如何学习 Emacs

马陆骋

目录

关于这本 Emacs 入门指南

这本入门指南针对想要掌握 GNU Emacs 文本编辑器的程序员。

据说 Emacs 的学习曲线已经不是那么陡峭了。然而万事开头难，比起其他编辑器而言，要学习 Emacs 还是要难一些的。

不过这个坎并不难过。本手册从基础出发，但是真正的目的是帮助你达到下一个等级 — 在几个月而不是几年后就可以编程 Emacs 自己的行为。

重点是自己学习 Emacs 内置的帮助文档，调试工具和源码。

我对你的要求是承诺（每个星期专注的学习几个小时，使用 Emacs 作为你的日常编辑器）和耐心（愿意离开你的 IDE 至少一个或两个月）。

如果你不确定这样做是否值得，请阅读下一章。

为什么是 Emacs

我相信未来的软件工程，如果我们要在日益增加的复杂性前存活，关键在于领域特定语言，它能用几百行而不是数万至数十万行的代码来表示一个解决方案，像现在的软件系统。

为什么这一定要用 Emacs 来做呢? 恩, 如果你频繁的使用数十种语言, 甚至创建过自己的语言, 你想要一个可编程的，方便扩展的编辑器。 Emacs 的核心是一个 Emacs Lisp 的解释器, 一个成熟的编程语言. 以及极好的集成帮助, 代码带行和一个调试器. 因为解释器内置在 Emacs 内部, 你可以交互式的编写你的工组和定制项; 你做的改变会立刻起作用.(如果你熟悉其他解释型语言你就已经知道并爱上了方便的 REPL). Emcas 的库和 API 接口数量是巨大的, 但是 Emacs Lisp 本身是很简单的.

Emacs vs. _

"写一个 Eclipse 或者 Intellij （或者 Visual Studio）的插件是非常困难的，所以几乎没有人这样做。这意味着没有构建和自定义你自己的工具的社区 。。。更有甚者，创建一个插件的高难度导致人们只做有真正重大意义的程序，然而在 Emacs 中，一个『插件』可以是任何大小，从一行代码到一个巨大的系统和框架 [1: Steve Yegge, XEmacs is dead. Long live XEmacs!. Read all of Steve’ s Emacs articles for some equally conflicted (but much funnier) Emacs advocacy.]"

当然，有很多合情合理的理由 **不去** 学习 Emacs。如果你几乎只做 Java，Windows/.NET 或者 OS X/iOS 编程，那就有足够的理由使用 Eclipse/Intellij，Visual Studio，或者 XCode。你肯定会在使用IDE的特性方面获得更多的经验。

或者你仅仅是厌恶你使用的工具的复杂：我的早期导师，我最尊重的的程序员之一，同时使用几十个记事本和终端窗口和一个超长的任务栏在屏幕的左边来区分它们。真实的故事。

关于生命周期的思考

…**你的语言:** 即使你不买『数十种领域特定语言』的帐，想想你 10 年以后，你会仍然只用 Java 或者 C++ 写程序吗?

…**你的编辑器:** GNU Emacs 已经存在30年了，在更早 10 年还有更早的版本。它是开源的 (自由的)软件。至今仍有活跃的开发者，有一个巨大的，多才的社区为每一种新的语言写插件，你可以想象 TextMate 或者 Sublime Text 或者其他的编辑器有这么长的生命周期吗?

安装正确的Emacs

安装最新版的 GNU Emacs (目前是24.5)。不是XEmacs，也不是 EmacsW32，不是 AquaMacs，也不是 Carbon Emacs。

Linux

Emacs 很可能已经安装了；如果没有，使用你的包管理器（yum, apt-get 等等）。

如果你的发行版只有旧的 Emacs 包，你可以尝试使用一个第三方的仓库（比如下面列出的），或者 从源码构建最新的版本

OS X

如果你使用 macports，安装 `emacs-app`，它给了你一个本地的 OS X 程序。默认情况下 macports 会安装`Emacs.app` 到 `/Applications/MacPorts`，如果你使用笔记本电脑，我建议你使用 `fullscreen` 变体（比如： `port install emacs-app +fullscreen`）来增加命令 `ns-toggle-fullscreen`。

如果你使用 homebrew

`brew install emacs --with-cocoa`。

另外，你可以使用一个 预编译的emacs，但是考虑下使用 macports 或者 homebrew 这样就可以很容易的使用它们来安装其他 Unixy 工具。

Windows

可能你是一个 Windows 用户，但是我不是。我很抱歉如果后面的一些例子在你的系统上不能工作。因为我没有在 Windows 上测试过。

你可以从 http://ftp.gnu.org/pub/gnu/emacs/windows/ 下载最新的 `emacs-xx.x-bin-i386.zip` 并且解压。然后在 `bin` 目录下运行 `addpm.exe`，记得用管理员权限。

有些 Emacs 功能需要 Unixy 工具比如 `find` 和 `grep`，你可以通过安装可移植性操作系统接口仿真环境比如 Cygwin。 你需要确认可移植性操作系统接口工具在系统 PATH 中，这样 Emacs 才能找到它们。

更多帮助参考 GNU Emacs FAQ for MS Windows 和 (经常过时的) EmacsWiki。

移除任何存在的 .emacs 配置文件。

如果你有任何存在的 Emacs 配置在一个 `.emacs` 文件或者一个 `.emacs.d` 目录中，你应该删除它，如果你想要你的 Emacs 和手册中的例子拥有一样的行为的话。

基础知识

通读 Emacs 的教程：启动 Emacs 然后按下 `C-h t`。

如果教程有点沉闷，坚持下去。它只会占用你最多30分钟；而且里面的东西是理解Emacs的关键。 你将学会的鼠标移动和文本操作键位在 bash shell 中也起作用。（或者任何使用readline库的程序）。 如果你使用 OS X，在文本域或者任何原生GUI程序中这些都管用。

不要进入本手册的下一章直到你通读了教程，然后你会很熟悉下面的这些键位：

Table 1. Table 键位

操作	键位
访问（打开的）和保存的文件:	C-x C-f and C-x C-s
在缓冲区之间切换:	C-x b and C-x C-b
使用标记和点去设置区域:	C-SPC
剪切和粘贴:	C-w, C-k, C-y, M-y
向前和向后搜索:	C-s, C-r
通过名字调用命令:	M-x
撤销:	C-/
取消输入到一半的命令:	C-g
获得编辑模式的帮助，按键绑定和命令:	C-h m, C-h k, C-h f, C-h a (只要记住 C-h 然后阅读小缓冲区的提示)
退出 Emacs:	C-x C-c

不要担心你不能记住在教程中的 **每一个细节** 。到目前为止没有必要记住所有的鼠标移动键位；箭头键就可以做到。 如果你忘了打开或者储存文件的按键。文件菜单会提醒你。你可以使用鼠标（停留在右手边的窗口的模式显示行并且阅读显示的帮助文字）而不是记住 `C-x 1` 来移除窗口。我会让你在一个月以后去重读整个教程，来发现差异； 在此期间，尝试学习键位而不是依赖菜单和鼠标。

如果你在使用一台运行 OS X 的 Mac 电脑，你是幸运的：因为你有独立的 Meta(alt/option)， Control，和 Command(⌘) 按键在你的键盘上，后者是为标准 OS 键位保留的。你可以使用 `⌘-s` 来保存，`⌘-c` 来复制，`⌘-v` 来粘贴，`⌘-f` 来搜索。不过，你最终应该学习上述命令的 Emacs 替代键位，因为它们更强大。对 Emacs 来说，⌘ 按键叫做 `super`，缩写是小写的 `s`，比如`s-f`。

下面这段原文已删除 如果你不使用Mac，你要抵制住使用 cua-mode 的诱惑，cua-mode 重绑定了按键诸如 `C-z`，`C-x`，`C-c` 和 `C-v` 到 它们的标准 Windows 定义。`cua-mode` 也改变了 `region` 的行为，使用 `shift-箭头按键` 高亮文本，并且输入 文本代替激活选择，就像你在其他编辑器中期待的一样。要习惯使用 Emacs 的方式做上面的事情。

基本的 Unix/C 工作流

本章使用了 GUN Radline 库的源码来作为操作的例子。 你可以通过命令 `git clone git://git.savannah.gnu.org/readline.git` [2: 你需要 git, 一个源码控制系统; Git Book 有安装说明.] 来获取它， 或者也可以使用你自己的 C 或者 C++ 代码。

c-mode

启动 Emacs 并且打开(C-x C-f)文件 `readline/examples/rl.c`.

