

# An Emacs Tutorial for Vim User

w0mTea

February 1, 2015

# Contents

<b>1 前言</b>	<b>2</b>
1.1 为什么离开 vim 改用 emacs . . . . .	2
<b>2 emacs 的安装</b>	<b>3</b>
<b>3 基础知识</b>	<b>4</b>
3.1 快捷键的约定 . . . . .	4
3.2 常用快捷键 . . . . .	4
<b>4 简单配置</b>	<b>6</b>
<b>5 过渡——evil-mode</b>	<b>7</b>
5.1 安装 . . . . .	7
5.2 启用 . . . . .	9
<b>6 emacs 中的包管理</b>	<b>10</b>
6.1 el-get . . . . .	10
6.1.1 安装 . . . . .	10
6.1.2 更新 . . . . .	10
6.1.3 删除 . . . . .	11
6.1.4 recipe 文件 . . . . .	11
6.1.5 本节参考资料 . . . . .	11
6.2 ELPA . . . . .	11
6.2.1 简单配置 . . . . .	11
6.2.2 安装/删除/升级 . . . . .	12
6.2.3 本节参考资料 . . . . .	12
6.3 手动安装 . . . . .	12
<b>7 保护你的手指</b>	<b>14</b>
7.1 用手掌外缘按 ctrl . . . . .	14
7.2 改键 . . . . .	14

7.3 踏板	15
<b>8 编程语言配置</b>	<b>16</b>
8.1 通用配置	16
8.2 C/C++	16
8.2.1 缩进	16
8.2.2 各种扩展	17
8.3 Python	17
8.3.1 python.el	17
8.3.2 python-mode.el	18
8.3.3 便利的小插件	19
8.3.4 参考资料	20
8.4 Lisp	20
8.5 Perl	20
8.6 Ruby	20
8.7 Markdown	20
8.7.1 插件及安装	20
8.7.2 配置	20
8.7.3 另一种写 Markdown 的方式	21
8.8 Java	21
<b>9 重量级应用——org-mode</b>	<b>22</b>
9.1 安装	22
9.2 文档结构	22
9.2.1 标题	22
9.2.2 折叠循环	23
9.2.3 标题的结构化操作	23
9.3 列表	24
9.4 表格	24
9.4.1 基础	25
9.4.2 计算	25
9.4.3 快捷键	25
9.5 链接	26
9.5.1 内部链接	26
9.5.2 外部链接	26
9.6 代码块	27
9.6.1 基本用法	27
9.6.2 画图	27
9.7 元数据	29
9.8 To-Do	30
9.8.1 基础	30
9.8.2 快捷键	32
9.9 导出	32
9.9.1 导出后端设置	33
9.9.2 导出通用设置	35
9.9.3 HTML	36

<i>CONTENTS</i>	3
9.9.4 PDF . . . . .	36
9.9.5 Markdown . . . . .	39
9.9.6 ODT . . . . .	39
9.10 参考资料 . . . . .	39
<b>10 文档和资料</b>	<b>40</b>
<b>11 致谢</b>	<b>41</b>
<b>12 结尾</b>	<b>42</b>

# Chapter 1

## 前言

写这篇教程的起因在于向许多 `vimer` 推荐 `org-mode` 时，他们总是觉得虽然 `org-mode` 功能强大，可是使用 `emacs` 总是有着一些障碍。作为一个同样从 `vim` 转向 `emacs` 的人，我觉得或许分享我的经验可以让他们更快的接受 `emacs`，从而体验 `emacs` 的美妙。

本文默认读者所使用的是类 `unix` 操作系统，因此对于 `windows` 用户来说，如果某些操作在你的电脑上无法执行，请不必大惊小怪。

### 1.1 为什么离开 `vim` 改用 `emacs`

在绝大多数 `unix/linux` 教程都教我们用 `vi/vim` 的情况下，许多人从开始就习惯了 `vim` 的一切：它的简洁、高效，它的模式，它简单直接的按键绑定……于是就这样，一批批的 `vim` 用户诞生。然而对于一直和 `vim` 并称的 `emacs`，我们往往只是知道它很强，却不清楚它到底强在哪里。如果你想清楚的了解这个 `vim` 的老对手到底凭什么能和 `vim` 分庭抗礼，那么，你可以试着使用它。

如果你是一个 `lisp` 爱好者，那么你绝对不能错过 `emacs`。或许经过漫长的配置，`vim` 同样可以很好的支持 `lisp`，可是在 `emacs` 上，`lisp` 天然就可以被良好的支持。同时，`emacs` 也可以被 `lisp` 扩展。因此，你写的每一行 `lisp` 代码，或许都可以让你的 `emacs` 变的更好用。写 `lisp`，也是让我离开 `vim` 转向 `emacs` 的契机。

如果你厌倦了 `unix` 哲学，厌倦了做一件事只能用无数的小工具来组合；或者你喜欢一个大而全的东西，可以帮你做各种各样的事情，那么 `emacs` 无疑是一个更好的选择。事实上，往往 `emacs` 用的越久，每天对着它的时间也越长，它帮你做的事情也越超出简单的文本编辑。从开始的写文档写代码，到收邮件玩游戏写幻灯片，甚至到煮咖啡，`emacs` 能帮我们做越来越多的事情。

## Chapter 2

# emacs 的安装

emacs 并不像 vim/vi 那样几乎被所有类 unix 系统内置，因此我们往往需要手动安装 emacs。

对于有包管理器的系统，使用包管理器通常都可以成功安装 emacs。需要注意的是，某些发行版的仓库默认不安装 emacs 的 GUI 版本，因此需要手动安装 emacs-x11 或类似名字的包。

需要注意的是，emacs 的 GUI 版本并不是 xemacs。我们使用的 emacs 实际上是 gnu emacs 这个分支，而 xemacs 则是另一个 emacs 的分支。虽然 xemacs 和 gnu emacs 有着相当的兼容性，可是在某些时候难免会碰到奇怪的问题。

对于没有包管理器或仓库没有 emacs 的系统，可以从 <http://ftp.gnu.org/gnu/emacs/> 下载。

# Chapter 3

## 基础知识

毫无疑问，对于一个从未是用过 `vi/vim` 的人来说，使用 `emacs` 并不是一个特别让人困惑的事，最起码开始不是，因为 `emacs` 和他之前用过的任何文本编辑器（比如 `windows` 下著名的 `notepad`）从表面上看并没有太大的不同。然而对于习惯了 `vi/vim` 模式操作的人来说，这样的操作模式真是让人无比烦恼：如果不使用一系列快捷键，效率就会变的无比低下；可是如果使用快捷键……那都是什么鬼东西！

为了让我们可以更快的接受 `emacs`，我们需要了解一些最最基础的东西。这些知识并不繁琐，但却总是有很大的帮助

### 3.1 快捷键的约定

由于没有 `vim` 那样的模式之分，`emacs` 的快捷键总是需要使用组合键。可是网上查到的 `C-n`，`C-x C-s` 都是啥意思呢？

`emacs` 中，快捷键的表示都遵循了一些固有的约定。`C-x` 表示同时按着 `ctrl` 和 `x`，`C-x C-s` 表示先按 `ctrl+x`，然后按 `ctrl+s`。当然，也可以按着 `ctrl` 不放，然后依次按 `x` 和 `s` 咯。

同样的，还有 `M`-系列的快捷键和 `S`-系列的快捷键。`M-x` 表示同时按 `alt` 和 `x`（`alt` 在不同键盘上可能不同，有可能也叫 `meta` 之类的）。`S-x` 表示同时按 `shift` 和 `x`。在这样的约定里，还有一些其他的特殊键，比如 `ESC`、`RET`（回车）等

### 3.2 常用快捷键

这里列举一些最简单同时也最最常用的快捷键。

快捷键	功能
C-x C-f	打开某文件
C-g	取消正在输入的命令
C-x C-c	关闭 <b>emacs</b>
C-x C-s	保存当前文件
M-x	运行命令



