vi 编辑器的学习使用

vi 编辑器是 Unix 的世界中一个相当强大的可视化编辑器，有人曾这样的说过在世界上有三种人：一种是使用 Vi 的，另一种是使用是 Emacs 的，剩下的是第三种人。
由此可以看出 Vi 和 Emacs 的强大。在有关 Linux 的论坛中关于 Vi 和 Emacs 的论争也是一直不断的。现在看来这样的论争也实在是没有办法的，这两个编辑器是同样的强大，同样有不同的人在支持。在个人看来就是同时喜欢这两个的。然而他的这种强大也是要会付出一定的代价，他的学习和使用也是相当的复杂的，在学习的起点常常会面临着一个巨大的学习困难。但是只要过了这样的一段时间你就可以来体会 Vi 的强大了。

一 Vi 的启动与退出

在现在的 Linux 世界中还有一个发行版本被称为是 Vi 的改进版本，所以被称为是 Vim，也就是 Vi Improved 的意思。在现在的一般的 Linux 的发行版本中常常是 Vim 的。要启动 Vi 可以终端窗口输入 vi 或是 vim，这样就可以终端窗口打开一个 Vi 的编辑窗口。或者是输入 gvim，这样就可以打开一个类似于 gedit 这样的文本编辑器，他有一些菜单，但是大部分的功能仍是通过命令行的方式来完成的。在 vi 中有两种模式：一是命令模式，一是编辑。命令模式是用来输入命令行来完成工作的。而编辑模式是用来编辑文本的。在两种模式中切换可以通过 Esc 来完成。在我们完成了文本编辑以后可以这样的来退出：

：q 这是退出的命令，如果你已经做过了改动，那么就会提示错误。

：q! 这也是一个退出命令，与上一个不同的是，我已经做过了改动，但是我想放弃这些改动，从而用这样的命令。

：w 这是文件写入的命令，但是执行完这个命令后并没有退出 vi.

:wq 这个命令的意思是保存并退出

二 Vi 的基本编辑命令

在启动了 Vi 以后是在命令模式的，这是可以输入 i（insert)进入插入模式。这时会在 Vi 窗口的下端显示出你这时的状态。这时你就可以来输入文本了。在这样的情况下，你可以用 Backspace 来 Delete 来删除一个字符。用方向键来控制光标。其实在知道了这样的几个命令后就可以用来编辑文档了。但是这样的几个命令并不能保证你能高效的来完成你的工作。而事实上在命令模式下我们用键盘来移动光标而不用将手离开键盘去按方向键，这样就可以大大的增强你的编辑速度。我们可以用 h(left),j (down),k(up),l(right)这几个键来移动光标。我们可以将光标放在一个字符上同时按下 x 键来删除这个字符。我们可以将光标放在某一行同时按下 dd，这样就可以将这一行删除。当然有的时候也许我们做了一些事情，不过我我们觉得以前的要更好一些，我们想要恢复到以前的状态，在这样的情况下我们可以 u 和 U 这两个命令。u 要撤销上一次的修改，而 U 将是撤销所记录的所有的修改。而有的时候我们觉得现在的工作要好一些，我们希望撤销刚才所做的撤销工作，也就是我们要重做这些工作，这时我们可以使用 CTRL_R 命令来达到我们的目的。有时我们可以使用新增的命令，使用 a 和 A 来完成这样的工作。a 在当前光标所在的字符后面进入插入状态，而 A 是在一行的末尾进入插入状态。使用这两个命令可以方便我们进行文本的插入操作。在 vi 的编辑模式中，是将回车换行看作新的一行的开始。有时我们希望新插入一行，这时可以使用 o 和 O 这两个命令来完成。o 是在文本的下面新增一行并进入插入模式，而 O 是在文本的上一行新增一行并进入插入模式。有了这些命令，现在的我们就可以比较方便的来完成我们的文本编辑工作了。但是有时候得到在线的帮助对于我们来说显得更为重要，要得到 vi 的帮助，可以在命令的模式下输入:help,这样就可以得到 vi 的在线帮助了。要想退出帮助，可以输入退出命令，:q.为得到更明确的帮助，我们可以明确的指明我们所需要知道的内容。例如我们

想知道 x 一些更详细的内容我们可以输入:help x.我们要想得到其他的帮助，我们就可以这样来得到我们想要的帮助。在 vi 中可以使用数字和命令组合的方式得到新的命令，例如 3h,这样就可向左移动 3 个字符。同样可以使用数字和其他的移动键进行组合来达到快速移到的目的。也可以是数字和其他的命令组合形成新的命令，例如 3x 就可一次删除 3 个字符。为了我们更快速的掌握 vi 的使用，vi 本身也提供了一个学习的教程，只要你能耐心的做下来，我想使用 vi 对你来说应不再是一件难事了。进入 vi 的学习教程，可以在终端输入 vitutor.这样就可以进入 vi 的学习教程，为了得到更多的帮助信息，可以在 vi 的窗口内输入：help tutor.这样就会得到更多的关开 Tutor 的帮助信息的。

在上一节的学习中，我们只是学习一些使用 vi 进行文本编辑的基本的命令。有了这些的基本命令我们就可以完成一般的文本编辑任务。在这一节中我们要学习一些其他的一些编辑命令，这些命令将包括其他的一些光标移动命令，如何在一行中快速的查找我们想要的东西，其他的一些文本删除和更改的命令，键盘宏和特殊字符的输入。啊：）

在 vi 的编辑中，我们可以有多种的光标移动命令：

我们可以用 w 命令向前移动一个字符，用 b 命令向后移动一个字符。就像其他的 vi 命令一样，我们也可以用数字来做前缀从而组成新的命令，来快速的移动。例如 4w 就是向前移动 4 个单词，而 5b 则是向后移动 5 个单词。而我们在编辑的过程中又如何来快速的移到一行的开始或是结尾处呢？在 vi 中$和^可以来完成这样的工作。$可以使光标移到一行的结尾处，而^可以使光标移到一行的开始处。$命令可以和数字进行组合形成新的移动命令，而^也可以和数字进行组合，但是组合后组成的新的命令中数字却不起任何的作用。

在我们的文本编辑中我们就会移动光标是我们经常要做的事情，但是我们很快就会发现查找我们要找的字符串也是我们经常要做的一件事。那么如何在文本编辑中快速的查找到我们想要的字符呢？在 vi 的编辑命令有几个这样的小命令可以帮助我们来完成这样的工作：f 是向前搜索的命令。例如 fx 是向前搜索字母 x.利用 f 向前搜索的命令我们也可以快速的移动到指定的位置。而 F 是向左搜索的命令，也就是向后搜索。例如 Fx 是向后搜索字母 x。与 f 和 F 这两个命令相类似的是 t 和 T 这两个命令。t 命令类似于 f 向前搜索命令，所不同的是 t 搜索到时并不是将光标停在目标字符上，而是停在目标字符的前一个字符上。和他相反的就是这个 F 命令。当然这几个命令都可以和数字组合来形成新的命令来完成我们的工作。在搜索的工作过过程我们可以使用 ESC 来退出搜索而开始我们新的工作。

在我们的工作中常常要求我们移动到指定的行，那么我们如何来做到这一点呢？我们当然可以使用数字和方向键组合来完成。虽然这种方式不够快速，但是确实可以来实现。而在 vi 中提供了一个新的命令来完成，那就是 G。例如 3G 可以使我们快速的移到第 3 行。而 1G 则可以使我们移到文章的最顶端，而 G 则是定位到文章的最后一行。那么在 vi 的编辑中我们又如何来知道我们在第几行呢？我们可以使用：set number 来叫 vi 加上行号，这样我们就可以很容易的知道我们所在的行号了，取消行号的命令为：set nonumber。那么在没有行号的文章中我们又如何来知道我们所处在的位置呢？我们可以使用 ctrl+G 命令来知道。这个命令可以清楚到告诉我们总共有多少行，而当前我们又在多少行，以及所占的百分比等信息。在我们进行编辑的过程中我们可以使用 CTRL-U 和 CTRF-D 来进行上下的翻页。当然这样的功能也可以通过功能键来实现。

在我们的文本编辑过程中另一件我们要常做的事情就是文本的删除了。我们可以使用 dd 来删除一行，我们还可以使用 dw 来删除一个字符。删除的命令操作 d 是一个相当灵活的命令，我们可以用他来进组合来完成工作。例如 d3w 则是一次删除 3 个字符，而 3dw 则是指一次删除一个字符，而这样的操作进行 3 次，即总的是也是删掉 3 个字符。而在 d$则是删除从当前位置到本行结束的所有字符.也 d 命令相类似是的 c 命令，这是一个更改的命令，所不同的是他在删除一个字符的同时进入插入状态，这样我们就可以进行另外的文本操作了。另一个有兴趣的命令则是.命令。.命令可以使 vi 重复执行刚才执行的命令。

在我们进行文本编辑的时候，有时要用到合并行的命令，也就是将几行合并为一行，这时我们可以使用 J 命令。这个命令可以将本行和下一行合并为一行。当然，就像大多数的 Linux 命令一样，我们可以使用数字来使几行合并为一行。例如 3J 就可以将当前行下的三行（包括当前行）合并为一行。那么我们又如何时来做替换文本的工作呢？我们可以使用 r 的命令。例如 rx 就可以当前光标下的字符替换为 x。我们当然也是可以用数字来组合以形成新的命令来进行工作，例如 5rd 就是将当前光标以后的 5 个字符替换为 d。有时我们要进行大小写的转换，这时我们就可以用~命令。这个命令可以实现在大小写的转换。

在 vi 中一个比较有趣的东西便是键盘宏了，这个可以使我们实现多个命令的记录，有时这样可以高效的完成我们的工作。例如我们现在的文本是 stdio.h stdlib.h math.h 但是我们都知道在 C 语言的编辑中我们所需要的是 #include #include #include 如何来做到？如果你有足够的耐心可以一句一句的来加，但是在这里我们可以使用键盘宏来完成我们的工作，这样我们就可以体会到他的强大之处了。开始输入 qa.其中的 a 是一个宏的名字，我们可以用任何我们喜欢的字母来代替，q 是开始录制宏的命令标志。这样我们就可以开始我们的宏输入了： ^ 移到一行的开始 i#include 在一行的开始插入#include 在结束处加入> j 移到下一行 q 结束宏的录制这样当我们要使用宏时就可以输入@a，这样就可以执行这个宏了。我们还可以在执行命令前加上数字来告诉 vi 执行几次。这样我们就可以快速的完成我们的一些工作了。在 vi 的编辑中我们还可以输入一些由平常的键盘不可以输入的字符,有关这样的信息我们可以输入：help digraphs 得到更多的信息。 (注：在 vi 中我们通常所指的一行是以回车做为标志的，即只有输入回车才算是一行的结束，从而开始新的一行）附：打印特殊字符现在还弄不懂，看了那书也还是弄不出来，希望朋友们能把成功的做法分享一下

vi 学习使用笔记之三

我们在使用 vi 进行编辑文本的时候常作的一件事就是要在所编辑的文本中进行查找。如何快速的查找到我们想要的东西呢？在 vi 中我们可以使用 f，F 和 t，T 来进行向前或是向后查找。

除了这些命令，我们还可以使用其他的一些命令来快速高效的完成我们的工作。在 vi 的编辑操作中，我们可以使用/string 命令来查找字符串 string，打下回车后，光标就能跳到正确的地方。在这个命令中/后的字符是我们想要查打的字符，而回车键则表明了命令的结束。

但是有时我们所想要查找的字符具有特殊的意义，例如.*[]^%?$~等等，那么我们又如何来查找这些具有特殊意义的字符呢？这时我们可以使用放在所要查找的字符前，这样再使用/来查找就可以正确的查找了。

有时我们在进行查找操作时想要查找的内容并不仅在一处，我们想要做的是在整个文章中进行查找。那么我们又如何来进行我们刚才的查找命令呢？我们可以这样的来做：/。这样我们就可以继续我们刚才的查找操作了。在这里回车是命令结束的标志。我们还可以使用 n 命令来继续刚才的查找命令。这两个命令都能达到同样的效果，但是显然用 n 可以有更少的键盘操作。偷懒嘛：）

在 vi 中他还具有记录查找命令历史的作用，这样我们就不用输入刚才输入的查找命令了，而是

只需要在他所记录的查找命令进行一下查找就可以了。例如你刚才做过的三次查找分别是：/one,/two,/three.而现在输入/，然后按方向键的上或是下我们就看到刚才查找的内容显示在 vi 的下面，这时只要打下回车我们就会找到我们要找的内容了。当然在 vi 中还在一些其他的查找选项，例如我们可选择高亮的显示查找的文本，命令为：:set hlsearch,关闭高亮显示的命令为：:set nohlsearch.如果也想同时关掉刚才高亮显示的结果，可以用这样的命令：:nohlsearch。在我们进行查找的选项中，我们还可以有这样的命令：:set incsearch。在我们打开这个选项以后，我们在进行查找时就会显示出不断匹配的过程。例如你想查找的内容是 include，在打开这个选项后你可以输入/i，光标定位在 i 上，再继续输入 n 光标定位在 in 上，如此直到查打到我们所要求的。关闭这个选项的命令为：:set noincsearch。一般来说我们在进行查找时总是在向前查找，那么又如何来向后查找呢？我们可以使用？命令。这个命令就是向后查找的命令。而还有一个命令 N 是逆向查找的命令，他可以实现立即反向查找。（注：在查找的时候我们还可以用一些其他的表达式来进行查找，例如/^string 是在开头进行查找，而/string$是在一行的末尾进行查找。不过看书我的理解是这样的，不过总是试验不成。而/c.m 则是查找所有第一个字母为 c，而第三个字母为 m 的字串，由此可以实现一些查找的匹配)

我想我们在接触了 vi 以前一定会用一些其他的编辑器，在那些的编辑器里复制，剪切和粘贴都是最平常的操作，而在 vi 中这些操作也是同样的存在的。在 vi 编辑器有一个注册的概念（concept of register),正是这个概念使我们可以时行多次的剪切和粘贴等的操作。在一般的编辑器中我们被限制只有一个剪切板可以用，而在 vi 中我们最多时可以有 26 个剪切板可以来使用，这样就会大大的提高我们的完成工作的效率。而在 vi 中更是有一个相当强大的功能那就是他可以同时处理多个文件。如此强大的功能我们又来如何的操作呢？
我想我们在接触了 vi 以前一定会用一些其他的编辑器，在那些的编辑器里复制，剪切和粘贴都是最平常的操作，而在 vi 中这些操作也是同样的存在的。在 vi 编辑器有一个注册的概念（concept of register),正是这个概念使我们可以时行多次的剪切和粘贴等的操作。在一般的编辑器中我们被限制只有一个剪切板可以用，而在 vi 中我们最多时可以有 26 个剪切板可以来使用，这样就会大大的提高我们的完成工作的效率。而在 vi 中更是有一个相当强大的功能那就是他可以同时处理多个文件。如此强大的功能我们又来如何的操作呢？在 vi 的编辑中我们可以使用 d 或是 x 来删除文本，但是经过这样的操作所删除掉的文本并没有被 vi 所丢弃，而是被保存起来。我们可以使用 p 命令来粘贴刚刚所删掉的内容。下面我们可以来试一下看一下他是如何工作的。我们可以在 vi 中随意的输入几行，然后我们移动到其中的一行，用 dd 命令来删掉其中的一行，这时我们就不会再在 vi 中看见他。如何叫他再回来？我们可以将光标移到任意的地方，然后用 p 命令，我们就会看到刚才被删除掉的内容又回来了。同样我们使用 x 命令来删除的东西也可以这样的粘贴回来。所不同的就是我们用 dd 来删除一行再用 p 命令时是在当前光标的下一行粘贴，而删除一个单词再用 p 命令来粘贴时是在当光标处粘贴。有了这样的命令有时我们就可以来处理我们输入错误的单词了。例如我们不小心将 the 输入成了 teh，这时我们可以将光标移到 e 的位置，用 x 命令删掉，再用 p 命令粘贴回来，这时我们就会发现现在的输入正是我们想要的输入了。p 命令可以在一个文件中使用几次，每一次都可以将刚删除的内容重新粘贴在我们所希望的地方。当然这个命令也可以使用数字做前缀来明确的指出所要执行的次数。在使用 vi 进行文本编辑的过程中我们还可以对某一行进行标记，做法为我们将光标移到某一行，用命令 ma 来进行标记。在这里 m 是标记的命令，a 是我们对这一行所做的标记的名称，当然我们也可以使用其他的标记名称，必竟我们是有 26 个字母可以用的嘛：）。在做了这样的标记以后我们可以快速的移到被标记的地方，‘a 就可以使我们快速的移到刚才我们所做标记的地方。这里’是单引号，这时我们就会移到被做标记那一行的行首。我们也可以使用`a 来移到我们所做标记的地方，这里`是反引号，也就是数字键 1 左边的那一个，不要弄错了噢：），这时我们就会移到所做标记时光标所在的位置，