如果你不小心再次调用了 C-x C-f，或者其他等待你输入的命令，记住你可以通过按下 C-g (确保聚焦在 minibuffer) 来取消掉。

因为这个是一个 C 语言的源文件，Emacs 已经自动激活了一个编辑模式叫做 `c-mode`，它提供了一些自定义的按键绑定和 C 语言的缩进规则和语法高亮的知识。 我们在后面会更深入的学习 c-mode。

shell

运行 shell 命令 (就是 M-x shell RET)。

这会创建一个新的缓冲区来运行环境变量指定的 shell。 [3: 在 Windows 系统中你需要一个 Cygwin 或类似的东西提供的 shell.] 这个缓冲区被称为 **shell** (星号是名字的一部分, 所以你不得不在使用 `C-x b` 切换时输入它们)。

cd 到 readline 的目录然后运行下面的命令：

```
./configure
make
```

这个 **shell** 缓冲区的编辑模式是 shell-mode。你可以在 shell-mode 中运行任何 shell 命令；例外的情况是执行一些需要真正终端支持的，比如分页 (less) 和基于鼠标的程序。 你失去了 bash 的 readline 补全和其他任何自定义的 bash-completion 脚本, 或者在这个模型下你的 shell 中的等效选择(虽然 Emacs 提供了它自己的命令历史和 tab 补全)。出于这些原因，我倾向于使用一个真正的终端(在 Emacs 之外)运行一些任务，就像在一个 shell 中运行 Emacs 一样。 (尝试去在 Emacs 中做所有的事情就得不偿失了)。

shell-mode 的一大优势就是所有的输出都是可以被你搜索，复制，粘贴到的，或者其它行为，都可以用 Emacs 的标准命令来操作；你不需要移动你的手到鼠标那儿仅仅为了选择一些文字。

shell-mode 加入了一些按键绑定：

输入 C-h m 然后阅读 shell-mode 的文档和它的按键绑定。

（目前为止, 不要操心去阅读帮助文档中关于自定义的部分 — shell-mode 提供的变量和钩子去改变它的行为。在 shell-mode 之后的文档还有关于所有的激活状态的副模式的文档；你也不需要阅读它们）。

特别有趣的是按下 RET 或者 C-c RET 在前一个输入行中；C-<up> and C-<down> (or M-p and M-n) 来循环上一个命令；M-x 来循环目录。

> 指出如何发送信号(比如: C-c) 给 shell。(提示：搜索帮助缓冲区的内容，关键字 "interrupt" 和 "stop")
>
> 
>
> 指出 C-M-l(字母 l, 不是1) 和 C-c C-s 是做什么的 (这些都是在运行一个 shell 命令产出大量的输出时很有用的) [5: 这些按键绑定不是拍脑袋想出来的. C-c C-s 是 C-x C-s的镜像, 你知道这是用来保存整个缓冲区到一个文件的("全局"的按键绑定趋向于使用 C-x 开头, 反之 C-c 是一个模式专用按键绑定的开头. 并且 C-M-l 在其它编辑模式中有同样的表现函数, 在 c-mode 中它试图把光标周围的函数都带入 view 中. 如果你感到被按键绑定的数量淹没了你的脑容量, 不要担心, 你总是可以使用 C-h m 找到它们)]

eshell

如果你不是特别的执着于 bash 或者其它任何 shell，除了 shell 以外你可以考虑使用 eshell，一个完全由 Emacs lisp 实现的 shell。先不说其他优势，在 Windows 上运行 eshell 不需要 Cygwin；你可以输入 lisp 代码或者通过名字直接在 shell 提示符上执行任何 Emacs 命令；而且你可以重定向命令输出到 Emacs 剪切板或者任何打开的 Emacs 的缓冲区。

我自己不使用 eshell，但是很多人会用。目前为止它可能是最简单的可以(运行在 Emacs 内部的)长期使用的 shell。

ansi-term

在 Emacs 中运行一个 shell 的最后一个选择就是 ansi-term 了。这是一个完全的 terminal 模拟器，而且它将会把大多数的按键直接发送给 terminal 中运行的程序。 包含 TAB，所以会是 shell，而不是 Emacs，来执行 tab 补全。

C-x 和 C-c 仍然会被 Emacs 处理；对于 M-x 你就得输入 C-x M-x 来实现了。 你可以从 raw 字符模式转换到 line 模型来获得普通的 Emacs 行为（比如：移动光标周围这样你可以复制一些前面的输出）；直到你输入回车，没有东西会被发送给终端。

想要获得帮助，你不能使用 C-h m 因为 C-h 被传给 shell 了（很可能意味着回退）。应该使用 f1，或者通过名字调用 describe-mode 。

shell-command

想要运行一次性的 shell 命令而不想打开一个完整的 shell 缓冲区的话，使用 shell-command （M-！）。

> 
>
> M-! date RET

shell-command-on-region (M-|) 比较类似但是它是发送当前的选择区到 shell 命令的标准输入。 让我们使用这招来计算 rl.c 的 main 函数的行数吧：

切换到 rl.c 的缓冲区（使用 C-x b）。

移动光标到 main 函数（通过 C-s 来搜索）。现在移动光标让它位于 main 函数的左花括号前。

按下 C-SPC 来设置标记。

按下 C-M-f 来移动光标到对应的右花括号前面。现在选择区覆盖了 main 函数的整个方法体。

M-| wc -l RET

如果你忘记了这些命令的名字，输入 M-x shell TAB TAB。

阅读 shell-command-on-region 的帮助。你可以通过按键绑定 (C-h k) 或者函数名字 (C-h f) 来查找到。

这个帮助文档特别的长，但是重要信息都在顶部。无视编码系统那段，和一切关于非交互式的参数（就是关于从 lisp 脚本调用这个函数的，对应我们要使用的交互式调用）。

“Prefix arg means replace the region with [the output of the shell command]” 说的是提供命令一个数值参数。 你将会在本教程中学习数值参数。在这个例子中参数的值并没有意义，所以 M-2 M-| 或者 C-u M-| 都可以。

试试看！

现在撤销 (C-/) 你在缓冲区做的事情。注意显示模式的那行的指示器的变化当缓冲区有未保存的修改时。

sh-mode

打开文件 file readline/configure。

如果你不是特别喜欢 shell 脚本，我很抱歉把你丢进一个 1200 行的程序生成的脚本，但是这是最接近我需要的例子了。

编辑 shell 脚本的模式叫做 sh-mode （对应运行交互式 shell 会话的 shell-mode，我们前面看过了）。如果 Emacs 没有辨认出那个文件是一个 shell 脚本 （也许它没有在第一行显示 shebang）你可以输入。。。你猜对了。。。M-x sh-mode。

我对 sh-mode 真没啥好说的。它会语法高亮和缩进各类 shell；并且它提供按键绑定来运行脚本，和插入某些 shell 结构（case，loops 等）和纠正当前 shell 的语法。 如果 Emacs 搞错了它（由于没有 shebang）你可以告诉 Emacs 去用哪个 shell。

现在你知道在哪里去发现如何解决问题了。

Info documentation

回到配置脚本，调用 info-lookup-symbol 然后根据提示输入 test。

假设 bash shell 的 Info 文档被正确的安装在你的系统上（至少在任何 Linux 或者 OS X 系统上是这样的） 你将会看到一个 Info 窗口展示 bash 手册在 test 命令那行。

根据链接跳转到 『Bash Conditional Expressions』。如果你不想要使用鼠标，用 TAB 移到到下一个链接点然后回车也行。

和往常一样，C-h 将会显示所有的 info 模式的按键绑定。再说一次，不要去死记硬背所有的按键绑定，也不要通读单调乏味的 Info 教程。你所需要知道的就是：

- SPC 和 DEL 来滚动；当你达到了一个 『节点』（一页或者一个章节）的末尾，再次按下 SPC 就会顺序到达下一个节点。我必须警告你，DEL 有时会很淘气当你达到一个章节的第一个节点时。
- l（字母 ell）回退历史。
- u 或者 ^ 向上移动到最接近的目录。重复按下 u 会把你带到主 Info 目录，那包含所有的 Info 手册在你的系统中。（Emacs 相关的手册在互联网上可以能找到。）
- 使用常见的机制来搜索 (C-s 和 C-r)；当你达到一个节点的结尾时，再次按下 C-s 将会继续搜索余下的手册。

正如你可以在 Info 目录下看到的，有很多 Info 手册，包含 Emacs 自己的手册，一个 Emacs Lisp 的参考手册和介绍，和更复杂的 Emacs 编辑模式的手册，比如 cc-mode。

Info 文档简直太好用了。如果你在维护一个 Makefile 你可以查找深奥的符号比如 $@ 的意义。需要写一些 socket 相关的程序？在后面的章节中我们将会安装 glibc (the GNU implementation of the Unix standard library) 的 Info 手册。

info-lookup-symbol 使用当前缓冲区的模式来定义查看哪个手册。有时候它也说不清楚，所以它会提示你：输入 sh-mode 或者 makefile-mode 或者 c-mode 或者其他 （按下 TAB 来显示所有可选项）。使用前缀 C-u 来调用 info-lookup-symbol 的话它会总是询问你。

compile

有必要的话切换到 rl.c 缓冲区（使用 C-x b）。

移动光标到 main 函数（使用 C-s 搜索）。故意在 { 花括号那里制造一个错误，就像下面的例子一样，然后保存你的修改。

M-x compile (并且接受默认的 shell 命令 make -k)。

让激活的鼠标仍然处于 rl.c 的窗口，输入 C-x 0（零）来最大化 **compilation** 窗口。

(C-x 1, 你在教程中学到过的, 隐藏了所有的窗口除了当前被选中的那个；C-x 0 做了相反的事)

之后, 使用 C-x 2 和 C-x 3 C-x o (字母 o) 来在窗口之间跳转。

好了, 回到 **compilation** 窗口. Compilation-mode 当然有它自己的特定的按键绑定:

使用 C-h m 来获得 compilation-mode 的帮助和它的按键绑定。 好消息是：这个模式的帮助文档很短。然后，不要去无聊的阅读所有的激活状态的副模式, 除非你确定你想看。

试试看 compilation-next-error 和 compilation-previous-error 的按键绑定。如果 next-error-follow-minor-mode 听起来很迷人，也可以试试。

