

# Emacs 实践笔记

aborn

2018-05-09 00:03

## Contents

<b>1</b>	<b>基本数据类型</b>	<b>1</b>
1.1	Lisp 的数据类型 . . . . .	1
1.2	标识类型 (Symbols) . . . . .	1
1.2.1	标识类型的组成 . . . . .	1
1.2.2	定义标识类型 . . . . .	2
1.2.3	标识符操作函数 . . . . .	2
1.2.4	标识符属性 . . . . .	3
1.3	列表 . . . . .	3
1.3.1	关联列表 alist (Association Lists) . . . . .	3
1.3.2	属性列表 plist (Property Lists) . . . . .	4
1.3.3	对列表进行排序 . . . . .	5
<b>2</b>	<b>控制结构</b>	<b>5</b>
2.1	控制结构 . . . . .	5
2.2	顺序结构 . . . . .	5
2.3	条件语句 . . . . .	6
<b>3</b>	<b>函数</b>	<b>6</b>
3.1	什么是函数? . . . . .	6
3.2	定义函数 . . . . .	6
3.2.1	检查一个函数是否定义 . . . . .	6
3.2.2	函数参数 . . . . .	6
3.3	函数调用 . . . . .	7
3.3.1	funcall . . . . .	7
3.3.2	apply . . . . .	7
3.3.3	映射函数 (Mapping Functions) . . . . .	8
3.4	匿名函数 . . . . .	8
3.4.1	lambda 宏 . . . . .	8

3.4.2	function 特殊表达式 . . . . .	9
3.5	获取函数单元内容 . . . . .	9
3.5.1	symbol-function . . . . .	9
3.5.2	fboundp . . . . .	9
3.6	特殊表达式 (Special Forms) 和宏 . . . . .	9
3.6.1	内建函数 (primitive) . . . . .	9
3.6.2	special form . . . . .	10
<b>4</b>	<b>文件</b>	<b>10</b>
4.1	文件及访问 . . . . .	10
4.1.1	打开文件 . . . . .	10
4.1.2	文件保存 . . . . .	10
4.2	文件基本信息函数 . . . . .	11
4.3	文件与目录 . . . . .	11
4.3.1	创建、复制和删除目录 . . . . .	11
4.4	文件名 . . . . .	11
4.4.1	文件名扩展 . . . . .	12
<b>5</b>	<b>org 实践</b>	<b>12</b>
5.1	org 模式简介 . . . . .	12
5.2	文档结构 . . . . .	12
5.2.1	目录结构 . . . . .	12
5.2.2	显示与隐藏 . . . . .	12
5.2.3	列表 . . . . .	13
5.2.4	块结构 . . . . .	13
5.3	表格 . . . . .	13
5.4	超链接 . . . . .	13
5.4.1	链接格式 . . . . .	13
5.5	待办事项 . . . . .	14
5.6	日程表 (Agenda View) . . . . .	14
5.6.1	日程文件 (Agenda files) . . . . .	14
5.6.2	分发按钮 . . . . .	14
5.6.3	内建 Agenda 视图 . . . . .	14
5.6.4	计划 Schedule . . . . .	14
5.7	Org 快速记录 . . . . .	14
5.7.1	如何使用 org-capture? . . . . .	15
5.7.2	org 条目复制与移动 . . . . .	15
5.7.3	记录模板 . . . . .	15
5.8	Org 的导出功能 . . . . .	16
5.8.1	导出的 Dispatcher . . . . .	16

5.9	org-capture.el . . . . .	16
<b>6</b>	<b>书签</b>	<b>16</b>
6.1	emacs 的书签功能 . . . . .	16
6.1.1	设置一个书签 . . . . .	17
6.1.2	列出保存的书签 . . . . .	17
6.1.3	跳转到一个书签 . . . . .	17
6.1.4	删除一个书签 . . . . .	18
6.1.5	保存书签 . . . . .	18
6.1.6	其他设置 . . . . .	18
6.1.7	bookmark+ . . . . .	18
<b>7</b>	<b>dired 实践</b>	<b>19</b>
7.1	dired 文件管理 . . . . .	19
7.1.1	常用命令 . . . . .	19
7.1.2	标记与操作 . . . . .	20
7.1.3	批量执行 Shell 命令 . . . . .	20
7.1.4	dired 的扩展 . . . . .	20
<b>8</b>	<b>magit 实践</b>	<b>20</b>
8.1	magit 模式简介 . . . . .	20
8.2	常用命令 . . . . .	21
8.3	分支操作 . . . . .	21
<b>9</b>	<b>包管理</b>	<b>21</b>
9.1	Emacs 的 Package-Mode . . . . .	21
9.2	包列表 . . . . .	21

