值。
* (block head
    (format t "a")
    (return-from head 'val)
    (format t "b"))
a
VAL

;; 使用 defun 定义的函数主体，都隐含在一个与函数同名的区块，所以可以使用 return-form
* (defun foo ()
    (return-from foo 27))


;; 另一个我们用来区分表达式的操作符是 let。它接受一个代码主体，但允许我们在主体内设置新变量。
* (let (( x 7)
        (y 2))
    (format t "Number")
    ( + x y))
Number
9

;; 概念上说，一个 let 表达式等同于函数调用。
;; 函数可以用名字来引用，也可以通过使用一个 lambda 表达式从字面上来引用。由于 lambda 表达式
;; 是函数的名字，我们可以像使用函数名那样，把 lambda 表达式作为函数调用的第一个实参
* ((lambda (x) (+ x 1)) 3)
4

;; 所以，上面的 let 表达式实际上等价于
* ((lambda (x y)
    (format t "Number")
    (+ x y))
  7
  2)
Number
9

;; 这个模型清楚的告诉我们，由 let 创造的变量的值，不能依赖其它由同一个 let 所创造的变量。
* (let ((x 2)
        (y (+ x 1)))
    (+ x y))
```

```
;; 在 (+ x 1) 中的 x 不是前一行设置的值，因为这个let 表达式其实等价于
* ((lambda (x y) (+ x y)) 2
                        (+ x 1))

;; 从这里可以清晰的看出 (+ x 1) 中的 x 其实是外部的变量，而非 let 表达式的内部变量

;; 如果想要新变量的值依赖同一个表达式所设立的另一个变量，可以使用 let*
* (let* ((x 1)
         (y (+ x 1)))
    (+ x y))
3

;; 一个 let* 功能上等同于一系列嵌套的 let 。这个例子等同于
* (let ((x 1))
    (let ((y (+ x 1)))
      (+ x y)))
3

;; let 与 let* 将变量初始值都设为 nil 。nil 为初始值的变量，可以不赋值。
* (let ( x y)
    (list x y))
(NIL NIL)

;; destructuring-bind 宏是通用化的 let 。其接受单一变量，
;; 一个模式(pattern，一个或多个变量所构成的树)，并将它们与某个实际的树所对应的部份做绑定。
* (destructuring-bind (w (x y) . z) '(a (b c) d e)
    (list w x y z))
(A B C (D E))

;;;;;;;;;;; 条件表达式

;; if 表达式，测试表达式求值返回真时，则对主体求值
* (if (oddp 9)
    (progn
      (format t "odd")))
odd
NIL

;; when 表达式与 if 表达式等价，测试表达式求值返回真时，则对主体求值
* (when (oddp 9)
    (format t "odd"))
odd
NIL

;; unless 表达式与 when 表达式相反，测试表达式求值返回假时，则对主体求值
* (unless (oddp 10)
    (format t "not odd"))
not odd
NIL

;; cond 表达式，允许多个条件判断，与每个条件相关的代码隐含在 progn 里
* (defun check-num (num)
    (cond ((< num 0) (format t "num < 0"))
    ((= num 0) (format t "num = 0"))
    (t (format t "num > 0"))))

* (check-num -1)
num < 0
NIL
* (check-num 0)
num = 0
NIL
* (check-num 1)
```

```
num > 0
NIL

;; case 表达式
* (defun check-num (num)
    (case num
    ((-1) (format t "num = -1"))
    ((0) (format t "num = 0"))
    (otherwise (format t "num unknown"))))

* (check-num -1)
num = -1
NIL
* (check-num 0)
num = 0
NIL
* (check-num 1)
num unknown
NIL

;; do 表达式，do 的第一个参数必须是说明变量规格的列表
;; (variable initial update)
;; initial 与 update 形式是选择性的。若 update 形式忽略时，
;; 每次迭代时不会更新变量。若 initial 形式也忽略时，变量会使用nil 来初始化。
* (defun show-squares (start end)
    (do ((i start (+ i 1)))
        ((> i end) 'done)
        (format t "~A ~A~%" i (* i i))))