编辑命令会在运行在 buffer 的相关的目录的下。通常是包含了 buffer 的文件的目录。想要运行一个不同的 Makefile， 你可以指定 -C (--directory) 或者 -f (--makefile) 的 make 选项；或者你可以使用 M-x cd 改变 buffer 的默认目录在 M-x compile 之前。

rgrep

为了搜索所有的文档中的一个字符串（或者正则表达式）, 使用 rgrep [6: 在 Windows 系统中你需要一个 Cygwin 或类似的东西提供的 find 和 grep 命令.] 这输出一个 matches 在一个新的缓冲区中，你可以导航就像编辑缓冲区(事实上, grep-mode 从 compilation-mode 继承了它的函数和按键绑定）。

使用 rgrep 来搜索 readline 的源码（确保你搜索的是 readline 的目录，而不是 readline/examples 这个目录）。

你可以精确控制 grep 命令的用法: 阅读 rgrep 的文档来找答案, 或者使用 grep 或者 grep-find 来代替. C-h 文档是一个足够的参考, 但是 Info 手册 (C-h F rgrep) 提供了一个更好的介绍.

vc

Emacs 有一些列命令, 都以 "vc-" 开头, 它们为不同的版本控制系统提供了一致的命令. 我们正在教程中使用的 readline 文件是在一个 git 仓库下的, 但是基本命令和 subversion 或者 cvs 一样. 更高级的命令(比如更改仓库设定, 或者 push 或者 pull 远程仓库在 git 或者其他分布式 vs 系统中) 你仍然需要在 Emacs 之外进行操作.

如果有必要, 切换回 rl.c 缓冲区(使用 C-x b).

(如果你发现你失去了可以在缓冲区之间切换的可点击的标签, 记得使用 C-x 的扩展性更强 - 当你有上百个缓冲区被打开时. 等会儿我们将会看到输入 C-x 会节省很多击键)

rl.c 大概已经有了很大的变化了. 调用 vc-revert 来撤销到仓库的最新版本.

由于参数的缘故, 假设我们查询 rl.c 的文档关于 "-u" 参数有一些令人费解:

使用(C-s)来查到文档的同一个实例关于 "-u" 并且替换单词 "unit" 为 "fd". 保存你的变更.

让 Emacs 来展示你一个文件的 diff. (如果你需要一个提示, 它在本节的第一段)

diff 会在 diff-mode 中被显示, 它也会被用于查看补丁文件. diff-mode 有很多有用的诀窍 - 阅读它的文档如果你有空的话.

搜有的 vc 命令都被限定在 C-x 开头的按键序列中. 如果你有想过通过全名来调用 diff 命令, 你可能已经注意到了 Emacs 告诉你对应的按键绑定了.

按下 C-x v C-h 来查看所有的以 C-x v 开头的按键绑定.

这也能工作于其他的前缀. 试试 C-x 4 - 你可能注意到一些类似的你已经知道的按键绑定.

为了提交文件, 看看 C-x vv (vc-next-action). 或者输入 vc-dir-mode 使用 C-x v d 并且找出如何标记文件, 为了能在多个文件一起执行 vc 行为.

vc 在 Emacs 手册中有 一个很长的章节.

个人观点, 我更喜欢使用 gitk 和 git 图形界面工具, 或者直接使用 git 的命令行接口, 来 adding, staging, committing, reverting, merging, branching. 但是我会大量使用 Emacs 的 vc-diff 和 vc-print-log (和 vc-annotate!) .

因为 vc 是被限制在通用特性的后端系统支持的, 人们已经卸了自定义的模式对于特定的版本控制系统. Magit 是就是一个这样的模式针对 git; 但是因为我们还没有学会如何去安装扩展, 我们还是坚持使用 vc. 无论如何, vc 可以做一到两件甚至 magit 也不能做的事情 - 我提到 vc-annotate 了吗?

vc-annotate

在几章节之前, 我让你去 grep readline 的源码搜索 “rl_insert_comment”.

切换到 **grep** 缓冲区如果你还在绕着它, 或者做一次新的搜索.

记住, 当使用 C-x b 来切换缓冲区时, 你不得不输入星号作为名字的一部分. (对了, 缓冲区的星号和磁盘上的文件并没有关联. 这只是一个管理; 你总是可以重命名 **grep** 缓冲区为一个不带星号的名字—查看 rename-buffer—或者保存到磁盘上如果你想要这样做的话).

第一个提示应该在 emacs_keymap.c 中, 跳到那个点.

让我们查找哪个版本的 readline 加入了 Meta-# 按键绑定:

M-x vc-annotate

删除所有的 Emacs 窗口除了注解窗口(你可能也需要调整 frame 的尺寸让它足够大)

把光标移到 "Meta-#" 这行, 按下 d 来查看这一行上一次修改的版本的 diff.

现在我们在 diff-mode 了. 看上去这个版本由大量的空格组成. 找到确切的 "Meta-#" 的改变. 你可以按下 C-c C-w 来 隐藏只有空格的改变在光标周围的大片地方, 然后按下 n 跳到下一块.

不, 这个版本不是. 按下 q 离开 diff 缓冲区, 然后再次运行 annotate , 从在这一行的前面开始.

现在确保你的光标在右边那行(如果新的版本和较新的那个差别足够大, 我们所感兴趣那行可能会移动). 然后再次按下 d, 在 diff 模式下一直按 n 知道你找到了对的那行. 对, 这就是我们在寻找的!

回到 annotate 缓冲区(按下 q). 按 l 来查看这个版本的日志信息. 看, readline 的 "Meta-#" 绑定追溯到版本 2.1! 如果你想要一些上下文你可以按下 D (就是 shift-d) 来查看这个版本改变的所有文件.

当然, annotate 更加有用当提交是更颗粒状的并且提交信息更具有描述性, 以及 bug 跟踪器的链接等等. 但是你懂的.

现在我不期待你可以记住所有的这些按键绑定, 但是我希望你知道如何找到它们当你需要时.

ediff

ediff 是一个更强大的模式来查看不同的文件和版本的区别.

如果有必要起换到缓冲区 rl.c.

M-x ediff-revision. Accept the default values of rl.c, its latest revision, and its current state.

这打开了一个新的 frame 从你控制 ediff 的地方; 总是要确保 ediff frame 是被聚焦的当你发出一个命令时.

按下 ? 来获取帮助. 找出如何关窗每个 diff. 找出如何显示文件一边一个. 当你完成时按下 q 退出.

当调用 ediff-revision 时你可以提供任意两个版本, 不仅仅是最新版和当前的工作区. 你可以 diff 任意两个缓冲区 (ediff-buffers) 或者文件 (ediff-files).

我建议你停留在 ediff-directories (如果你想知道为什么, 用用看就知道了).

etags

etags 是一个程序用来索引源文件的, 然后创建一个 TAGS 文件这样 Emacs 可以使用它来找到定义的变量, 函数和类型.

运行 shell 命令 make TAGS 在 readline 目录中 (从一个 shell 缓冲区或者使用 M-x compile).

大多数开源项目的 Makefile 都包含 TAGS 目标. 你可以使用一个

你刚才运行的 etags 是 Emacs 自带的一种. 也有其他的可选实现叫做 "Exuberant ctags", 它支持更多的语言. 使用你的包管理器来安装它(比如 port 或者 yum install ctag) 然后使用 ctags -e 来调用它.

回到 Emacs, 调用 find-tag (M-.). 输入 rl_insert_comment 作为 tag 来查找, 然后你刚才生成的 TAGS 文件的位置.

根据 find-tag 的帮助, 找出如何回跳到你调用命令之前的位置, 和如何找到下一个符合的位置如果大于一个的话.(你可以找到文档从 C-h F find-tag 更清楚比从 C-h f).

要使用不同的 TAGS 文件, visit-tags-table.

如果你是一个 C++ 程序员, 你很快就会发现 ctags/etags 不是很完美当碰到类和命名空间时. 在 90% 的情况下它表现良好. 希望有人可以开发出一个基于 clang 的索引.

gdb

我们要在 rl 程序中运行 GNU 调试器, gdb.

首先在 readline 中编译然后在 readline/examples中.

只要你没有引入任何错误到任何源文件中, 编译应该是成功的. [7: 我没有在 Windows 上测试过, 但是我假设 Cygwin 安装了必要的编译器, 头文件和库.] 如果有错误, 你知道该如何回滚.

M-x gdb. 当使用命令行提示符时, 指定程序 example/rl 作为 gdb 的参数; 如果你在 readline/examples 目录, 命令行看起来就像 gdb --annotate=3 ./rl. gdb 的默认选项可能会在你的系统上不同, 所以你可能想要使用它们.

现在你正在运行 gdb 在 Emacs 中, 所以标准 gdb 命令运行着. 让我们设置一个断点在 readline 函数中. 开始 rl 程序, 我们会到达断点.

在 gdb 的提示符下输入下面的命令:

break readline run print rl_pending_input next step backtrace frame 1

注意 Emacs 是如何显示对应的源文件和行在一个分开的窗口的.

上面的命令都是直接被 gdb 程序解释的. gdb 允许你缩写它们分别用 r, p, n, s, bt, 和 f. 在一个空行按下回车会重复上一个命令—这在要重复执行命令时很有效. 输入 help 会显示 gdb 的内置帮助.