---

## 1 基本数据类型

### 1.1 Lisp 的数据类型

Lisp 的对象至少属于一种数据类型。Emacs 里最基础的数据类型称之为原始类型 (primitive type)，这些原始类型包括整型、浮点、cons、符号 (symbol)、字符串、数组、哈希表 (hash-table)、subr、二进制编码函数 (byte-code function)，再加上一些特殊的类型，如 buffer。同时，每种原始类型都有一个对应的函数去校验对象是否属于其类型。

## 1.2 标识类型 (Symbols)

标识类型是一种有唯一名的（命名）对象。它常用于变量及函数名。判断一个对象是否为标识类型用 `symbolp object` 方法。

### 1.2.1 标识类型的组成

每个标识对象有四部分组成，每部分称之为单元，每个单元指向其他对象。

1. 名字 即标识对象名，获取标识对象名函数为 `(symbol-name symbol)`
2. 变量值 当标识对象作为变量时的值
3. 函数 标识函数定义，函数单元可保存另一个标识对象、或者 keymap、或者一个键盘宏
4. 属性列表 标识对象的属性列表 (plist)，获取属性列表函数为 `(symbol-plist symbol)` 注意，其中 **名字**为字符串类型，不可改变，其他三个组成部分可被赋值为任意 lisp 对象。其值为属性列表 (plist)。一个以冒号开头的符号类型称之为 keyword symbol，它常用于常量类型。

### 1.2.2 定义标识类型

定义标识对象是一种特殊的 lisp 表达式，它表示将标识类型用于特殊用途。

1. `defvar` 和 `defconst` 它们是一种特殊表达式 (Special Forms)，它定义一个标识作为全局变量。实际应用中往往使用 `*setq*`，它可以将任意变量值绑定到标识对象。
2. `defun` 用于定义函数，它的作用是创建一个 lambda 表达式，并将其存储在标识对象的函数单元里。
3. `defmacro` 定义标识符为宏，创建一个宏对象并前对象保存在函数单元里。

### 1.2.3 标识符操作函数

常见的与标识类型相关的函数有 `make-symbol` 和 `intern`

1. `make-symbol`

`make-symbol name`

这个函数返回一个新的标识对象，它的名字是 **name** (必须为字符串)

## 2. intern

```
intern name &optional obarray
```

这个函数返回一个被绑定的名字为 **name** 的标识对象。如果标识符不在变量 **obarray** 对应的对象数组 (obarray) 里，创建一个新的，并加入到对象数组里。当无 obarray 参数时，采用全局的对象数组 obarray。

### 1.2.4 标识符属性

标识符属性记录了标识符的额外信息，下面的函数是对标识符属性进行操作：

1. get symbol property 获取标识符属性为 property 的属性值，属性不存在返回 nil
2. put symbol property value 设置标识符属性 property 的值为 value，如果之前存在相同的属性名，其值将被覆盖。这个函数返回 value。下面是一些例子：

```
(put 'fly 'verb 'transitive)          ;; 'transitive
(put 'fly 'noun '(a buzzing little bug)) ;; (a buzzing little bug)
(get 'fly 'verb)                      ;; transitive
(symbol-plist 'fly)                   ;; (verb transitive noun (a buzzing little bug))
```

3. 标准标识符属性 下面列的一些标准标识符属性用于 emacs 的特殊用途
  - (a) :advertised-binding 用于函数的 key 的绑定
  - (b) interactive-form 用于交互函数，不要手工设置它，通过 **interactive** 特殊表达式来设置它
  - (c) disabled 如果不为 nil，对应的函数不能作为命令
  - (d) theme-face 用于主题设置

## 1.3 列表

列表是由零个或者多个元素组成的序列，列表中的每个元素都可由任意的对象组成。

### 1.3.1 关联列表 alist (Association Lists)

关联列表是一种特殊的列表，它的每个元素都是一个点对构成，如下示例：

```
(setq alist-of-colors
      '((rose . red) (lily . white) (buttercup . yellow)))
```

关联列表可以用来记录 key-value 这样的 map 结构；对每个元素做 car 操作拿到 key，做 cdr 操作即拿到相关系的 value。

#### 1. 关联列表操作

- (assoc key alist) 获取列表第一个 key 所关联的值；下面是一个例子：

```
ELISP> (assoc 'rose alist-of-colors)
(rose . red)
```

注意：这里用得比较是 equal 函数，如想用 eq 函数，请采用 (assq key alist) 这个函数

- (rassoc value alist) 获取列表第一个 value 为 **value** 所关联的值；
- (assoc-default key alist) 获取列表中第一个 key 为 **key** 的 value；

```
ELISP> (assoc-default 'rose alist-of-colors)
red
```