# Chapter 4

## 简单配置

emacs 采用 `emacs lisp` 作为配置语言，因此在配置里看到大堆的括号请不要惊讶。通常来说 `emacs` 的配置文件以及各种插件都以 `.el` 为后缀名。

`emacs` 的启动文件（类似于 `vim` 的 `.vimrc`）可以是下列三个中的一个：

- `~/.emacs`
- `~/.emacs.el`
- `~/.emacs.d/init.el`

虽然说使用哪一个配置文件都可以，可是我还是建议使用最后一种。因为这种方案下，你可以把 `emacs` 相关的所有配置都放在 `.emacs.d` 这个文件夹下，而不是零散的东堆西散。尤其在你的配置文件变的很大的时候，你可以轻松的把启动文件中乱七八糟的配置代码拆分成单独的模块，每个模块单独占有一个文件，并且统一放在这个文件夹里。

另外，下文提到的包管理中，最好也把其相关文件放在 `.emacs.d` 文件夹下。

至于具体的配置，可以根据自己的需求来弄。后文会提供一些好的站点帮助大家完成自己的配置文档。而一些简单的配置，会在后面的内容里提到。

# Chapter 5

## 过渡——evil-mode

对于刚接触 **emacs** 的 **vimer** 来说，最难习惯的估计就是光标移动了。如果还能像 **vim** 那样操作无疑会愉快的多。而像 **vim** 一样操作 **emacs** 并不是你一个人的想法，因此早就有别的高手实现了这一功能，那就是 **evil-mode**。

注：在下文中，配置代码在 **pdf** 中有可能无法复制，如果不想手打可以参考 **emacsWiki**: <http://www.emacswiki.org/emacs/el-get>

### 5.1 安装

**emacs** 有着若干种安装扩展的方法，具体的会在下一节讲到。这里只讲一种我最常用到的也是感觉最方便的方法：**el-get** 安装。

在你的配置文件中加入下列部分：（需要注意的是，**el-get** 的默认位置也在 **.emacs.d** 文件夹内）

```
(add-to-list 'load-path "~/.emacs.d/el-get/el-get")

(unless (require 'el-get nil t)
  (url-retrieve
   "https://github.com/dimitri/el-get/raw/master/
    el-get-install.el"
   (lambda (s)
     (end-of-buffer)
     (eval-print-last-sexp))))

;; now either el-get is `require'd already, or have been
;; `load'ed by the
;; el-get installer.

;; now set our own packages
(setq
```

```

my:el-get-packages
'(el-get                ; el-get is self-hosting
  switch-window          ; takes over C-x o
  auto-complete           ; complete as you type with
    overlays
  zencoding-mode          ; http://www.emacswiki.org/
    emacs/ZenCoding
  color-theme             ; nice looking emacs
  color-theme-tango))     ; check out
    color-theme-solarized

;;
;; Some recipes require extra tools to be installed
;;
;; Note: el-get-install requires git, so we know we have
    at least that.
;;
(when (el-get-executable-find "cvs")
  (add-to-list 'my:el-get-packages 'emacs-goodies-el)) ;
    the debian addons for emacs

(when (el-get-executable-find "svn")
  (loop for p in '(psvn                ; M-x svn-status
                  yasnippet           ; powerful snippet mode
                  )
        do (add-to-list 'my:el-get-packages p)))

(setq my:el-get-packages
      (append my:el-get-packages
              (mapcar #'el-get-source-name el-get-sources
                      )))

;; install new packages and init already installed
    packages
(el-get 'sync my:el-get-packages)

```

上述代码段会自动检查是否安装了 **el-get**，并自动在未安装的情况下安装。注意，这段代码需要系统中安装过 **git** 才能运行。同时为了在安装其他扩展时不会出问题，建议安装 **svn** 或 **cvs**。把上述代码段保存后，重新运行 **emacs**，就会自动安装 **el-get**

**el-get** 安好了，那么怎么安装 **evil-mode** 呢？回到上面那段代码，可以看到

```

;; now set our own packages
(setq
  my:el-get-packages

```

```
'(el-get
  switch-window
  auto-complete
  zencoding-mode
  color-theme
  color-theme-tango))
```

只要在这段代码内添加上我们想要的扩展，而且这个扩展恰好在 `el-get` 的仓库内，那么我们就可以自动的安装并启用对应扩展。大多数常见扩展都可以被 `el-get` 自动找到，`evil-mode` 也不例外。因此只要在这段代码中加上 `evil-mode` 就可以。搞定后和下面的差不多：

```
(setq
my:el-get-packages
'(el-get
  switch-window
  auto-complete
  evil
  zencoding-mode
  color-theme
  color-theme-tango))
```

之后重启 `emacs`，就可以安装了。

## 5.2 启用

安装成功后，只需要在配置文件中加入

```
(require 'evil)
(evil-mode 1)
```

就可以全局启用 `evil-mode`。如果想手动启动 `evil-mode`，把上面的 `1` 改成 `0`，在需要启动的时候按 `M-x evil-mode RET` 即可。

现在，`vim` 熟悉的操作，不就回来了吗？

# Chapter 6

## emacs 中的包管理

在上一章，我们已经使用了 **el-get** 来安装扩展。只需要在列表中加入你需要的包名就可以自动安装，岂不是爽的很？这一节会介绍一些 **el-get** 的其他用法。

不幸的是，并不是所有的软件包都可以通过 **el-get** 安装，因此还需要介绍一些其他方法来弥补这一小小的缺陷。

### 6.1 el-get

#### 6.1.1 安装

除了之前提到的配置文件中加入包名的方法。除此之外还可以在 **emacs** 中实时安装扩展。

**M-x el-get-install RET** 并在出现的 **Package install** 中输入想要的包名即可安装。注意：打包名时要善用 **tab** 补全哦，不仅可以省事，还可以检查是否打错以及该包是否在 **el-get** 的仓库内。

**el-get** 安装的扩展包会被记录在一个文件中，无论通过哪一种方式安装扩展，所以是否加入包名到配置文件并不会影响使用。但是我仍建议仍加入到配置文件中的包列表中去，因为那样在其他环境需要安装时，你只需要复制你的配置并运行 **emacs** 即可安装所有之前安装过的插件。但若是实时安装的插件没有加入配置，在更换环境时会遗漏一些东西。

注：在更换环境时把整个 **.emacs.d** 文件夹拷贝过去也可以避免遗漏实时安装的插件。

#### 6.1.2 更新

**M-x el-get-self-update RET** 即可更新 **el-get**

**M-x el-get-update RET** 再输入包名即可更新选定包。

**M-x el-get-update-all RET** 即可更新安装记录中所有已安装的包。注，网速慢慎用，可能会被更新信息刷屏好久

### 6.1.3 删除

删除包列表中要删除的包名，使用 **M-x el-get-remove RET** 再输入包名即可。

### 6.1.4 recipe 文件

el-get 使用一系列的 recipe 文件来处理安装包。每一个 recipe 文件都描述了安装包的名字、下载地址、版本、安装后的初始化动作等信息。这些 recipe 文件就相当于包管理器的软件源元数据，我们查询、安装等操作都需要用到它。

默认情况下，recipe 文件放在 `.emacs.d/el-get/el-get/recipes` 文件夹下。

对于 el-get 默认没有的扩展，一个安装方法便是自己写一个简单的 recipe 文件。具体的做法可以参考 [emacsWiKi el-get](#) 页。

另外，对于发布在 [emacsWiKi](#) 上的插件，可以使用 **M-x el-get-emacswiki-refresh** 来获取/刷新其 recipe 文件。因此如果要安装的包列在了 [emacsWiKi](#) 上，那么就不用自己麻烦的去安装啦。

### 6.1.5 本节参考资料

本节仅列出了少数用法，更多用法请参考下列网站：

- EmacsWiKi: <http://www.emacswiki.org/emacs/el-get>
- Github: <https://github.com/dimitri/el-get/>