对于严肃的调试, Emacs 可以打开额外的窗口来显示当前的堆栈, 断点, 本地变量和注册者. 可以查看 Emacs 的手册获取更多细节: C-h r 开发 Emacs 手册在 Info 浏览器, 然后搜索章节标题 "GDB Graphical Interface".

家庭作业

你现在可能觉得要奔溃了. 花几天时间来使用 Emacs 作为你的主力编辑器, 这个过程可能是痛苦的, 在查看本手册的下一章之前. 如果你忘了一个我教过你的命令, 试着靠你自己找出它 (使用 C-h m, C-h a, C-h f, C-h k and C-h F) 在查看这里之前.

作为你学习 Emacs 的承诺之一, 尽量的尝试在 Emacs 中运行你的 shell.(如果你需要帮助来配置你的 shell 能在 Emacs 中工作, 或者配置 Emacs 来很好的在你的 shell 中工作, 我们将会重读 shell-mode 在 General customization”章节.

cc 模式的自定义

我应该带领你了解 Emacs，Emacs Lisp 的关系，和 CC 模式的手册，但是它们是那么的。。。长！ Emacs手册的目录都比我们的前一章 "Unix/c workflow" 要长。 [8: 不信试试: lynx --dump http://www.gnu.org/software/emacs/manual/html_node/emacs/index.html | sed '/^References/,$ d' | wc -w]

然而我会向你展示我是如何尝试去发现我需要的信息在Emacs的源码中的（谢天谢地，大多数都是lisp而不是C）。 我刚才贬低的手册是非常有用的。因为一个开源的项目有这么广泛的文档是非常棒的！ 但是一页一页的阅读这些手册是不可能的；你必须学会如何有效的找到正确的信息。 我们已经涉及到了基础的工具，现在我们需要更多的练习。

顺便说一下， c-mode, c++-mode, objc-mode, java-mode，和其他一些名字都是 cc－mode 的别名， 只是需要一些微小的配置修改来支持它们各自的语言。从现在开始我会使用 cc-mode 这个名字。

指导编码风格

你喜欢缩进2个，3个，4个，还是8个空格，或者使用 tab？ 一个 tab 是多大？无论你喜欢什么样的，我想都不是 cc-mode 默认提供的那样，所以我们可以试试去改变它。

访问(打开)一个 C 或者 C++ 文件 (你可以使用 reading/example/rl.c 在前面的章节)

在任何地方按下 TAB 来根据当前的缩进规则达到缩进的位置（要插入一个字面tab，使用 C-q 也称 quoted-insert）

让我们看看 Emacs 在我们按下 TAB 时在后台做了什么:

C-h k TAB

TAB 运行了命令 c-indent-line-or-region, 这是一个交互的被编译的 Lisp 函数在 `cc-cmds.el` 中.

如果你没有看到 cc-cmds.el 的链接, 说明你没有安装 elisp 的源码. 使用你的系统包管理工具去安装 emacs-el 包并再次尝试.

跟随 cc-cmds.el 的链接. 这会把你定位到 c-indent-line-or-region 被定义的地方. 如果有必要, 把整个定义带到视图中 (C-M-l).

```
(defun c-indent-line-or-region (&optional arg region)
  "Indent active region, current line, or block starting on
this line.
In Transient Mark mode, when the region is active, reindent
the region.
Otherwise, with a prefix argument, rigidly reindent the
expression
starting on the current line.
Otherwise reindent just the current line."
  (interactive
   (list current-prefix-arg (use-region-p)))
  (if region
      (c-indent-region (region-beginning) (region-end))
    (c-indent-command arg)))
```

Emergency elisp

关于上面的代码有一句题外话需要解释一下. 让我们从结尾处的 if 条件开始:

```
(if region
    (c-indent-region (region-beginning) (region-end))
  (c-indent-command arg)))
```

让我们把它简化一下:

```
(if region
    a
  b)
```

如果你还是不知道这意味着什么, 就当它是一个函数吧, 然后它看上去就像这样:

```
if(region, a, b)
```

移动圆括号到函数的左边大概是 Lisp 被世界接受的最大的障碍.[2]

参数部分的 a 和 b 是什么意思? 猜猜看, 可以用 C-h f if 来查看答案.

COND, THEN 和 ELSE 的定义是非常清楚的. 但是 “if is a special form” 是什么意思?

这会证明它是否是一个常规的函数. elisp 对于"普通"表达式的赋值-"表达式"意味着给一个"模型"加上括号, 就像 (a b c) 或者 (a b (cd)) - 就是给每个参数赋值.

让我们假设函数 + (elisp 没有特殊操作符, 所以 + 只是一个函数而已). 如果 x 是一个变量包含值 5, 而 y 是一个变量包含值 2, 那么下面的两个表达式是等同的.

```
(+ x y)
(+ 5 2)
```

函数 + 看不懂 x, 它只看到 5.

对表达式 (+ 5 2) 求值: 切换到 **scratch** 缓冲区, 输入 (+ 5 2) 在它自己的一行中, 然后按下 C-M-x 来求值表达式并且显示结果在 echo area 中.

在表达式的第一个元素 (这里指 +) 也被求值了. + 实际上是一个变量, 它的值就是给数字做加法的函数.

好了, 回到 if. if 是一个 "特殊" 的表达式, 它的意思是 elisp 的解释器会把 if 当做一种特殊的情况. 考虑一下, 很明显普通的函数求值规则不适合它: 我们不会想要去求值 ELSE, 和它的特殊副作用, 当 COND 是 "true" 是(non-nil).

这些都在 Emacs Lisp 的手册里有解释:

C-h S if

在 **info** 缓冲区按下 i （代表 index）来进入特定的表达式。

回到 cc-cmds.el.gz 缓冲区。

启用 eldoc 辅模式 (M-x eldoc-mode)。 现在把光标定位到 if 条件将会显示一些简短的文档在回显区域。

也启用 show-paren-mode。这会帮助看清楚 THEN 和 ELSE 从句的开始和结束。

现在，回到 defun：

```
(defun c-indent-line-or-region (&optional arg region)
  "Indent active region, current line, or block starting on this line.
In Transient Mark mode, when the region is active, reindent the region.
Otherwise, with a prefix argument, rigidly reindent the expression
starting on the current line.
Otherwise reindent just the current line."
  (interactive
   (list current-prefix-arg (use-region-p)))
  (if region
      (c-indent-region (region-beginning) (region-end))
    (c-indent-command arg)))
```

你有三种方法在 defun 上获得帮助: eldoc 在回显区域的总结, C-h f 提供的参考，和 C-h S 提供的更详尽的 Info 手册。随你挑。

如果你发现了单词 "lambda", 这和 javascript 中的匿名函数中的关键字 "function" 是一个意思.

重要的事情说三遍:

- defun 定义了一个函数的名字.
- ARGLIST 函数的一个参数列表. 在 elisp 中, 一个 list 是由闭合的括号包住的：(a b c). 这里不是作为求一个函数调用的值，因为 defun 是一个特殊的表达式用来对这个特定的 list 做出一些特殊对待。当定义一个没有参数的函数时，ARGLIST 会是一个空的列表 （）。
- 选项 DOCSTRING 被 C-h f 帮助系统使用 (是的，即使对于函数你定义了你自己！).
- BODY 是一或多个被求值的列表当你调用函数时。
- 。。。除了对于（交互式。。。）表达式。

C-h S interactive

不要被解析陷得太深; 阅读一两段就够了. 学习只找到你需要的信息, 不然你很容易就会淹没. 现在我们不需要知道如何使用交互式终端, 只要知道它的意义.

配置缩进

嗯. c-indent-line-or-region 是一个函数接收参数 arg 和 region, 当函数被交互式地调用时(比如通过按下 TAB 键), 被设置为前置参数(如果被 C-u 或类似的行为所指定) 并且 "true" 如果 region 是激活的.

现在我们关系缩进当操作不是在一个 region 而是单独一行 (比如. region 是 nil), 那么让我们看看 ELSE 从句:

```
(if region
    (c-indent-region (region-beginning) (region-end))
  (c-indent-command arg))
```

使用 find-function 来跳转到 c-indent-command 的定义.

这是一个又长又吓人的函数, 但是幸运地是它有着很好地文档. 现在我们知道了这个函数的名字, 我们可以在帮助缓冲区中查看同样地文档, 使用 C-h f c-indent-command.

文档提到了两个有趣的变量: c-basic-offset 和 indent-tabs-mode.

C-h v c-basic-offset

谈论了变量 “buffer-local” 和 “file local” . 什么?

C-h S buffer-local

要从 elisp Info node 中找到 Buffer-Local Variables, 搜索 “file local” (你可以使用 C-s 或者 index i).

正如你所见, 有时候需要尝试不同的搜索来找到正确地信息. “Buffer-local” 正好在索引中, 所以符号搜索 (C-h S) 可以找到它; “File local” 有一个空格, 但是符号搜索不允许有空格, 搜索你不得不在 Info 缓冲区中搜索.

你也可以从 “Buffer-Local Variables” Info 节点网上扫描到 “Variables” 目录.

让我看看当前的 c-basic-offset 的值:

切换到缓冲区 rl.c(c-basic-offset 是本地缓冲区, 所以我们在哪个缓冲区是要紧的).

M-x eval-expression RET c-basic-offset RET

现在改变它为 4:

M-x set-variable c-basic-offset 4

找到一行要重新缩进的然后按下 TAB.

对变量 indent-tabs-mode 重复相同的行为.