可以说是更精确啊。这也是这两个命令的不同之处。在进行文本编辑时我们可以列出当前所做的所有的标记。命令为 ：:marks。这时 vi 就会列出当前的所有的标记。当然如果我们将做了标记的那一行删除的话所做的标记也就不存了。我们用标记不仅可以快速的移到做了标记的行，而且还可以用标记来删除掉一行，例如我们在某一行用 ma 做了标记，然后我们移到这一行的底部，这样我们就可以用在 d‘a 来删掉这一行。可以说这也是标记命令提供给我们的方便吧。在 vi 中还有一个命令可以提供复制的操作，那就是 y 命令。yy 命令可以复制一行，然后我们可以用 p 命令来粘贴。这时我们也可用标记命令来组合使用，我们可以在一行用 ma 标记此行，然后将光标移到这一行的底部，用 y’a 命令就可以来复制这一行了。然后到我们所希望的地方用 p 命令来粘贴。我们也可以使用 Y 命令来复制一行，或是用数字做前缀来明确的指明复制几行，然后用 p 命令粘贴到我们所希望的地方。在 vi 中还有一个比较有趣的命令便是！命令。这个命令告诉 vi 编辑器我们要来运行一个系统命令，这时 vi 就会将以后的输入做为一个命令来运行，运行的结果也就是输出将替代当前光标所在的行的内容。例如我们在 Linux 中知道 sort 是一个排序的命令，他是将一个文件或是一些输入进行排序后输出到屏幕或是其他的文件。那么我们想对 vi 中的内容进行排序来如何做呢？这时我们可以将光标放在文本的开头的一行,输入！10G，这时 vi 就知道要到 10 行，也就是我们要操作的内容是第 1 行到第 10 行，这时在 vi 的下端就会显示出!，这时我们就可以输入命令了，sort。也就是我们要输入的完整的命令应为：!10Gsort。这样回车以后 vi 就会对文本中的 10 行进行操作，将操作的结果替换掉现在 vi 中的文本来显示。而!!是在一行运行命令，并且输入的结果为当前行的内容。例如我们输入!!ls，就会将 ls 的执行结果显示在 vi 中，并且是当前的内容，如果此行原先有内容将会被替换掉。如果我们完成一个文件的编辑而要开始一个新的编辑时我们最平常的做法就是退出当前的文件而重启 vi 开始一个新的编辑。事实我们可以直接在 vi 中输入：vi file 而开始一个新文件的编辑。如果当前的文件你没有保存，vi 会给出警告的信息，这时你可以输入：write 来保存当前的文件。你也可以用：vi！file 强制开始一个新文件的编辑。与 vi 相类似的一个命令是：view，所不同的是他以只读的方式打开一个文件，这时 vi 会给出警告信息，但是你也可以进行修改，只是你不能进行保存，如果你要保存，vi 就会给出提示。当然在这样的情况下你可以用命令：write!来强制保存。我们还可以使用 vi 来编辑多个文件。我们可以在终端输入 vi file1 file2 file3，这样我们就可以来编辑多个文件了，在默认的情况下 vi 来显示第一个文件，如果要切换到下一个文件我们可以输入：next，这样我们就可以切换到第二个文件了。如果你没有保存，vi 会给出提示信息，这时也就不可能切换到第二个文件了。这时我们可以输入:write 进行保存然后再切换到第二个文件，或者是我们输入:write:next.来保存后再切换到第二个文件。或者是我们可以用：wnext 来简写这个命令。当然我们也可以用命令:next!来强制切换到第二个文件。当然这样你所做的改动也就会丢失掉。为了避免这样的情况，我们可以打开 vi 的自动保存功能。:set autowrite.这样我们就不会为没有保存而收到提示信息了。关闭自动保存的命令为：:set noautowrite.当然 next 命令也可以用数字做前缀来指明所要执行的次数。如何来确定我们在编辑哪一个文件呢？我们可以用这样的命令来明确我们所编辑的文件：:args.这个命令会显示出我们所打开的文件，同时会显示我们正在编辑的文件。如果我们想要回到上一个文件我们可以用这样的命令：:previous 或是:Next.如果你要保存当前的文件并切换到前一个文件可以用这样的命令：:wprevious 或是:wNext.如果我们要编辑第一个文件我们可以用：first 或是:rewind 来快速的切换到第一个文件，同理如果我们要编辑最后一个文件我们可以用:last 来快速切换。如果我们在一个文件中进行一些改动，再切换到另一个文件中进编辑，这时我们就可以用 CTRL_^来切换这两个文件。

在使用 vi 进行文本编辑的时候我们也可以打开多个窗口进行编辑。这也正是 vi 编辑器的强大这处。那么我们如何来打开多个窗口进行文本编辑呢？我们又如何在多个文本窗口中进行切换呢？如何来控制窗口的大小呢？在 vi 编辑器还有一个缓冲区的概念，利用缓冲区我们可以进行多文

本的编辑。
在使用 vi 进行文本编辑的时候我们也可以打开多个窗口进行编辑。这也正是 vi 编辑器的强大这处。那么我们如何来打开多个窗口进行文本编辑呢？我们又如何在多个文本窗口中进行切换呢？如何来控制窗口的大小呢？在 vi 编辑器还有一个缓冲区的概念，利用缓冲区我们可以进行多文本的编辑。打开一个窗口最简单的方法就是下面的命令：:split。输入这样的命令后 vi 就会将当前的窗口平分为两个。并且在这两个窗口中显示的是同一篇文章。如果你在其中的一个窗口进行文编辑，那么另一个窗口也会同步的显示出你所做的工作。如何在两个窗口中进行切换呢？在 gvim 中要方便得多，只要用鼠标就可以进行窗口的切换。而在 vim 中则需要用命令来实现。CTRL_Ww 这个命令可以实现在两个文本窗口中进行切换。切换窗口还有另外的命令：CTRL_Wj 可以将光标定位在下一个窗口中，而 CTRL_Wk 可以将光标定位在上一个窗口中。如果想关闭 一个窗口可以用命令 ZZ 或是:q.当然了 CTRL_Wc 也可以做到同样的事情。我们打开一个窗口一般并不是要在两个窗口中显示同一个文件，我们常常需要的是在两个窗口中显示两个文件来加快文件编辑的工作。如何在新窗口中打开一个文件来进行编辑呢？我们可以用这样的命令：:split file.这样我们就可以在另一个窗口中打开文件 file 开始我们的编辑工作了：)。那么我们如何来控制窗口的大小呢？我们在输入 split 命令时可以带上一个参数，这个参数 也就指定了打开的窗口的大小。例如我们可以这样的输入：:3 split file。这样我们就可 以在一个新窗口中打开文件 file，而这个窗口的大小有三行。当然我们也可以将这个命令中的空格去掉，也就是写成：:3split file 这样也可以达到同样的作用。与 split 命令相类似的一个命令就是:new 命令。所不同的就是 split 命令打开一个新窗口，但是在两个窗口中显示同一个文件，而 new 命令则是打开一个新窗口开始一个新文件的编辑.我们还可以打开一个新窗口，在这个窗口中打开一个文件来读。命令为:sview。这个命令是:split 和:view 这两个命令的组合。在这样的多文本窗口中进行操作时我们常做一个工作就是要改变窗口的大小。可以使我们来改变窗口大小的命令为：CTRL_W+这个命令增大窗口，默认增量为 1 CTRL_W-这个命令减小窗口，默认值为 1 CTRL_W=这个命令可以将几个窗口的大小变得相等。另外还有一个命令 countCTRL_W_可以使得当前窗口变得 count 这样的高。如果没有指定 count 将会使得当前窗口变得尽可能的最大。 :buffers 这个命令就会列出当前的编辑中所有的缓冲区状态。在这个状态列表中，前面的数字就是缓冲区的数字标记，第二个标记就是缓冲区当前的状态，而后一个则表明与空上缓冲区所关联的文件名。他的一些状态是这样的： - 非活动的缓冲区（Inactive Buffer） h 隐藏的缓冲区（Buffer is hidden) % 当前的缓冲区（current buffer) # 交换缓冲区（Alternate buffer) + 文件已经被修改如果我们要选择一个缓冲区我们可以用这样的命令：:buffer number number 就是缓冲区状态列表中所显示出来的数字。我们也可以用文件名字来选择缓冲区： :buffer file 关于缓冲区有一些有用的命令可以快速的完成我们的工作：我们可以用下面的命令来分割当前的窗口开始编辑一个缓冲区：：sbuffer number 如果我们指明了数字，那么当前窗口就会显示数字所标记的那缓冲区中的内容，而如果没有指明数字，那么就会利用当前的缓冲区。当然这个命令我们也可以用文件名来做为参数进行操作。对于缓冲区还有一些其他相关的命令： :bnext 到下一个缓冲区 :count bnext 到下一个缓冲区执行 count 次 :count sbnext 命令:split 和 :count bnext 组合形成的命令 :count bprevious 到前一个缓冲区 count 次 :count sbprevious :count bNext 与:bprevious 作用相同 :blast 到最后一个缓冲区 :brewind 到第一个缓冲区

在现在的 vi 版本，或者是说是在 vim 中，与其先辈区分开来的一个特征就是现在的版本中有一个可视模式。这种可视模式可以使你加亮一个文本块然后整体进行命令操作。例如你可以用高亮显示出一个文本块，然后用 d 命令来删除这个文本块。这种可视模式与其他的编辑相比的一个好处就是你可以在做改动以前看到你的操作对于所编辑的文本产生的影响。那么我们如何为用这种有趣的可视化模式来进行我们的文本编辑工作呢？

vi 编辑器学习使用之六在现在的 vi 版本，或者是说是在 vim 中，与其先辈区分开来的一个特征就是现在的版本中有一个可视模式。这种可视模式可以使你加亮一个文本块然后整体进行命令操作。例如你可以用高亮显示出一个文本块，然后用 d 命令来删除这个文本块。这种可视模式与其他的编辑相比的一个好处就是你可以在做改动以前看到你的操作对于所编辑的文本产生的影响。那么我们如何为用这种有趣的可视化模式来进行我们的文本编辑工作呢？要进入可视化模式，我们可以输主命令 v，这样在 vi 的状态行就会显示我们已经进行可视化模式了。在这样的模式下，我们移动光标，那么在光标起始的位置和光标现在的位置之间的文本都会以高亮显示的。这时我们可以对这些高亮显示的文本整体进行命令操作，例如这时我们可以输入 d 命令，那么这些高亮显示的文本就会被删除掉。一般来说可以化模式可以具体的分为三种可视化模式，一种就是我们刚才用 v 命令进入的所谓的字符式可视模式（character-by-character visual mode)。在这种模式下，我们在进行文本选择以高亮显示时是以字符为单位的,我们可以一个字符字符的来选择。而第二种就是所谓的行可视化模式（linewise visual mode)，这时我们可以输入 V 命令来进入这种可视化模式。这时我们所进行的操作是在以行为单位来进行的。还有一个命令可以使我们进入可视化模式，这就是 CTRL_v,这就是我们所要说到第三种可视化模式，他可以使一个矩形内的文本高亮显示，然后以这些文本为整体进行编辑操作。在可视模式下我们也可以得到一些相关的帮助信息。当然在输入命令时要在所希望帮助的名称前有 v_做为前缀。例如我们想得到一些关于在可视模式下进行删除操作的命令，我们可以这样的来得到帮助信息：:help v_d 当我们要退出可视化模式时，我们可以按 ESC 键退出。当然 CTRL_c 也可达到同样的作用。与 ESC 作用相同的还有 CTRL_和 CTRL_N. 在可视化模式下我们可以进行各种各样的编辑操作。如 d 命令可以删除那些高亮显示的文本内容，而 D 命令只是来删除一行，哪怕这一行中只有部分文本是高亮显示的。与 d 和 D 命令类似的是复制 y 和 Y 命令。y 命令是将高亮显示的文本复制下来，而 Y 只是将一行文本复制下来。c 命令可以将以高亮显示的文本删除掉然后进入编辑模式，而 C 命令的作用与其相类似，所不同的只是 C 命令只是删除一行文本。我们还可以利用可视化模式将几行文本合并为一行。J 命令可以将高亮显示的文本内容合并为一行，同时以空格来区分各行，而如果我们不希望看到这些空格，我们可以使用 gJ 命令。我们在可视模式下进行文本编辑时也可以进行可视化模式的选择和切换。你可以在任何情况下进行这种切换，而切换的做法只是简单的输入那种可视化的命令操作。例如现在我们在字符模式的可视化模式下进行操作，而现在我们想切换到块模式的可视化模式，这时我们只是要简单的输入 CTRL_v 可以了。当然我们也可以先用 ESC 来关闭当前的可视化模式，然后再选择我们所希望的可视化模式. 对于程序员来说似乎这种可视化模式会有更大的用处，我们可以用一种可视化模式来高亮显示文本，然后用>命令来进行文本的缩进，我们还可以用，这个命令中 I 是插入文本的命令，string 是我们要插的文本，而 Esc 则是结束插入操作的命令。这时就会看到我们所输入的文本显示在文本块的左侧，也就是开头的地方，当我们结束插入操作时我们就会惊奇的发现我们所输入的文本也会同时出现我们所定义的文本块内所包含的其他行的开头部分。同样当我们用 c 命令来操作时我们也会发现类似的情况。当然了，在用 c 这个命令进行操作时你所输入的文本应不超过一行，否则的话将会只有第一行的文本会被改动。而 C 命令也会有相类似的情况。我们也可以类似的来得到一些关于块操作的命令帮助：例如：:help v_b_r

vi 编辑器学习使用之七

vi 是一个强大的编辑器，他不仅可以用来处理我们平时的文本工作，他还可以用来写程序文件。在 vi 中有许多命令可以方便的完成我们的程序处理工作。在用 vi 进行程序处理时，vi 充分的显示出来了他的强大之处，他可以实现在语法加亮显示，实现自动缩进，实现括号匹配，还可以在程序中实现查找和跳转。

vi 编辑器学习使用之七

vi 是一个强大的编辑器，他不仅可以用来处理我们平时的文本工作，他还可以用来写程序文件。在 vi 中有许多命令可以方便的完成我们的程序处理工作。在用 vi 进行程序处理时，vi 充分的显示出来了他的强大之处，他可以实现在语法加亮显示，实现自动缩进，实现括号匹配，还可以在程序中实现查找和跳转。