## 6.2 ELPA

ELPA 也是一个 emacs 的包管理工具，在 emacs24 及以上版本已经默认集成 (package.el)。

### 6.2.1 简单配置

ELPA 需要添加一些仓库源，如下：

```
(require 'package)
(setq package-archives '(("gnu" . "http://elpa.gnu.org/
packages/")
                        ("marmalade" . "http://
marmalade-repo.org/packages/"
                        )
)
```

```
("melpa" . "http://melpa.milkbox
.net/packages/"))
```

这样就会添加 `gnu` 官方源、`marmalade`、`melpa` 三个源。

在某些情况下，启动 `emacs` 时会显示 `package.el` 没有被初始化，可以通过加入下列代码在启动文件的非末尾位置解决：

```
(setq package-enable-at-startup nil)
(package-initialize)
```

### 6.2.2 安装/删除/升级

1. **M-x list-packages RET** 会列举所有包。通过 **C-s xxx** 可以快速找到你想安装的包
2. 光标移动到对应包名上，按：
  - **RET**：会显示包的介绍
  - **i**：标记该包为待安装
  - **u**：取消标记
  - **d**：标记为待删除
  - **U**：标记为待升级。只有可升级的包才可被标记
  - **x**：执行，会删除 **d** 标记的包，安装 **i** 标记的包
  - **r**：刷新列表
  - **q**：退出列表

### 6.2.3 本节参考资料

- EmacsWiki: <http://www.emacswiki.org/emacs/ELPA>
- ergoEmacs: [http://ergoemacs.org/emacs/emacs\\_package\\_system.html](http://ergoemacs.org/emacs/emacs_package_system.html)

## 6.3 手动安装

有些时候，会有一些冷门的包或者自己写的包无法在前面讲过的仓库里找到，而你也不想写 `el-get` 的 `recipe` 或者 `elpa` 的本地仓库，那么就会用到本节的知识。

首先，你需要让 `emacs` 可以找到你的扩展文件。假如你的文件在 `~/Documents/emacs-package` 目录下，那么在配置中加入：

```
(add-to-list 'load-path "~/Documents/emacs-package")
(load "you-package") ; . 后缀最好省略 el
```

这样就可以启用你的扩展了。

由于手动安装有着诸多不便，而且使用也较为少，因此这里仅列出最基本的用法。更复杂的用法，请见：

- ergoEmacs: [http://ergoemacs.org/emacs/emacs\\_installing\\_packages.html](http://ergoemacs.org/emacs/emacs_installing_packages.html)



# Chapter 7

## 保护你的手指

emacs 需要大量使用 `ctrl` 和 `alt` 两个键，但在大部分 `qwerty` 键盘上，`ctrl` 的位置都在很难按到的角落里。据说，如果长期使用小指按角落的 `ctrl` 会很容易导致手部健康出现问题。因此，我专门加入了这一章来列举一些常用的方法来避免 `ctrl` 和 `alt` 位置不当带来的伤害。

### 7.1 用手掌外缘按 `ctrl`

由于许多键盘中 `ctrl` 处于左下角，所以可以把左手外翻向左下角压去，这样就可以按到 `ctrl`。

- 优点：简单，不需要特别的准备
- 缺点：笔记本键盘很难用，按着 `ctrl` 时左手几乎无法按其他任何键
- 推荐度：2/5

### 7.2 改键

仔细观察，不难发现我们的键盘上总有一些位置很黄金却很少用到的键，这之中典型的例子之一 `caps lock` 键。因此，我们不妨更改键位设置，把使用频度更高的键更换到这些位置上。

一些常用的改键方案包括：

1. 左 `ctrl` 和 `caps lock` 交换：似乎是网上流传最广的改键方法
2. 右 `alt` 和右 `ctrl` 交换：这种改法最适合空格两边都是 `alt` 的键盘，这样大拇指稍稍移动就可以按到 `ctrl` 和 `alt`

改键方法视具体环境不同而有所不同。在 **windows** 下，可以使用各种改键软件完成这一工作。在 **\*nix** 下，对于使用 **xorg** 的用户来说也可以使用 **xmodmap**。如果使用 **DE**，那么很有可能在设置中心内也有调整键盘布局的选项。

- 优点：效果不错，可以根据自己的情况自由配置
- 缺点：需要自己进行一些准备；偶尔使用被改的冷门键可能会不方便
- 推荐度：4/5

## 7.3 踏板

有一类被称为踏板的神奇道具，可以定义踩下时发出的按键信号，这类踏板用于 **emacs** 那真真是极好的，可以极大的减轻手部负担。

- 优点：简单方便，效果超群，直接减少手的工作量
- 缺点：相比上面的方法来说最贵；不同系统驱动可能有潜在问题
- 推荐度：3/5

# Chapter 8

## 编程语言配置

几乎可以肯定，`emacs` 用户中程序员占了大多数，因此没有相应配置怎么能行？本章会列举一些常用的语言的用到的常用扩展以及简单配置，更具体的配置请参照 `emacsWiki` 上仔细进行。

### 8.1 通用配置

本节主要介绍一些独力于具体语言之外的通用配置，如自动补全、版本控制、括号匹配等

### 8.2 C/C++

写 `linux` 内核那帮家伙有不少都是用 `emacs` 的，`emacs` 是 `c` 写的，`GNU` 的许多东西是 `c` 写的，所以 `emacs` 默认就可以不错的支持 `C`，而 `C/C++` 的相关插件堪称多如牛毛……这里仅列出一些常用插件，大家可以一一尝试，选择最复合自己习惯的。

#### 8.2.1 缩进

写代码，缩进搞不好，心情绝对好不了，这里就写写 `emacs` 中的 `c` 的缩进。

## 8.2.2 各种扩展

## 8.3 Python

似乎每一个流行的语言在 `emacs` 都有一大堆的插件可以使用，因此如何选择这堆插件往往让人头疼。本章会尽量详细的介绍一些主流插件的特点以便各位选择，同时给出一些小而好用的插件。

最常见的 `python` 扩展就是 `emacs24.2` 以上自带的 `python.el` 和需要自己安装的 `python-mode.el`。下面会分别说说这两个插件

### 8.3.1 `python.el`

`python.el` 默认包含在了 `emacs24.2` 以上的版本内。如果想要确保自己使用的是最新的 `python.el`，可以使用 `el-get` 安装。安装时会发现有 `python` 和 `python24` 两个选项，其中 `python24` 是一个分支版本，更稳定却没有加入新的特性。**注意**，如果你正在使用 `emacs24.4` 或以上的版本，还需要同时安装 `cl-lib`。

`python.el` 支持许多特性，包括但不限于：

- 自动检测缩进
- 内置 `python shell`（支持 `python2` 和 `3`）
- `python shell` 补全
- 支持调试利器 `PDB`
- `S-exp` 类似的移动方式

解释执行类语言著名的 `REPL`（`Read-Evaluate-Print-Loop`）怎么可以不能在 `emacs` 里实现呢？

在写完 `python` 代码后，输入 **C-c C-c** 就可以执行整个文件的内容，选中部分代码后使用 **C-c C-r** 就可以执行选中部分（选中可以用鼠标拖动，用 `evil-mode` 可以按 `v`，不用 `evil-mode` 可以按着 `shift` 移动光标）。

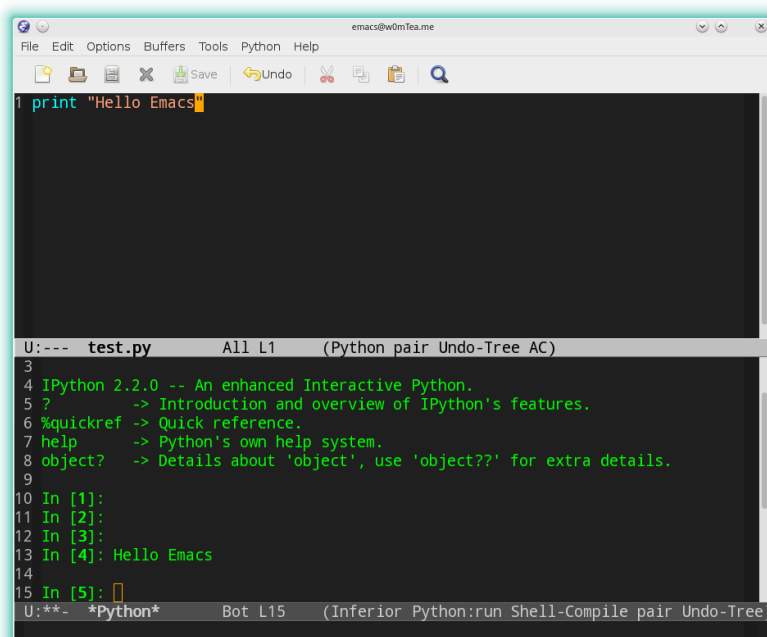
默认情况下，`python.el` 会使用 `python` 来执行你的代码。但是你也可以使用 `iPython`，方法如下：

