从零开始——Emacs 安装配置使用教程 2015

字数 11995 阅读 49658 评论 20 喜欢 95

教程存档 Github

序|Preface

先来一篇有趣的简介：Emacs 和 Vim：神的编辑器和编辑器之神 - 51CTO.COM

为何写这篇教程？

作为一个彻头彻尾的 emacs 新手，尽管有些薄弱的编程经验，但上手这么一个黑客级别的编辑器还是难免一段阵痛期。虽然网上有很多非常好的教程，比如这篇著名的文章，一年成为 Emacs 高手(像神一样使用编辑器)，虽然提供了一个很好的学习框架，但具体的学习内容还是需要你自己寻找。这篇教程，在某种意义上可以被视为按照那个学习框架进行的学习实践。

在实践过程中，我发现相关基础知识的优秀教程散布在互联网的各个角落，缺乏一个有条理的组织，更别提有些排名靠前的教程早已过时，里面提到的操作和方法已经不适用于最新版的 emacs。

虽然 emacs 可以作为一个简单的文本编辑器开箱即用，但陡峭的学习曲线主要体现在让它发挥最大功效的个性化定制之中。这篇教程整理了我在配置 emacs 过程中学到的知识，方便新手循序渐进的学习。建议你将它作为一个有内在结构的课程来对待。整个教程大约会花掉你 18 个小时。

谁该阅读这篇教程？

我学习 emacs 的初衷是为了做日程管理，记笔记，并且写研究论文。我的学习和研究经常涉及到各种编程语言，特别是各种统计软件，比如 R，SAS，Stata，Matlab 等，当然还有 Python。虽然 R 和 Python 都支持“文学编程”模式[1]，但如果想将多种语言整合进一个文档的话，emacs 的 org-babel 模式无疑是第一选择。而既然是想写研究论文，对于 Latex 的支持必不可少。所以，我配置的 emacs 会以方便“文学编程”，多格式导出（主要是 Latex 和 html）以及各种文档模板为重点，主要围绕 emacs 的两个插件 org 和 auctex 来展开，可能还会涉及到知识整理，以及同 Evernote 互动的内容。另外，许多配置步骤在 Unix 系统下会方便很多，但很不幸，我使用的是 windows。

所以，如果你是 emacs 小白，但有一点编程经验，有同我类似的需求，并且也使用 windows，那么这篇文章就是为你量身定制的。欢迎阅读！

为什么不直接使用高手写好的配置？

这个教程本身就借鉴了许多高手的配置。我坚信“授人以鱼不如授人以渔”的格言。我相信我对你最大的帮助是这篇教程本身，而不是那些配置文档。希望作为新手的你，在阅读完这篇教程后，能获得足够的信心和动力来打造一个独属于你的 emacs。

如何学习这篇教程？

请注意我使用了“学习”而不是“阅读”。这意味着你需要自己动手，实践那些优秀教程提到的操作。自己写一些代码，而不是单纯的复制粘帖。

凡是附在“参考”后的文章，尽管大部分是英语文章，都请你仔细阅读。当然，你可以先浏览我写在后面的笔记和总结，来获得一个初步理解。你甚至可以只阅读我的总结来基本理解我要谈的事情，这看上去会节省你大量的时间，但其实你在不知不觉中错过了很多我没提及但对你很重要的细节。No pain, no gain...如果你坚持不读参考文章，请至少记得它们的标题。等你遇到问题时，至少知道该用什么样的关键词来搜索答案。

附在“延伸阅读”后的文章都很有价值，有助于你深入理解前文提到的概念。加了“必读”标签的请优先阅读。标签"TD"代表文章中出现了很多我还没提及的技术细节（Technique details），需要更多的背景知识。所以，如果你在阅读中感到困惑，可以先跳过，等学习完整个教程后再来阅读。因此，TD 还有一层 TODO 的意味……

在阅读我给出的文章时，你可能希望完成一个“知识闭环”后再回来继续学习。所谓“知识闭环”，就是持续了解阅读过程中遇到的新概念，直到不再遇到新的概念为止。显然，这会耗费大量的精力，而且你的疑问通常会在后续的教程中得到解决。我在撰写这个教程时，并没假定你完成了“知识闭环”后再继续学习，所以，建议你只对最大的疑惑做扩展阅读，比如**选择性**的浏览文章中包含的超链接。

下载安装|Download and Installation

官网：http://ftp.gnu.org/gnu/emacs/windows/

打开网页后，顶部是一段关于如何安装的说明文档。

总结：

- 不需要安装，解压缩到某个路径就可以了
- 为了方便全局调用，请添加 bin 路径到环境变量（比如我的，`C:\emacs\bin`）。你可能需要先了解下环境变量和命令行的基本知识。搜索关键词“windows 环境变量 PATH”，“启动 cmd”
- 测试下，在 cmd 里，输入 `emacs -nw`[2]，以终端模式来运行 emacs；只输入 `emacs`，以 GUI 模式来运行
- 文档介绍了 bin 目录下各个 exe 文件的功能，也介绍了怎样完全卸载，直接删除就好
- 解压缩完成后，可以运行 `bin\addpm.exe`，这样会自动生成配置文件 `.emac` 和目录 `.emacs.d`，并且在启动菜单里添加应用程序快捷方式。另外，官方文档里说还会添加注册表的相关条目。不过在我的电脑上，注册表并未新增相应条目
- 可以在桌面上新建一个快捷方式（shortcut），位置（location）填入 `emacs` 的安装路径`\bin\runemacs.exe --debug-init`。加了 flag `--debug-init`，是为了方便调试（debug）配置文件。不推荐为 `emacs.exe` 建立快捷方式，因为会额外启动一个命令行窗口。

请选择 24 以上的版本

延伸阅读： FAQ 3.2

其他版本

个人推荐

- 64 bit version
  http://sourceforge.net/projects/emacsbinw64/
  本人使用的版本，后面的配置都是基于这个版本
  版本号：24.4.91