我们可以用这样的命令在 vi 中打开语法加亮显示的功能：:syntax on。这样以后我们在输入的字符中，vi 就会自动的识别出关键字，字符串以及其他的一些语法元素，并以不同的颜色来显示出来。这对于程序员来说是一个巨大的帮助。当然你可以自定义语法加亮显示的颜色。

一般情况下，vi 的这种语法加亮显示的功能可以工作的很好，但是有时我们也遇到一些小小的麻烦。也许我们都遇到过背景是白色而字体是浅黄色的情况，在这样的情况下是非常难读的。vi 编辑器有两种语法加亮的办法，一种是当背景为浅色时用，而另一种是当背景为深色时用的。当我们启动 vi 时，他会检测我们所使用的端是哪一种背景颜色，是浅色还是深色，然后再应用语法加亮的颜色方案。当然了，有的时候 vi 也是可以检测出错的。我们可以用这样的命令来知道我们的背景的情况：:set background?。这样 vi 就会在底端给出我们具体的显示。如果 vi 检测的并不正确，我们可以用这样的命令来为他赋下正确的值：:set background=light 或是:set background=dark.当然我们要清楚的知道，这样的操作要在打开语法加亮的命令之前来执行。vi 实现语法加亮显示的功能是通文件的扩展名来文件的类型从而实现功能的。但是有时我们在编辑一个 C 程序文件时并没有使用传统的扩展名，那么我们如何来告诉 vi 我们正在编辑的文件的类型呢？解决的办法就是用 filetype 这个选项。例如我们要告诉 vi 我们正在编辑的是一个 C 程序文件我们可以这样的来做：:set filetype=c。这样以后 vi 就会知道我们正在编辑的是一个 C 程序文件并正确的加亮语法显示。

除了语法加亮显示的功能以外，vi 还提供了缩进的功能。命令<<将使当前行向左移动一个移位宽度，而命令>>将使当前向右移动一个移位宽度。这个所谓的是移位宽度具体是多少呢？在 vi 中默认的是八个空格的宽度。然而平时的经验可以表明当缩进是四个空格宽度时最有利于程序的阅读。那么我们如何将这个移动宽度定为四个空格的长度呢？这时我们就可以用下面的命令来做到：:set shiftwidth=4.而这时的命令仅是对当前的一行有效，我们也可以像其他的 vi 命令一样的在命令前用数字做为前缀还指定命令作用的范围。如 5<<就将缩进五行。

在 vi 中还有许多自动进行缩进的选项，一般有以下的几种：C 缩进（cindent)：这是 C 语言的缩进形式，采用这样的缩进方式的程序语言有：C，C++,Java 等。当采用这种缩进格式时，vi 就会自动的采用标准的 C 语言形式。还有一种形式是 smartindent:在这种缩进模式中，每一行都和前一行有相同的缩进量，同时这种缩进形式能正确的识别出花括号，当遇到右花括号（}），则取消了缩进形式。另外的一种缩进形式便是所谓的自动缩进（autoindent):在这种缩进形式中，新增加的行和前一行有相同的缩进形式。vi 编辑器可以很好的识别出 C,C++,Java 以及其他的一些结构化程序设计语言，并且能用 C 语言的缩进格式来很好的处理程序的缩进结构。我们可以用这样的命令来打开 C 语言形式的缩进结构：:set cindent.这样以后 vi 编辑器就会用 C 语言的缩进形式来正确的处理程序文件。一般而言 C 缩进结构可以满中绝大多数人的需要，当然了不同的人有不同的编程风格，如果你不喜欢这样的缩进结构，你可以自己定义自己的缩进形式。也许我们做程序设计的人并不想每一次想要编辑 C 程序文件时都要用命令:set cindent 来打开 C 缩进形式，为了解决这样的问题，我们可以改写 vi 的配置文件来使 vi 能够自动的完成这样的工作。我们在.vimrc（UNIX/LINUX）或是_vimrc（WINDOWS）中加入下面的几句：

```
:filetype on
:autocmd FileType c,cpp:set cindent
```

第一行是打开 vi 文件类型识别功能，第二行是如果所识别的文件类型为 C 或是 C++文件那么便打 C 缩进形式。

在 vi 中除了 C 缩进形式以外我们还有 smartindent 缩进形式可以来用。在 smartindent 这种缩进模式中能够正确的识别出{和}。同时增加了识 C 语言关键字的功能。如果一行是以#开头的，那么这种格式将会被特殊对待而不采用缩进格式。这种缩进格式不如 cindent，但是却强于 autoindent。另外的一种缩进形式便函是 autoindent。在其他的一些结构化程序设计语言如 Pascal，Per 或是 Python 语言中，我们所希望的是新的一行能和上一行有相同的缩进形式，这时我们便可以打开 autoindent 缩进形式，:set autoindent。这样就可以很好的来满足我们的要求了。

vi 不仅有这些功能，而且还有更多的功能来快速的处理程序文件。其中一个便是可以快速的在文件中定位变量及宏定义等。其中一些常用到的命令有：

[CTRL_I/]CTRL_I　　在本文件中以及在由#include 命令包含进来的文件中进行查找光标下面的文字

gd/gD　　查找变量的定义

[CTRL_D/]CTRL_D　　跳到宏定义的地方

[d/]d/[D/]D　　显示宏定义

这几个命令中有一些需要说明的地方：[CTRL_I/]CTRL_I 命令用来查找光标下面的文字，查找的范围不仅是在本文件中进行查找，而且还要查找由#include 所包含进来的文件中进条查找。变显查找的命令 gd 和 gD 有时并不能完美的完我们想要做工作。这是因为 vi 在理解 C 和 C++的语法方面有一些小的限制。但是大多数的情况下这几个命令还是可以很好的来完成我们工作的。[d/命令可以显示以当前光标下的内容为为名字的第一个宏定义，]d 也有着同样的作用，所不同的只是后者是从当前光标处查找下一个宏定义。当然这两个命令也可以在由#inlcude 所包含进来的文件中进行查找。[D/]D 命令可以用来列出文件中的宏定义。这两个命令有着同样的作用，所不同的只是前者是列出当前光标以后的宏定义，而后者是从当前光标处的下一个宏开始列出文件中的宏义。(注：此处由此书看来似乎是这样的，但是自己亲自来做时并没有出现这样效果）

我们在编写程序时常常要做的一件事便是括号的匹配，我们在 vi 中可以用%命令来确定匹配的括号。例如我们将光标定位在其中一个括号处，然后执行命令%，光标就会定位在与其相匹配的括号处。有时我们在编写了一段程序后却想着要将这一段程序进行缩进，这时我们可以用这样的命令来做到：将光标定位在第一个或是最后一个包含着所要缩进的程序块的括号处，然后执行下面的命令：>%，这样就可以将这个程序右缩进一段距离。但是我们发现执行这个命令后花括号也一样的进行了缩进，有时我们这并不是我们想要的，我们想要的是仅括号内的文本进行缩进。这时我们又应如何来做呢？这时我们可以用这样的命令：>i{.这个命令是说仅括号内的文本进行缩进。我们还可以用可视化的模式来缩进文本块，具体的做法是这样的：

1 将光标定位在左括号或是右括号处。

2 进入可视化模式：v

3 选中括号内的文本：i}

4 缩进：>

当我们用 vi 来编辑我们的程序文件时，他提供给了我们定位程序函数的功能。这对于我们想要理解一个程序是非常有帮助的。vi 所定位的函数被包含在一个由 ctags 的程序建立的文件当中。要建立一个名为 tags 的这样的文件，我们可以输入这样的命令来完成：$ctags *.c

这样以后当我们在 vi 中来编辑我们的程序文件时我们就可以任意的跳转到我们想要去的函数处，当然这得要求我们想要到函数存在。我们可以用这样的命令来到达我们要到的地方：

:tag function

这个命令甚至可以查找得到在其他文件中的函数。

在完成函数跳转的功能命令中还有几个强大而有趣的命令。CTRL_]命令跳转到以光标下的字符串为函数名的函数处。这样就大大的方便了我们在 C 程序查找函数的需要。例如你正在看一段名为 write_block 的函数程序，而在这个程序中调用了函数 write_line。这个被调用的函数具体是

来做什么的呢？我们可以将光标定位在这个函数上，然用执行命令 CTRL_]。这样 vi 就会马上跳转到这个函数的定义处。我们就可以清楚的看到这个函数的实现的方法。而在这个函数中又调用了函数 write_char，这时我们可以用同样的方法来查看这个函数的定义过程。命令:tags 可以列出我们已经访问过的函数名称。但是我们在走了一段路以后想回去又该怎么来做呢？命令 CTRL_T 可以使我们回到上一个访问的函数处。我们为了在函数跳转的命令，我们要用命令 ctags 建立这文件来存放在 C 程序中所出现在的函数名，这时 vi 要建立一个栈来存入函数名。我们用命令 CTRL_T 回到上一个访问的函数处，这时我们还可以用命令:tag 来向前走一步，即回到前一个访问的函数处。我们也还可以像其他大多数的 vi 命令一样在此命令之前用数字来做前缀，指明要执行的次数。我们还可以用这样的命令：:tag function。这样我们就可以来到指定的函数处。

当我们用命令:tag 时是将当前窗口中的内容由函数的内容所代替了。我们也可以打开一个新窗口来显示函数的内容。这时我们就要用到下面的命令：:stag tag。这个命令是要打开一个新窗口来显示 tag 指定的函数内容。我们也可以用命令 CTRL_W_]来打开一个新窗口显示以光标下的字符串为函数名的函数内容。我们也可以在这个命令前加下数字 count 做为前缀，这打开的窗口就是 count 这样的高度。

我想我们常会有这样的情况，那就是我们想要跳到一个函数处却记清他的名字究竟是什么？这是我们要怎么办呢？在 vi 中有一个很好的解决办法，那就是我们可以用:tag /name 来找到你想要的内容。例如在我们上面所举过的例子中曾提到了函数 write_line。但是现在我们记不得他的全名了，只是记得好像有一个 line。这时我们如何来找到呢？我们可以用这样的命令来做：:tag /line。这样 vi 就会将我们带到 write_line 定义的地方。我们还可以借助于一些表达式来精确的定位我们要去的地方。例子如我们似乎记得有一个函数是以 read 开头的，这时我们可以这样的来查找：:tag /^read。这个是说 read 所要查找的内容是以 read 开头的。或者说我们不能十分清楚的记得一函数的名称是 DoFile,do_file 还是 Do_File。这时我们也可以这样的来查找：:tag /DoFile|do_file|Do_File。或者是我们可以写成这样的表达式：:tag /[Dd|o_=[Ff|ile。这样我们就可以找到我们想要的内容了。一般情况下我们用这样的命令来查找时并不能精确的得到我们想要的东西，而是得到许多匹配的选项。而我们可以用命令:tselect 来列出所有这些合要求的内容。这个列表一般由这样的一些内容组成：

第一列的数字是这些标记（tag）的编号，第二列这些内容的优先级，他一般包含三个字母：F 完全匹配，如果没有则是忽略的情况；S 静态标记，如果没有，则是全局标记；C 则是说是这个标在当前的文件中。在执行完这个命令后，:tselect 会给我们一个机会，我们可以选择到标号为哪一个的内容处去，或者是选择离开。命令 g］与命令:tselect 相类似，只是他是以光标下的内容为查找内容的。命令:tjump 的作用与:tselect 命令相同，所不同的只是当执行的结果只有一项内容时，则自动的选中这个结果。命令 gCTRL_]与此相同，只是他是当前光标下的内容为查找对象的。其他的一些相关的命令如下：

:count tnext 到下一个标记处

:count tprevious 到上一个标记处

:cout tNext 到上一个标记处

:cout trewind 到第一个标记处

:cout tlast 到最后一个标记处

命令:stselect 与:tselect 命令相同，所不同的只是前者将打开一个新窗口来显示执行的结果。与其类似的命令还有:stjump.

当我们在编写 makefile 文件时我们所需要的缩进是一个 tab，而不是 8 个空格。这样的区别是很难在屏幕上看出来的。这时我们可以用这样的命令：:set list，这样以后 tab 则显示为^I,同时在每一行的末尾显示$，这样我们就会很好的来区分这些细小的分别了，同时还有助我们来检查我们程序的正确性。当然我们也可以选择用其他的字符来显示，通过 listchars 选项我们可以做到这一

点。如果我们设置了 expandtab 选项，那么我们输入一个 tab 时，vi 实际上插入的空格。这对于我们编写 makefile 文件时是很不方便的。如果我们想真正的输入一个 tab，而不是插入空格，这时我们可以输入 CTRL_V<Tab>，这时的 CTRL_V 告诉 vi 不要忽略以后的字符。有时在一个 makefile 文件中包含许多的文件，我们可以对这些文件进行排序，具体的做法如下：

1 将光标放在文件列表的起始点

2 用命令标记这个位置：ma

3 到这个列表的底部

4 执行命令排序：!'a sort

我们也可以在可视模式下来排序：

1 移动到要排序的文本的顶部

2 进入可视化模式：V

3 移动到文本的底部

4 执行命令：!sort

vi 编辑器还可以允许我们在 vi 中执行编译程序的 make 命令，其命令为:make，这时就会来编译程序，同时会显示出错误来。这时我们就可以移到到出错的地方来修正我们的错误，然后再重新编译，直到成功。如果我们的程序编译有错误，vi 就会显示出错误的提示信息和错误的地方，同时 vi 会自动到第一个出现错误的地方。我们在修改完错误以后，可以用命令:cnext 来到下一个出错误的地方继续我们的工作。命令:cprevious 和命令:cNext 可以回到上一个错误的地方。命令:clast 可以到最后一个出错的地方，而命令:crewind 可以到第一个出现错误的地方。而命令：cnfile 可以到下一文件第一个出错的地方。如果我们此时忘记了出现在的错误是什么，我们可以用这样的命令来显示出错信息：:cc，如果我们想看到一个错误的列表，我们可以用这样的命令来完成：:clist。我们还可以用命令来显示一定范围内的错误。如：

:clist3,5　　显示第三行到第五行的错误

:clsit,5　　显示第一行到第五行的错误

:clsist5,　　显示第五行到最后一行的错误

如果我们已经运行 make 命令并且生成我们自己的错误信息的文件，我们可以用这样的命令来告诉 vi 这些情况：:cfile error-file。error-file 是 make 或是编译输出的文件。如果我们没有 error-file 文件，那么我们可以使用 errorfile 这个选项在。退出错误的状态可以用命令：:cquit.这些命令对于我们在使用一个集成的开发环境时会显得更有用一些.

errorfile 选项会建立一个默认的文件，这个文件会被命令:clist 和-q 命令行选项所使用，但是这个文件并不会成为:make 命令的输出文件。如果我们想要建立自己的默认错误文件可以使用下面的命令：:set errorfile=error.list

:grep 与:make 相类似，他是执行外部的命令程序 grep 并显示输出。例如我们要在程序文件中查找变量 ground_point 我们可以使用下面的命令：