1. 在未启动 `python` 解释器时使用 `C-c C-c` 等命令会让你输入解释器路径，此时可以输入 `iPython` 的路径来启动 `iPython`
2. 设置如下代码

```
(setq
  python-shell-interpreter "ipython"
  python-shell-interpreter-args ""
  python-shell-prompt-regexp "In\\[[0-9]+\\]:_"
  python-shell-prompt-output-regexp "Out\\[[0-9]+\\]:_"
  python-shell-completion-setup-code
```

```
"from IPython.core.completerlib import
    module_completion"
python-shell-completion-module-string-code
" ' '.join(module_completion(''%s''))\n"
python-shell-completion-string-code
" ' '.join(get_ipython().Completer.all_completions(''%s''))\n")
```

下面来个例子：



### 8.3.2 python-mode.el

python-mode 需要自己安装，可以通过 el-get 来搞定。安装后，打开.py 文件就会自动加载 python-mode。

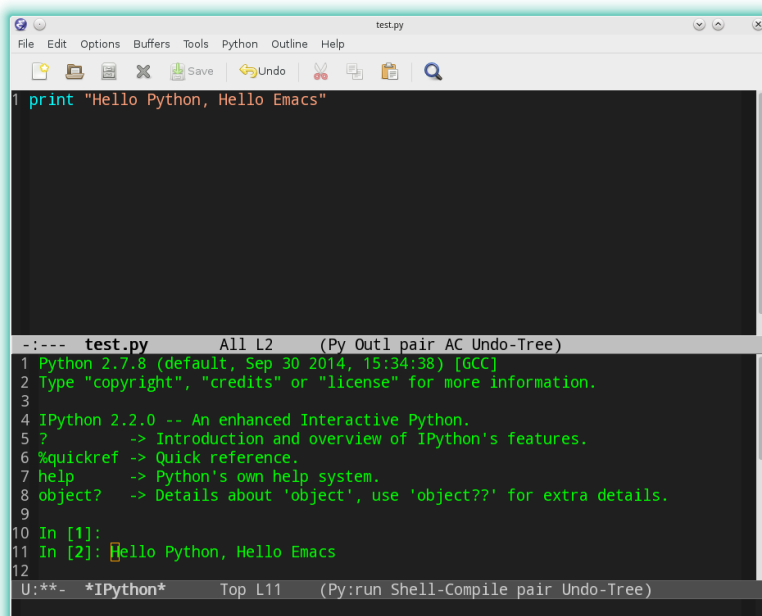
python-mode 可以很容易的更改使用其他 python 解释器，下面以 ipython 为例：

```
;python mode
(require 'python-mode)
;use IPython
(setq-default py-shell-name "ipython")
(setq-default py-which-bufname "IPython")
```

在 `python-mode` 里如何进行 REPL 呢？很简单，在写了 `python` 代码后，你可以

1. 选中你要执行的代码块点击工具栏的 `python` 标签，点 `Execute Region`，就可以看到结果啦（这个动作可以用快捷键搞定）
2. 点工具栏的 `python` 标签，点 `Execute Buffer`，会把整个文件执行一遍（同样可以快捷键搞定）

附上一张实例图（这里使用的 `ipython`）：



```
test.py
File Edit Options Buffers Tools Python Outline Help
[Icons: Save, Undo, Copy, Paste, Find]

1 print "Hello Python, Hello Emacs"

-:--- test.py All L2 (Py Out1 pair AC Undo-Tree)
1 Python 2.7.8 (default, Sep 30 2014, 15:34:38) [GCC]
2 Type "copyright", "credits" or "license" for more information.
3
4 IPython 2.2.0 -- An enhanced Interactive Python.
5 ? -> Introduction and overview of IPython's features.
6 %quickref -> Quick reference.
7 help -> Python's own help system.
8 object? -> Details about 'object', use 'object??' for extra details.
9
10 In [1]:
11 In [2]: Hello Python, Hello Emacs
12
U:**~ *IPython* Top L11 (Py:run Shell-Compile pair Undo-Tree)
```

基于 `python-mode` 可以构建一个 `python` 的 IDE，这个有视频演示（需翻墙）：<http://www.youtube.com/watch?v=0cZ7szFuz18If>

### 8.3.3 便利的小插件

1. `Jedi` `Jedi` 是一个 `python` 补全库，可以为 `python` 提供快速而准确的补全，而且可以被各种工具使用（不仅限于 `emacs`）。而 `emacs` 中基于 `Jedi` 的补全插件就有好几种，这里就说说 `Jedi.el`

安装 `Jedi.el` 很简单，通过 `el-get` 或 `ELPA` 都可以搞定。同时需要安装 `Jedi` 库（包管理器或者<https://github.com/davidhalter/jedi>）

需要注意，`jedi` 对 `python.el` 的支持更好，在 `python-mode` 下可能会有一些问题。

### 8.3.4 参考资料

本节涉及到的插件以及插件组合过多，难免有所遗漏谬误，因此建议各位如有疑问，不要忘了查询下列资料：

- EmacsWiki: <http://www.emacswiki.org/PythonProgrammingInEmacs>
- EmacsWiki: <http://www.emacswiki.org/emacs/ProgrammingWithPythonModeDotEl>
- PythonWiki: <https://wiki.python.org/moin/EmacsEditor>
- python.el main page: <https://github.com/fgallina/python.el>

## 8.4 Lisp

## 8.5 Perl

## 8.6 Ruby

## 8.7 Markdown

### 8.7.1 插件及安装

Emacs 有 `markdown-mode` 来提供对 `markdown` 的支持。此插件可以通过 `el-get` 安装。若 `el-get` 无法找到此插件，记得同步 `emacsWiki` 源（方法见此）

安装后，打开 `.md` 或 `.markdown` 文件时会自动加载 `markdown-mode` 并提供高亮支持。

### 8.7.2 配置

如果你常写 `markdown`，那么对 `markdown` 文件强制进行一些静态语法检查是十分有用的。举个例子，在链接中大量使用的方括号和圆括号很容易被遗漏掉后半边。因此我们可以要求 `emacs` 检查括号：

```
(add-hook 'markdown-mode-hook
  (lambda ()
    (when buffer-file-name
      (add-hook 'after-save-hook
        'check-parens
        nil t))))
```

### 8.7.3 另一种写 Markdown 的方式

其实还有一种更省事也更好用的写 markdown 的方法，那就是用 org-mode 导出生成。想了解更多的读者可以看重量级应用——org-mode 以及 org-mode 导出的 markdown 部分。

## 8.8 Java

事实证明，如果需要写 java，请出门左转 eclipse 或出门右转 intellij……emacs 写 java 并不是一个很好的选择，而且相关插件开发的人也是寥寥无几，毕竟把 emacs 弄到 eclipse 或 intellij 那样好用绝对是一个艰难的事，所以广大的 emacs 都机智的选择了上述两种 IDE 之一。



# Chapter 9

## 重量级应用——org-mode

**org-mode** 是啥呢？它是一个很强大的东西，可以快速高效的通过纯文本文件来完成做笔记、TO-DO list、项目计划等一系列事情。它有些类似 **vim** 下的 **vimwiki** 和 **markdown**，但比前两者强大的多。