Setting variables from elisp code

切换到 **scratch** 缓冲区.

; 开头的就是注释.

在这个表达式中输入 C-M-x 来求值: (set indent-tabs-mode nil)

你触发了一个错误, 然后 Emacs 在 elisp debugger 中提出了回溯.

你尝试给 nil 设置一个常亮, 这当然是一个错误.

使用你之前对于 indent-tabs-mode 和 elipse 对函数求值的知识来解释这是为什么.

我们可以把变量的名字括起来这样它就不会被求值了:

(set 'indent-tabs-mode nil)

阅读更多关于 quoting:

C-h S quote

C-h S setq

下面的表达式是一样的; 更好的表达式是 setq.

```
(set 'indent-tabs-mode nil)
(set (quote indent-tabs-mode) nil)
(setq indent-tabs-mode nil)
```

最后一件事: indent-tabs-mode 是 buffer-local, 所以在这里设置它只对 **scratch** 缓冲区有影响. 要让改变全局生效, 你必须使用 setq-default.

Init file

在你重启 Emacs 后你对这些变量的改变会丢失. 你需要把你的设定放在一个初始化文件中这样 Emacs 就会在每次启动时加载它.

In the Emacs manual (C-h r) table of contents, search for “init file” and read the first paragraph.

有几个地方你可以放置你的初始化文件; 我建议的地方是 ~/.emacs.d 文件夹, 这样你就可以通过同一目录下不同的 elisp 文件在组织管理你的自定义设置, 并且在主初始文件中加载它们. 把它们纳入版本控制.

访问(打开)你选择的初始化文件(如果文件不存在, Emacs 会创建它如果你保存了缓冲区).

把下面的几行加上:

(setq-default c-basic-offset 4) (setq-default indent-tabs-mode nil)

(或任何你选择的值).

重启 Emacs, 访问 rl.c, 并且确保你的设置生效了.

钩子

默认情况下，cc 模式会自动重新缩进行当你输入一个字符比如 ; 或者 } 时。 这些被成为 “electric characters” 不过你可以禁用这个行为在一个特定的缓冲罐你去中 使用 c-toggle-electric-state (C-c C-l)。

想要总是禁用 electric characters 我们可以让 Emacs 每次加载 cc 模式时 调用 c-toggle-electric-state。

C-h m （从 rl.c 缓冲区， 或任何其他 cc 模式的缓冲区）来找出 cc 模式提供的钩子的名字。

一个钩子是一个变量包含一个要去运行的函数的列表，通常是一个特定编辑模式的入口。 对于 C 语言的代码我们有两个钩子：一个是所有被 cc 模式支持的语言，另一个仅仅对于 C 语言。 我们将会使用第一个， c-mode-common-hook。

把下面的内容加入你的初始化文件:

```
(defun my-disable-electric-indentation ()
  "Stop ';', '}', etc. from re-indenting the current line."
  (c-toggle-electric-state -1))
(add-hook 'c-mode-common-hook 'my-disable-electric-
indentation)
```

首先我们定义了一个无参函数并且调用 (c-toggle-electric-state -1)。 然后我们加入这个函数到 c-mode-common-hook 。

参数 -1 告诉 c-toggle-electric-state 被禁用了，而不是被 electric behavior 所触发 （我是从 C-h f c-toggle-electric-state 中学到的；有些函数想要 nil，但是这个要的是一个负数 ）。

你可以直接给一个钩子加入一个匿名函数:

```
(add-hook 'c-mode-common-hook
          (lambda () (c-toggle-electric-state -1)))
```

但是之后你就没有办法通过名字在引用这个函数了， 所以你不能使用 remove-hook 来把它从钩子中移除了。

cc 模式的风格系统

说起代码风格，可不只是缩进的尺寸那么简单。 左花括号放在哪里？他们也需要被缩进吗?

c-basic-offset 的 C-h v documentation 提到了一个 『风格系统』， 并且指引我们到 c-default-style 。

C-h v c-default-style

```
c-default-style is a variable defined in `cc-vars.el'.
Its value is ((java-mode . "java")
 (awk-mode . "awk")
 (other . "gnu"))
```

Elisp 语法对于一个 list 是 (a b c)， 对一个 pair 是 (a . b)。 Pairs 被称作 “cons cells” 在 lisp 术语中， 并且你可以使用函数 car 来获取第一个元素， 函数 cdr （发音 “could-er”） 来获取第二个。

所以 c-default-style 的值是一个包含了 3 个 pair 的列表； 着就像 java 模式中的查询字典，awk-mode 和其他是键，"java", "awk" 和 "gnu" 是 值 （这里，每个编辑模式的风格名字被 建 所代表了）。 这些查找字典被称为 “alists”。

C-h S alist

在你的初始化文件中你可以设置 c-default-style 这样默认的风格对于其它就不是 "gnu" 了.

如果你需要自定义任何不在内建的风格内的东西(包含 c-basic-offset 和 indent-tabs-mode), 我推荐你定义你自己的风格: 这样如果你在不同的项目中使用不同的分割, 你就能很方便的切换(使用 c-set-style). 我们先前设置 c-basic-offset 的方式会自动创建一个叫做 "user" 的风格.

要获取帮助请查看 “Configuration basics”, “Customizing indentation”, 和 “Sample .emacs file” 在 CC 模式手册中.

绑定按键

我常常在不同的源代码之间有的文件需要一个 8 个空格长的 tab，另一个在同目录的文件下 甚至同一文件的不同行，需要一个 4 个空格长的 tab。 让我们创建一个函数来在 tab 宽度在 2，4，8 个空格之间循环。

（我发现变量 tab-width 通过使用 apropos-variable 来搜索 “tab”.）

```
(defun my-tab-width ()
  "Cycle tab-width between values 2, 4, and 8."
  (interactive)
  (setq tab-width
        (cond ((eq tab-width 8) 2)
              ((eq tab-width 2) 4)
              (t 8)))
  (redraw-display))
```

这个条件表达式等于 2 如果 tab-width 等于 8； 变为 4 如果 tab-width 等于 2；否则就是 8. 在 Info 手册中查找 cond 和 eq 如果你喜欢。

我把搜索我写的函数命名为 “my-something”， 因为 elisp 没有单独的命名空间 为每个模式或包；通过这种方法我可以确保我的函数不会意外的被某个编辑模式所依赖的 一个已经存在的函数所覆盖。

现在让我们绑定新的函数到一个按键序列，这样我们就能方便的调用它了。 C-c 跟随一个字母就是用来让用户去定义的，那我们就使用 C-c t:

```
(global-set-key (kbd "C-c t") 'my-tab-width)
```

其中 global-set-key 中 “global” 的意味很明显了。 如果喜欢一个只在 cc-mode 有用的按键绑定， 使用 define-key 来添加 绑定到这个模式的 keymap:

```
(define-key c-mode-base-map (kbd "C-c t") 'my-tab-width)
```

我发现 c-mode-base-map 使用 C-h v c-mode- TAB TAB. 也有一个 c-mode-map 仅仅用于 C 语言，而不是所有被 cc 模式支持的语言。

关联文件扩展名和一个编辑模式

假设你想要 .h 文件被打开时使用 c++ 模式而不是 c 模式:

C-h v auto-mode-alist

C-h f add-to-list

这复杂的部分会正确处理正则表达式 - elisp 没有正则对正则表达式的文字的语法, 所以你不得不把它放在一个字符串中, 然后用反斜杠转义, 这样让正则更加糟糕了. 查看 Emacs 手册中的 "Regexps".

家庭作业

如果你有 30 分钟时间, 阅读 Steve Yegge 的 Emergency Elisp.

如果你有 6 个月的时间来完成计算机程序的结构与解释, 一个 MIT 的有名的课本教授了重要的(也很先进)的编程概念和技术使用一个简单的 Lisp 方言叫做 Scheme. 如果你喜欢数学方面的东西比如用埃拉托斯特尼筛法来找出素数, 海伦的方法来计算平方根, 或者蒙特卡罗模拟法来估算 pi 值, 那你会爱上这本书的.

如果你在阅读本手册之前在网上有一大堆的 .emacs 自定义配置, 试着去彻底的理解它们.

将来, 当你需要一个特定的自定义选项, 试着去发现解决方案从手册或者 elisp 的源码在用谷歌搜索之前.

不要尝试去修改 Emacs 发行版自带的 elisp 文件. 第一, 要合并你的改变到一个新的 Emacs 版本并不容易. 第二, 文件是被编译过的, 所以你需要重新编译它们. 第三, 就算你这样做了也没什么用因为核心的 elisp 函数都是被内置在 Emacs 的镜像中的, 所以你不得不重新编译整个程序. 你应该做的, 是使用变量和钩子来提供自定义配置.

不要把注意力放在 Emacs 的手册上当它告诉你去使用 'Easy customization' 这个功能时(在一些帮助缓冲区中它会说“you can customize this variable”). 它指向一个非常糟糕的半图形化界面来设置 Emacs 的变量. 最好保持你所有的自定义配置在你的 init 文件中.

修复那个糟糕的配色方案

Emacs 默认的配色方案被亲切地称为 "生气的水果沙拉".

默认字体

"Fonts" 在 Emacs 手册中:

在 'Options' 菜单中点击 'Set Default Font'. 要在将来的会话中保存, 点击 'Save Options' 在 'Options' 菜单. [10: If you have disabled the menu, perhaps because you’ re using an init file copied from someone else or something like the Emacs starter kit, you can re-enable it just for this session with M-x menu-bar-mode.]