:grep -w ground_point *.c

-w 是告诉 grep 程序包仅查找单词 ground_point，而不是任意匹配的结果。与前面的命令相类似，:cnext，:cprevious,:cc 等命令可以在这个匹配的列表中进行移动，:crewind,:clast 分别移到到列表中的第一个和最后一个。:cnfile 到下一个文件是的第一个。

vi 编辑器可以很好的完成一些我们要做的一些重复性的工作。例如我们可以在 vi 中设置缩写，这样当我们在输入一个单词时只需输入其中的一部分，而要 vi 来完其余的部分。当然了，我们可以将我们喜欢的一些设置写入 vi 的配置文件，这样就不要我们每一次都要通过命令来完成了。除了这个功能以外，我们还可以在 vi 中定义我们自己的功能按键，而不会影响到系统中的功能

键。这样的功能是不是很有趣呢？

我们可以在 vi 中用一个缩写的字符来代替一个单词，然后在我们想要输入这个单词时只要输入这个缩写的字符就可输入我们想要的单词了。我们如何来做到这些呢？这时我们要用到的 vi 命令是:abbreviate.例如我们可以用 ad 来 vi 编辑器可以很好的完成一些我们要做的一些重复性的工作。例如我们可以在 vi 中设置缩写，这样当我们在输入一个单词时只需输入其中的一部分，而要 vi 来完其余的部分。当然了，我们可以将我们喜欢的一些设置写入 vi 的配置文件，这样就不要我们每一次都要通过命令来完成了。除了这个功能以外，我们还可以在 vi 中定义我们自己的功能按键，而不会影响到系统中的功能键。这样的功能是不是很有趣呢？
我们可以在 vi 中用一个缩写的字符来代替一个单词，然后在我们想要输入这个单词时只要输入这个缩写的字符就可输入我们想要的单词了。我们如何来做到这些呢？这时我们要用到的 vi 命令是:abbreviate.例如我们可以用 ad 来代替 advertisement。这样我们在想输入 advertisement 的时候只要输入 ad 然后打一下空格或是 tab 就可以输入 advertisement 了。具体的做法是这样的：
:abbreviate ad advertisement 这个命令就是在告诉 vi 我们设置 advertisement 的缩写为 ad，这样我们在要输入 advertisement 的时候只要输入 ad 就可了，剩下的工作 vi 会为我们完成的。当然了这个命令也可以为多个单词设置缩写。例如我们可以设置 Jack Berry 的缩写为 JB。命令为：:abbreviate JB Jack Berry 这样我们在输入了 JB 以后打下空格或是 Tab，vi 就会自动的用我们设置的单词来替换 JB。对于程序员来说这样命令为我们提供了巨大的便利。例如我们可以做出这样的设置：
:abbreviate #b /**********
:abbreviate #e **********/
这个缩写可以在加快我们添加注释的速度。我们在编写程序文件时常有这样的习惯，那就是在程序的开头总是要加上一个注释块，来标明我们此程序的目的等。在这个缩写中有一点要我们注意的地方，那就是我们希望在写程序的注释块时下一行要和上一行对齐，要做到这一点就要求第二行的注释开头的两个字母要是空格，但是在这个缩写命令中会忽略到空格的作用，因而我们在写这个缩写时在开头写上，这样就会满足我们的要求了。也许有时我们会在一个文件中设置了多个缩写，我们可以命令:abbreviate 来列出我们这个文件中所有的缩写的设置。
另一个比较有趣和强大的命令就是:map 命令，这个命令可以使得我们将键盘上的一个按键与 vi 中的命令绑定在一起。例如我们现在将一个单词用花括号括起来，例如我们要将 amount 变成{amount}的形式，这时我们就可以这样的来应用这个命令：
:map i{ea}
在这个命令中:map 是 vi 中的命令，而 F5 则是说将下面的命令与 F5 键绑定，后面的则是具体的命令内容，i{是说插入字符{，然后退回到命令状态。e 是移到单词的结尾处，a}则是增加字符}然后退至命令状态。
在我们做过这样的工作以后我们就可以来执行我们的命令了，我们将光标定位在一个单词上，例如 amount，按下 F5 键，我们就会发现这时就变成了{amount}的形式。
在这里我们要注意的一点就是我们最好不要将在 vi 中有特殊命令的功能热键与某些命令进行绑定，如果是这样的话就会给我们带来麻烦。
同上面的:abbreviate 命令相类似，我们也可以用命令:map 来列出在这个文件中所有的功能键，包括我们自己定义的以及系统定义的。
我们在做过这样的设置以后，有时希望这样的设置在下一次使用时会继续有效，这时我们就要用命令:mkvimrc 将我们所做的这些设置写入文件中。这个命令的格式如下：
:mkvimrc file
在这里 file 就是我们要将这些设置写入的文件名。我们可以用命令:source file 来读出文件并执行

文件中的内容。
在 vi 的启动过程中，vi 会首先查找初始化文件。如果找到就会自动执行文件的内容。而这些初始化文件一般来说是这样的一些文件：
$HOME/.vimrc
$HOME/_vimrc
$HOME/.exrc
$HOME/_exrc
而在 Windows 和 Dos 系统中则是这样的一些文件：
$HOME/_vimrc
$HOME/.vimrc
$VIM/_vimrc
$VIM/.vimrc
$HOME/_exrc
$HOME/.exrc
$VIM/_exrc
$VIM/.exrc
而如果我们用一些图形界面的话，那么还会读一些其他的配置文件。如果我们要执行 gvim 那么 $VIMRUNTIME/menu.vim 也会被读进来。用命令:version 我们可得到这些配置文件的信息。
例如我们可以在.vimrc 中写入以下的一些语句：
:syntax on
打开语法加亮功能
:set shiftwidth=4
设置缩进宽度为 4
:ab #d #define
将#define 缩写为#d
:ab #b /*************************
:ab #e **********************/
我们还可以加入一些其他的东西，这样就可大的方便我们的工作了。而这也正是 vi 的强大之处 。

现代的 Vim 编辑器是基于一个古老的名为 vi 的编辑器设计而成的，而 vi 基于一个更古老的基于命令行方式的 ex 编辑器设计而成的。ex 编辑器是在屏幕产生以前相当流行的一个编辑，他是为那时古老的标准打印而设计的。
仅管 ex 是基于命令行方式的，然而他也是真正意义上的一个强大而高效的编辑器。即使是在今天他也是发挥着作用。仅管现在的 Vim 编辑器有一套的命令系统，可是有些事情还是要用 ex 的命令方式才可以更好的来完成。因而现在的 Vim 编辑器设计了与 ex 编辑器的接口，我们可以在 Vim 中使用 ex 方式的命令。而那些以冒号开头的命令就是一个 ex 方式的命令。

如果你想执行单一的命令行方式的命令，那么你只输入：然后输入命令就可以了。例如我们在前面讨论的:set number.事实上他就是一个命令模式的命令。在执行完这个命令以后，Vim 编辑器又回到了他原来的状态。我们可以通过命令:ex 选择进入命令行模式。Q 命令也有同样的作用。如果要选择回到正常模式（normal mode)也就是可视化模式（Visual mode)，我们可以使用命令:visual.
命令:print（简写为:p)可以打印出选定的行。如果没有传递参数，他仅是打印当前的行。打印完成以后回到打印行的开头。我们也可以指定要打印的行的范围。例如命令:1,5 print 就是要打印 1 到 5 行。严格来说你不必须在数 5 和 print 之间输入空格，但是如果这样做后我们就会现这个命

令看起来会更好。如果你仅仅是想打印第 5 行，你可以用这样的命令：:5 print.当然了我们也可以使用一些特殊的数字标记，例如字符$是指最后一行，因而如果我们想要打印全文的时候我们可以用这样的命令：:1,$ print.而字符%是指整篇文章（1,$)，因而我们也可用这样的命令来打印整篇文章：:% print,而.则是指当前行，我们要打印当前可以用:print 命令而不带任何参数，也可以用这样的命令:.print。我们还可以用指定句子中内容来打印选定的行。例如在我们的文章中我们可以用这样的命令来打印含有字符 ex 的行：:/ex/ print。这样就会打印出含用 ex 的行，并会高亮显示出 ex。同样命令:?ex? print 也会达到同样的作用，而命令:?ex? print 也正是打印含有 ex 字符的命令格式。

在前面的学习中我们曾学过标记命令 m，例如我们可以在任何地方用命令 ma 做上标记，然后在其他的地方可以用命令‘a 回到做了标记的地方。这个命令也可以与 print 命令组合。例如我们可以在一个地方用命令 ma 做上标记，然后在其他的地方用命令 mb 做上另外的标记，然后我们就可以执行下面的命令来打印这两个标记之间的内容了：:'a,'b print

我们也可以在可视化的模式下来选定要打印的内容。例如我们用命令 V 进入可视化模式并选定一段内容，这时我们输入：就会在 Vim 的底部显示<,>两个字符，这两个字符分别指我们的选定内容的开头和结束部分。

命令:substitute 可以使我们将指定的字符换成其他的字符。这个命令的格式是这样的：

:range substitute /from/to flags

在这个命令中 range 是指定了范围，也就说是在哪些行里做替换。而后是说将字符串 from 替换成字符串 to。在默认的情况下，这个替换命令仅是将一行中第一个出的的字符替换成给定的字符。而如果我们想将所有出现的字符都替换成给定的字符我们就用 g 这个标记命令。例如：:% substitute /from/to/g.这个命令就达到将所有出现 from 的地方全部替换成 to。其他的标记（flags）包括：p（print），这个命令是告诉 substitute 打印所做的改动。c（confirm），这个命令是告诉 substitute 命令在做出这样的改动以前要询问是否要做这样的改动例如如果我们执行下面的命令：:1,$ substitute /Professor/Teacher/c.在 Vim 就会显示我们将要做改动的文本，并显示下面的内容：

Prfessor: You mean it's not supposed to do that?

replace with Teacher(y/n/a/q/^E/^Y)?

这时你可以做出以下这样的回答：

y 执行这个替换

n 跳过这个替换

a 执行所有的替换不要做询问

q 退出，不做任何改动

CTRL—E 向上翻滚一行

CTRL-Y 向下翻滚一行

在 Vim 的命令还有一些命令可以帮助我们很好的完成我的工作：例如命令:read filename 可读进一个文件并将读进和内容插在当前行的后面。而命令:write 是将文件写入。这是一个保存我们工作的方法。我们也可以用命令:write newfile 将当前的文件内容写入一个新的文件。一般情况下:write 命令并不会覆盖已经存在的文件。我们可以用强制操作（!）选项来完成我们所需要的操作并覆盖当前已经存在的文件。而这个命令对于我们要将一个大的文件分拆为小的文件时显得更有用。我们可以用可视化模式选定一个范围然后将这个选定的范围写入新的文件，从而实现了分拆的目的。

我们还可以 Vim 中不需要退出而执行 Shell 命令。命令:shell 可以使们进入命令终端执行我们需要的命令。当我们要退出终端回到 Vim 中时可以执行 exit 命令。我想这对于程序人员来说真是一个巨大的帮助。:）

在我们谈起 Vim 编辑器似乎只是觉得他只是一个类似于一个命令行方式的文本编辑器。而事实上不是这样的。Vim 在窗口环境中也可以完美的完成我们的工作。在窗口环境下，我们不仅可以使用那些在 Vim 文本方式下的命令来完成工作，而且还有许多的菜单和其他的选项。这些都可以使得我们完美的完成我们的工作。

我们要启动图形界面的 Vim 可以用下面的命令：gvim file。这样就可以打开图形界面来编辑文本 file。图形界面下的 Vim 编辑器的外观因你所用的操作系统的不同而有所不同，就是同样的操作系统也会因你所使用的工具集不同（Motif,Athena,GTK)而会呈现不同的外观。而值得向大家推荐的是 GTK 版本的 Vim 编辑器，当然其他版本的也是可以来完美的完成我们的工作的。

在 Windows 系统中我们可以在标准形式下用鼠标来选择文本，而在 X Window 系统中我们也会有一个标准的系统来使用鼠标，但是这两种标准是不同的。然而比较幸运的是我们可以定制我们的 Vim 编辑器。我们可以使得我们的 Vim 中的鼠标的行为看起来像是 X Window 系统中的鼠标的行为。下面的命令是设置鼠标的行为为 X Window 风格的鼠标行为：:behave xterm。而下面的命令则是使得鼠标的行为是 Windows 风格的行为：:behave mswin。在 UNIX 和 Linux 系统中的鼠标风格是 xterm 风格，而在 Windows 中的鼠标风格是在安装过程中选择的。为了控制鼠标的行为，命令:behave 有以下这些选项：

Option	Setting for :behave mswin	Setting for :behave xterm
'selectmode'	mouse,key	(empty)
'mousemodel'	popup	extend
'keymodel'	startsel,stopsel	(empty)
'selection'	exclusive	inclusive

xterm 的鼠标风格的行为主要有以下一些：

左键：　　移动光标

拉动左键：　　在可视化模式下选择文本

右键：　　选中光标处到右键点击处的文本

中键：　　在光标处粘贴选中的文本

Windows 的鼠标风格的行为主要有以下一些：

左键：　　移动光标

拉动左键：　　在选择模式下选中文本

<S-Left Mouse>　　选中到光标处的文本

<S-Right Mouse>　　显示弹出菜单

中键：　　将系统剪切板中的文本粘贴到文件

(注：其中的 S 为 Shift 键）

其他的一些特殊的用法：

Shift+左键：　　向前搜索光标处的文本

Shift+右键：　　向后搜索光标处的文本

Ctrl+左键：　　跳转到以光标处的文本为名字的标记（tag）处

Ctrl+右键：　　跳转到前一个标记处

在图形界面的 Vim 编辑器还有一个有趣的特征，那就是当我们点开一个菜单时就会发在子菜单的第一行有一条虚线，点击此虚线就可以将这个菜单移其他的地方。关闭后 Vim 的菜单结构又恢复到正常的状态了。在 GTK 版本和 Windows 版本中的图形界面的 Vim 还有一个工具栏，这些工具可以快速的完成我们的工作。

虽然现在的专业的文字处理软件例如 MS Word,Star Office,Open Office 等可以很好的来完成的一些文档处理的工作,但是人们仍然是喜欢用纯文本的形式来处理手中的文本.这里因为用纯文本处理的文件比较容易阅读,很不你那些专业的字处理软件,有专门的文件存储格式,少了许多的麻烦.Vim 是一个强大的文本编辑器,他也可以像那些专来的字处理软件一样来处理我们手中的文本工作,从而使得我们的工作来得更漂亮.