### 1.3.2 属性列表 plist (Property Lists)

属性列表是由成对元素 (paired elements) 组成的列表，每个元素对关联着一个属性的名及其对应属性值。下面是一个例子：

```
(pine cones numbers (1 2 3) color "blue")
```

这里 pine 关联其值为 cons，numbers 关联其值为 (1 2 3)，一般每个元素对的关联值是由 symbol 类型组成的。

#### 1. 属性列表的操作

- (plist-get plist property) 返回属性列表中属性名为 property 的属性值：

```

ELISP> (setq pl '(pine cones numbers (1 2 3) color "blue"))
(pine cones numbers
  (1 2 3)
  color "blue")
ELISP> (plist-get pl 'pine)
cones
ELISP> (plist-get pl 'numbers)
(1 2 3)

```

- (plist-member plist property) 如果属性列表 plist 中含有属性 property, 则返回 non-nil。
- (plist-put plist property value) 保存属性 property 及值 value 的属性对

```

(setq my-plist '(bar t foo 4))           ;; => (bar t foo 4)
(setq my-plist (plist-put my-plist 'foo 69))   ;; => (bar t foo 69)
(setq my-plist (plist-put my-plist 'quux '(a))) ;; => (bar t foo 69 quux (a))

```

### 1.3.3 对列表进行排序

对列表进行排序可以采用 sort 这个函数 (**sort list predicate**)。不过这个函数是有副作用的, 这个函数调用后会改变原有 list 的结构。第三个参数 predicate 传入的是一个比较函数, 它接收两个参数。如果是想递增排序, 当第一个参数小于第二个参数时返回 non-nil, 否则返回 nil。注意这个 sort 函数对 list 的排序, 始终保持 car 部分不变。下面是一个例子:

```

ELISP> (setq nums '(1 3 2 6 5 4 0))
(1 3 2 6 5 4 0)
ELISP> (sort nums '<)
(0 1 2 3 4 5 6)
ELISP> nums
(1 2 3 4 5 6)

```

注意这里的 nums 排序后, 的 car 与原来 list 的 car 是一样的。所以一般采用重新赋值的方式 (**setq nums (sort nums ' <)**)

## 2 控制结构

### 2.1 控制结构

Lisp 程序由一系列表达式结成, Lisp 解释器解释并执行这些表达式。在执行这些表达式过程中用到了控制结构, Lisp 里的控制结构都是特殊表达式

(Special Forms)。最简单的控制结构是顺序执行，也是符合人的书写和线性习惯。其他控制结构有：条件语句、迭代。

## 2.2 顺序结构

顺序结构是最简单的控件结构，如果想自己定义顺序结构，可以采用 **progn** 这个特殊表达式：

```
(progn a b c ...)
```

它的执行结构是最后一句的结果。与之类似有另一个特殊表达式 (**prog1 form1 forms...**) 它也是顺序执行，不过它的返回值是 form1 的返回值。同时还有一个特殊表达式 (**prog2 form1 form2 forms...**) 它效果也是一样，不过它返回的是 form2 的值。

## 2.3 条件语句

ELisp 提供四种条件语句：if、when、unless 和 cond

# 3 函数

## 3.1 什么是函数？

函数是有传入参数的可计算的单元。每个函数的计算结果为函数返回值。大部分计算机语言里，每个函数有其自己函数名。从严格意义来说，lisp 函数是没有名字的。lisp 函数其本质是一个对象，该对象可关联到一个标识符（本书把 Symbol 翻译成标识符），这个标识符就是函数名。

## 3.2 定义函数

定义一个函数的语法如下：

```
defun name args [doc] [declare] [interactive] body. . .
```

### 3.2.1 检查一个函数是否定义

检查一个变量是否绑定到函数，**fboundp symbol**，还有一个函数 (**functionp OBJECT**)

```
(fboundp 'info)                ; t
(fboundp 'setq)                 ; t
(fboundp 'xyz)                  ; nil
(functionp (lambda () (message "Anonymous Functions"))) ; t
(fboundp (lambda () (message "Anonymous Functions"))) ; *** Eval error ***
```



### 3.2.2 函数参数

有些参数是可选的，当用户没有传时，设置一个默认值，下面是一个例子：

```
(defun cookbook/fun-option-parameter (a &optional b &rest e)
  (when (null b)
    (message "parameter b is not provided")
    (setq b "ddd")) ;; set to default value
  (message "a=%s, b=%s" a b))
```

函数 `cookbook/fun-option-parameter` 中，`a` 为必传参数，`b` 为可选择参数，`e` 为其余参数，当实际传入的参数大于 2 时，其他参数将组成一个 list 绑定到 `e` 上。