用 **org-mode** 写文档类的东西，你需要关注的只有你的文档的结构和内容，而其他的，都有 **org-mode** 帮你搞定，毫无疑问，这是一种相当爽的感觉。而且 **org-mode** 写完后，可以轻松的导出成各种格式：**html**、**markdown**、**pdf**、**odt**（进而通过 **libreoffice** 等转换成 **MS Office** 格式）、**beamer** 等等。或许你已经猜到了，你正在看的这个 **pdf**，也是 **org-mode** 生成的。相信如果没有 **org-mode**，我是不会有勇气开始写这个文档的。

由于 **org-mode** 的内容十分多，多到即使我再写一年也未必写的全的程度，所以我只能写一些基本的用法并给出一些参考资料，大家若有需要，可以查阅 **org-mode** 资料。

### 9.1 安装

**Emacs23** 之后已经默认包含有 **org-mode** 了，但是默认包含的版本往往比较低，因此建议安装一个新的 **org-mode**。安装方法如包管理部分所讲那样，可以用 **el-get** 安装，也可以用 **ELPA** 安装，也可以手动安装，这里就不多说了。

安装成功后，使用 **emacs** 打开任意一个后缀名为 **.org** 的文件就会自动启动 **org-mode** 了。

### 9.2 文档结构

#### 9.2.1 标题

文档结构的骨架就是一级级标题组成的树状结构，这里就来说说标题。

org-mode 里标题的表示十分简单，\* 表示标题，几个 \* 就是几级标题。需要注意的是，星号后需加一个空格。每次手打星号其实是很烦人的事，因此 org-mode 给出了一些快捷键帮我们搞定：

- **C-<RET>**：插入一个同级标题。也就是说，上一个标题是二级标题的话，这个快捷键也会插入二级标题哦
- **M-<RET>**：插入一个同级同类标题。这个和 **C-RET** 基本一样，一些细微的区别会在列表部分给出

### 9.2.2 折叠循环

当文档写长以后，就会有无从下手的感觉，可是通过良好的折叠，可以让我们仅面对少部分内容，并且可以清楚的看到文档的结构，为此 org-mode 提供了一系列折叠功能。

在一个标题上按 **TAB**，这个标题之下的内容都折叠起来，只留下标题。在标题上按 **TAB**，所执行的操作是一个循环，如图：



除了在标题上按 **TAB** 外，还可以使用 **S-TAB** 或者 **C-u TAB** 来完成全局折叠。多次按此快捷键也是循环操作，依次在总览、各级标题和所有内容间切换，如图：



这里仅列举的最基础的用法，更多用法请见参考资料。

### 9.2.3 标题的结构化操作

文档规模上去以后，经常需要以子树为单位在结构上进行一些操作，因此 org-mode 提供了一些结构化操作来帮助我们。

#### 1. 标题间移动

- **C-c C-n**：下一个标题
- **C-c C-p**：上一个标题
- **C-c C-f**：下一个同级标题。只会在同子树的标题间移动

- C-c C-b: 上一个同级标题。只会在同子树的标题间移动
- C-c C-u: 上一级标题
- C-c C-j: 保持当前位置不变的跳转。使用该指令后, 会打开搜索, 输入内容即可找到对应位置。浏览完后, 按 C-g 就可以退出浏览状态, 回到原位置

## 2. 结构化编辑

- M-<left>: 当前标题升级
- M-<right>: 当前标题降级
- M-S-<left>: 当前子树升级
- M-S-<right>: 当前子树降级
- M-S-<up>: 当前子树上移
- M-S-<down>: 当前子树下移

## 9.3 列表

org-mode 中的列表分为有序和无序两种。有序的列表以"1." 这样的形式开始, 无序的列表以 \* + 或者 - 开始。

需要注意的是:

1. 需要在列表符号后加上空格才能生效
2. 有序列表中的数字可以随便写
3. 无序列表用哪个符号不影响最终显示结果
4. 用 \* 来开始无序列表, 那所在行必须有缩进, 否则会被认为是标题
5. 列表可以嵌套, 用缩进来表示层级

列表同样可以使用结构化操作。一些细微的不同在于, 列表的子树上下移动不是 M-S-xx 而是 M-xx。

除此之外, 针对列表还有一些新的操作:

- C-c -: 转换列表符号, 会在有序、无序各符号间循环切换
- C-c \*: 把列表项转换为标题
- C-c ^: 给当前列表排序

## 9.4 表格

org-mode 中还提供了强大的表格功能。

### 9.4.1 基础

新建表格很简单，只要在新的一行连着打两个或多个 | 就好啦。只需要在 | 之间填上内容就 ok。

每次想弄出新的格子都得打 |，这多麻烦呀。其实 org-mode 提供了简单的操作。在第一行的 | 已经弄好的情况下，在格子内按 TAB 就会自动跳到下一个格子，如果这个格子已经是行尾了，则会自动在下一行自动补齐以整行表格。下面是一个例子：

```
| 1 | 2 | 3 |
在3里按TAB后
| 1 | 2 | 3 |
|   |   |   |
```

在表格里按 TAB，还有自动对齐表格的功能（注：表格里如果有中文，对齐往往会出现问题，这是中文和英文显示大小不一造成的，可以通过设置 emacs 的显示字号来笨笨的解决这一问题），所以多按 TAB，不仅省事还整齐啊。

除了按 TAB 可以横向移动以外，按 RET 也可以向纵向移动。

如果想在上下两栏间插入分割线，可以按 C-c -

### 9.4.2 计算

org-mode 表格还提供了计算的功能。

设想这样一个情景，你需要做一个简单的统计，给出物品的单价和数量，算出总金额，如何让 org-mode 帮做这个计算呢？请看例子：

```
| 单价 | 数量 | 总额 |
|-----+-----+-----|
|    10 |    10 | =$1*$2 |
```

这样，在表格中按 tab 就会自动计算出结果了。其中 \$1 表示第一列，\$2 表示第二列。

可是这样需要手动给每一个需要计算的格子打上公式，还是很麻烦，能不能批量的计算呢？当然可以啦。在表格下面加上这句话：

```
#+TBLFM: $3=$1*$2
```

在需要进行全表格计算的时候，在上面那句话上按 C-c C-c 就可以看到结果啦

如果想要知道某一列所有数的和呢？在那一列上，按 C-c + 就会算出结果，并且可以直接粘贴。

### 9.4.3 快捷键

这里仅罗列一些常用快捷键哈：

- C-c |：插入表格

- C-c C-c: 在表格内是对齐表格，在上面说的那句话上是计算表格
- <TAB>: 移到下一格
- S-<TAB>: 移到上一格
- <RET>: 移到下一列
- M-a: 移到一格的头部
- M-e: 移到一格的尾部
- M-<left>/M-<right>: 把一列左移/右移
- M-S-<left>/M-S-<right>: 删除一列/插入一列
- M-<up>/M-<down>: 一行上移/下移
- M-S-<up>/M-S-<down>: 删除一行/插入一行
- C-c -: 插入水平分割线
- C-c +: 计算一列之和
- C-c ^: 排序

## 9.5 链接

就像 html 一样，org-mode 提供了丰富的链接功能。

链接的基本格式如下：

`[[link][description]]` 或 `[[link]]`

其中 link 是要链接的东东，description 是描述。

上面的两个链接出来的样子是这样的：description 或 9.5

### 9.5.1 内部链接

如果你的链接不像一个 url，那么 org-mode 会把它按内部链接处理。上面例子中的链接会链接到本节，为什么呢？因为我在这一节内加了一个锚点：

`<<link>>`

这样点击链接就会找到这个锚点并跳转过去了。

### 9.5.2 外部链接

org-mode 支持各种各样的外部链接，这里仅列出几个常用的，全部的链接支持在这里：org-mode 手册

```
http://xxxxx
file:path/to/some/file
path/to/some/file
```

## 9.6 代码块

`org-mode` 还提供了非常非常强大的代码块功能，除了基础的显示代码的功能以外，还有诸多花式玩法（本文档里的图就是用 `org-mode` 的代码块写出来的哦）。这一节只说说基础用法和一些简单的花式（比如说画图），更复杂的玩法可以去 `org-manual` 好好看看哈。