下面这两个版本可以省去大部分配置的麻烦事。不过多劳多得，请自行选择

- All-in-one Emacs Binary by Vincent Goulet
  http://vgoulet.act.ulaval.ca/en/emacs/
- An Emacs Starter Kit for the Social Sciences by Kieran Healy
  http://kieranhealy.org/resources/emacs-starter-kit/

更多版本

基本操作

打开 emacs，同时按下 `Ctrl` 和 `h`，然后键入 `t`，阅读新手教程，熟悉界面，基本术语和操作。
请不要跳过这一步！（但不要求熟练掌握）
本文后面的部分已经假定你阅读了这个教程，所以默认遵循 emacs 的术语规范。

`C` 代表 `Ctrl` 键。`M` 代表 `Alt` 键。`RET` 代表 `Enter` 键（回车键）。`C-x` 代表同时按下 `Ctrl` 和 `x`。`C-x d RET` 代表先同时按下 `Ctrl` 和 `x`，再按下 `d`，最后再按下 `RET`。我在后文的按键描述中，会经常省略最后一步的回车操作。另外，请留意描述所用的英文字母的大小写。

Emacs 里的大部分地方都支持自动补全，快捷键是 TAB。

配置篇|Configuration

编程基础

你可能会奇怪，为什么配置 emacs 还需要编程？一般配置一个程序，不都是通过菜单栏打开一个对话框，然后修改里面提供的选项么？在 emacs 里，的确有这么一套配置系统，详见 Emacs's Customization Tutorial。但个人不推荐使用。因为，第一，里面提供的选项并不完全，许多配置只能通过编程才能做到；第二，它也是通过在你的配置文件中加入一些代码来实现的。

配置 emacs 的所有代码构成了一个配置文档。Emacs 的配置文档是用 elisp 语言写的。elisp 是 lisp 的一种方言。至于 lisp 语言，有人说它是黑客的语言。不过你并不需要完全理解 elisp 才能配置 emacs。至少我对 elisp 谈不上熟练。不过我倒是看过一些 lisp 语言的入门教程，所以能够看得懂 elisp 的官方帮助文档。你可以参考 Learn X in Y minutes 来快速入个门。由于 elisp 的函数命名规则大部分都很直观，所以

只要了解了基本语法，大部分配置语句对你来说都会变得很直白。下面列出几个配置文档时的常用函数，只是让你熟悉下 elisp 的语法。更多的函数会在用到时讲解。

参考 set

- 变量赋值。比如`(set 'a 5)`相当于 a=5

参考 setq

- 这个其实就是为了偷懒，在一次执行多个赋值操作时少打几个'

参考 let

- `let` 的意义在于批量执行函数时定义共享参数。考虑下面这个使用情景，你希望连续调用函数 A，B，C，它们都接受一个字符串参数 s，s 代表某个路径。A 负责打印字符串 s，并提示接下来要利用 s 做什么事情；B 负责切换到 s 指定的位置然后执行一些操作；C 负责将 s 加入到某个全局列表中。你当然可以不用 `let`，在调用函数前加上一句`(set 's 一个字符串)`即可。但这时定义的 s 会成为全局变量，进而污染你的变量空间。

HOME

参考 The Emacs Initialization File

- Emacs 配置文档常见的文件名有两个，`.emacs`，`init.el`，虽然本质上它们都是 elisp 脚本（像 python 脚本那样）。
- 你可以用任何文本编辑器来编辑他们。个人推荐使用 Notepad++，支持语法高亮，列编辑。Notepad++可以很方便的进行区域注释（`Ctrl+q`，`Shift+Ctrl+q`），这对调试配置文档很重要。当然，在你熟悉 emacs 后，emacs 也许会成为你的唯一编辑器
- 当你在使用 Notepad++编写自己的配置文档时，可能经常需要执行注释或者反注释某段代码的操作。注释时请用 `Shift+Ctrl+q`，反注释时请用 `Ctrl+q`。前者会保证在每一行代码前都加一个`;`，后者则是，如果本行代码以`;`开头，就删掉一个`;`，如果不以`;`开头，则添加一个`;`。容易理解，当代码块中包含注释时，你肯定不希望在注释掉代码的同时反注释掉那些注释。

接下来，让我们先来了解 emacs 在哪里寻找配置文档，以及会具体选择哪种格式。

参考 How Emacs Finds Your Init File，了解 emacs 启动时配置文件的加载规则

1. Emacs 会在系统中寻找一个名为 HOME 的变量，然后拷贝一个副本供自己使用，并在其指定的路径下寻找配置文件
2. 各个平台的默认 HOME 路径请参考 HOME and Startup Directories on MS-Windows
3. windows 平台，在 cmd 使用 `echo %userprofile%`来查看 HOME[3]
4. The MS-Windows System Registry 介绍了 emacs 寻找默认参数的路径的先后顺序
    - 注意，环境变量是第一位的，如果没有才会在注册表中寻找。也就是说，如果环境变量和注册表都包含 HOME 的话，emacs 会拷贝前者作为自己的副本。这通常不是一个好消息。一方面，你希望尽量按照自己的意愿来设置 emacs 的 HOME 变量；另一方面，你可能已

经为别的应用程序创建了系统级别的 HOME，以至于不得不把 emacs 的配置文件也放在那里

- 个人认为，一个更合理的加载逻辑应该是，顺序检测一系列路径，后面检测到的值覆盖前面的。这样你就可以通过创建注册表的方式来避免与系统环境变量的冲突
- 对于 windows 7/8/8.1，如果你的环境变量和注册表里都没有 HOME，emacs 会把`%userprofile%`的值设置为 HOME，一般是 `C:\Users\your-user-name\AppData\Roaming`

1. 通常，emacs 会优先加载`.emacs`，如果找不到，并且存在文件夹`.emacs.d`，会尝试加载其中的`init.el`
2. 基于前面的介绍，一个比较好的安装配置方案如下：