## 3.3 函数调用

最通用的函数的调用方式是对 list 进行求值，如对 list (`concat "a" "b"`) 进行求值，相当于用参数“a”和“b”调用函数 `concat`。这种方式用在你清楚程序上下文中调用哪个函数、传递哪个参数。但有时候你需要在程序运行时才决定调用哪个函数。针对这种情况，Emacs Lisp 提供了另外两种方式 **funcall** 和 **apply**。其中 `apply` 一般用在运行时行决定传递多少个参数的情况。

### 3.3.1 funcall

`funcall` 它的语法如下：

```
funcall function &rest arguments
```

这里 `funcall` 本身是一个函数，因此 `funcall` 在调用前，它的所有参数都将事先做求值运算，对 `funcall` 来说它不知道具体的求值过程。同时请注意第一个参数 **function** 必须为一个 Lisp 函数或者原生函数，不能为特殊表达式 (Special Forms) 和宏，但可以为匿名函数 (lambda 表达式)。下面为一个例子：

```
(setq f 'list)           ;; list
(funcall f 'x 'y 'z)     ;; (x y z)
```

### 3.3.2 apply

`apply` 的定义如下：

```
apply function &rest arguments
```

`apply` 与 `funcall` 作用一样，唯独有一点不一样：它的 `arguments` 是一个对象列表，每个对象作为单独的参数传入，如下例子：

```
(setq f 'list)      ;; list
(apply f 'x 'y 'z)   ;; Wrong type argument: listp, z
(apply '+ 1 2 '(3 4)) ;; 10
```

### 3.3.3 映射函数 (Mapping Functions)

映射函数操作是指对一个列表或者集合逐个执行指定函数，这节介绍几个常的映射函数：`mapcar`, `mapc`, 和 `mapconcat`。

`mapcar function sequence`

这个函数功能有与 javascript 里的 `array.map` 操作类型，对 **sequence** 里的每个元素执行 **function** 操作，返回操作结果列表。这个函数应用非常广泛，以下几个应用举例：

```
(mapcar 'car '((a b) (c d) (e f))) ;; (a c e)
(mapcar '1+ [1 2 3])                ;; (2 3 4)
(mapcar 'string "abc")               ;; ("a" "b" "c")
```

`mapc` 与 `mapcar` 调用方式一样，唯一不同的点是它始终返回的是 **sequence**。

`mapconcat function sequence separator`

`mapconcat` 对 **sequence** 里的每个元素调用 **function** 最后将结果拼接成一个字符串作为返回值，采用 `separator` 作为拼接符。

## 3.4 匿名函数

在 elisp 里有三种方式可以定义匿名函数：`lambda` 宏、`function` 特殊表达式、`#'` 可读语法。

### 3.4.1 lambda 宏

它的定义如下：

```
lambda args [doc] [interactive] body. . .
```

这个宏返回一个匿名函数，实际上这个宏是自引用 (self-quoting)。

```
(lambda (x) (* x x)) ;; (lambda (x) (* x x))
```

下面是另一个例子：

```
(lambda (x)
  "Return the hyperbolic cosine of X."
  (* 0.5 (+ (exp x) (exp (- x)))))
```

上面的表达式被计算成一个函数对象。

### 3.4.2 function 特殊表达式

定义如下：

```
function function-object
```

这是一个特殊表达式 (Special Forms)，表示对 **function-object** 不作求值操作。其实在实际使用中我们往往采用它的简写 **#'**，因此下面三个是等价的：

```
(lambda (x) (* x x))
(function (lambda (x) (* x x)))
#'(lambda (x) (* x x))
```

## 3.5 获取函数单元内容

当我们把一个标识符 (Symbol) 定义为函数，其本质是将函数对象存储在标签符号对应的函数单元（标识符还有一个变量单元用于存储变量），下面是介绍函数单元处理方法：

### 3.5.1 symbol-function

定义如下：

```
symbol-function symbol
```

这个函数返回标识符 **symbol** 对应的函数对象，它不校验返回的函数是否为合法的函数。如果 **symbol** 的函数单元为空，返回 **nil**。

### 3.5.2 fboundp

用于判断 **symbol** 对应的函数单元是否为 **nil**

```
fboundp symbol
```

当 **symbol** 在函数单元有一个对象时返回 **t**，否则返回 **nil**。

## 3.6 特殊表达式 (Special Forms) 和宏

有些与函数看起来很像的类型，它们也接受参数，同时计算出结果。但在 Elisp 里，他们不被当成函数，下面给出简单介绍：

### 3.6.1 内建函数 (primitive)

是用 C 语言写的，可被调用的函数；

### 3.6.2 special form

一种类型的内建函数，如 if, and 和 while

## 4 文件

### 4.1 文件及访问

文件是操作系统永久保存数据的单元，为了编辑文件，我们必要告诉 Emacs 去读取一个文件，并将文件的内容保存在一个 Buffer 里，这样 Buffer 与文件就关联在一起。下面介绍与文件访问相关的函数，由于历史原因这些函数的命令都是以 **find-** 开头的，不是以 **visit-** 开头。