* (show-squares 1 3)
1 1
2 4
3 9
DONE

;; 当do 表达式同时更新超过一个变量时，如果一个 update 形式，引用到一个拥有自己的
;; update 形式的变量时，它获得迭代之前的值。
* (let ((x 'a))
    (do ((x 1 (+ x 1)) ;; update x
         (y x x))    ;; update y
      ((> x 5))
      (format t "(~A ~A) " x y)))
(1 A) (2 1) (3 2) (4 3) (5 4)
NIL

;; 但如果使用 do* ，它有着和 let 与 let* 一样的关系。任何 initial 或 update 形式
;; 可以参照到前一个子句的变量，并会获得当下的值
* (do* ((x 1 (+ x 1))
        (y x x))
    ((> x 5))
    (format t "(~A ~A) " x y))
(1 1) (2 2) (3 3) (4 4) (5 5)
NIL

;; 使用 dolist 进行迭代。当迭代结束时，初始列表内的第三个表达式(此处是 done)
;; 会被求值并作为 dolist 的返回值。缺省是 nil
* (dolist (x '(a b c d) 'done)
    (format t "~A " x))
A B C D
DONE

;; 类似的迭代操作符还有 dotimes，给定某个 n，将会从整数 0，迭代至 n-1
* (dotimes (x 5 'done)
```

```
   (format t "~A " x))
0 1 2 3 4
DONE

;; mapc 可以用来遍历多个列表
* (mapc #'(lambda (x y)
               (format t "~A ~A  " x y))
   '(a1 a2 a3)
   '(b1 b2 b3))
A1 B1 A2 B2 A3 B3
(A1 A2 A3)

;; 在 Common Lisp 里，一个表达式可以返回零个或多个数值，
;; 最多可以返回几个值取决于各编译器实现，但至少可以返回 19 个值。

;; values 函数返回多个数值，它一个不少地返回你作为数值所传入的实参
* (values 'a nil 10 (+ 1 2 3 4))
A
NIL
10
10

;; 要接收多个数值，我们使用 multiple-value-bind，
;; 如果接收的值数量大于返回的值数量，剩余的变量会是 nil
* (multiple-value-bind (x y z) (values 1 2 3)
   (list x y z))
(1 2 3)
* (multiple-value-bind (x y z) (values 1 2)
   (list x y z))
(1 2 NIL)

;; 可以借由 multiple-value-call 将多值作为实参传给第二个函数
* (multiple-value-call #'+ (values 1 2 3))
6

;; multiple-value-list 看起来像是使用 #'list 作为第一个参数的来调用 multiple-value-call
* (multiple-value-list (values 'a 'b 'c))
(A B C)

;; 可以使用 return 在任何时候离开一个 block。但有时候想在数个函数调用里将控制权转移回来。
;; 要实现这个功能，可以使用 catch 与 throw 。一个 catch 表达式接受一个标签(tag)，
;; 标签可以是任何类型的对象。
* (defun super ()
   (catch 'abort
     (sub)
     (format t "catch abort")))

* (defun sub ()
   (throw 'abort 100))

* (super)
100
;; 可以看到，这里只输出了 100，跳过了 (format t "catch abort") 这条语句的执行。
;; 一个带有给定标签的 throw ，为了要到达匹配标签的 catch ，会将控制权转移
;; (因此杀掉进程)给任何有标签的 catch 。如果没有一个 catch 符合欲匹配的标签时，
;; throw 会产生一个错误。

;; 有时候想要防止代码被 throw 与 error 打断。可以使用 unwind-protect ，确保像是前述的中断，
;; 不会让程序停在不一致的状态。一个 unwind-protect 接受任何数量的实参，并返回第一个实参的值。
;; 然而即便是第一个实参的求值被打断时，剩下的表达式仍会被求值：
* (setf x 1)
1
* (catch 'abort
```

```
   (unwind-protect
     (throw 'abort 100)
     (setf x 2)))
100
* x
2
```