    - 将 emacs 的压缩包解压到某个路径
    - 运行 bin 路径下的`runemacs.exe`
    - `c-x d ~ RET`，编辑区域左上角的文件路径即 emacs 的 HOME。或者键入`C-h v user-init-file`并查看返回值
    - 在 HOME 路径下，emacs 会自动生成`.emacs.d`文件夹，如果没有请自己建立
    - 在该文件夹下新建`init.el`，输入如下代码

    ```
    ;; This file is only for windows 7/8/8.1
    ;; The only thing it does is to set the HOME directories for emacs,
    ;; then trigger the init.el in the directory specified by HOME to
    ;; accomplish the true initialization
    ;; You should put this file in the **default** HOME directory right
    after
    ;; emacs is installed
    (setenv "HOME" "C:/emacs/") ;; you can change this dir to the place
    you like
    (load "~/.emacs.d/init.el")
    ```

    - 最后一行代码中，~代表 emacs 的 HOME 路径。由于前面已经重新设定 HOME，所以这行代码相当于调用`C:/emacs/.emacs.d/`下的`init.el`。关于`load`命令，后面有详细解释
    - 顺便删掉前面几步中你见到的任何`.emacs`文件，保证 emacs 利用`init.el`启动

这样做的好处是，除了可以自定义`.emacs.d`所在的路径，还可以方便的备份整个文件夹，因为插件通常会被安装到这个文件夹下。如果需要换到其他电脑甚至平台时，只需要把整个文件夹复制过去，然后类似于上述步骤那样，想办法让真正的`init.el`发挥作用即可。
使用`init.el`而不是`.emacs`来配置，可以保证配置文件的结构化和模块化，方便维护。

最后规定后文要经常用到的几个代指

- ~代指**重定义后**的 emacs 的 HOME 路径
- `user-emacs-directory`指代`~\emacs.d`，该路径可以在启动 emacs 后通过`C-h v user-emacs-directory`来查看。
- `init.el`代指`user-emacs-directory`下的版本，**是我们要配置的版本**

延伸阅读：

- General Variables 必读
- DotEmacsDotD
- FAQ 3.4 3.5 3.6
- The Emacs Initialization File TD
- Summary: Sequence of Actions at Startup TD

PATH

从这个章节开始，对于提到的非 emacs 程序，都假定这些程序的主要可执行文件（exe）所在路径已经被添加到系统的环境变量 PATH 中。 对于 python，R，pandoc，cygwin 等，网上有很多安装并配置环境变量的教程。仍不熟悉基本操作的可以先看看“延伸阅读”的第一篇文章。

在向 `init.el` 写入任何代码之前，先打开 emacs 试用一下。键入 `M-x python`，如果没报错的话，就成功进入了 python 模式。Emacs 并不自带 python，那它是怎么知道去哪里调用 `python.exe` 的呢?

参考 Emacs: Set Environment Variables within Emacs

原来 Emacs 继承了 windows 的环境变量 PATH。输入 `M-x getenv RET PATH` 查看 PATH[4]。

实际上，当你在 emacs 中运行 shell 时[5]，各个指令的搜索路径是 PATH。而当 emacs 自身需要寻找某个可执行文件时，比如 python，搜索路径是 `exec-path`，而默认，在 windows 平台下，emacs 会**直接**拷贝系统的环境变量。也就是所说，在 `init.el` 中修改 emacs 的 PATH 副本并不会同时修改 `exec-path`。

当我们安装了一些只想同 emacs 结合使用的软件时，如果不想修改系统的环境变量，可以在 `init.el` 中加入：

```
(setenv "PATH"
  (concat
   "C:/Program Files (x86)/Notepad++" ";"
   (getenv "PATH")
  )
)
```

这样，你就可以在 emacs 中打开一个 shell，然后键入 `notepad++`来调用它了。注意，这个修改并不会在 `exec-path` 中追加相应的路径。如果你希望 emacs 也能调用 notepad++，还需要同步修改 `exec-path`，具体方法请参见原文。

如果你像我一样不想同步 `exec-path`，**最简单的方案就是把相关程序的安装路径添加到系统的环境变量中**。

延伸阅读：

- Windows Environment Variables Tutorial 必读
- Environment Variables

加载

对于任何软件，一个得心应手的配置基本基本都要用到插件，比如 Chrome。

对于 emacs，新安装的插件经常要你自己去启动并配置。这是 emacs 上手难的重要原因之一。考虑一个最简单的安装流程，你从网上下载了某个`**.el` 文件，然后在 `init.el` 中 `load` 这个文件。是不是 `load` 那一步显得很别扭？而功能更强大的插件可能由更复杂的文件结构组成，需要你做更多的准备工作才能正常使用。这个时候，一个插件管理系统就很必要了。24 以上的版本都集成了一个插件管理器 `elpa`，可以方便的通过 `M-x list-packages` 来安装插件。不过别高兴的太早，通过 `elpa` 安装的插件通常仍需要你手动来加载和配置。

注意，是**加载**，而不是**激活**。回忆下你是怎么使用 Chrome 的插件系统：安装插件，插件的图标出现在浏览器地址栏的右侧，点击插件的图标来使用插件（激活其功能），有的插件甚至默认激活。这个过程中，所有加载和初始化配置的工作都由软件自动完成，你唯一需要做的就是选择用不用（激活）而已。

然而，`elpa` 要求你自己完成加载和配置的步骤。一般来说，常见的载入命令有，`require`，`load`，`autoload` 等。而所谓的配置就是初始化一些参数。

emacs 一般称“插件”为"package"或者"library"。本质上，它们都提供一堆定义好的函数，来实现一些操作，进而实现某个功能。这里多说几句。在 emacs 中，连移动光标这种最底层的操作都有对应的函数。比如，你在 emacs 中可以键入 `C-f` 来将光标向右移动一个字符，同时也可键入 `M-x forward-char` 来实现。任何复杂的功能，比如给文档生成一个目录，都可以被分解为一个个操作，或者说调用一个个函数，而这些函数顺序执行下来功能就得到了实现。

当 emacs 想要加载某个插件时，归根到底需要**定位并运行**一个（也许是一些）脚本文件，那个脚本里定义了实现插件功能所需的变量和函数。emacs 将它们转变为可供自己使用的对象（elisp object），放到运行环境中等待调用。而脚本自身还可以在内部进一步加载其他脚本。下面，来了解加载脚本的几个语句，`load`，`require`，`load-file`，`autoload`。

参考 Emacs Lisp's Library System: What's require, load, load-file, autoload, feature?