#### 4.1.1 打开文件

如果想在 buffer 里打开一个文件，其命令是 **find-file** (C-x C-f)。当文件已经在 buffer 中存在时，这个命令返回文件对应的 buffer。如果当前没有 buffer 对应文件，则，创建一个 buffer，并将其文件内容读到 buffer 中，并返回这个 buffer。字义如下：

```
find-file filename &optional wildcards
```

这个函数有一个对应的 hook 变量，叫 **find-file-hook** 它的值是一个函数列表。这些函数在文件被打开后依次执行。

#### 4.1.2 文件保存

文件被载入到 buffer 后，我们可以对其进行修改；修改完了后，将内容保存回文件，其对应的函数为：

```
save-buffer &optional backup-option
```

文件保存对应有两个 hook 变量，为：**before-save-hook** 和 **after-save-hook** 分别表示保存前的 hook 函数列表和保存后的 hook 函数列表。与之类似的还有一个函数 **write-file**

```
write-file filename &optional confirm
```

这个函数的功能是将当前 buffer 的内容写入到 filename 对应的文件中，并将当前 buffer 与这个文件进行关联

## 4.2 文件基本信息函数

下面介绍一些与文件基本信息相关的函数

1. 文件是否存在

```
file-exists-p filename
```

与之类似的有: **file-readable-p**、**file-executable-p**、**file-writable-p**、**file-directory-p** 这几个函数。

## 4.3 文件与目录

判断文件是否在一个目录下，怎么做？

```
file-in-directory-p file dir
```

如果 file 是一个在目录 dir 或者 dir 子目录下的文件，则返回 t。如果 file 与 dir 处于同一目录，也返回 t。如果想列出一个目录下的所有文件，那就要用到 **directory-files** 这个函数，其定义如下：

```
directory-files directory &optional full-name match-regexp nosort
```

这个函数按字母顺序返回目录 directory 下的所有文件。参数 full-name 不为 nil 时，则返回每个文件的绝对路径，否则返回相对路径。match-regexp 如果不是 nil，该函数返回只与 match-regexp 相匹配的文件列表。nosort 如果不为 nil，则不按字母排序。

### 4.3.1 创建、复制和删除目录

对目录的创建、复制和删除都有相关的处理函数，下面一一介绍：

```
make-directory dirname &optional parents
```

**make-directory** 创建一个目录名为 dirname 的目录

## 4.4 文件名

下面介绍一些与文件名操作有关的函数

`file-name-directory lename`

**file-name-directory** 返回的文件名里的目录部分，如果文件名里没有包含目录部分，则返回 `nil`。与这个函数对应的一个函数为 **file-name-nondirectory**，它返回非目录部分。

### 4.4.1 文件名扩展

**expand-file-name** 这个函数将文件名转成绝对文件名：

`expand-file-name filename &optional directory`

如果 `directory` 参数存在，将 `filename` 作为其相对路径，否则使用 **default-directory** 变量。这个函数在写 `elisp` 代码时经常用到，下面是一些例子：

```
(expand-file-name "foo")
"/xcssun/users/rms/lewis/foo"
(expand-file-name "../foo")
"/xcssun/users/rms/foo"
(expand-file-name "foo" "/usr/spool/") "/usr/spool/foo"
```

## 5 org 实践

### 5.1 org 模式简介

Emacs 的 `org-mode` 可用于记笔记、管理自己的待办事项 (TODO lists)，同时，也可用于管理项目。它是一个高效的纯文本编辑系统。

### 5.2 文档结构

Org 是基于 `Outline-mode`，并提供灵活的命令编辑结构化的文档。其文档结构语法跟 `markdown` 很类似。

#### 5.2.1 目录结构

Org 的目录结构在每行最左边以星号标记，星号越多，标题层级越深。下面是一些例子：

```
\* 一级目录
\** 二级目录
\*** 三级目录
\* 另一个一级目录
```

### 5.2.2 显示与隐藏

目录结构下的内容可以隐藏起来，通常用采用 **TAB** 和 *S-TAB* 这两个命令来切换。

### 5.2.3 列表

Org 提供三种类型的列表：有序列表、无序列表和描述列表

1. 有序列表以 '1.' 或者 '1)'
2. 无序列表以 '-', '+' 或者 '\*'
3. 描述列表

### 5.2.4 块结构

在 Org 文档中，加入代码块这种类型的块结构，都是采用 begin...end 这种模式，下面是一个例子：

```
\#+BEGIN_EXAMPLE  
\#+END_EXAMPLE
```