Vim 在默认的情况下是不会自动换行的,这也就是说我们在默认的情况下我们不得不自己决伫回车的位置.这样的设置对于处理程序文件来说是一件相当好的事情,因为这样可以由我们自己来决定回定回车换行的位置,从而可以很好的来完成我们的程序设计工作.但是如果我们是在处理一些文档工作,那么这个问题就有一些成为困挠我们的问题了.幸运的是 Vim 提供了这样的功能可以使我们来解决这样的问题.当我们指定了 textwidth 这个选项后,Vim 就会自动的在相应的地方加上换行符.例如我们可以用下面的命令来指定一个只有 30 列的文本:

:set textwidth=30

这样以后当我们再输入文本时如果达到 30 这个限制,Vim 就会自动的依据情况来加上换行符.在 Vim 中我们可以有两种方法来选择换行的方式.例如下面的命令是告诉 Vim 从左面算起 30 个字符时换行::set textwidth=30

而下面的命令则是告诉 Vim 从右面算起当达到 margin 个字符的空白时要换行:

:set wrapmargin=margin 这里的 margin 是空白字符的个数.例如如果你有一个 80 个字符宽的文本,下面的两个命令则是起到了同样的作用:

:set wrapmargin=10

:set textwidth=70

Vim 并不是那些专业的字处理软件,在我们指定了文本宽度的情况下,当我们将一行中的前几个文字删掉的话,Vim 并不会像那些专业的字处理软件那样将后面行中的文本移到前面的行上,而是形成了一些长短不一的段落.这样看起来不是很好,因为我们需要的是有同一风格的文本形式.在 Vim 中这样的情况可以有几种处理方法:一是在可视化模式下选中这些文本,然后用 gp 命令来格式化选中的段落.另一种方法就可以使用 gqmotion 的命令来完成格式.例如我们要格式化 5 行,我们就可以用下面的命令:gq4j.这个命令是告诉 Vim 编辑要格式化本行,同时要格式化下面的 4 行.这样就达到了格式化 5 行的目的.在这样的情况下,向前移动的命令}这时就会为我们提供更大的便利了.这时我们的做法是这样的:将光标放在我们要格式化段落的第一行上,然后执行命令 gq}.这样就可以达到我们的目的了.这样方法要简便得多,因为我们不必要再数细数我们要格式化多少行了.命令 gqip 可以格式化当前的段落.这个命令要比上一个的格式命令还要简便一些,因为在这样的情况下我们不必要将光标放在一个段落的第一行了.最后如果我们要想格式化一行的可以使用命令 gqgq,当然了我们也可以简记为 gqq.

我们都知道在专业的文字处理软件中有文本对齐的选项,而这在 Vim 当中也是可以做到的.如果要将文本居中对齐我们可以用这样的命令::range ceter width.在这个命令中如果我们没有指定 textwidth 的值,他将使用我们设置的 textwidth 的值,如果我们也没有设置这个值,那么他将使用系统定义的 80.例如我们要将 1 到 5 行的内容居中对齐,我们可以使用下面的命令:

:1,5 center 30

同理右对齐的命令也可以类似的写成:

:1,5 right 30

但是左对齐的命令就与这两个命令有一些不同了,左对齐的命令为:

:range left margin
在这个命令中左对齐的参数并不是文本的宽度,而在文本左边的空白字符的宽度,如果为 0,那么将紧靠屏幕的左边沿.在 Vim 中并没有内置的方法来对齐文本,但是我们可以使用一个简洁的宏包来完成这样的工作.要使用这个包先执行下面的命令:
:source $VIMRUNTIME/macros/justify.vim
在这个包中定义了一个新的可视化的命令_j.要对齐一个文本块,可以在可视化模式中高亮显示这个文本块,然后执行命令_j.
J 命令可以使两行合并为一行,同时用空格为分格这两行.在文本处理中有一个 joinspace 的选项,如果设置了这个选项,那么可果一行是以标点符号来结尾的,那么在用这个命令后会用两个空格来区分这两行.也就是说如果我们用命令:set nojoinspace ,用 J 命令来合并这两行时会用一个空格来区分.但是如果我们用命令::set joinspace ,用 J 命令来合并这两行时会用两个空格来区.这就是这个选项所要起到的作用.
在 Vim 编辑器中我们可以通过设置 formatoptions 选项来设置 Vim 中的文本和注释的换行方式.这个命令为:
:set formatoptions=character
在这个命令中 character 一组格式化的标记,他可以是下面的一些字符:
t 文本自动换行
c 注释自动换行,同时自动在行首添加注释标记
r 当添加新行时自动添加注释符
o 当用 O 和 o 开始新的一行时自动在行首添加注释符
q 允许使用 gq 来格式化文本
2 第二行缩进两个字符
v 采用老的 vi 换行方式,当你输入空格时换行
b 在达到 textwidth 以前当输入空格时换行
l 在插入模式下不换行,只用 gq 来完成相应的工作
下面让我们来看一下这些选项是如何来工作的:
当需要文本自动换行时要打开 t 标记,当需要注释自动换行时要打 c 标记.所以对于一个要处理程序文件的人来说打开注释自动换行似乎要有更大的帮助:
:set formatoptions=c
这样在注释中的一个长句就会自动换行,而正常的程序文本就不会被自动换行.事实上我们常做些这样的选项:
:set formatoptions=cq
这个选项是告诉 Vim 编辑器不仅注释要自动换行,而且这些注释可以使用 gq 命令来格式化.
Vim 编辑器可以很好的处理注释的问题.在 C 程序风格的程序文本处理过程中,当注释自动换行时 Vim 会自动在注释的开头加下注释标记.但是在这样的设置也还是存在一个问题的,那就是当你回车时 Vim 就不会在下一行的开头自动加上注释标记了.如果这一行仍写的是注释,那么就要你亲自来加上了.但是当我们打开 r这个标记后就会解决这个问题了.这时如果你打回车,Vim 编辑器还会在新的一行加上注释标记.这时如果你要想在下一行写程序文本,那么就不得不动手删除注释标记了.如果你希望当用 O 或是 o 添加新的一行时要自动添加注释标记就要打开 o 这个格式标记了.
格式标记选项是告诉 Vim 编辑器在格式文本中要从第二行开始而不是第一行.设置这个选项的命令为:
:set formatoptions+=2
而 v 标记则控制一个句子从哪里进行分裂.例如现在我们有一个句子:
This is a test of the very long warpp.

现在我们要在这个句子的后面新增一个词 logic
如果没有 v 标记结果就会变成:
This is a test of the very
long line warpping logic.
而如果打开 v 标记,结果就会变成:
This is a test of the very long line wrapping
logic.
令管已经存在的文本已经远远的超过了 textwidth 的限制,但是因为设置了 v 标记,所以 Vim 编辑器也不会换行,相反只有你新增了一个词时才会换行.
在默认的情况下 vim 编辑器是使用内部的格式程序来模式文本,当然了我们也可以使用外部的格式程序来格式我们的文本.在 UNIX 和 Linux 系统中有一个标准的程序 fmt 可以很好的来做这样的工作.如果我们想用命令 gq 来执行这个处部命令,我们可这样的来进行设置:
:set formatprg=fmt
即使是没有进行这样的设置,我们也可以使用命令!来格式文本.例如如果我们想用程序 fmt 来格式一个段落,我们可以用这样的命令:!}fmt.!是开始了一个过滤命令,而}是告诉 Vim 过滤一个段落.fmt 是我们要使用的命令的名字.
在早期的打印机的时代,开始新的一行要占用两个字符的时间.如果到了一行的结尾处,你要他快速的回到新的一行的开头,打印的针头在纸面上飞快到掠过,常常就会在纸面的中间留不希望的污点.这个问题的解决办法就是用两个字符来解决:一个字符<Return>来移到第一列,而字符<Line feed>来新增一行.计算机产生以后,存储较为昂贵,在如何解决回车换行这个老问题上,人们产生了不同的意见.UNIX 人认为在到达一行的结尾时新增一行(<Line feed>),而苹果人则认同<Return>的解决办法,MS 则坚持古老的<Return>,<Line feed>的方法.这就意味着如果你将一个文件从一个系统转移到另一个系统,你就面临着回车换行的问题.而 Vim 编辑器则会自动的认出这种文件格式方面的区别,并会为我们做出相应的处理.
fileformats 选项包含了许多当编辑一个新文件时会用到的一些文件格式.例如下面的命令就是告诉 vim 编辑器将 UNIX 的文件格式做为第一选择,而将 MS-DOS 的文件格式做为第二选择:
:set fileformats=unix,dos
检测到的文件格式会被存放在 fileformat 选项中,我们可以且下面的命令来查找我们所使用的文件格式:
:set fileformat?
我们还可以应用 fileformat 这个选项将一个文件从一种文件模式转换成另一种文件格式.例如我们有一个名为 readme.txt 的 MS-DOS 文件格式的文件,而我们想将他转换为 UNIX 文件格式的文件.我们可以按照下面的方法来做:
首行编辑该文件:
$ vim readme.txt
其次将文件格式改为 UNIX 文件格式:
:set fileformat=unix
然后保存该文件,此时这个文件就转换成为了 UNIX 文件格式的文件.
在默认的情况下 Vim 编辑器认为我们的文件是由行组成的,也就是他认为文件中的最后一行是以<EOL>为结束符的.有时我们会遇到不包含有结束标记行的文件,当 Vim 遇到这样的文件时,他就会设置 noendofline 选项,而当遇到正常结束符的文件时则会设置 endofline 选项.如果你想设置你的文件以<EOL>结束符结尾则可以用下面的命令:
:set endofline
如果你想设置一个文件不以<EOL>结束符来结尾,则可以用下面的命令:

;set noendofline
在 Vim 中,我们有一系列的命令可使得我们在文件中进行移动,例如)命令向前移动一个句子,而(是向后移动一个句子,}向前移动一个段落,而{是向后移动一个段落.
曾经 Troff 是 UNIX 系统上专业的字处理程序,虽然在现代这个程序已经很少用了,可是 Vim 还是包含了一些选项使得我们可以用这种曾经的方式来处理文本.Troff 需要用宏来告诉他应做一些什么.其中的一些宏开始一个新的段落.因为 Troff 要用许多的宏包来,因而 Vim 需要知道哪些宏将开始一个新的段落.paragraphs 选项可以做这些工作.这个选项的格式如下:
:set paragraphs="macromacro..."
每一个宏是两个字符的 Troff 宏名,例如:set paragraphs="P LP"是告诉 Vim 宏.P 和.LP 开始一个新行.在默认的情况下 paragraph 的选项如下:
:set paragraphs=IPLPPPQPP LIpplpipbp
这个命令列出下面的一些开始一个新段落的宏名:
.IP .LP .PP .QP .P .LI .pp .lp .ip .bp
我们可以使用命令[[和[]向前移动一个区间,而使用命令]]和][向后移动一个区间,一个区间是用页分隔符(CTRL-L)定义的文本.我们也可以用 Troff 的宏来定义一个区间,section 选项有些类似于 paragraph 选项,所不同的是前者定义一个宏用来分隔区间,而后者是段落.默认情况下是这样的;
:set section=SHNHH HUnhsh
在用 Vim 进行文本处理时我们还可以对文本进行加密处理,这种加密处理是最弱的一种加密处理方法,常常用新闻的发布.这时我们要用到的命令为 g?motion.例如我们可以用命令 g?g?或者是 g?? 来加密当前行.当我们对一文本进行两次加密处理就为解密处理了.

我们在用 Vim 来处理文件时可以使用 Vim 的自动完成功能来大大加速我们的工作速度.所谓的自动完成也就是说当我们输入一个单词的一部分以后,按 CTRL -P,Vim 就会自动的来完成剩下的部分.我们在前面的学习过程中曾用:abbreviate 命令来简记某一个单词来达到自动完成的目的,而在这里我们将看到是一个更加强大的自动完成功能.Vim 能非常简单和灵活的来决定要用哪一个单词来自动完成. 我们在使用 Vim 这个强大的自动完成功能的同时,还可以自己定义我们的自动完成的特征,而且还可以使用不同类型的自动完成功能.
我们在用 Vim 来处理文件时可以使用 Vim 的自动完成功能来大大加速我们的工作速度.所谓的自动完成也就是说当我们输入一个单词的一部分以后,按 CTRL-P,Vim 就会自动的来完成剩下的部分.我们在前面的学习过程中曾用:abbreviate 命令来简记某一个单词来达到自动完成的目的,而在这里我们将看到是一个更加强大的自动完成功能.Vim 能非常简单和灵活的来决定要用哪一个单词来自动完成. 我们在使用 Vim 这个强大的自动完成功能的同时,还可以自己定义我们的自动完成的特征,而且还可以使用不同类型的自动完成功能.
如果我们在编写 C 程序,而我们所谓得到的下面的一个句子:
total=ch_array[0]+ch_array[1]+ch_array[2]
这时我们输入 total=ch_array[0]+ch_,然后按下 CTRL-P,Vim 就会自动的替我们完成其余的部分,这时我们得到将是
total=ch_array[0]+ch_array
由此可以看到我们在处理文件时用这样的方式可以大大的加快我们的处理速度.
那么 Vim 是如何找到匹配的单词的呢?在默认的情况下,Vim 在查找一个单词时是按照如下的步骤:
1 在当前文件中进行查找
2 在其他窗口中进行查找

3 在其他的已装入的缓冲区中进行查找
4 在没有装入缓冲区的文件中进行查找
5 在当前的标记(tag)列表是进行查找
6 在所有的由当前文件的#include 包含进来的文件中进行查找
当然了我们也可以自定义我们的查找顺序.
我们在使用自动完成功能时的命令 CTRL-P 是向后查找匹配的单词,而还有一个命令 CTRL-N 是向前查找匹配的单词.他们有同样的功能和作用,所不同的只是查找方向上的不同.
Vim 还提供了许多的命令可以使得我们来自定义我们的一些查找上的特征.例如我们可以用下面的命令来告诉 Vim 在在自动完成的查找过程中忽略大小写的区别:
:set ignorecase
这样以后如果我们输入 ins,Vim 就会认为是 INSERT,Inside 或者是 instep.当然了前提是在我们所编辑的文本中含有这些词,要不就会找得到了.
为了能够精确的进行查找来自动完成,我们并不希望上面的情况出现,我们可以设置一些选项来告诉 Vim 区分大小写的情况.这时我们要用到下面的命令:
:set infercase
这样以后如果我们再输入 ins,与其相匹配的列表就成为了 instep,inside,insert.我们可以通过按 CTRL-P 或是 CTRL-N 来进行匹配完成.
在大多数情况下,Vim 默认的设置可以很好的来完成工作,但是有时我们要定义自己的一些完成的选项,这时我们就要用到 complete 这个选项了.这个选项的格式如下:
:set complete=key,key,key
而这个命令中可能出现的 key 值如下:
. 当前文件
b 已被装缓冲区,但是没有在窗口内的文件
d 在当前的文件中定义和由#include 包含进来的文件
i 由#include 包含进来的文件
k 由 dictionary 选项定义的文件
kfile 名为{file}的文件
t 标记(tags)文件
u 没有载入的缓冲区
w 在其他窗口中的文件
我们可以使用 path 选项来告诉 Vim 如何来查找我们在当前文件中所包含进来的文件.我们还可以指定一个字典,这个选项的格式如下:
:set dictionary=file,file,....
这个选项定义了由命令 CTRL-P 和 CTRL-N 进行匹配查找时所要查找的文件.在 Linux 系统中这个定义文件在/usr/dict/words 中,所以如果我们要将这个文件添加进来进行查找的话,我们就要用到下面的命令:
:set dictionary=/usr/dict/words
如果我们要使用一个我们自己的文件也可以这样的来设置
:set dictionary=/home/oualline/words,/usr/doc/words
我们也可以指定一个字典文件和 k 选项组合使用:
:set dictionary=k/usr/oualline/words
我们也可以多次的使用 k 这个标记选项:
:set dictionary=k/usr/dict/words,k/usr/share/words
在上面提到的 CTRL-P 和 CTRL-N 进行查找匹配时查找的范围比较的宽范,我们当然也可以使用

命令进行一些比较严格的查找.这时我们可以使用命令 CTRL-X.当我们输入 CTRL-X 时我们会进入 CTRL-X 的一个子模式.这时我们可以使用下面的命令进行查找:

CTRL-D 宏定义
CTRL-F 文件名
CTRL-K 字典
CTRL-I 当前文件以及由#include 包含进来的文件
CTRL-L 整个行
CTRL-] 标记(tags)
CTRL-P 向前查找,与没有 CTRL-X 命令时相同
CTRL-N 向后查找,与没有 CTRL-X 命令时相同
CTRL-X

CTRL-D 命令查找宏定义.他也会查找#include 文件.当我们执行完这个命令以后就可以使用 CTRL-P,CTRL-N 来进行匹配查找.