- `load` 一个位于硬盘上的文件，意味着执行这个文件里的所有 elisp 语句，然后将执行结果放进 emacs 的运行环境
- Feature 可以理解为“特色功能”，比如，你在苹果的 App Store 里查看应用程序简介时，一般都会看到一个以 Features 开头的段落。单数形式，feature，一般对应一个插件的名字，因为一般插件的名字直接表明它实现的功能。复数形式，features，是一个用来存储 feature 的列表，这个列表可以告诉 emacs 哪些插件经被加载了。一般情况下，一个插件的启动脚本的结尾会调用 `(provide '<symbol name>)`，将`'<symbol name>`加入到 features 中去。`'<symbol name>`一般就是插件的名字
- `(require '<symbol name>)`会先查看 features 里面是否存在`<symbol name>`。如果存在，语句执行完毕。如果不存在，基于它来猜一个文件名，或者由 `require` 的第二个参数直接指定文件名，然后 `load` 文件。注意，`load` 完成后，`require` 函数会再一次查看 features

列表中是否存在'<symbol name>，如果发现还是不存在，视参数<soft-flag>来决定是否报错

- require 的意义在于避免重复加载。比如，某个插件的启动脚本中需要用到另一个插件提供的某个函数。那么它就会 require 这个插件，保证插件已被载入，然后再执行后续语句。
- load 会搜索 load-path，load-file 需要指定文件路径，autoload 在一个函数被 call 后再 load 指定文件

延伸阅读 Required Feature

其实，连整个 emacs 的启动都可以概括为一句话：加载一系列脚本。只不过这些脚本有的是内置的（built in），有的是你安装的插件包含的，有的是你自己写的。

配置 emacs 归根结底是在配置各种各样的脚本。

接下来，请思考如下问题。
你可以在 init.el 就 load 各种各样的脚本，使得 emacs 在**启动时**就把整个使用过程中**可能**用到的函数**一次性**准备好。但这样真的好么？

参考 Autoload

- autoload 告诉 emacs 某个地方有一个定义好的函数，并且告诉 emacs，先别加载，只要记住在调用这个函数时去哪里寻找它的定义即可
- 这样做的一个好处是，避免在启动 emacs 时因为执行过多代码而效率低下，比如启动慢，卡系统等。想象一下，如果你安装了大量的有关 python 开发的插件，而某次打开 emacs 只是希望写点日记，你肯定不希望这些插件在启动时就被加载，让你白白等上几秒，也不希望这些插件在你做文本编辑时抢占系统资源（内存，CPU 时间等）。所以，一个合理的配置应该是，当你打开某个 python 脚本，或者手动进入 python 的编辑模式时，才加载那些插件
- 一个简单概括：“只注册函数名而不定义函数本身”

前面介绍了几种加载机制。加载的目的在于定义变量和函数以供使用。任何插件，只有先被加载才能被使用。而且通常，你都希望先加载一个插件，再来配置它。考虑如下情景。

你的插件中定义了一个变量 a，默认值是 1，插件内定义的许多函数都在内部使用了 a。你希望在自己使用这些函数时，用到的 a 的值是 2。有两种实现途径。一种是直接到插件的脚本文件中修改 a 的值为 2。这叫做"hard coding"，有很多坏处。比如，每次更新插件，都要重新修改。另一种方法是，等这个插件已经被加载后，修改相应的 elisp object。那自然，你得先让这个对象存在于 emacs 中，然后才能修改。所以要先加载，让需要配置的变量得到定义，再去修改变量的值。

下面，让我们来看看这些脚本文件究竟长什么样子。打开 emacs 内置插件的文件夹，emacs 安装路径\share\emacs\24.4.91\lisp，你会看到一些子文件夹，一些后缀名为 gz 的压缩文件，以及一些后缀名为 elc 的文件。压缩文件中存放的其实是同名的 .el 文件，也就是前面一直在提的脚本。.elc 是这个脚本编译好的版本，可以加快载入速度，不适合人类阅读。所以，如果你想查看一个插件的源代码，请查看 .el 文件。.el 被放在压缩包是为了避免源代码被修改，进而造成各种问题。另外，加

载插件时，总是会优先加载编译好的版本，其默认的文件扩展名即`.elc`；如果不存在，才会加载`.el`或者其他格式的文件。

延伸阅读

- Features TD
- How Programs Do Loading TD
- Libraries of Lisp Code for Emacs TD
- Byte Compilation TD

Elpa

有了前面铺垫的基础概念后，让我们来学习使用 elpa。Elpa(Emacs package system)也是一个插件，只不过它是管理插件的插件。在 emacs24 和更高的版本中，elpa 是一个内置插件，脚本文件`package.el`位于`emacs`安装路径`\share\emacs\24.4.91\lisp\emacs-lisp`。有些插件因为由多个脚本构成，会被放在一个单独的文件夹中。初始化这个脚本的主脚本的文件名通常由插件名加上`.el`构成。注意，如果你修改了一个脚本文件，并且同名`.elc`存在，那么必须重新编译该脚本才能使改动生效。