## 5.3 表格

## 5.4 超链接

Org 模式提供了比较好用的超链接方式，可以链接到普通网页、文件、email 等。

### 5.4.1 链接格式

Org 模式支持两种链接，即，内部链接和外部链接。它们有相同的格式：

[[链接]][描述]] 或 当只有链接没有描述 [[链接]]

一旦链接编辑完成，在 org 模式下，只显示 **描述**部分，而不会显示整体（后一种是只显示链接）。为了编辑链接和描述，需要通过快捷键 **C-c C-l** 来完成（注意：编辑结束后按 Enter 完成修改操作）。

1. 内部链接 内部链接是指向当前文件的链接，它的链接格式：

[[#链接ID]]

其中 **链接 ID** 是文档中唯一的标识 ID

2. 外部链接 Org 支持的外部链接有很多中形式，如文件、网页、新闻组、电子邮件信息、BBDB 数据条目等。它们以一个短的标识字符串打头，紧接着是一个冒号，冒号后面没有空格字符。
3. 链接处理相关命令
  - org-store-link 保存当前位置的一个链接，以备后面插入使用
  - org-insert-link 插入链接，绑定的快捷键为 C-c C-l，如果光标正在一个链接上，那么 C-c C-l 的行为是编辑这个链接及其描述

## 5.5 待办事项

Org 模式用来管理自己的 TODO list 非常方便

## 5.6 日程表 (Agenda View)

我们可以用 Org 来安排自己的行程

### 5.6.1 日程文件 (Agenda files)

变量 org-agenda-files 保存了一个文件列表，这些文件用来记录日程，下面是一些操作函数：C-c [ 将当前文件加入到 agenda 文件列表最前页面 org-agenda-file-to-front C-c ] 将当前文件从 agenda 文件列表中删除 org-remove-file

### 5.6.2 分发按键

默认采用 C-c a，接下的默认的命令有：

- a 创建一个日程
- t/T 创建一个 TODO items
- L 对当前文件生成 timeline

### 5.6.3 内建 Agenda 视图

### 5.6.4 计划 Schedule

用 org 来安排日程

- org-schedule 将当前 TODO 添加计划时间



## 5.7 Org 快速记录

有时候，突然想到一些待办事项，或者一些突发的灵感。这时，我们想用 emacs 快速记录它，Org-Capture 提供这个好用的功能。它的前身是 org-remember.el（注：从 org 8.0 开始，org-remember 被 org-capture）替代。

### 5.7.1 如何使用 org-capture?

快速记录的命令为 **M-x org-capture**，默认绑定的快捷键为 **C-c c**。当这个命令被调用后，你可以使用自己定义好的模板快速创建记录。一旦完成内容的输入，按下 **C-c C-c** (org-capture-finalize)，来完成。然后，你就能继续做你当下的事。如果想跳转到刚刚创建的记录的 buffer，用 **C-u C-c C-c** 来完成。如果想中途中止输入，只要按下 **C-c C-k** (org-capture-kill)。

### 5.7.2 org 条目复制与移动

有时候，我们想将当前的某条目转移到其他文件或者其他项目里。这时，我们会用到 org-copy 和 org-refile 这两个命令。它们对应的快捷键分别是 **C-c M-w** 及 **C-c C-w**。这里有一个问题是，目标文件如何配置？目录文件的配置由一个变量决定，**org-refile-targets**，我自己的配置如下：

```
(setq org-refile-targets
      '((nil :maxlevel . 3)          ;; 当前文件的最大层级
        (aborn-gtd-files :maxlevel . 3)))
```

注意：我这时将文件放在 aborn-gtd-files 文件列表里。

### 5.7.3 记录模板

记录的模板为一个列表变量，org-capture-templates，列表的每条记录由如下几段组成：

```
("t" "Todo" entry (file+headline (expand-file-name org-default-notes-file org-directory)
                                   "* TODO %?\n  创建于:%T  %i\n"))
```

1. 快捷键 如例子中的那样，“t”表示对应按键 t 这个快捷键。它能帮助我们快速地选中哪条模板进行快速记录。
2. 描述 接下来是一段简单的描述
3. 类型 第三段表示类型，有五种类型：entry item checkitem table-line plain

- entry 普通的 Org 结点，保证目标文件为 org-mode 文件，插入的时候将作为目录结点的子结点

(如果没有，将做为顶级结点)；

- item 与 entry 类似，不同点在于它的目标文件可以为简单的纯文本文件；
- checkitem 复选条目；
- table-line 在目标文件中的第一个 table 中插入新行；
- plain 纯文本记录