例如我们可以编辑下面的测试文件:

include.h 文件中的内容

```
#define MAX(x,y) ((x)<(y)?(y):(x))
#define MIN(x,y) ((x)<(y)?(x):(y))
int sum(int i1,int i2)
{return (i1+i2);}
```

main.c 文件中的内容:

```
#include "include.h"
#define MORE "/usr/ucb/more"
```

这时我们开始编辑 main.c 文件,如果我们按下 CTRL-X 我们就会进入 CTRL-X 的子模式.如果我们要查找一个宏定义,我们可以按下 CTRL-D,这时就会在屏幕的底部简单的显示出有多少匹配的选项.这样我们就可以用 CTRL-P 和 CTRL-N 来进行自动完成的功能了.而命令 CTRL-X CTRL-]则是查找下一个标记(tag),标记是一个 C 函数的定义.我们可以用命令 ctags 命令来生成一个 C 函数定义的列表.我们可以这样的来使用这个命令:

```
$ctags *.c *.h
```

这样以后我们就可以在插入模式入下用 CTRL-X CTRL-]命令来进行标记的查找和匹配了.

在默认的情况下,vim 编辑器只是简单的显示出标记的名字,我们可以执行下面的命令,这样以后就可以显示出整个标记了:

```
:set showfulltag
```

我们可以使用 CTRL-X CTRL-F 命令来匹配文件名.他会在当前的目录下查找文件并会显示出匹配的内容,这时你就可以用 CTRL-P 和 CTRL-N 命令来选择你想要的匹配选项了.

到目前为止我们所说还只是对单词进行操作,我们可以用命令 CTRL-X CTRL-L 对一行进行匹配操作,同样的我们也可以用 CTRL-N 和 CTRL-P 来进行选项的匹配.我们还可以在输入 CTRL-X 命令后用 CTRL-Y 向下滚动文本,而用 CTRL-E 向上滚动文本.

Vim 最大的一个优点就是他的灵活性.使得他具有如此灵活性的一个重要原因就是自动命令.所谓的自动命令就是一条命令,所不同的只是他可以在某些事件发生时自动执行.

例如通过 Vim 的这样自动命令,我们可以使用 Vim 来编辑压缩文件,这是因为我们可以定义一个自动命令在我们读取文件时解压缩,在我们写入文件时进行压缩.

我们在处理文件时有时希望文件在保存时在文件的结尾处插入当前的日期,而这在 Vim 当中我们

可以使用自动命令来完成.我们可以定义这样的一个函数:

```
:function DateInsert()
:       $read !date       "在文件的结尾处插入日期
:endfunction
```

当我们要保存文件时可以调用这个函数:

```
:call DateInsert()
```

然后我们就可以保存退出了.

我们还可以将这个功能绑定在一个按键上:

```
:map <F12>:call DateInsert()<CR>|:write<CR>
```

这种方法使用问题的解决更简单,因为我们只需在要保存文件时按一下 F12 键就可以了.

但是也许有时我们会忘记这样的步骤,而使用正常的保存命令,那么这时这个函数也就失去了意义.

我们希望这个插入日期的命令可以自动被执行,这也正是自动命令的意义所在.

下面的命令就可以来满足我们的要求了:

```
:autocmd FileWritePre * :callDateInsert()<CR>
```

这个命令会使得所有的文件在保存之前调用这个插入日期的函数.我们并不需要每一次都要输入:write 来保存文件,也就是说当我们定义了这个命令以后,当我们输入一次:write 命令,vim 就会自动检查所有未保存的文件并执行我们定义的这个命令,然后执行保存命令.

:autocmd 命令的一般格式如下:

```
:autocmd group events file_patter nested command
```

在这个格式中组名(group)是一个可选项,他被用于管理和调用命令.参数事件(events)是一个事件列表,指明引发命令的事件,而嵌套标记(nested)可以允许嵌套自动命令,最后是所要执行的命令.

命令:augroup 可以使得我们定义一组自动命令.这个组名最好应是一个和我们要定义的一组自动命令相关联的名字,如下面的例子:

```
:augroup cprograms
:       autocmd FileReadPost *.c :set cindent
:       autocmd FileReadPost *.cpp :set cindent
:augroup END
```

在这里因为这些自动命令是在:augroup 的作用域之内的,所以他们被放在 cprogram 这个组内.这组自动命令是在读取 C 和 Cpp 之后执行的.如果我们想在这个组中添加关于头文件的自动命令,我们可以使用:augroup 命令,或者是在我们的定义中包含这个组名.

如果我们正在编辑的文件名为 sam.cx,而我们想将他当作 C 程序文件来对处理.这时我们可以通过命令来告诉 Vim 编辑器在文件读入以后将他与*.c 文件相匹配.我们所用的命令如下:

```
:doautocmd group event file_name
```

在这个命令执行的时候,Vim 是将假定当前文件的名字为 file_name,而实际上这并不是他真正的文件名.如果在这个命令中我们并没有指明组(group),那么所有的组都会被用到.而且我们必须指明事件,并且 Vim 会认为这个事件已经发生.与命令:doautocmd 相类似的是命令:doautoall,所不同的只是后者是针对每一个缓冲区(buffer).这个命令的格式为:

```
:doautoall group event file_name
```

我们可以使用下列的命令来引发命令:

```
BufNewFile                  当编辑一个新文件时引发命令
BufReadPre      BufReadPost      在读入之前或是读入之后引发命令
          BufRead           BufReadPost 的另一种说法
BufFilePre      BufFilePost      在通过:file 命令更改缓冲区名字之前或是之后引发
FileReadPre     FileReadPost
```

在用:read 命令读入文件之前或之后.在文件读入之后,在文件的开头和结尾处有由[和]来标记
FilterReadPre FilterReadPost 在用过滤命令(filter)读入文件之前或之后
FileType 当 FileType 设置时有效
Syntax 当 Syntax 设置时有效
StdinReadPre StdReadPost 由标准输入设备读入或是输出
BufWritePre BufWritePost 在将整个缓冲区写入文件之前或是之后
BufWrite BufWritePre 的另一种说法
FileWritePre FileWritePost 将部分缓冲区内容写入文件之前或是之后
FileAppendPre FileAppendPost 将一个文件由过滤命令输入之前或之后
FileChangedShell
这个事件是在 Vim 运行 Shell 命令而文件的修改时间已经改变时引发
FocusGained FocusLost
这个事件是在 Vim 编辑器得到或是失运输入光标时引发.这意味着 vim 运行图形界面,并且是 Vim 成为当前窗口或是其他的成为当前窗口
CursorHold
用户停止输入的时间长于由 updatetime 所指定的时间时引发
BufEnter BufLeave 当进入或离开缓冲区时引发
BufUnload 在缓冲区未载入之前引发
BufCreate BufDelete 在缓冲区产生以后或是在缓冲区删除以前引发
GuiEnter 启动 GUI 时引发
VimEnter Vim 启动而初始化文件还没有读入进引发
VimLeave 退出 Vim 编辑器而.viminfo 还没有改写时引发
FileEncoding fileencoding 已经被设置时有效
TermChanged term 选项被更改时引发
User
并不是一个真正的事件,而是命令:doautocmd 一起使用的一个事件
当存文件时,只有下列事件成对出现时才引发:
BufWritePre BufWritePost
FilterWritePre FilterWritePost
FileAppendPre FileAppendPost
FileWritePre FileWritePost
当读文件进,只要下列事件中的一个被设置则会引发:
BufNewFile
BufReadPre BufReadPost
FilterReadPre FilterReadPost
FileReadPre FileReadPost
文件名类型与 UNIX 标准系统相匹配.下面列出是一些特殊字符匹配的情况:
* 匹配任意长度的任意字符
? 匹配单个字符
' 分隔交替的类型
? 字符?
, 字符,
character
将这个字符看作是一个要查找的类型字符,例如:a+可以匹配,aa,aaa 等等

一般情况下一个命令的执行结果并不会引发另一个事件.例如 Syntax 的执行结果并不会引发 FileReadPre 事件.但是如果我们在自动命令中加入 nested 关键字就可以来引发了,例如下面的命令:

:autocmd FileChangedShell *.c nested e!

我们可以使用命令:autocmd 列出所有的自动命令.

如果我们想要得到所有这些命令中的一个子集命令,我们可以使用如下的命令:

:autocmd group event pattern

在这个命令中如果我们指定了 group,那么就会所有与 group 相匹配的命令.event 可以是以前定义的或是所有的事件,pattern 指了与类型相匹配的文件选项.只有那些与命令相匹配的命令才会被列出来.

我们可以建立我们自己的自动命令,当然了我们也可以移除那些我们不再需要的自动命令.例如命令:autocmd!就可以移除所有的自动命令.

我们也可以用下面的命令来移除指定的组命令:

:autocmd! group

在这个命令中我们当然也可以为这个组指定事件或是类型

:autocmd! group event pattern

event 可以用*代替所有的事件

我们可以用命令:autocmd!来移除那些已经存在的自动命令,我们还可以同时在一个命令中再新建一个我们自己的自动命令.这个语法的格式如下:

:autocmd! group event pattern nested command

这个命令等价于下面的两个命令:

:autocmd! group event pattern

:autocmd group event pattern nested command

有时我们并不想引发某个命令,在这样的情况下我们可以设置 eventignore 选项来指定那些要忽略的事件.例如下面的命令将进入窗口和离开窗口的事件忽略掉:

:set eventignore=WinEnter,WinLeave

如果要忽略所有的事件,可以用下面的命令:

:set eventignore=all

Vim 编辑器的不仅可以使得我们编辑各种各样的文件,而且还有一些附加的功能来完成我们的作.例如说 Vim 可以实现在对文本文件进行加密,而且我们还可以通过一命令行参数来控制我们对文件的处理.我们还可以通过交换文本来实现对文件的修复.

我们在用 Vim 时常用一些有用的命令行参数,在这些命令行参数中最有用的一个要算是--help,这个命令行参数可以简单的列出一个帮助屏幕,在这个屏幕上列出所有的命令行参数.这个命令的执行结果如下:

这个命令的格式如下:

$ vim --help

如果要查出我们所使用的 Vim 的版本,我们可以使用下面的命令:

$ vim --version

我们可以用-R 参数来在只读状态下打开一个文件,命令格式如下:

$ vim -R filename

在大多数的系统中,这个命令与下面的命令是相同的:

$ view filename
-x 参数可以告诉 Vim 编辑器对某一个文本文件进行加密处理.例如我们想要建立一个文件来保存一些我们自己的小秘密时就可以使用下面的命令:
$ vim -x filename
这时 Vim 编辑器就会要你来输入加密和解密时所需要的密码.这样以后我们就可以在正常的状态下来编辑我们的文本文件了,当我们完成编辑工作要退出时,Vim 就会将文本进行加密处理后退出.
当我们想用 cat 或是 type 命令来打印出已进行过加密处理的文件时会得到一些无意义的信息.
我们当然是可以在加密和非加密两种状态下进行选择和切换.选项 key 包含了加密时的密码.如果我们将这一项设为空,那么也就意味着我们关闭了加密选项.这个命令为:
:set key=
如果我们设定了一个密码,那么我们就打开了加密选项.例如下面的命令:
:set key=
保存退出.此时的文件就没有密码,我们可以通 cat 和 type 命令来打印文件中的内容.
而我们也可以通过样的命令来重置我们的加密密码.例如下面的命令:
:set key=secret
但是这样的设置并不是一个好主意,因为在输入这个命令时你所输入的密码是可以明文显示的,别人在旁边晃动一下肩膀也许就可以看到了,因而这是一个不安全的加密方法.为了避免这样的问题,我们可以使用:X 命令,这个命令可以将我们所输入的密码用*打印出来,因而不易被人看到,所以安全得多.
但是我们要清楚的认识到,Vim 所采用的这种加密算法是很弱的.也许他可以防住偶尔路过的小偷,但是不可能防住一个很有时间的密码学专家.同时我们还要认识到我们在编辑文件时所使用的交换文件(swap file)并没有经过加密,因而当我们在进行文件编辑时,一个有着超级权限的人可以通过交换文件得知我们所编辑的内容.一个解决的办法就是我们在编辑文件时不使用交换文件.如果我们在命令行参数中指定-n 参数,我们就可以在编辑文件时不使用交换文件,这时我们所输入的内容是存在内存中的.例如我们可以使用下面的命令来编辑一个加密的文件同时不使用交换文件:
$ vim -x -n filename
但是我们应该知道在不使用交换文件的情况下,防止了别人偷看我们编辑器的内容,而我们也不可能通过这个交换文件来恢复我们的文件了.
因为不使用交换文件时我们所输入的内容存在于内存中,因而就百以文本方式存在的.任何人都可以通过查看编辑器的内存来发现我们文件中的内容.如果你真的是想保密你的文件,那只能在使用一个不用联网的电脑,使用好的加密工具,当电脑不用的时候我们要安全的锁起来了.要不我们可要怎么办呢?!:-)
想一下,如果我们要处理许多的文件,而要将这些文件中的字符串-person-换成 Jones.我们应怎么样来做呢?一个办法就是我们要输入许多次来手工的更改(想一想这样的事情有多的恐怖),另一个解决办法就是我们可以写一个 Shell 脚本或是批处理文件来做这样的工作.Vim 作为一个原始的屏幕编辑器在正常模式下启动可以极好的完成这样的工作.然而要批处理,Vim 并不会产生一个空的有注释的文件,因而我们要使用 ex 的命令模式.这种方式给了我们一个极好的命令行的接口可以很容易的放入批文件中.
在我们这个例子中我们所需要的命令如下:
:%s/-person-/Jones/g
:write
:quit
我们将这些命令放入一个名为 change.vim 的文件中,然后在批模式下运行 Vim,这时我们可以使用这样的命令:

$ vim -es file.txt<change.vim
这个是告诉 Vim 是 ex 命令模式下运行处理文件 file.txt 并从 change.vim 中读入文件.-s 标记告诉 Vim 不要给出任何的输出提示一类的内容.
还有另外的一些命令行参数使得我们可以更好的来控制 Vim 编辑器:
-R 打开一个文件只读
-m 允许修改.这个选项允许我们可以设置 write 选项和修改文件
-Z
受限模式.这个命令参数可阻止我们使用:shell 命令或是其他的一些命令来使用外部的 Shell.但是这个选项并不会阻止我们使用:vi 命令来编辑文件.
另外还有一些命令行参数可以允许我们决定读取哪一个初始化文件:
-u file
使用 file 为初始化文件,而不是使用.vimrc 作为初始化文件.如果没有这个文件,那么就不会用任何的初始化文件.
-U file
使用 file 而不使用.gvimrc 作为初始化文件.如果没有这个文件,同样不使用任何的初始化文件.
-i file 使用 file 而不使用.viminfo 作为初始化文件.
在 UNIX 系统中,Vim 编辑器实际上是一个有着不同的名字或链接的文件.Vim 编辑器在哪种模式下启动,取决于我们用什么样的命令或是名字来启动他.一些常用的命令如下:
vim 在终端模式下启动(在当前窗口内启动编辑)
gvim 在图形模式下启动(编辑器启动他自己的窗口来进行编辑)
ex 在 ex 模式下启动
view 在正常模式下启动,只读
gview 在图形模式下启动,只读
rvim 在终端模式下启动,受限
rview 在终端模式下启动,只读,受限
rgvim 在图形模式下启动,受限
rgview 在图形模式下启动,只读,受限
vi Linux 下的启动命令,同 vim
我们也可以通过命令行参数来设置初始化模式:
-g 在图形模式下启动 vim(与命令 gvim 相同)
-v 在可视化模式下启动(与命令 vim 相同)
-e 在 ex 模式下启动(与大多数系统上使用 ex 命令相同)
我们还可以使用一些命令行参数来调试我们的 Vim 编辑器,常用的命令行参数有下面的一些:
-Vnumber
显示额外的信息以使我们知道在编辑器的内部都做了一些什么.数字越大,我们得到的输出信息也就越多.这个参数常用来调试我们的 Vim 编辑器脚本.
-f
前景.我们不要在背景模式下启动图形界面.这个启动对于那些只有一个程序执行完毕才开始执行另一个程序的情况显得尤为有用.当然了这个选项对调试也是相当有用的.
-w script
将用户输入的所有字符存入脚本文件.如果这个文件已经存在,那么就追加在文件后面.
-W script -W 相类似,只是这个选项会覆盖掉已经存在的数据
-s script 用-w 选项来恢复脚本记录
-T terminal

设置终端类型.在 UNIX 系统中,这会改写$TERM 的环境变量(当然了,如果$TERM 环境变量错误,其他的许多程序都将会崩溃)
我们还有一些兼容的命令行参数.这些参数可以使得我们的 Vim 运行起来更像是 Vi;
-N
这个参数可以使得 Vim 以自己的方式来运行,而不是不是像 Vi 的方式运行.这个参数是被.vimrc 文件默认定义的.
-C
兼容模式.这个参数关掉了许多 Vim 的自己的特征,而是尽量的像 Vi 的方式一样的运行.
-l
Lisp 模式.这种模式完全是由老版本的 vi 延续下来的.他设置了 lisp 和 showmatch 选项.这时的 Vim 的与文件类型相关的命令都可以很好的处理 Lisp 程序,并且这是可以自动完成的.
最后还有一些不知道该将他们归为哪一类的命令行参数:
-d device 打开编辑给定的设备文件
-b 二进制模式.设置了 noexpandtab,textwidth=0,nomodeline,binary
在一般的情况下,vim 是不产生备份文件的.如果我们希望他产生备份文件,我们可以使用下面的命令:
:set backup
产生的备份文件的名称是在原始文件名的最后加上了~.例如我们有一个名为 data.txt 的文件,那么由 Vim 所产生的备份文件名即为:data.txt~.
如果我们不喜欢这个备份文件的扩展名,我们可以定义我们自己的备份文件扩展名.这时我们要到下面的命令:
:set backupext=string
例如如果我们设置了 backupext 的值为.bak,那么新的备份文件名即为 data.txt.bak
如果我们设置了 patchmode 选项,那么 Vim 编辑器就会以当前文件名加上 patchmode 的值为文件名备份正在编辑的文件.但是我们要注意的是只有以这个文件名为文件名的备份文件不存在时才会产生这样的备份文件.例如我们执行下面的命令:
:set patchmode=.org
如果我们是第一次编辑一个已经存在的文件:data.txt.当我们执行了这样的命令保存退出想要看一下事实是否是这样的.因为以前这个备份文件并不存在,所以会产生一个备份文件,名为:data.txt.org.但是当我们再一次编辑这个文件保存退出后,因为这个备份文件已经存在,所在现在的备份文件名:data.txt~.
通常情况下 Vim 编辑器会将备份文件放在与文件相同的目录下,我们可以通过设置 backupdir 选项来选择我们的备份文件存放的地方,例如我们想将备份文件放在~/tmp 目录下,我们可以通过执行下面的命令来做到:
:set backupdir=~/tmp/
但是这样的设置有时也会产生问题的,如果我们在不同的目录下编辑具有相同文件名的文件,当我们保存退出时,Vim 会将备份文件放在~/tmp/目录下,名字的冲突会使得老的备份文件丢失.同时我们还要知道的就是这个选项可以同时设置几个值,中间用逗来分隔.Vim 会将备份文件放在第一个目录下.
一般情况下,当 Vim 保存文件时,会执行下面的步骤:
1
Vim 要检查 Vim 外面的文件是否被做了改动.例如也许有的人已将这个文件重新命名了,如果发生这样的情况,Vim 就会给出警告并询问是否继续.
2

如果设置了 writebackup 或是 backup 选项,Vim 就会将旧的备份文件移除,同时产生一个当前文件的副本做为新的备份文件
3 将缓冲区的内容写入文件.
4
如果设置了 patchmode 选项而不存 patch 文件,那么 Vim 就会将备份文件重命名成为 patch 文件
5 如果没有设置 backup 选项,而是设置了 writebackup 选项,就会移除备份文件
Vim 覆盖已经存在的文件的原因是因为要保护 UNIX 系统上的硬链接.在非 UNIX 系统上,备份文件只是通过重命名当前文件来产生
注意:如果我们设置了 nobackup 和 nowritebackup 选项,Vim 会覆盖已经存在的文件.这在磁盘已满而更新文件时会造成数据的丢失.
在默认的情况下,Vim 设置了 writebackup 选项.这就意味着 Vim 很难会丢失数据.通过采用这样的方法,在磁盘已满的情况下,我们就没有机会丢失文件.也许我们会不能写入新文件,但是我们不会丢失我们的旧文件.
我们在用 Vim 来编写时,Vim 会在我们编写的过程中产生一个临时的交换文件,这个交换文件中包含着我们所做过的所有修改,当我们完成编写工作,保存退出后,这个临时文件会被删掉.但是如果 Vim 遇到了意外情况而退出时,这个临时文件会因为来不及删除而存在于硬盘上.当 Vim 启动时他会检查在当前目录中是否存着交换文件,如果存在,则意味着有一个 Vim 正在编辑此文件,或者是我们在 Vim 编辑器的过程中遇到意外而退出,从而留下交换文件.这时 Vim 就会给出警告信息,并会给我们机会要我们自己来决定我们下一步要怎么做.这时我们可以有以下四个选项:
Open Read-Only(以只读方式打开)
这个选项会告诉 Vim 以只读方式打开.如果我们想要看到文件中的内容或是有另一个编辑过程正在运行,我们可以选择这个选项
Edit anyway
如果我们选择这个选项我们可以对这个文件进行编辑.我们最好不要选择这个选项,除非我们对我们正在做的事情有着绝对的把握.这时我们应该知道,如果同时有两个或是多个编辑过程同时编辑一个文件,只有最后一个保存的编辑过程有效
Recover
如果我们正在编辑我们的文件,而由于系统故障或是其他的原因而导致 vim 意外退出时我们可以选择这个选项.此时 Vim 会检查交换文件,并试着从我们意外退出的地方重新开始
Quit 取消对此文件的修改
在我们选择了其中的一项后我们就可以正常的开始我们的编辑工作了.如果我们选择了 Recover 我们要十分小心,因为我们以前所做过的修改并不一定被保存下来.
如果我们记得 Vim 意外退出时我们正在编辑的文件,我们可以用-r 命令参数在修复模式下启动 Vim.例如我们在编辑文件 commands.c 时 Vim 意外退出,我们就可以用下面的命令在修复模式下启动 vim:
$ vim -r commands.c
如果我们想得到一个可以修复的编辑器程序列表,我们可以用下面的命令:
$ vim -r
这样 Vim 就会在当前目录和标准的临时的目录下查找交换文件,命令的执行结果就像下面的样子:
$ vim -r
找到以下的交换文件:
在目前目录:
1. .vi14.txt.swp
所有者: mayuelong 日期: Wed Jul 27 19:19:39 2005

文件名: ~mayuelong/Documents/vi14.txt
修改过: 是
用户名: mayuelong 主机名: localhost.localdomain
进程 ID: 3070 (正在执行)
在目录 ~/tmp:
-- 无 --
在目录 /var/tmp:
-- 无 --
在目录 /tmp:
-- 无 --

Vim 在意外退出的情况下并不会覆盖旧的交换文件.例如第一次编辑时产生的交换文件名为.file.txt.swp.如果我们再编辑又遇到意外退出时所产生的交换文件名为.file.txt.swo,到第三次时所交生的交换文件为.file.txt.swn,如此类推.
我们在启动 Vim 时可以指明用哪一个交换文件来修复文件,命令如下:
$ vim -r .file.txt.swo
如果想知道我们当前正在使用的交换文件的名称可以使用下面的命令:
:swapname
这样就会显示交换文件的名称.
通常情况下,交换文件会每 4 秒或是每隔 200 个字符保存一次.这个数值是由 updatecount 和 updatetime 选项来控制的.我们可以用下面的命令来设置交换文件每 23 秒保存一次:
:set updatetime=23000
(注:这个数值是以微秒计的)
或者是我们可用下面的命令来设置 vim 每 400 个字符保存一次:
:set updatecount=400
如果我们将 updatecount 的值为 0,那么交换文件就不会被保存了.
事实上我们可以来控制是不是要在编辑的过程中产生交换文件,例如下面的命令是在产生交换文件,而这也正是 Vim 所默认的:
:set swapfile
我们也可以用下面的命令来使 Vim 不产生交换文件:
:set noswapfile
我们可以对每一个编辑的文件将这个选项设置或是重置.如果我们正在编辑一个大文件而我们又不想可以修复,我们可设置 noswapfile.如果我们同时正在另一个窗口编辑一个文件,那么这个窗口中的文件仍是使用交换文件的.
在 UNIX 或是 Linux 系统中,当我们要保存文件时,通常数据被装入内存缓冲区,并肯是在系统认为是一个合适的时才会被写入文件.这通常只是几秒钟的事情.如果我们想要确认数据到达了磁盘,我们可以使用下面的命令:
:set swapsync
这个命令是告诉 Vim 编辑器在每一次将文件写入交换文件的同时写入磁盘.swapsync 选项可以是 fsync 或是 sync,这个取决于我们要保存文件时的系统调用.
通常情况下,Vim 是在和当前文件相同的目录下产生交换文件,我们可以通过 directory 选项来更改交换文件产生的目录.例如下面的命令可以将产生的交换文件放在/tmp 目录下:
:set directory=/tmp
这并不是一个好主意,因为如果我们是在不同的目录下编辑具有相同文件名的文件时会产生名字

冲突.
我们也可以将这个选项设成一个目录列表,中间用逗号来分隔.最好的方法就是将当前目录(.)设为目录列表的第一个选择.在存放交换文件时首先放在列表的第一项指的目录处,这样交换文件首先会被存放在当前目录下.
如果我们已经做许多的改动,我们想保存我们当前所做工作,我们可以使用下面的命令:
:write
但是这个命令是用我们当前所做的改动来重写已经存的文件,与其相关的一个命令是
:preserve
这个命令是将我们所做工作存入交换文件,而原始的保持不变,直到我们用:write 或是 ZZ 命令退出时才会被重写.在这样的情况下,如果 Vim 遇到意外,我们可以用交换文件来修复我们所做的工作,哪怕是原始文件已经丢失.如果没有用这个命令,我们就不得不同时修复原始文件和交换文件.
我们不仅可以在启动 vim 时修复文件,还可以用下面的命令来修复文件:
:recover file.txt
这个命令与下面的有着同样的作用:
$ vim -r file.txt
如果我们用:recover 命令试图修复我们正编辑的文件则会返回错误.如果没有指定文件名,则默认的当前缓冲区中的文件.如果我们想要放弃我们所做的修改并试图修复时,我们可以使用下面的命令:
:recover! file.txt

Vim 大量的工作是通过命令行的方式来完成,这样的命令行方式对于则接触 Vim 新手来说也许会觉得难于操作,可是当我们习惯了这样的工作方式后,我们就不得不佩服 Vim 的强大功能,正是这些命令使得我们可以高效的来完成我们的工作.而在 Vim 当中还有一些其他的命令.
命令:ascii 或者是 ga 可以用 ASCII 码和八进制及十六进制打印出来.当我们编辑多字节文件时,例如说我们的汉语,这个命令就可以打印所有的字节.
命令 countgo 可以到达当前文件中由 count 所字的字符数的位置.例如命令 3go 就可以到达文件中的第三个字符处.而命令 gCTRL-G 可显示出当前文件中的字符数的信息,同是显示出当前行,当前列以及其他的一些信息.
而命令:goto offset 可以将光标置于由 offset 所指定的字符处.而命令 gg 则与我们以前见到过的 G 命令相类似,他也可以到达由 count 所指定的行.例如 5gg 可以到达第 5 行.gg 命令与 G 命令所不同的地方只是在没有指定 count 值时前者回到第一行,而后者要到最后一行.
而命令 CTRL-L 可以起到重画屏幕的作用.这个命令在我们使用终端窗口或是在屏幕上存在着一些系统信息时显得尤为有用.而命令:sleep time 可以使得我们的 Vim 编辑器在指定的时间不做任何事情(Sleep 嘛:-)).在这个命令中如果时间是以 m 结尾则是指的微秒.这个命令在我们想暂停执行宏时显得更为有用.而命令 count gs 命令也有同样的作用.
在大多数的终端上,CTRL-S 可以停止输出,如果我们要重新启动他,则要 CTRL-Q 命令.这两个命令并不是 Vim 命令中的一部分.为了避免键盘的冲突,我们在 Vim 中并不用这两个命令.我们最好也不要试着用:map 命令将一些功能和这两个键进行绑定,因为在执行这两个命令时是键盘得到命令而不会到达 Vim.
如果我们是在 UNIX 或是 LINUX 系统的终端模式下进行工作,我们可以用命令 CTRL-Z 来中止我们的正常编辑状态.如果我们要继续编辑则要使用 Shell 命令 fg.而命令:suspend 也会有同样的作用.
在一般的情况下我们可以使用:help 或是 F1 键来显示帮助屏幕,从而得到一般的帮助信息.