参考 Emacs: How to Install Packages Using ELPA, MELPA, Marmalade

- 默认的插件安装路径是`~/.emacs.d/elpa`
- 默认情况下，elpa 的相关函数已经在启动 emacs 时注册（回忆`autoload`）。直接键入`M-x list-packages`即可调用
- 由于在启动时只是注册函数名，所以 elpa 的启动脚本并未加载。如果你想在配置文档中修改脚本中定义的变量，比如`package-archives`，请先`(require 'package)`。**该原则适用于其他插件的配置**。也就是说，如果你想在`init.el`中修改某个插件的某个变量的值，请保证 emacs 在执行这条修改语句时，相关变量已经得到定义
- 一般用来初始化该插件的主脚本的文件名都是插件名`.el`

为了保证你可以自行试验后文的操作，现在请你到`init.el`中添加一段代码：

```
(require 'package)

;;; Standard package repositories

;; We include the org repository for completeness, but don't normally
;; use it.
(add-to-list 'package-archives '("org" . "http://orgmode.org/elpa/"))

;;; Also use Melpa for most packages
(add-to-list 'package-archives '("melpa" .
"http://melpa.milkbox.net/packages/"))
(add-to-list 'package-archives '("melpa-stable" . "http://melpa-
stable.milkbox.net/packages/"))
```

上述代码给 elpa 添加了几个额外的插件来源。不用理会其中的语法，反正在后面配置`init.el`时我会提醒你删掉这段代码。

需要注意，elpa 智能但不傻瓜。

参考 Emacs 24 Package System Problems

- 安装一个插件后，elpa 会自动在插件所在目录下生成一个 autoloads 文件。这个文档本意是方便你调用插件的。比如，你可以在 `init.el` 中加入`(load 某某插件-autoloads)`来加载该插件
- 如果你希望用 `require` 的方式来加载插件，并且还希望 `require` 这个 autoloads 文件，会出现一个问题。autoloads 的结尾并没有`(provide '某某插件-autoloads)`，所以 `require` 一定会报错。而且这样做也没什么意义。因为你的目的在于将插件本身的名字放到 features 列表中，而不是“插件名-autoloads”。所以，请 `load` 而不是 `require`autoloads 文件
- 当然，你也可以直接加载插件的主脚本，比如`(require 'auto-complete)`而不是 `(load 'auto-complete-autoloads)`。不过，这样做有两个坏处。第一，有些插件可能会指导 elpa 在生成 autoloads 文件时加入一些配置代码。在这种情形下，有可能你通过 `load` 这个 autoloads 文件能成功初始化插件，而直接 `load` 或者 `require` 插件的主脚本则不能。第二，autoloads 由 `autoload` 函数构成，`autoload` 的好处如前所述，可以轻便化 emacs 的启动

load-path

下面来谈一个很重要的变量，`load-path`，其变量类别是“列表”，作用范围是“全局变量”。打开 emacs，键入 `C-h v load-path RET`。如果你是在刚安装完 emacs 后键入这个命令，得到的返回值应该类似这样：

```
("c:/emacs/share/emacs/24.4.91/site-lisp" "c:/emacs/share/emacs/site-lisp"
此处省略若干行
```

中文部分是我自己加上的，告诉你我为了节省空间，删掉了许多行。

每次使用 elpa 安装插件后，这个值都会发生改变。比如，在**初次**使用 elpa 安装完 ack 插件后，`load-path` 会变为：

```
("~/.emacs.d/elpa/ack-1.3/" "c:/emacs/share/emacs/24.4.91/site-lisp"
"c:/emacs/share/emacs/site-lisp"
此处省略若干行
```

请自行把~脑补为 HOME 路径。

通过对比，不难发现，emacs 在启动时，会将 `user-emacs-directory/elpa` 路径下的的所有文件夹加入到 `load-path` 的**头部**。由于 elpa 的默认安装路径是`~/.emacs.d/elpa`，所以第一行会是`~/.emacs.d/elpa/ack-1.3/`。你用 elpa 安装的任何插件，其所在路径都会位于 `load-path` 头部。我想强调，这个位置，非常重要。

在 emacs24 及更高的版本中，emacs 自带了一个 org 插件，位于 `emacs` 安装路径`\share\emacs\24.4.91\lisp\org`，这个插件后面会详细讲解。每次启动 emacs，这个路

径都会被添加到 `load-path` 中。在 emacs 中键入 `M-x org-mode` 会调用 org 插件，让编辑区域进入 org 模式。

org 插件有很多相关插件。假设现在，你想通过 elpa 安装某个相关插件，比如，bog，执行如下操作：

- 键入 `M-x list-packages RET`，出现选择编码的提示，键入 `RET`
- 定位 bog：键入 `C-s Extensions for research notes in Org mode`，然后键入 `C-s RET`
- 在 emacs 窗口左侧，点击光标所在行出现的小个左箭头，然后点击 bog

你会在新出现的窗口看到语句 `Requires: org-8.0.0, dash-2.5.0`，表明该插件依赖额外的两个插件 org 和 dash。elpa 会智能的安装所有依赖插件。注意，尽管你的 emacs 自带 org，elpa 还是会选择安装自己的插件源中的版本。所以，最后 `load-path` 会变为：

```
("c:/emacs/.emacs.d/elpa/bog-0.6.0/"
"c:/emacs/.emacs.d/elpa/dash-20150311.2355/"
 "c:/emacs/.emacs.d/elpa/org-20150316/"
 "c:/emacs/.emacs.d/lisp"
 "c:/emacs/share/emacs/24.4.91/site-lisp"
此处省略若干行
"c:/emacs/share/emacs/24.4.91/lisp/org"
此处省略若干行
```

elpa 安装的 org 排在了 emacs 自带 org 的前面。

`load-path` 如其名字所示，告诉 emacs 在加载任何脚本时，如果没有指明脚本所在路径，那么就去 `load-path` 所含的路径中寻找。然后使用**第一个**找到的脚本。也就是说，此后你调用 org 插件时，使用的都会是 elpa 安装的版本，即插件的一个版本'shadow'了另一个版本。