“DejaVu Sans Mono” 是一个很好地选择 (还有 Bitstream Vera Sans Mono”, 或者 OS X 10.6+ 上的 “Menlo”). 要选择一个等宽的字体, 即使你喜欢用等比字体编程. 我们之后会提到 Emacs 对等比字体的处理.

菜单 “Save Options” 命令自动用新的设置修改你的初始化文件(也会在 echo 区域告诉你, 现在在 **Messages** 缓冲区中如果你错过了).

有一个 OS X 的 bug 会阻止 “Save Options” 保存默认字体. OS X 用户就不得不使用我下面会解释的自定义字体机制来制定默认的字体名字.

语法高亮颜色

Font Lock 模式是一个辅模式负责语法高亮. 你可以阅读 font-lock 模式如果你想要了解 Emacs 是如何判断注释, 关键字和变量的, 或者如果加入你自己的关键字.

但是仅仅去修改颜色:

指出你想要修改的字体的名字: describe-face (默认是当前字体) 或者 list-faces-display.

修改字体会提示你使用一个字体和每个要改变的属性. 保留属性为 "未指定" (默认情况) 来从默认字体继承. 至于前景色和背景色, 你可以使用预定义的颜色通过 tab 自动补全, 或者指定 RGB 值比如 #3f7f5f.

自定义字体带来了"很容易自定的"借口我前面章节中警告过你不要去用. 点击 State 按钮然后选择为将来的会话保存.(你也可以在这个修改, 但是我更倾向 modify-face 因为它给你 tab 补全为每一个可能的值).

除了 descirbe-face, 我在帮助菜单中找到的, 我再 Emacs 手册的 “Faces”, “Standard Faces” and “Customizing Faces” 找到了其余的.

现在让我们弄明白如何疏通 “easy customization” interface:

访问你的初始化文件查看 “easy customization” 加入了些什么. 如果你已经打开了文件, 你需要 revert-buffer 来看看最新的修改.

```
init.el
(custom-set-variables
  ;; custom-set-variables was added by Custom.
  ;; If you edit it by hand, you could mess it up, so be
careful.
  ;; Your init file should contain only one such instance.
  ;; If there is more than one, they won't work right.
 )
(custom-set-faces
  ;; custom-set-faces was added by Custom.
  ;; If you edit it by hand, you could mess it up, so be
careful.
  ;; Your init file should contain only one such instance.
  ;; If there is more than one, they won't work right.
 '(default ((t (:height 120 :family "Menlo"))))
 '(font-lock-comment-face ((t (:foreground "#3f7f5f")))))
```

你可以通过给 custom-set-faces 表单添加参数来配置维护 font-lock 字体, 但是它们可能被覆盖如果你在将来使用了 easy customization framework. 受到 custom-set-faces 这个名字的启发, 我搜索了以 set-face 开头的函数并且发现了 found set-face-attribute:

```
(set-face-attribute 'default nil :family "Menlo" :height 120)
(set-face-attribute 'font-lock-comment-face nil :foreground "#3f7f5f")
(set-face-attribute 'font-lock-string-face nil :foreground "#4f004f")
(set-face-attribute 'font-lock-constant-face nil :foreground "#4f004f")
(set-face-attribute 'font-lock-keyword-face nil :foreground "#00003f")
(set-face-attribute 'font-lock-builtin-face nil :foreground "#00003f")
(set-face-attribute 'font-lock-type-face nil :foreground "#000000")
(set-face-attribute 'font-lock-function-name-face nil
                    :foreground "#000000" :weight 'bold)
(set-face-attribute 'font-lock-variable-name-face nil
                    :foreground "#000000" :weight 'bold)
```

上面的配置产生了一个非常保守的 dark-on-white 配色方案. 我喜欢它因为在很多语言中它高亮了被定义的变量和函数而不是它们使用的.

查看 Emacs Lisp 手册中的 “Face Attributes” 可以找到所有的字体属性名字. 对于 elisp 语法, 关键字符号比如 :foreground 和 :weight 是常量. (要对它们求值所以你不必用引号包含它们).

diff-mode

说起颜色, 让我们加入一些有用的配色在 diff-mode 中 (我们之前提到的, 在章节 vc version control interface).

[11: This customization—with several others—is borrowed from the Emacs Starter Kit.]

```
init.el
(eval-after-load 'diff-mode
  '(progn
     (set-face-foreground 'diff-added "green4")
     (set-face-foreground 'diff-removed "red3")))
```

eval-after-load 顾名思义: diff-mode 第一次被加载, 对后面的形式求值. Emacs 不会在启动时加载每一个 elisp 包, 而是等到你真正使用 diff-mode 才去加载它.

如果你直接在 init 文件里使用 (set-face-foreground 'diff-added …) 地话. 你会得到一个错误 “invalid face diff-added”.

其实你可以在初始化文件里加载 diff-mode (使用 require), 但是这会增加 Emacs 的启动时间, 所以当你需要在终端启动 Emacs 来作一个快速工作时会有轻微的延迟即使你不使用 diff-mode.

eval-after-load 由一个简单的求值构成, 但是我们想要创建两个函数调用, 所以我们把他们封装到 progn 中, 仅仅对一系列表达式连续求值. 我们把他们用引号括起来, 这样它们就不会被立刻执行, 直到被传到 eval-after-load 时.

diff-mode.el.gz

> provide the package (provide 'diff-mode) eval-after-load 的第一个参数需要匹配包提供的名字(这通常匹配被模式启动的命令的名字), 但是重复检查下: C-h f diff-mode, 跟随 diff-mode.el.gz 的链接, 发现靠近底部提供的表达式.

安装第三方的 elisp 包

我假设你已经把你的 ~/.emacs.d 纳入 git 版本控制了. [12: cd ~/.emacs.d; git init .; git add init.el; git status; git diff --cached;git commit -m "My emacs init file."]

color-theme 的安装说明推荐你使用包管理器(apt-get install emacs-goodies-el 或者 port install color-theme-mode.el 之类的) 但是我更喜欢在一个地方管理我所有的 Emacs 扩展. 那个地方就是我的 ~/.emacs.d 目录, 因为我终究会碰到一个我的系统包管理器没有提供的扩展.

Download the color-theme.6.6.0.tar.gz and extract into ~/.emacs.d.

```
git add color-theme.6.6.0;
git commit -m "color-theme 6.6.0 from
http://www.nongnu.org/color-theme/";
```

Add the color-theme directory to your Emacs load-path:

```
init.el
(add-to-list 'load-path "~/.emacs.d/color-theme-6.6.0")
```

And actually load it:

```
(require 'color-theme)
```

Now you can select a theme with color-theme-select.

从 github 安装包

Solarized 的 Emacs 配色主题在 github 上. 如果你使用 git 管理你的 ~/.emacs.d 目录, 你可以使用 git submodules 来简单地管理这样的第三方包.

```
cd ~/.emacs.d;
git submodule add https://github.com/sellout/emacs-color-
theme-solarized.git;
```

以后你可以在 ~/.emacs.d/emacs-color-theme-solarized/ 使用 git pull 来获取最新的更变.

Add ~/.emacs.d/emacs-color-theme-solarized/ to your Emacs load-path, and require 'color-theme-solarized, as you did for the color-theme package.

现在你可以使用 color-theme-solarized-light (或者 -dark) 来激活主题了.

Increasing the font size

C-x C-+, C-x C--, and C-x C-0. See “Temporary Face Changes”.

variable-pitch-mode

你可以在 fixed- 和 variable-width 字体在一个特定的缓冲区中使用 variable-pitch-mode 来切换. 自定义的 face 是 variable-pitch.

想要自动启用 variable-pitch-mode, 把它加入到所有你要生效的主模式的钩子中. 比如, 在 text-mode-hook 里编辑纯文本.

如果你要在任何地方都使用它, 我先假定给一个等比字体设置默认字体是无害的. variable-pitch-mode 的巨大优势是, 它很容易就可以在等比和等宽字体之间切换当你在一个注释中使用一些字符画和字符表格时.

[1]: I found this—after some dead ends—with Info’s index command, typing “font”, tab-completing, and trying whatever looked promising. I could just have explored the Options menu instead, but—silly me—I had disabled the menus because people on the internet said “real Emacs power users disable the menus.” That might make sense on a text terminal, where you can’t click the menu anyway, but on OS X, where there’s only one menu bar at the top of the screen, it’s just silly. See the next footnote.