4. 目标文件 第四个字段配置目标文件

5. 模板 第五个字段表示模板，模板参数 含义如下：

- %t 只有日期的时间戳
- %T 日期 + 时间的时间戳
- %u,%U 如上，只不过它们是 inactive 的
- %i 初始化文本，当前上下文将作为初始化文本

6. 属性 properties 最后一个字段表示属性列表，支持以下属性配置：

- :prepend 一般一个记录条目插入在目标文件的最后，这个属性可以将条目插入在最前
- :immediate-finish 立刻完成，没有交互
- :clock-in 对这个条目设置闹钟
- :kill-buffer 如果目标文件没有相应的访问 buffer, 插入后，自动关闭 buffer

## 5.8 Org 的导出功能

Org 文件支持导出多种格式的目标文件，如 ASCII 文件、HTML 文件 (用于发布为 Web)、PDF 文档等。

### 5.8.1 导出的 Dispatcher

任何导出命令都有一个前缀按键，我们称之为 Dispatcher，为 **C-c C-e**

## 5.9 org-capture.el

Org 8.0 以后版本采用 org-capture.el 取代原有的 org-remember.el

## 6 书签

### 6.1 emacs 的书签功能

emacs 的书签用于记录你在文件中的阅读位置。它有点类似寄存器，跟寄存器一样，因为它也能记录位置位置。但同寄存器有两点不一样：1. 它有更长的名字；2. 当 emacs 关闭的时候，它会自动持久化到磁盘。

#### 6.1.1 设置一个书签

当我们阅读一个很长的文档，没能一口气读完时。我们希望记住当前文档的最后阅读的位置，以便下次再用 emacs 阅读的时候能快速定位到。那么，我们设置一个书签，通过 **bookmark-set** 对应快捷键为 **C-x r m**

#### 6.1.2 列出保存的书签

**bookmark-bmenu-list** 对应快捷键为 **C-x r l**，它将打开一个 \*Bookmark List\* 的 buffer 同时列出所有保存的书签。

1. 书签列表 \*Bookmark List\* 在 \*Bookmark List\* 这个 buffer 里，有以下快捷键可以使用：
  - a 显示当前书签的标注信息；
  - A 在另一个 buffer 中显示所有书签的所有标注信息；
  - d 标记书签，以便用来删除 (x - 执行删除)；
  - e 编辑当前书签的标注信息；
  - m 标记书签，以便用于进一步显示和其他操作 (v - 访问这个书签)；
  - o 选中当前书签，并显示在另一个 window 中；
  - C-o 在另一个 window 中切换到当前这个书签；
  - r 重命名当前书签；
  - w 将当前书签的位置显示在 minibuffer 里。

#### 6.1.3 跳转到一个书签

使用 **bookmark-jump** 函数，可以跳转到一个特定的书签，它绑定的快捷键为 **C-x r b**。如果你的 emacs 中安装了 helm 这个插件，你也可以使用 **helm-bookmarks** 这个命令来快速查找书签，并跳转到书签位置。

1. helm-bookmarks 通过 helm-bookmarks 命令来查找并跳转书签如下图：

2. 修改默认排序 书签查找和跳转的时候，默认的书签排序是按字母排序的。如果想将最近访问的书签放在最前面，将下面代码添加到你的 emacs 配置文件中。

```
(defadvice bookmark-jump (after bookmark-jump activate)
  (let ((latest (bookmark-get-bookmark bookmark)))
    (setq bookmark-alist (delq latest bookmark-alist))
    (add-to-list 'bookmark-alist latest)))
```

#### 6.1.4 删除一个书签

删除一个书签对应的命令为 **bookmark-delete** 。

#### 6.1.5 保存书签

最新版本 emacs（老版本的书签保存在 `~/.emacs.bmk`），在退出的时候会自动保存书签。如果想手动保存书签的话，可以采用 **bookmark-save** 这个函数命令。默认的情况，emacs 会将书签保存在 **bookmark-default-file** 变量对应的文件中。在我的机器中，对应的文件如下：

```
ELISP> bookmark-default-file
"/Users/aborn/.emacs.d/.cache/bookmarks"
ELISP>
```

#### 6.1.6 其他设置

有一个变量 **bookmark-save-flag**。如果这个变量的值为一个数值，它表示修改（或新增）多少次书签后，emacs 会自动保存书签到磁盘。当这个变量的值被设置为 1 时，每次对 bookmark 的改动，emacs 就会自动保存内容到磁盘相应位置（这样可以防止 emacs 突然 crash 时 bookmark 的丢失）。如果这个值设置为 nil，表示 emacs 不会主动保存 bookmark，除非用户手动调用 **M-x bookmark-save**。