'shadow'现象很常见。除了前面提到的'shadow'内置插件，elpa 安装的插件的新版本会'shadow'旧版本。请记住一个非常有用的命令，`list-load-path-shadows`，它可以总结所有插件当前的'shadow'状态。现在，请你自己键入 `M-x list-load-path-shadows RET`，然后阅读下返回的信息。

'shadow'之所以发生，是因为 `load-path` 中包含了同一个插件多个版本的脚本路径，哪个版本排在前面就使用哪个。

总结下，在配置插件时，请时常反问自己如下问题：

- 当我想加载一个插件时，emacs 知不知道它的所在路径？
- 当我想修改插件定义的某个参数时，是否已经加载了这个插件？
- 会不会某个已经存在的版本，shadow 了我想使用的版本？

最后，学习下修改 `load-path` 的常用操作。

参考 Modifying List Variables

- 优先关注 `add-to-list` 的语法。

延伸阅读

- Library Search TD
- How to Install Emacs Packages Manually
- ELPA

牛刀小试

整套配置文件的思路参考 Emacs 配置文件——新手攻略。

虽说是新手攻略，还是太简洁了些。不过，请你大概阅读一遍，并将作者的配置文件下载到本地，解压，然后将 `emacs.d-master` 文件夹下的文件所有文件拷贝到你的 `user-emacs-directory`。这会覆盖你自己的 `init.el`，不要紧，当然你为了保险可以备份下。下面用之前建立的**专门用来调试配置文档的快捷方式运行 emacs**。emacs 会按照 `init.el` 的指导自动安装并配置相关插件。但不知你的运行结果怎样，我的会报错。

Required feature ... was not provided

```
Debugger entered--Lisp error: (error "Required feature `switch-window-autoloads' was not provided")
require(switch-window-autoloads)
eval-buffer(#<buffer  *load*-432260> nil "c:/emacs/.emacs.d/lisp/editing-utils/init-switch-window.el" nil t)
```

有了前面铺垫的基础，你应该能很好理解错误的原因：应该 `load` 一个 autoloads 文档，而不是 `require`。定位到出错的文档，把`(require 'switch-window-autoloads)`修改为`(load "switch-window-autoloads")`。注意，根据 `require` 和 `load` 的语法规则，我把 `switch-window-autoloads` 从一个符号（Symbol）改成了一个字符串（String）。

顺便检查下同文件夹下的其他配置文档，更正相同的错误。关闭 emacs 再次运行。你会发现，后续还会在各种各样的 init 文档中出现同样的错误。请一一更正。

"Cannot open load file" ... "org-exp"

```
Debugger entered--Lisp error: (file-error "Cannot open load file" "no such file or directory" "org-exp-blocks")
  require(org-exp)
  (progn (require (quote org-exp)) (require (quote org-clock)) (require (quote org-fstree)))
  (lambda nil (progn (require (quote org-exp)) (require (quote org-clock)) (require (quote org-fstree))))()
  eval-after-load(org (lambda nil (progn (require (quote org-exp)) (require (quote org-clock)) (require (quote org-fstree)))))
  eval-buffer(#<buffer  *load*-5658> nil "c:/emacs/.emacs.d/lisp/init-org.el" nil t)
```

这里的 `quote` 指单引号字符`'`。请打开文档定位出错语句。然后 Google 搜索"org-exp"，发现只有 org-exp-blocks，估计"org-exp"是作者自己写的吧。请注释或删除`(require 'org-exp)`。

在读过 org-exp-blocks 的帮助文档后，你可能非常想加载这个插件。不过请注意，文档中提到：

> make sure that the path to org's contrib directory is in your load-path and add the following to your .emacs.

什么是"contrib directory"？检索下本地的 org 插件所在文件夹，无论是 elpa 版本，还是内置的，都没有"contrib directory"。Google 后发现，这个目录里包含了许多 org 用户写的插件，因为不是 org 官方开发者写的，所以没被包含在前面的两个版本中。

到这里，也许你会以为 org-exp-blocks 也在"contrib directory"中。恭喜你，上当了。

参考 Org-mode Contributed Packages

- 请看"Moved to core"那一部分，很容易找到下面这句话
  "Org-exp-blocks is now part of the Org core. Link to raw file."
- 也就是说，现在不用手动调用 org-exp-blocks 了。所以，你其实什么也不用做
- 不得不说，有点坑爹。吃一堑长一智，请记住这个页面，以后配置文件出现问题时，也许不是被'Moved to core'，就是被'Obsolete'了

做完以上操作，再次启动 emacs，应该能顺利进入欢迎界面了。不过，要知道，还是有很多未被'Moved to core'但非常有用的插件，一般只包含在 org 官网提供的 beta 版本中。那应该怎样获取呢？

Build Org

参考 Org 官网

- 想获得官方的 beta 版本，需要用到工具 Git。下载并安装好。
- 用桌面上出现的 Git 快捷方式打开 Git，键入 `pwd`，记住当前的工作路径。或者你也可以通过 `cd` 命令来切换到你想要的工作路径
- 记住当前的工作路径。键入 `git clone git://orgmode.org/org-mode.git`。等待 beta 版本的 org 被下载到本地。提示：也许在你的 git 中，粘帖操作被绑定为鼠标右键
- 将工作路径下的 org-mode 文件夹重名为 org-beta，拷贝到 `user-emacs-directory`。重命名那步没什么特别含义，只是为了区分。如果你选择拷贝到其他路径，请自行调整后续命令
- 打开 `init-org.el`，在第一行加入`(add-to-list 'load-path (expand-file-name "org-beta\lisp" user-emacs-directory))`，相信你不用查阅帮助文档也能理解 `expand-file-name` 的作用。这行代码将 org-beta 的核心脚本所在路径添加到 `load-path`，相当于让 beta 版本'shadow'其他版本
- 你还需要把 org-beta 目录下下的 `org-contrib\lisp` 添加到 `load-path`，因为这个目录即前面所说的"contrib directory"。在第一行下面额外添加代码：`(add-to-list 'load-path (expand-file-name "org-beta\org-contrib\lisp" user-emacs-directory))`
- 到这步，你应该可以正常使用 emacs 了。不过，为了真正的“安装”org-beta，请继续执行下述操作
- 额外下载并安装 Cygwin。参考 Cygwin 详解“Cygwin 在线安装指南”一节。一定要执行“Cygwin 中模块的各种分类”一节提到的操作，即安装 Devel 这个部分的模块，因为要用到其中的 automake 模块。记得安装完后配置环境变量
- 打开 emacs，键入 `M-x pwd`，返回路径如果不是 org-beta 所在的那个，就切换过去。具体操作，键入 `C-x d ~ RET .emacs.d/org-mode/ RET`