常用自定义选项

ido-mode

其他的编辑器和 IDE 们有好看的文件夹和文件的树状视图，也有 tabs 来在缓冲区之间切换。 Emacs 有一个比较丑陋的树形视图 M-x speedbar。如果你想要看看有多丑的话。 但是真正的 Emacs 的答案去在大型项目中导航是 ido 模式。

```
init.el
(ido-mode 1)
```

ido 模式重新绑定了 find-file 和 switch-to-buufer 的按键为更强大的版本。 你只需要输入文件或者缓冲区的名字的一些字母（不必一定是名字的开头，也不必一定是相邻的字母）。 一个符合的列表，按照最近使用的排序，就会显示在 minibuffer 中；使用左右箭头（或者 C-s 和 C-r） 来在符合的列表中导航。如果没有项目的项目，在一个简明的（客配置）的暂停后，ido 会从之前使用的目录中搜索。

因为回车会打开第一个符合的文件，想要打开一个目录你需要使用 C-d。或者你可以使用 C-f（在 ido-find-file 提示下）来进入普通的 find-file

ido 和 Emacs 和搭配但是在手册中没有出现过；阅读 ido-file-file 的在线帮助。 想知道更多细节和配置说明，使用 C-h f ido-mode 来找到它的 elisp 实现， 并且阅读 elisp 文件的很长的开头注释。

Emacs 用户倾向于在使用完缓冲区后还是让它们保持打开状态，然后过了一段时间有了上百个打开的缓冲区 那么 ido-switch-buffer 就变成了有效的 『在项目中找到文件』。

要保存你的打开文件列表在不同的 Emacs 调用之间，或者分别管理不同套的打开文件（如果你同时工作在多个项目之间）， 查看 Emacs 手册中的『保存 Emacs 会话』。

朴素简约

想要最大化你的屏幕空间，考虑一下禁用工具栏和滚动条：

```
init.el
(tool-bar-mode -1)
(scroll-bar-mode -1)
```

类似的，你可以禁用菜单栏（menu-bar-mode -1），所让我觉得菜单栏在探索 Emacs 的特性时很有用； 主模式和辅模式常常会把它们自己的菜单加到菜单栏中。正如前面提到的，你肯定不应该在 OS X 中禁用菜单栏； 如果你想要在不同的环境中共享 Emacs 的初始化文件，你可以根据变量 system-type 和 window-system 来 有选择性的禁用菜单栏。

有一些命令比如 revert-buffer 会强制你通过输入 yes 来确认； 如果可以只输入 y 来确认那想必是极好的。 如果你查看 revert-buffer 的定义你会发现它调用了 yes-or-no-p, 你可以重定义为 y-or-n-p：

```
(defun yes-or-no-p (prompt)
  (y-or-n-p prompt))
```

或者更简单

```
(defalias 'yes-or-no-p 'y-or-n-p)
```

另一件事人们在安装好之后就会做的就是阻止 Emacs 的自动保存一个 ~ 开头的备份文件：

```
(setq make-backup-files nil)
```

应该使用你的版本控制工具来备份。

在 Emacs Lisps 之间导航

在前面几章中我已经鼓励你来探索 Emasc 的功能通过直接学习 elisp 代码。 为了让这个工作更容易，我们已经看到了 show-paren-mode 和 eldoc-mode; 让我们在全局启用它们因为我们在其他编程语言中也用得上。

```
(show-paren-mode 1)
(eldoc-mode 1)
```

我们也重新绑定了 M-。从它原本的 find-tag 到 find-function-at-point，但是仅仅对用 elisp 文件而言， 因为你不需要一个 tags 的 table 因为 Emacs 已经知道了所有它已经加载的 elisp 函数。

```
(define-key emacs-lisp-mode-map
  (kbd "M-.") 'find-function-at-point)
```

如果你计划写很多 Lisp 的话，paredit-mode 总是能让你的括号对等，对于重构时的大范围移动也很有用。

shell

如果你尝试过运行一个程序比如 git 在 M-x shell 下，你会看到一个警告 『terminal is not fully functional』 在不能使用的行为之后。因为 git 发送了它的输出通过一个分页（通常是 less），这需要一个真正的终端模拟器。

设定 PAGER 环境变量到 /bin/cat （但是仅仅在 Emacs 中这样）能解决这个问题：

```
init.el
(setenv "PAGER" "/bin/cat")
```

这也能让你在 M-x grep 中使用 git grep。

确保你不会覆盖 PAGER 在你的 ~/.gitconfig 文件或者 GIT_PAGER 环境变量中（和 man 程序的 MANPAGER 等等).

如果你需要在你的 ~/.bashrc 文件(或你选择的shell相符的配置文件)中自定义，你可以在环境变量 INSIDE_EMACS 中测试 。 参考手册如何配置 Emacs 使用的 shell。

其它想法

其他的你可能想做的自定义被 Emacs 手册包含了：

- Minor Modes.
- Making Buffer Names Unique.
- Displaying The Cursor (in particular global-hl-line-mode).

你可以查看其他人的初始化文件来获得更多的灵感，在 Emacs wiki 和http://sites.google.com/site/steveyegge2/my-dot-emacs-file[因特网上有很多]。 一些我体积的自定义配置来自 Emacs Starter Kit, 一个 elisp 文件的集合用来提供 『一系列你平常使用 Emacs 更舒服的默认配置』。

Info 文档

Info 文件的搜索路径

Info 手册告诉我们多个 Info 默认的目录列表：

```
(eval-after-load 'info
  '(progn
     (push "/opt/local/share/info" Info-default-directory-
list)
     (push "~/.emacs.d/info" Info-default-directory-list)))
```

第一个是我的包管理器（macports）安装的 info 文件的目录；根据你自己的系统修改这个路径。 第二个是我们手动安装的 Info 文件的目录。

glibc

glibc 是 GNU 的 libc 实现，Unix 的标准 C 类库。 不要和 glib 搞混了，那是一个 GNOME 桌面的类库。

如果你不是一个 GUN/Linux 系统的开发者，但是你没有发现你系统自己的 libc 的 info 格式的文档， 你也会发现 GNU 文档有用的地方。glibc 遵循 ISO C 99 和 POSIX 标准，并且它的文档 总是很仔细的指出哪个特性不是标准的。常用的 Unix 变量都被包含了（比如。 BSD 和 SysV 系统的信号处理）。

从 GNU 的官网下载 "Info document (gzip 文件)"

Untar into ~/.emacs.d/info/

```
cd ~/.emacs.d/info
install-info libc.info dir
```

install-info 是由 textinfo 包提供的， 你可以使用你的系统的包管理器来安装。

生成的目录文件包含一个入口，每个函数和常量在 libc 里。 这些入口会被加入到你的顶层 Info 菜单。 要从顶层菜单中移除它们是很简单的（你仍然可以发现它们通过 Info 的 index 命令或者搜索）： 目录文件是一个纯文本，所以你可以编辑它并且移除这些入口。 离开单一的 libc 入口在 『库』 中。

现在访问一个 C 文件并且使用 info-lookup-symbol 来查找， 比如说， socket。 来到主要的 sockets 节点来介绍材料。

Python

安装 pydoc-info.el （我们在上一节中提到了 elipse 包的安装）并且跟随在 pydoc-info 的 README 的说明在 " 配置和安装" 这节。

Perl

我已经发现了 perl 5.6(落后了10年了) 的 info 文档。 如果你知道一个更加新的版本， 或者知道如何从最新的 perl 文档构建 info 文件， 请告诉我详细的做法。

并且让全世界都知道！这是一个耻辱，这样一个良好的集成文档系统没有接收更多的支持从流行的软件发行版。

为 Emacs 作贡献

请考虑为 Emacs 贡献补丁，即使你是一个 Emacs 新手。特别是如果你是一个 Emacs 新手，没有人 比你更有资格来判断介绍文档的质量了。现在就贡献吧，在斯德哥尔摩综合症来临之前。

报告一个 bug

M-x report-emacs-bug 将会打开一个 邮件 的缓冲区带有模板方便你填充内容。你大概还没配置 Emacs 来 发送邮件，但是你可以复制这个缓冲区的内容然后粘贴到你常用的邮件客户端中。

跟随这个缓冲区的说明. 特别是, 确保你可以在使用 emacs -Q(用于确保 bug 不是由于你的 Emacs 配置或者第三方包造成的) 的情况下重现这个 bug.

一个简单的文档修改

ido-find-file 的文档这样说到：

在阅读后我依然不知道 C-j （在 ido-find-file 提示时）实际上做了什么。 实际上它意味着 『使用任何你输入的每个字』而不是使用 ido 的模糊匹配。 我会提交一个不到给文档。

上世纪80年代的补丁制作方法

M-x find-function ido-find-file.

保存一个这个缓冲区的拷贝（使用 C-x C-w） 到 ~/.emacs.d/ido.el 和另一份拷贝到 ~/.emacs.d/ido.el.orig。

修改 ~/.emacs.d/ido.el。要测试这个修改，求值（C-M-x）defun 从你的改变中。

使用 diff -cp ido.el.orig ido.el 生成一个补丁 [14: 不要问我为什么 Emacs 的维护者们不喜欢一元化的 diffs (diff -u). 他们指定要 diff -c]

创建一个 21 世纪的补丁

[15: 21 世纪需要半 G 磁盘空间和 30 分支的下载]

理想情况下你最好直接给最新的(未发布)的 Emacs 源码版本打补丁，而不是当前的 release 版。 因为其它人可能已经修复了你的问题！ Emacs 的源码在 savannah.gnu.org 然后你可以从官方的 Bazaar 仓库获得源码：

```
bzr bracnch bzr://bzr.savannah.gnu.org/emacs/trunk
```

或者，如果你更喜欢 git， 可以从 git 的镜像(急说只比 Bazaar 的仓库进度晚一天)：

```
git clone git://git.savannah.gnu.org/emacs.git
```

注意上面的任何一个命令都要花掉一些时间，但是你只要这样做一次。 我下面会使用 git 作为说明。

给你的修改创建一个私有（本地）分支：

```
git checkout -b ido-el-documentation --track master
```

直接在源码上做修改，并测试。

```
git commit -am "* ido.el: Documentation for C-j in ido-find-file and ido-
switch-buffer."
git format-patch master
```

将来，在开始另一个更变前你需要把最新的改变从上游拉下来：

```
git checkout master
git pull --rebase
```

[16: 这里 --rebase 是用来避免冲突如果你在本地有没有推送到 master 分支的提交。虽然你遵循了说明，你的本地修改可能只在私有分支上，而不是 master 分支。]

发送补丁

把补丁发送给和 M-x reprot-emacs-bug 同样的 email 地址。 但是首先阅读贡献的说明和 Emacs Lisp 的贴士和约定。

给第三方包做贡献

试着去包的网站上找到贡献指南：提交补丁的 email 地址，建议的补丁格式和 changelog 消息，等等。 如果这个包是在 github 上的， 包维护者可能开放了 github 的 pull request 接口。

人体工程学

下面提到的大多数建议都严重依赖 Control 键, 所以绑定你的 Caps Lock 到 Control键吧. 这在 OS X 和 Linux 下很简单, 在 Windows 上调用一个注册表.