#### 6.1.7 bookmark+

bookmark+ 是对 bookmark 的一个扩展的包。它有更多的功能：

1. 原始的 bookmark 只能对文件位置记录，bookmark+ 对孤立的 buffer(没有关联文件的 buffer) 也能保存书签；
2. 支持对书签进行打 tag；

3. 对文档的某个区域保存为书签，而不仅仅是某个位置;
4. 记录了每个书签的访问次数，及最后一次的访问时间，可以基于它们排序;
5. 多个书签可以有相同的名字;
6. 可以对函数、变量等加书签。

更多功能请参考: <https://www.emacswiki.org/emacs/BookmarkPlus#Bookmark%2b>

## 7 dired 实践

### 7.1 dired 文件管理

dired 的全称为 Directory Edit，即目录编辑，是一个非常老的模式。是 Emacs 下的一个文件管理神器！进入当前文件的 dired 文件管理，\*M-x dired\*。

#### 7.1.1 常用命令

1. 光标移动命令
  - **n** 下移
  - **p** 上移
2. 文件操作
  - **C** 拷贝文件，dired-recursive-copies 变量决定了拷贝的类型，一般为 top
  - **D** 删除文件，类似的有一个 dired-recursive-deletes 变量可以控制递归删除
  - **R** 重命名或者移动文件
  - **D** 删除文件或者目录
  - **+** 创建目录
  - **Z** gzip 压缩文件
  - **w** 复制文件名 (C-u 则复制相对于 dired 当前目录的相对目录)
  - **A** 对文件进行正则表达式搜索，会在第一个匹配的地方停下，然后使用 M-, 搜索下一个匹配。

### 3. 其他命令

- **RET** 打开文件或者目录
- **g** 刷新当前 dired buffer
- **k** 隐藏不想显示出来的文件
- **q** 退出

#### 7.1.2 标记与操作

dired 可以对多个文件进行标记，然后进行批量操作。一个典型的是采用 **d** 对当前文件打上删除标记，然后使用 **x** 命令来删除所有标记的文件。

##### 1. 标记操作命令

- **m** 以星标记当前文件
- **\* \*** 标记所有可执行文件
- **\* @** 标记所有符号链接
- **\* /** 标记所有目录 (不包括. 和.. )
- **\* s** 标记所有文件 (不包括. 和.. )
- **\* .** 标记具有给定扩展名的文件
- **% m REGEXP <RET>** 或 **\* % REGEXP <RET>** 标记所有匹配到给定的正则表达式的文件。
- **% g REGEXP <RET>** 标记所有文件内容匹配到给定的正则表达式的文件。

##### 2. 其他标记相关命令

- **u** 去除当前行的标记
- **U** 去除所有标记

#### 7.1.3 批量执行 Shell 命令

在 dired 模式下，可以对标记的文件批量执行 shell 命令（如果没有标记文件，则对当前文件执行 shell），运行命令 **dired-do-shell-command** (绑定的快捷键为 **!**)，相应的它有一个对应的异步操作的命令 **dired-do-async-shell-command** (绑定的快捷键为 **&**)。

#### 7.1.4 dired 的扩展

1. diredful diredful 可使得不同的文件显示不同的颜色, 是一个非常好的扩展
2. dired-icon dired-icon 根据文件类型显示相应 icon

## 8 magit 实践

### 8.1 magit 模式简介

magit 是 emacs 下版本管理的强大武器

### 8.2 常用命令

- **magit-dispatch-popup** 命令分发器, 在 spacemacs 里绑定到 **M-m g m**
- **magit-diff** 相当于 `git diff`, 当进入 diff-buffer 后按 **g** 更新之
- **magit-status** 相当于 `git status`, 进入 status-buffer 后按 **s** 添加文件或文件夹到本地仓库
- **magit-checkout** 切换分支
- **magit-branch-and-checkout** 从当前分支切一个新的分支

### 8.3 分支操作

常用的分支操作如下:

- (magit-branch-delete) **b k** 删除一个或多个 (本地) 分支
- (magit-branch-rename) **b r** 对当前 Branch 进行重命名
- (magit-get-current-branch) 获取当前分支名

## 9 包管理

### 9.1 Emacs 的 Package-Mode

当通过 `*M-x list-package*` 命令打开一个 `*Package*` 的 Buffer, 它有如下命令:

1. i 标识安装 (u 取消标识)
2. x 执行安装操作
3. d 标识删除 (x 执行删除操作)
4. U 标识要更新的 package
5. ~ 标识所有废弃包
6. M-x package-autoremove 删除那些无用的旧包

## 9.2 包列表

1. elisp-slime-nav 写 elisp 代码时, 可用于跳转到函数的定义