- 切换后，再次键入 M-x pwd，确认路径正确。然后键入 M-！ make。注意，Alt 和!要一起按，即同时键入 Alt，Shift 和数字键 1。make 命令源于 Cygwin 中的 automake 模块，它会把 org-beta 的所有核心脚本编译好，然后建立帮助文档的索引
- 打开 emacs，键入 M-x org-version RET，返回信息中包含的路径如果是 org-beta，即表明'shadow'成功
- 前面几步的操作也适用于编译其他插件

现在，请你执行如下操作：

- 删掉 user-emacs-directory 下的 elpa 文件夹
- 重新运行 emacs，让 emacs 在更正后的配置文档的指导下重新初始化

你会发现，居然又报错了！出错语句是配置文档 org-magit-autoloads 中的(eval-after-load "org" '(progn (org-add-link-type "magit" 'org-magit-open 'org-magit-export) (add-hook 'org-store-link-functions 'org-magit-store-link)))。我想你已经猜到了，这一定跟使用 beta 版本的 org 有关。注释掉 init-org.el 中的头两行代码，让 emacs 使用 elpa 版本的 org。然后打开 emacs 键入 M-x list-packages 来强制刷新下插件列表。最后再次删掉 elpa 文件夹并运行 emacs。如果以后你想使用 beta 版本，记得反注释掉头两行代码。

一阵繁忙的下载后，emacs 应该能不报错的完成初始化。但是看看编译记录（complied log），发现有大量的 warning 信息。请把 log 保存下来，以便以后分析。把光标切换到 complied log 区域，键入 C-x C-f，然后选择合适的路径和文件名，键入 RET 保存 log。

后文中我会以 init.log 来代指这个文件。

恭喜，现在你已经拥有了一个功能非常强大的 emacs 了。赶快探索下吧。

最后补充下我个人偏好的额外设置。

init.el

关闭烦人的警示音。禁止启动后的欢迎页面。

```
;; Turn off sound alarms completely
(setq ring-bell-function 'ignore)

;; disable welcome page
(setq inhibit-startup-message t)
```

在 custom-set-variables 区域添加代码，让 emacs 启动后自动全屏。请注意括号的匹配。

```
(custom-set-variables
其他代码
 '(initial-frame-alist (quote ((fullscreen . maximized))))
 )
```

当你通过 emacs 的自定义系统（本篇最开始提到）修改 emacs 设置后，emacs 自动将相关代码添加到 init.el 的 custom-set-variables 区域。这里我们直接添加代码来实现功能。

安装 Emacs Speaks Statistics: ESS，使 org 模式下可以运行 R，SAS 等（当然，你要额外安装这些统计软件）

参考 Installing ESS on your system

- 同编译 org-beta 的步骤类似。我把 `ess-14.09` 放到 `user-emacs-directory`，然后运行 emacs 并切换工作路径到 `ess-14.09`，最后 make
- 在 `init.el` 中添加

```
(add-to-list 'load-path (expand-file-name "ess-14.09" user-emacs-
directory))
(load "ess-autoloads")
```

- 运行 emacs，键入 `M-x R`。如果能进入 R session，那么就是安装成功
- `(load "ess-autoloads")`是最小配置，如果你希望用到 ess 的全部功能，请加载 `ess-site.el`（注意 `load-path`）

`init-org.el`

开启 org 模式下的代码高亮；导出代码块时不运行代码；跳过运行代码块时的确认步骤（可能有安全风险）。在注释`;; Various preferences` 下方添加代码

```
;; Various preferences
(setq
      其他代码
      ;; turn on the syntax highlight in the org mode
      org-src-fontify-natively t
      ;; when exporting the org file, do not evaluate the code block if the
exports header is both
      org-export-babel-evaluate nil
      ;; skip the confirmation step when evaluate a code block
      org-confirm-babel-evaluate nil)
```

导出 PDF 时代码高亮使用 minted，在上面的代码块下方添加

```
;; Include the latex-exporter
(require 'ox-latex)
;; Add minted to the defaults packages to include when exporting.
;; set snippet-flat to nil to exclude minted for latex preview
;; see http://orgmode.org/worg/org-tutorials/org-latex-preview.html
(add-to-list 'org-latex-packages-alist '("" "minted" nil))
;; Tell the latex export to use the minted package for source
;; code coloration.
(setq org-latex-listings 'minted)
;; Let the exporter use the -shell-escape option to let latex
;; execute external programs.
;; This obviously and can be dangerous to activate!
;; multiple compile in order to generate everything
(setq org-latex-pdf-process
      '("xelatex -shell-escape -interaction nonstopmode -output-directory %o %f"
        "bibtex %b"
        "xelatex -shell-escape -interaction nonstopmode -output-directory %o %f"
        "xelatex -shell-escape -interaction nonstopmode -output-directory %o
%f"))
```