### 9.6.1 基本用法

插入代码块的方法很简单哈，如下：

```
#+BEGIN_SRC c
#+END_SRC
```

之后在 `BEGIN` 和 `END` 之间按 `C-c` 即可进入代码块编辑模式，输入完后再按 `C-c` 即可退出编辑模式。注意，有时候使用上述快捷键时显示此快捷键未定义，很可能是你正在使用汉字输入法导致的，切换输入法即可。

来一段效果展示：

```
#include<stdio.h>
int main()
{
    printf("hello_org-mode!\n");
}
```

注意需要在 `BEGIN_SRC` 后加上语言名字，以使用该语言特有的语法高亮缩进之类的。每次都打带 `#` 的一长串东西多烦人啊，因此 `org-mode` 提供了快捷键：输入 `<s` 并在上面按 `<TAB>`，看看有没有惊喜？

### 9.6.2 画图

`org-mode` 的代码块可以调用 `ditaa` 来画图，对于一些简单的图来说只需要用 `org-mode` 就能搞定啦～

`org-mode` 需要使用 `ditaa` 来完成字符画，这个东西已经内置，所以不需要再下载。另外，`ditaa` 需要 `java` 环境支持。

在使用前，需要对 `emacs` 做一个小设置：

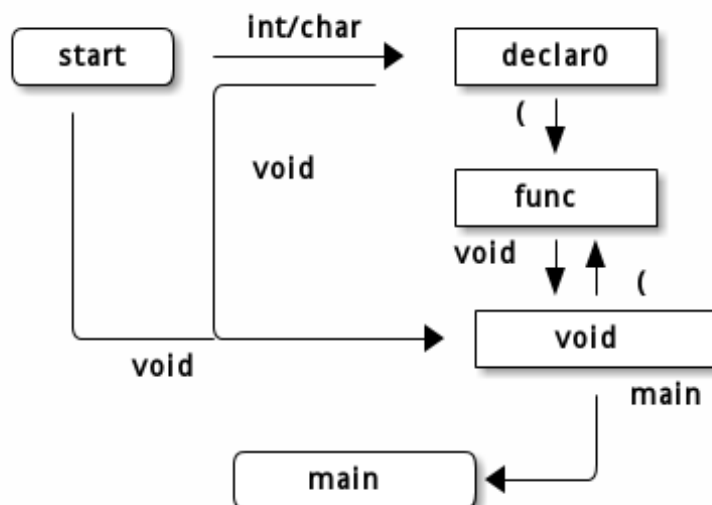
```
; org-mode ditaa setup
(org-babel-do-load-languages
 'org-babel-load-languages
 '((ditaa . t)))
```

现在就可以用 `ditaa` 画图啦。`ditaa` 有不少特性，这里仅给出一些简单的例子，更有意思的大家可以去它的官方看哈，网址<http://sourceforge.net/projects/ditaa/>

还记得上面的 `tab` 循环吗？其实它的源码是这样的

再来一个例子：

其效果如下：



## 9.7 元数据

上面已经出现过了诸如 `#+BEGIN_SRC` 这样的形式，其实这样的东东有一个特殊的名字，叫做元数据。这些元数据不会直接出现在文档中，而是起了各种各样的作用。这里我将会列举一些常用的元数据及其快捷键，更多的元数据的用途后面会陆续加入。

```

s    #+BEGIN_SRC ... #+END_SRC
e    #+BEGIN_EXAMPLE ... #+END_EXAMPLE
q    #+BEGIN_QUOTE ... #+END_QUOTE
v    #+BEGIN_VERSE ... #+END_VERSE
c    #+BEGIN_CENTER ... #+END_CENTER
l    #+BEGIN_LATEX ... #+END_LATEX
L    #+LATEX:
h    #+BEGIN_HTML ... #+END_HTML
H    #+HTML:
a    #+BEGIN_ASCII ... #+END_ASCII
A    #+ASCII:
i    #+INDEX: LINE
I    #+INCLUDE: LINE
  
```

上面的缩写前加上 `<` 在按 `tab` 就会自动填充元数据啦，就像上面说的 `<s` 一样。另外，有些元数据可以加参数，比如：

```
#+BEGIN_SRC c -n
```



```
int main()
{
    printf("hello org-mode\n");
    return 0;
}
#+END_SRC
```

其中的 `-n` 可以设置行号，效果如下：

```
1 int main()
2 {
3     printf("hello_org-mode\n");
4     return 0;
5 }
```

除了上面列举的那些和文档内容有关的元数据，更多的元数据被用来设置 `org-mode` 的工作方式，比如设置导出行为之类的。如果你打开仓库中的 `.org` 文档，你会看到类似于

```
#+TITLE: An Emacs Tutorial for Vim User
#+AUTHOR: w0mTea
#+EMAIL: w0mT3a@gmail.com
#+LATEX_CLASS:
#+LATEX_CLASS_OPTIONS:
```

的东西，这些都是不直接关乎内容的配置元数据。

最后附加一句，`org-mode` 中以 `#` 开始的行会被视为注释，如果想多行注释，可以用

```
#+BEGIN_COMMENT
```

```
#+END_COMMENT
```

## 9.8 To-Do

`org-mode` 除了用来写文档以外，还可以用作你的 `todo-list`。这里会说说 `org-mode` 中的 `todo`。

### 9.8.1 基础

`todo` 项是 `org-mode` 中 `todo` 的基本元素。`todo` 项是的样子和标题十分像，它其实就是以 `TODO` 关键字打头的标题，比如这样：

\* `TODO` 写代码

在 `org-mode` 中，`todo` 项会以彩色显示，十分醒目。

`todo` 项和标题一样，也是分级结构，并同样构成一颗树。

有了 `todo` 项我们又能干什么呢？首先，我们可以通过分级来把大的目标分解成小目标，之后，我们可以通过 `org-mode` 提供的操作来完成 `todo-list` 的管理。下面用一个例子来说明哈

假设 `w0mTea` 需要写一个 `c` 语言编译器，因此他列了这样一个 `todo-list`:

```
* TODO 写编译器
** TODO 词法分析
** TODO 语法分析
** TODO 语义分析
** TODO 中间代码
** TODO 生成汇编
** TODO 产生二进制码
```

经过漫长的努力，`w0mTea` 终于完成了词法分析和语法分析，因此他把光标分别移动到词法分析和语法分析那一行，按了 **C-c C-t**，然后 `todo-list` 就成了这样：

```
* TODO 写编译器
** DONE 词法分析
** DONE 语法分析
** TODO 语义分析
** TODO 中间代码
** TODO 生成汇编
** TODO 产生二进制码
```

这里出现的 **C-c C-t** 就是用来切换 `todo` 状态的，在普通标题、`todo`、`done` 三个状态间循环切换。

`w0mTea` 觉得这样看着不太直观，他需要直观的统计他的进度，因此他给 `todo` 项做了点改变：

```
* TODO 写编译器 [/]
```

之后他在 `[/]` 上按了 **C-c C-c**，成了这样：

```
* TODO 写编译器[2/6]
** DONE 词法分析
** DONE 语法分析
** TODO 语义分析
** TODO 中间代码
** TODO 生成汇编
** TODO 产生二进制码
```

这里的加上的内容用于显示子内容的完成度，除了

`[/]`

这样的格式外，还可以把 `/` 换成 `%` 用来显示百分比。之后的 **C-c C-c** 会刷新进度显示。

现在 `w0mTea` 觉得爽多了，可是他还想搞得更直观一点，于是他把 `todo-list` 弄成了这样：

```
* TODO 写编译器[%]
- [ ] 词法分析
- [ ] 输入正则表达式
- [ ] 构建 nfa
- [ ] nfa 转 dfa
- [ ] 语法分析
- [ ] 语义分析
- [ ] 中间代码
- [ ] 生成汇编
- [ ] 生成二进制
```

之后在他已完成的输入正则表达式上按 **C-c C-c**，成了这样：

```
* TODO 写编译器[0%]
- [-] 词法分析
- [X] 输入正则表达式
- [ ] 构建 nfa
- [ ] nfa 转 dfa
```