你可能选择交换左 Control 键和 Caps Lock 键, 或者仅仅让它们都变成 Control 键 - 在 Emacs 和 bash 中你可以使用 M-u (upcase-word) 来大写一个单词, 使用 C-x C-u (upcase-region) 大写一个区域.

现在你将会发现导航键比如 C-f, C-b, C-n 和 C-p 都和你键盘上的方向键很接近. 学习更大的导航跳跃使用 C-a 和 C-e (一行的开始和结束), M-f 和 M-b (前进和后退一个单词), C-M-f 和 C-M-b (前进和后退一个区块根据一个花括号或圆括号), 和 M-a 和 M-e (一个语句或表达式的开始和结束).

如果你需要移动到一个很远的地方, 使用 C-s 搜索通常会更快.

对于修改错字, C-t (transpose-chars) 很方便.

与其伸手去按退格键, 试试使用 C-h 来退格(就像在大多数 shell 中一样), 使用 C-w 来反向杀词(在大多数 shell 中也适用; 在 Emacs 中通常绑定到 C-backspace). C-d 删除下一个字符, 你甚至不需要自己去绑定它.

重新绑定 C-h 有一点棘手, 因为在你绑定之后你能在一些主模式或者辅模式中绑定一个新的序列使用 C-h, 取消你自己的绑定. 你可以使用 key-translation-map 来让 C-h 在 Emacs 中作为退格键, 忽视所有的现在或将来的 C-h 绑定.

```
(define-key key-translation-map (kbd "C-h") (kbd "<DEL>"))
```

你还是可以使用帮助命令使用 F1 f, F1 v, F1 s, 等等。

比如说 C-w， 你可以定义你自己的函数来保持默认行为当选择区被激活，然后做 backward-kill-word 当没有被激活时：

```
(defun kill-region-or-backward-kill-word (&optional arg region)
  "`kill-region' if the region is active, otherwise `backward-kill-word'"
  (interactive
   (list (prefix-numeric-value current-prefix-arg) (use-region-p)))
  (if region
      (kill-region (region-beginning) (region-end))
    (backward-kill-word arg)))
(global-set-key (kbd "C-w") 'kill-region-or-backward-kill-word)
```

这些小贴士有一部分是受到 Steve Yegge 的 Effective Emacs的启发。去看看吧。

OS X

Control 和 Command 键

Mac 的键盘很方便的拥有独立的 Control，Meta (也称 Option 或 Alt) 和 Command (⌘)键。你有传统的 Emacs 键位绑定在 Control 和 Meta 键上，也有 OS X 的键位绑定在 Command 键上。

正如前面讲到的，你非常应该考虑一下重新绑定 Caps Lock 和 Control 键，在系统层面上。在 系统配置 》 键盘 》 修饰键。

在笔记本的娇小的键盘上，右边是没有 Control 键的。好的人体工学实践是使用一个右手边的 Control 键和一个左手边的普通键， 而不是拉伸你的左手。你可以使用开源的 KeyRemap4MacBook 来绑定你的回车键作为一个额外的 Control 键仅仅在按下时； 单次的击键仍然被注册为回车。试试看吧！它没听起来那么疯狂。KeyRemap4MacBook, 尽管名称不是很准确，在任何近期的 Mac 上都能工作。

在你经历了最初的学习曲线并且习惯了 Emacs 的绑定对于 打开，保存，复制，剪切，粘贴，和撤销后，你可能想要重新绑定 Command 键 为额外的 Meta 键，让你的手指更容易的按到目标。

init.el

```
(setq mac-command-modifier 'meta)
(global-set-key (kbd "M-`") 'other-frame)
```

(第二行保留了每一个使用 ⌘- 的行为.)

用这种方式重新绑定 Command 键在一个终端中的 Emacs 是不管用的, 因为终端程序会拦截 Command 按键.

如果你选择不是用 Command 键, 你可能想要重新绑定 ⌘-q 因为它很接近经常使用的 M-q. 或者至少让 Emacs 在关闭前给个提示.

```
(setq confirm-kill-emacs 'y-or-n-p)
```

系统级的 PATH

当你从 Finder(或者 Dock, 或者 Spotlight)启动 Emacs.app 时, 环境不会包含你的自定义的 .bash_profile 或者 .bashrc 文件. 如果你运行一个 shell 在 Emacs 中 (M-x shell), 那个 shell 会加载你的 .bashrc, 但是其他的 Emacs 命令比如 shell-command, grep 和 compile 不会.

你可以修改 Emacs 的 environment 目录在你的初始化文件中:

```
(setenv "PATH" (concat (getenv "HOME") "/bin:" "/opt/local/bin:" (getenv
"PATH")))
```

你也可以设置环境变量去应用到任何 OS X 的应用在你的 login 会话中, 通过创建文件

~/.MacOSX/environment.plist:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>PATH</key>
  <string>/usr/bin:/bin:/usr/sbin:/sbin</string>
</dict>
</plist>
```

从命令行运行 Emacs

如果你使用 macports 安装 Emacs.app, 你会发现命令行版本在 /Applications/MacPorts/Emacs.app/Contents/MacOS/Emacs (当你需要运行 Emacs 和某种命令行的切换比如 -Q 时很有用)

为本手册做贡献

本手册基于 GNU Free Documentation License 发行. 文档的源码在 github 上. 无论多小的补丁, 错误修正或遗漏, 都可以很方便的做贡献.

我本打算写多个引导的章节类似 "基本的 Unix/C 工作流" 这样的: 一个是面向 Ruby/Rails 开发者的, 等等. 但由于没时间我放弃了, 但是如果你想贡献这么一个章节, 快做把. 这会是很复杂的因为我介绍概念按一个特定的顺序, 和一个介绍的章节来说明 Rails 开发将会需要安装第三方的包, 这块内容我很久没接触了.

非常简单的贡献小补丁的方法

如果你有一个小的单词拼写错误要修正, 你可以在 github 上浏览源文件, 点击按钮 "Fork and edit this file", 完成修改, 然后提交一个 "pull request". 这需要你有一个(免费的) github 帐号, 但是你甚至不需要检出源文件在你的电脑上.

或者直接邮件我这个修正. 除非你告诉我, 否则我会用你的名字和 email 地址在 changelog 中, 任何人从 github 上下载源文件都能看到.

更复杂的修改

查看 README.

词汇表

Emacs key names:

Table 2. Table 按键

按键	说明
C-x	means Control-x.
C-x C-s	means Control-x, then Control-s.
C-x k	means Control-x, then k on its own (without Control).
M-x	means Alt-x (Alt is called Option on some Mac keyboards).The M stands for “Meta” which is presumably what the Alt key was called in the 70s.
M-x help	means press Alt-x, then type help, then press return.
C-M-x	means Control-Alt-x
S-x	means Shift-x
s-x	means Command-x (the ⌘ Command key on Mac keyboards).The s stands for “Super”, from the days when keyboards had Meta, Super, and Hyper keys.
<RET>	means the Return or Enter key.
<SPC>	means the Space bar.
<DEL>	means Backspace(not to be confused with the delete-forward key)

Frames:

Emacs 称作你的window管理的windows。【Emacs 手册】

Windows:

一个 Frame 中的独立视图。【Emacs 手册】

Buffers:

你在编辑的文字。一个缓冲区和一个文件的区别是，他们的内容可能是不同的直到你保存你的改变；并且有些缓冲区根本没有对应的文件 （比如 一个 **compilation** 或者 **Help** 缓冲区）。【Emacs 手册】

Mode line:

在 modeline 上面的窗口. 把你鼠标悬浮在每一个指示器上来查看说明。【Emacs 手册】

Echo area:

frame 的最底下的一行，用来显示小的提供信息的消息。【Emacs 手册】

Minibuffer:

提示用户输入的 echo area。【Emacs 手册】

The point, the mark and the region:

point 就是光标的位置。使用 C-<SPC> 来设置标记，移动光标，那么 region 就是 point 和 mark 之间。【Emacs 手册】

Killing and yanking:

Killing 就是剪切. Yanking 就是粘贴 (对不住 vi 用户了!)。【Emacs 手册】

Major mode or Editing mode:

一个主模式就是一个自定义的 Emacs 对于编辑一个特定的分类(比如, 一个特定的编程语言). 一个缓冲区同一时间只能有一个激活的主模式。【Emacs 手册】

Minor mode:

辅助模式可以同时启用多个; 他们的功能从语法高亮到自动拼写检查到修改通用的 Emacs 命令。【Emacs 手册】