参考 Export org-mode code block and result with different styles

- 这段代码要求你安装了 Latex 和 Python。推荐使用 TeX Live 和 Anaconda。确保 Latex 安装了 minted 插件，Python 安装了 Pygments 插件。另外请配置好环境变量
- Latex 有时需要多次编译才能正确导出所有元素。因此会出现三个 `xelatex` 语句
- `bibtex` 命令可以生成 `.bbl` 文件，这个文件用来生成参考文献列表。放到中间是因为，它需要借助第一个 `xelatex` 生成的 `.aux` 文件，一个临时辅助文件，来实现转换。原理很简单。文献信息存储在格式为 bibtex 的 `.bib` 文件中。根据不同的文献引用标准和具体的引用条目（由 `.aux` 提供），`.bib` 内的信息在经过筛选、重组后被放入 `.bbl` 文件，用来生成最终的文献引用内容
- 因为 minted 包依赖 python，所以 latex 在编译时需要调用外部程序 python。latex 觉得这种行为存在风险，默认禁止。`-shell-escape` 允许 latex 运行"shell command"，进而允许调用 python

另外，如果你想在 org 模式下用 RefTex 来引用文献，有一个插件'ox-bibtex'，它可以在导出到 Latex 和 HTML 时自动生成参考文献附录。'ox-bibtex'在 org-contrib 中。所以如果要启用这个插件，请配合启用 org-beta 后再加载这个插件。
用这个插件导出 Latex 时，如果你遵照前面的配置，应该一切正常。HTML 导出功能需要用到 bibtex2html。许多人在使用这项功能时都会遇到错误 `Executing bibtex2html failed`。参考
Emacs: unifying citations between html and latex in org-mode，问题在于不能使用临时文件。这个问题最终也没得到很好解决。下面我给出一个 windows8.1+texlive 2014 使用环境下的解决方案，不保证其他环境也适用。

bibtex2html

其实方案很简单，安装最新版本的 bibtex2html 即可，目前是 1.98。这里只是给不熟悉 Unix 开发环境的同学们指个路。

参考 Github 上的说明文档

- 先去下载最新的开发者版本 bibtex2html-1.98.tar.gz，解压到本地文件夹中，比如 `bibtexdir`
- 运行 cygwin，`cd` 到 `bibtexdir`
- 键入 `./configure`，等待程序运行完毕
- 键入 `make`，等待程序运行完毕
- 如果你希望 cygwin 能在内部调用 bibtex2html，再键入 `make install`，这会把 bibtex2html 安装到 `cygwin64` 所在路径 `\usr\local\bin`
- 现在，`bibtexdir` 目录下会出现'bib2bib.exe'，'bibtex2html.exe' ，'aux2bib'
- 将三个文件拷贝到系统环境变量 PATH 中的某个路径，确保你在 cmd 中键入 `bibtex2html` 可以调用相关 `.exe` 文件
- 大功告成

`init-auctex.el`

使用 Sumatra PDF（请下载并安装）来预览 PDF。最大的好处是，可以从 PDF 逆向定位 TEX。即你编译完`.tex`文档并调用 Sumatra PDF 预览时，在 PDF 中双击某个位置，emacs 会自动打开对应的`.tex`文件并定位过去。

参考 Sync Emacs AUCTeX with Sumatra PDF，在`(load "auctex-autoloads")`下面添加

```
;; run latex compiler with option -shell-escape
(setq LaTeX-command-style '(("" "%(PDF)%(latex) -shell-escape %S%(PDFout)")))
;; use Sumatra PDF to preview pdf
(setq TeX-source-correlate-mode t)
(setq TeX-source-correlate-method 'synctex)
(setq TeX-view-program-list
   '(("Sumatra PDF" ("\"Sumatra 安装路径/SumatraPDF.exe\" -reuse-instance"
                     (mode-io-correlate " -forward-search %b %n ") " %o"))))
```

请将 Sumatra 安装路径替换为你自己的安装路径。并打开 Sumatra 的 option 界面，按照参考文章的回答设置`Set inverse search command line`。

其他配置

参考 Moving The Ctrl Key，绑定`ctrl`到`capslock`

我采用 AutoHotkey 的方式，并且将脚本放到 startup 文件夹来实现开机自启。我的电脑上，startup 的路径：
`C:\Users\xiaohang\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup`

延伸阅读 windows 下自定义配置的说明

初入江湖

经过以上配置，你的 emacs 应该已经比较好用了。不过在 emacs 世界里，此时的你还只是个初入江湖的小虾米。在相当的一段时间内，你会纠结于 emacs 复杂的按键组合，为千方百计也不能安装好一个小插件而抓狂。我想说，这都是正常现象。在这些痛苦中，你慢慢成长，从读官方文档开始，一点点熟悉 elisp，开始欣赏 emacs 的设计，甚至能自己写一个小插件。于是，你使用 emacs 越来越顺手，越来越想打造一个独属于自己的配置，最大化你在各个场景下的使用效率。

而我的教程到这里也要告一段落了。我已经把自己所知悉数传授给了你，从这里开始，我们处在同一个起跑线上。但我想，这套教程并不会结束，因为我还有很多承诺没同你兑现呢，比如分析`init.log`，比如讲解 org 模式。不过，相信你经过前面的学习，已经能靠依靠自己探索 emacs 中的大部分事物了。而我，也会逐渐积累自己的使用心得。

我计划如下呈现后续的教程：围绕一个具体的使用情景，我会向你描述我的插件选择，配置和操作习惯。

最后，如果你觉得这篇文章不错，请点击下方的**喜欢**按钮，谢谢支持！

好了，朋友们，下期再见～

---

[1]: R-markdown, iPython-notebook ↩

[2]: 选项 `nw` 代表"no window" ↩

[3]: 你也可以使用 powershell 支持的 `cd ~`来直接切换到主目录，从而得知主路径的位置…… ↩

[4]: 这些命令也可以：`M-: (getenv "PATH")`,`C-h v initial-environment` ↩

[5]: `M-x shell` 来激活一个 shell。你可以把它简单理解为一个运行在 emacs 里面的 cmd。 ↩