这个列表加 [] 的东东被成为 **checkbox**，这里的 **C-c C-c** 是用来切换 **checkbox** 完成状态的。注意此时词法分析项上的状态是 **-**，表明部分子项完成。

w0mTea 给词法分析也加上了进度框，于是：

```
* TODO 写编译器[33%]
- [X] 词法分析[3/3]
- [X] 输入正则表达式
- [X] 构建 nfa
- [X] nfa 转 dfa
- [X] 语法分析
- [ ] 语义分析
- [ ] 中间代码
- [ ] 生成汇编
- [ ] 生成二进制
```

### 9.8.2 快捷键

上面的例子里讲了一些最常用的用法，这里总结一下常用快捷键：

- S-M-<RET>：插入 **todo** 项
- C-c C-t：切换 **todo** 状态
- C-c C-c：切换 **checkbox** 完成状态、刷新进度

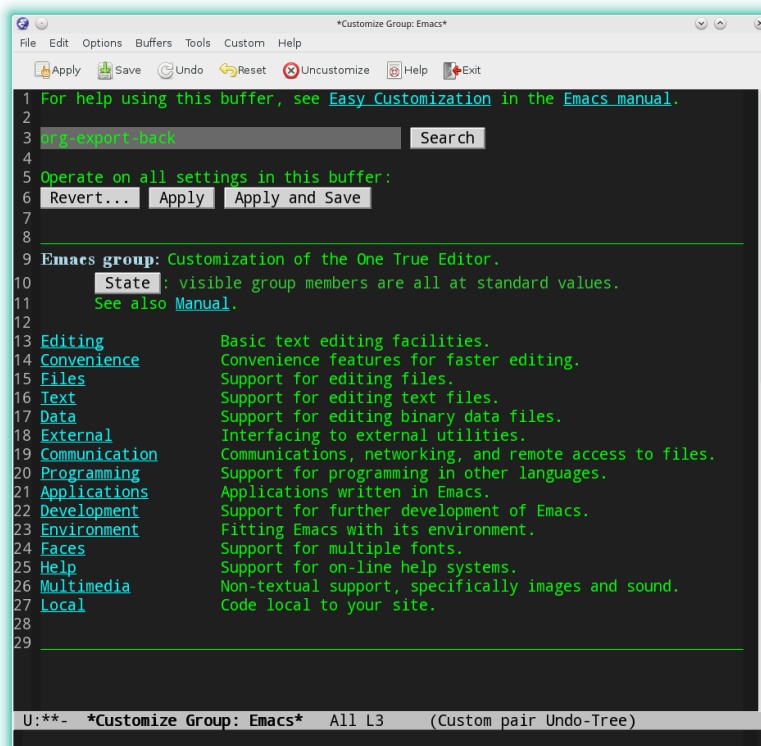
## 9.9 导出

上面写了一大堆，都只是纯文本的 **org** 文档而已，怎么才能导出成各种各样的其他格式呢？本节将会介绍一些常用格式的导出方法，更多的导出请参考 **org-manual**

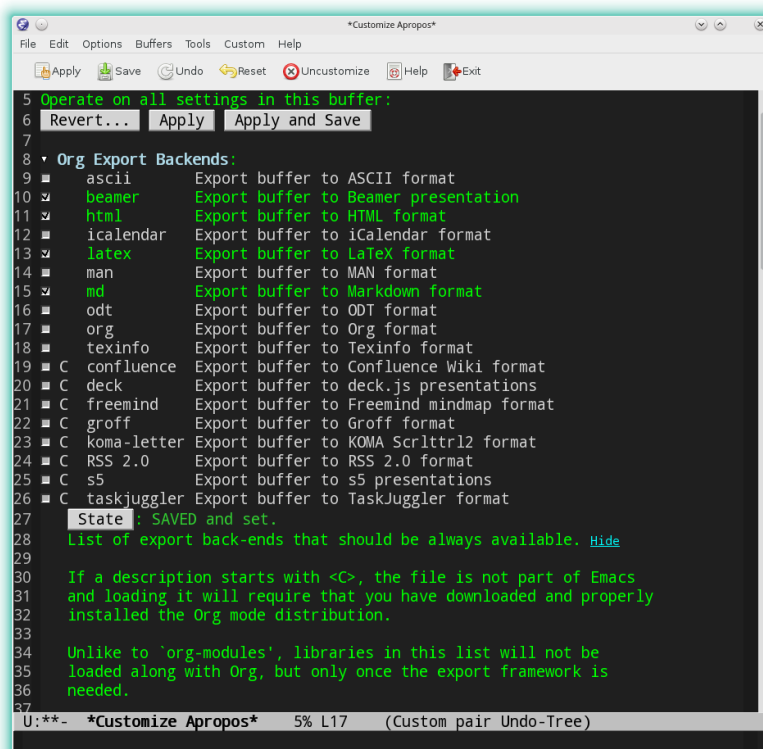
### 9.9.1 导出后端设置

每一种导出格式都需要对应的导出后端，但不是所有的后端都默认启用，因此需要激活对应后端。默认启用的后端有 `ascii`，`html`，`icalendar` 和 `latex`。

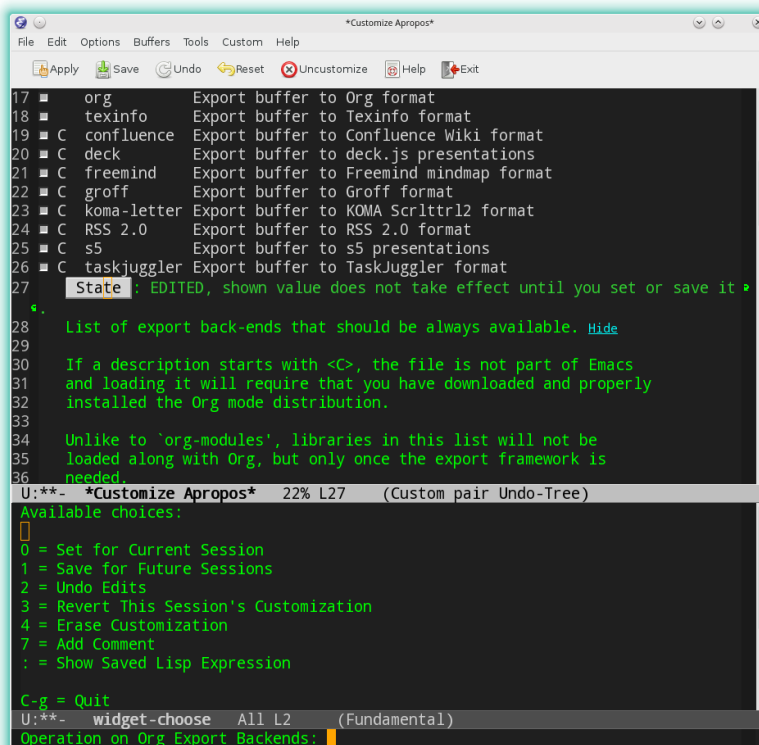
在 `emacs` 中，**M-x customize <RET>** 会进入配置模式，在搜索框中输入 `org-export-backend` 即可进入后端配置，如图：



之后勾选想要的后端，去掉不想要的后端，如图：



然后把光标移动到 **state** 按钮上按回车，选择 **0** 立即生效或 **1** 下次生效即可。



### 9.9.2 导出通用设置

导出前，最好通过元数据对导出进行一些通用设置。使用 **C-c C-e #**，然后输入 **default**，会自动插入通用设置模板，若输入特殊导出格式（比如 **html** 或 **latex**），则会插入特殊模板。通用模板如下：

```

#+OPTIONS: ':nil *:t -:t ::t <:t H:
3 \n:nil ^:t arch:headline
#+OPTIONS: author:t c:nil creator:comment d:
(not "LOGBOOK") date:t
#
+OPTIONS: e:t email:nil f:t inline:t num:t p:nil pri:nil prop:nil
#
+OPTIONS: stat:t tags:t tasks:t tex:t timestamp:t toc:t todo:t |:t
#+TITLE: example
#+DATE: <2014-12-17 三>
#+AUTHOR: w0mTea
#+EMAIL: w0mT3a@gmail.com
#+DESCRIPTION:

```

```
#+KEYWORDS:
#+LANGUAGE: en
#+SELECT_TAGS: export
#+EXCLUDE_TAGS: noexport
#+CREATOR: Emacs 24.3.1 (Org mode 8.3beta)
```

下面，大概说说其中主要的一些选项：

1. **OPTIONS:** `options` 中配置了许多行为，其中的选项用 `option:value` 的格式表示。普遍来说，`t` 表示启用，`nil` 表示禁用，其他值表示具体的设置
  - `author`: 导出时显示作者
  - `date`: 导出时显示日期
  - `timestamp`: 导出时生成时间戳
  - `toc`: 导出时显示目录
2. **TITLE** 文档的标题，默认情况下是文件名
3. **DATE** 日期，默认情况下是插入选项模板时的日期
4. **AUTHOR** 作者，默认和系统用户有关
5. **EMAIL** 邮箱，默认和系统用户和系统 `hostname` 有关

### 9.9.3 HTML

导出成 `html` 可以使用 **C-c C-e h**，之后根据需要输入

- `h`: 导出成 `html` 文件
- `H`: 导出成 `html` 临时文件，在 `emacs` 中打开
- `o`: 导出成 `html` 并打开

### 9.9.4 PDF

导出成 `pdf` 需要有 `latex` 环境，否则生成的 `tex` 文件将无法编译。

导出成 `pdf` 可以使用 **C-c C-e l**，之后根据需要输入：

- `L`: 导出成 `tex` 临时文件并打开
- `l`: 导出成 `tex` 文件
- `p`: 导出 `tex` 文件并生成 `pdf`
- `o`: 导出 `tex` 文件，生成 `pdf` 并打开

上面仅是导出成 `pdf` 的通用步骤，下面会说一些特殊的设置。

## 1. 元数据设置

使用 **C-c C-e #** 插入导出元数据时，选择 **latex** 就可以插入 **latex** 专用的元数据。主要有以下几个：

```
#+LATEX_CLASS:
#+LATEX_CLASS_OPTIONS:
#+LATEX_HEADER:
#+LATEX_HEADER_EXTRA:
```

下面分别说下它们的作用。

## • LATEX CLASS

这个是 **org-mode** 中设置的一个导出模板，这个模板会在 **tex** 文件的导言区设置一堆东西。默认的模板和 **latex** 默认文档类是一致的，有 **article report book** 等。自定义模板的内容在后文给出。

## • LATEX CLASS OPTIONS

这个用来传递 **LATEX CLASS** 用到的一些选项。比如在 **latex** 中可以设置

```
\documentclass[onside]{book}
```

如何把选项传递给 **book** 放在上面的 **[]** 中呢？很简单，把选项写在该项后就好：

```
#+LATEX_CLASS: book
#+LATEX_CLASS_OPTIONS: [onside]
```

## • LATEX HEADER

用 **latex** 经常需要一些包。如果这些包没在 **LATEX CLASS** 模板中列出，那么就需要自己加上。假如我们需要使用 **amsmath** 宏包：

```
#+LATEX_HEADER: \usepackage{amsmath}
```

该元数据可以出现多个

## • LATEX HEAD EXTRA

使用方法同上，直观上的区别就是加在这个里的 **latex** 代码会出现的相对靠后……另外的区别就是写在这里的宏包在 **latex** 片段预览中不会被用到。片段预览的内容见 **org-mode** 手册

## 2. 导出模板设置

注意：本部分内容和 **org mode** 版本强烈相关，版本低于 8 的 **org mode** 绝对无法成功按照本章所讲进行设置。

默认导出模板可以自己进行修改，同时你也可以自定义新的模板。

首先，在进行配置之前，需要在配置文件中加上下面这句话：

```
(require 'ox-latex)
(setq org-export-latex-listings t)
```



之后，可以通过如下形式把常用的宏包加入到包列表中。这样，在导出时就会把你需要的宏包加入。下面以 `hyperref` 为例：

```
(add-to-list 'org-latex-packages-alist
  '("" "hyperref" t))
```

通过如下格式可以自定义导出模板：

```
(add-to-list 'org-latex-classes
  '("模板名"
    "\\documentclass这里写你需要的{}
    [NO-DEFAULT-PACKAGES]
    [PACKAGES]
    [EXTRA]
    "
    ("\\section{%s}" . "\\section*{%s}")
    ("\\subsection{%s}" . "\\subsection*{%s}"
     ")
    ("\\subsubsection{%s}" . "\\subsubsubsection*{%s}")
    ("\\paragraph{%s}" . "\\paragraph*{%s}")
    ("\\subparagraph{%s}" . "\\subparagraph*{%s}")))
```

其中的 `[NO-DEFAULT-PACKAGES]` 表示不加载默认宏包。`org-mode` 中有着一系列的默认宏包，如果不加上此行会把那些包全加上，很容易导致包冲突，因此建议一定要加上这行，再手动把常用宏包加到你的列表中。

`[PACKAGES]` 表示需要加载你列出的宏包。

`[EXTRA]` 表示需要加载额外的宏包。

在上面的部分可以加入一些其他的 `latex` 代码做各种设置。具体的例子见 `README` 文档中列出的配置，那个就配置了许多东西。

再往后的部分内容表示了 `org-mode` 中标题结构的对应 `latex` 代码。上文列出的

```
("\\section{%s}" . "\\section*{%s}")
("\\subsection{%s}" . "\\subsection*{%s}")
```

表示一级标题对应 `latex` 中的 `section`，二级标题对应 `subsection`，后面的以此类推。上文只有 5 个部分，超出的标题等级会对应到有序列表。

最后，我把我的配置放在了 `README` 中，只要安装相应的宏包和字体，就可以直接使用此配置。具体的说明见 `README` 文档。此配置支持导出中文。

### 3. 更改编译程序

默认下导出成 PDF 会用 latex 编译。如果想改成其他程序（比如 xelatex）可以这么做：

```
(setq org-latex-pdf-process
      '("xelatex_Interaction_nonstopmode_%b"
        "xelatex_Interaction_nonstopmode_%b"))
```

注意，上文中的%b 表示对应的 tex 文件，-interaction nonstopmode 是程序参数。为了保证目录、引用等的正确，请务必把你的编译命令写两遍，像上文例子一样。

#### 4. 导出中的 code block

默认情况下，导出时会把 code block 转换为 latex 中的 verbatim，也就是原样输出。这样虽然保证了缩进等等，可是却无法使用代码高亮等。因此可以设置 code block 对应的 latex 内容。以 listings 包为例：

```
(add-to-list 'org-latex-listings '(" " "listings"))
(add-to-list 'org-latex-listings '(" " "color"))
```

### 9.9.5 Markdown

导出成 markdown 可以使用 **C-c C-e m**，之后根据需输入：

- M: 导出成 markdown 临时文件并打开
- m: 导出成 markdown 文件
- o: 导出 markdown 文件并打开

### 9.9.6 ODT

odt 是个啥子捏？这或就是开源的 office 格式，导出 odt，再用 libreoffice 等软件打开转存，就可以转存成 MS office 格式。

导出 odt 需要 zip 程序，使用前请确保已安装它。

导出可以使用 **C-c C-e o o** 来导出成 odt 文件。

### 9.10 参考资料

- org-mode manual:<http://orgmode.org/manual/index.html>

# Chapter 10

## 文档和资料

- emacs manual: [http://www.gnu.org/software/emacs/manual/html\\_node/emacs/index.html](http://www.gnu.org/software/emacs/manual/html_node/emacs/index.html)
- emacs wiki: <http://www.emacswiki.org/emacs/>
- A blog: <http://home.fnal.gov/~neilsen/notebook/orgExamples/org-examples.html>

Chapter **1 1**

致谢

# Chapter 12

## 结尾

本文仓促写成，错漏颇多，还望各位指出错误，让这份教程可以帮助更多的人。