我们可以使用 z height<CR>来调整当前窗口的高度.如果当前仅有一个窗口,这个命令只是影响到窗口中的行数.
我们在没有输入文件名的情况下启动 Vim 时会看到一个介绍的屏幕,这个屏幕在我们输入任一字符后消失,如果我们想再看到他,可以使用下面的命令:
:intro

Vim 是一个强大的文本编辑器,这个强大的文本编辑器的大部分工作是通过命令行的方式完成的.Vim 一系列的命令可以使得我们快速高效的完成我们的各种各样的工作.下面我们就来看一下在 Vim 的编辑中常出现的问题的一些解决办法.
在我们快速的输入文本的时候,很容易使得一些单词输入错误.例如我们本应输入的是 the,而我们却错输入成了 teh,这时我们可以通过简单的命令使得这 e 和 h 这两个字母交换一下位置来达我们改错的目的.我们可以将光标放在 e 上,然后输入命令 xp,x 命令删除掉字母 e,而 p 命令则是将他放在当前光标的后面,也就是 h 字母的后,从而就达到了改错的目的.
在我们进行文本编辑的时候有时要做一些文本替换的工作,例如是将文中所有的字符串 idiots 替换成 managers.这时我们可以使用下面的命令:
:1,$s/idiots/manages/g
这个是命令是以冒号(:)开头的,则表明这是一个 ex 模式的命令.所有的 ex 命令都指明了命令要作用的范围.在这种情况选定了当前的所有文本,从第一行到最后一行($).我们可以也可以用%来简单指整篇文章.而 s 命令是命令:substitute 的缩写.旧的文本放在前面,而新的文本则放在后面,g 标记则指明这是一个全局的替换,这样就不会出在一行中多交出现要替换的文本而只替换第一个的情况了.
但是有时我们希望在进行文本替换以前 Vim 可以向我们进行一些询问,然后由我们来决定下一步的操作.在这样的情况下我们可以按照下面的方法来做:
1 执行命令 1G 到文档的开始处.
2 执行命令/idiot 来查找文章中出 idiot 的地方
3 执行命令 cwmanager<ESC>,当我们执行 cw 命令时当前光标处的文本被删除并进入插入模式,这时我们就可以将我想要替换成的文本输入,并退回到命令状态.
4 执行命令 n 重复上一次的查找
5 执行命令.重复上一次的编辑操作,如果在这一步我们不想进行替换,我们可以跳过这一步,进行下一次的查找.
6 重复以上两步直到将文件中的所有字符 idiot 替换成 manager
我们还可以执行下面的命令来达到同样的作用:
:%s/idiot/manager/cg
在这个命令中我们是用%来指代文章中的所有的行,与上一个命令不同的地方是我们加入了标记 c,这个标记可以告诉 Vim 每一次在替换之前都要进行询问.
我们在进行文本编辑时想要进行文本的移动操作我们又该如何来做呢?这时我们可以按照下面的步骤来做:
1 将光标移到我们要移动的段落的开头部分
2 用命令 ma 在此处做上标记.
3 将光标移到这个段落的底部
4 执行命令 d'a 来删除刚才做了标记的文本.
5 将光标移到我们想放置文本的地方.
6 执行命令 p 将这段文本放在此处

这样我们就达到移动文本的目的.
我们可以按照下面的方法来做,可以达到同样的效果:
1 用光标放在第一行并将其标记为 a
2 用命令}移动到这个段落的底部,标记为 b
3 输入命令:'a,'b move 来移动文本.
老版本的 Vi 编辑器不能很好的来处理多文件.但是 Vim 在处理多文件上却显得优秀得多.我们有多种不同的方法在不同的文件之间进行文件拷贝.我们可以使用传统的 Vi 风格的命令,也可以使用 Vim 可视化模式.我们还可以利用系统的剪切板来进行不同文件间的文本拷贝,所有的这些方法都可以很好的来工作,采用哪一种方法这就要看我们个人的喜好了.
使用传统的 Vi 风格命令来在不同的窗口之间进行文本的拷贝可以按照如下的方法来做:
1 编辑第一文件
2 执行命令:split second_file 打开另一个窗口并开始编辑第二个文件
3 使用命令 CTRL-W p 回到含有原始文件的前一个窗口
4 将光标移动到要拷贝文本的第一行
5 用命令 ma 标记这一行
6 移动到要拷贝文本的最后一行
7 执行命令 y'a 来复制当前光标位置到所做标记之间的文本.
8 使用命令 CTRL-W p 回到将要放置文本的这个文件.
9 将光标移到将要插入文本的地方,复制的文本将会放到这一行的前面.
10 使用命令 P 命令将复制的文本粘贴到文件中.
(注:p 命令则是将文本放在光标所在行的后面)
用可视化模式在两个窗口中进行文本的拷贝可以按照如下的方法:
1 编辑第一个文件.
2 执行命令:split 开始编辑第二个文件.
3 使用命令 CTRL-W p 回到前一个包含有原始文件的窗口.
4 移动到将要复制文本的第一行.
5 执行命令 V 进入可视化模式.
6 移动到将要复制文本的最后一行,被选中的文本将会被高亮显示.
7 执行命令 y 复制选中的文本
8 使用命令 CTRL-W p 回到将要放置文本的文件中.
9 移动到将要插入文本的地方,所复制的文本将会被放置在光标所在行的前面.
10 使用命令 P 来放置所复制的文本.
在不同的 vim 程序间实现在文本的拷贝可以照如下的方法:
1 编辑第一个文件.
2 启动 Vim 程序并编辑第二个文件
3 回到含有第一个文件的窗口.
4 移到要复制文本的第一行.
5 执行 V 命令进入可视化模式.
6 移到要复制文本的最后一行,选中的文本将会被高这显示.
7 使用命令"*y 命令将文本复制到系统剪切板.
8 回到另一个窗口.
9 移动到将要放置复制文本的地方,复制的文本将会被放到当前光标的前面.
10 执行命令"*P 将复制的文本放在这个文件中.
(注:这个方法似乎只在 Gvim 中有效)

也许我们经常会编辑一些文件,其中含有一个名字的列表,这时我们希望做到的是将这个名字列表按照一定的顺序进行排列.例如我们可以按照字母的顺序进行排列,可是按照 ASCII 的顺序进行排列.我们可以按照下面的方法进行:

1 将光标移到要排列的内容的第一行.

2 使用命令 ma 进行标记.

3 将光标移到要排序的内容的底部.

4

执行命令!'asort 进行排序.!命令告诉 Vim 通过 UNIX 命令来执行.'a 则是告诉 Vim 这个命令作用的范围.

我们还可以按照下面的方法进行排序:

1 将光标移到要排序内容的第一行.

2 执行命令 V 进入可视化模式.

3 移到光标到要排序内容的底部,这时选择的文本将会被高亮显示.

4 执行命令!sort 进行排序.

Vim 编辑器是一个程序员开发给程序员的编辑器.我们可以用这个编辑器在 C 或是 C++程序文件中进行函数的定位.我们要想使用这个功能,我们首先要生成一个名为 tags 的文件,在这个文件中含 C 或是 C++程序文件中所有函数的信息.我们可以使用下面的命令来当前我们工作的目录下的所有 C 程序生成一个 tags 文件:

$ ctags *.c

如果是对于 C++文件,我们可以使用下面的命令来生成:

$ ctags *.cpp

如果我们是使用其他的扩展名,我们可以使用相应的扩展名,而不一定非要使用 C 或是 C++的扩展名.

在我们生成这个文件以后,如果我们要利用这个文件来编辑我们的程序文件,这样 Vim 就会查找相应的文件并会在函数中进行定位,例如我们要编辑的文件为 write_file,我们可以使用下面的命令来开始我们的编辑工作:

$ gvim -t write_file

假如我们正在看一个名为 write_file 的函数,而在这个函数中调用了函数 setup_data,而我们又想要知道这个函数的详细内容,这时我们可以将光标位在这个函数的开头部分,然后按下 CTRL-],这样 Vim 就会跳到这个函数定义的地方.哪怕是我们要查找的函数在其他的文件中,Vim 也可以为我们进行精确的定位.

如果我们编辑了当前文件在没有保存的情况下使用了这个命令,那么 Vim 会给出警告信息,并且会忽略这个命令.

有许多和标记函数相关的命令可以使得我们在所标记的函数中进行向前或是向后的跳转和搜索,还可以打开一个新窗口并将被调用的函数在新窗口中打开.

我们在编写程序的时候常常会在程序的开头部分写上一个注释的边框,在其中写一些表明程序用途等等的信息.在 Vim 我们可以利用在~/.vimrc 这个初始化文会写上一些命令来快速的完成这样的工作.例如我们可以其中加入下面的内容:

:ab #b /**********

:ab #e <space>**********/

这样就在 Vim 当中定义的一个简写的标记,在我们要写注释时,只要输入#b<Enter>就可以了.

而我们要输出下面的注释只要输入#e<Enter>就可以了.这样的命令对于那些每天要有大量程序写的朋友们来说是不是一个巨大的帮助呢!:)

我们还可以利用 vim 来读 man 帮助手册页,但是用这样的方法并不是太好,因为 man 显示的结果中

一些下划线在 Vim 中显示会有一些困难.为了去除这样的字符,我们可以使用标准的 UNIX 命令 ul -i.这样就会去除那些很难阅读的控制字符.例如我们可以用下面的命令来读 date 的 man 手册页:

$ man date | ul -i | vi -

我们还要用这样的技巧来使用 Vim:

:%s/.b//g

这个命令是告诉 Vim 移除那些退格符(b),从而使用文件更易读.

位于文件行后面的空格符或是制表符有时是没有用,而他们的存在也使得文件看起为不是太舒服,去除这些多余的符号我们可以使用下面命令:

:%s/s*$//

冒号是表明进入命令模式,所有的命令模式都要指明命令作用的范围,在这里我们是指整个文件(%),这个命令是使得 Vim 将文件行末的空白符(s)重复 0 次或是多数(*).

假如我们正在编辑一个文件,而且在这个文件中我们做了许多的改动.这是一个很重要的文件,我们不希望因为偶然的改动而造成损失,这时我们可以将这个文件进行写保护.当我们用 Vim 来编辑一个有写保护的文件时我们并不会得到警告或是只有很少的警告,但是当我们想保存退出出 Vim 会给出错误信息,指出这是一个写保护的文件,并且 Vim 并不会退出.这时我们要怎么办呢?我们可以使用命令:w!强行保存或是使用:w otherfilename 来以另一个文件名进行保存.

在 UNIX 或是 Linux 系统中我们可以使用 Vim 和 grep 命令组合来编辑一些含有同一个给定单词的文件.这对我们编写程序的朋友们来说是有着极大的帮助,因为有时我们也许可以希望查看或是编辑含有同一个变量的文件.例如我们所有含有 frame_counter 的 C 程序文件,这时我们就可以使用下面的命令:

$ vim `grep -l 'frame_counter' *.c`

grep 命令是在所有的文件中查找含有指定单词的文件,因为我们指定了-l 选项,所以这个命令只是会列出含这个单词的文件而不会打印出这一行的信息.这样 Vim 就会打开 grep 命令列出的文件进行编辑,而我们也就可以使用:n 或是:rewind 命令在这些文件中进行跳转.

我们还可以在 Vim 内部使用:grep 命令来查找我们想要的字符串,例如我们要在所有的 C 程序文件中查找含有 error_string 的字符串,我们可以使用下面的命令:

:grep error_string *.c

但是这个命令是使用外部的 UNIX 或是 Linux 命令,而且 Vim 会打开第一个匹配的文件,并将光标置于第一个查找到的字符串处.

在 Vim 编辑器有着相当丰富的命令和设置.有许多的命令设置可以说我们是根本就不会用到的.下面的只是简要的介绍一些这样的命令和设置的方法.

命令 cscope 可以检查 C 或是 C++程序文件并产生一个含有程序中函数和变量信息的数据库.我们可以使用 Cscope 程序来查看这个数据库从而可以得函数定义和使用的一些信息.Cscope 可以从下处得到:

http://cscope.soureforge.net

我们可以使用下面的命令来得一些详细的帮助信息:

:help cscope

Cscope 一些相关的命令如下:

:cs arguments

:cscope argument

处理一些与 Cscope 程序相关联的活动

:cstag procedure

定位到 Cscope 数据库中名为 procedure 的函数标记处
:set csprg=program
:set cscopeprg=program
定义 CScope 程序名(默认为 Cscope)
:set cst
:set cscopetag
:set nocst
:set nocscopetag
如果设置了 cscopetag 选项就可以在使用 Cscope 数据中使用命令(:tags,CTRL-])来浏览标记
:set csto=flag
:set cscopetagorder=flag
这个选项设置了 CScope 标记查询命令的查询顺序.如果是默认的 0,那么会先查 Cscope 数据库,然后是标记;如果是 1,则首先查在标记中查找.
:set csverb
:set cscopeverbose
:set nocsverb
:set nocopeverbose
如果设置了 cscopeverbose 选项,那么在 Vim 查找 Cscope 数据库并且查找失败时给出错误信息,而 Vim 默认的设置为 nocscopeverbose
OLE 系统是运行在 Windows 下面的程序彼此之间进行通信的方法.而 Vim 编辑器可以来扮演一个 OLE 服务器的角色.这就意味着我们可以来编写 Window 程序并与 Vim 通信.我们可以用下面的命令来得到更详细的帮助信息:
:help ole-interface
与 Perl 的接口可以使得我们在 Vim 中执行 perl 命令,同时可以提供给 Perl 程序一个接口,使得他可以访问 Vim 的功能.我们可以使用下面的命令来得一些更详细的帮助信息:
:help perl
Perl 的一些接口命令如下:
:pe command
:perl command　　执行单一的 perl 命令
:range perld command
:range perldo command　　在几行上执行 perl 命令
与 Perl 相类似是 Python.我们可以用命令:help python 得到更详细的帮助信息.
Python 的一些接口命令如下:
:range py statement
:range python statement　　执行单一的 Python 命令描述
:range pyf file
:range pyfile file　　执行文件中的 Python 程序
Sniff+的一些接口命令如下:
:sni command
:sniff command　　通过与 Sniff+的接口来执行命令.如果没有提供命令,则会显示出当前连接信息.
同样我们可以通过命令:help sniff 来得到 Vim 提供的帮助信息.
Tcl 的一些接口命令:
:tc command

:tcl command　　执行单一的 Tcl 命令
:range tcld command
:range tcldo command
在所列出的行中每行执行一次 Tcl 命令
:tclf file
:tclfile file　　在给定的文件中执行 Tcl 脚本
Vim 编辑器可以处理各种不同的语言.在这里列出我们一些常用字的用其他语言来编辑文件的命令.如果我们要想得到一些更详细的说明,则要查阅 Vim 文档或是我们的系统文档了.
下面的是一个固定的常用的命令:
<F8>　　在从左到右和从右到左两种模式间切换
:set rl
:set rightleft
:set norl
:set norightleft
通过这些选项的设置我们可以选择是从左到右的模式还是从右到左的模式
:set ari
:set allowrevins
:set noari
:set noallowrevins
通过设置这些选项我们可以通过 CTRL-_来设置 revins 选项.这个可以使得我们选择语言的输入的是从左到右还由右至左.
:set ri
:set revins
:set nori
:set norevins
通过这些选项设置,我们可以选择在插入模式下是由左至右还是由右至左.如果设置了 allowrevins 选项我们可通过 CTRL-_来在这几个选项间进行切换.
:set gfs=f1,f2
:set guifontset=f1,f2
定义英语使用 f1 字体,而另一种语言使用 f2 字体
这个选项只有在我们编译 Vim 编辑器时允许进行字体设置并且是只在 UNIX 系统才可以正常的工作.
:set lmap=ch1ch2,ch1ch2
:set langmap=ch1ch2,ch1ch2
为外文本设置键盘映射
Vim 编辑器对汉字的输入支持由左到右,由右到左几种模式.他还支持传统的中文和简体中文.与中文相关的命令如下:
:set fe=encodning
:set fileencoding=encoding
设置文件的编码.对于中文这个选项可以是对于传统中文的 taiwan 或是对于简体中文的 pre.
如果我们要编辑 Vim 编辑器时打开了 Farsi 的支持,我们就可以在用这种语言来编辑文件了.可以用-F 选项在启动 Vim 时进入 Farsi 模式:
$ vim -F file.txt
我们可以得到更详细的信息:

:help farsi
与 Farsi 相关的命令如下:
:set fk
:set fkmap
:set nofk
:set nofkmap
通过这些选项的设置我们可以告诉 Vim 我们正在使用 Farsi 键盘
:set akm
:set altkeymap
:set noakm
:set noaltkeymap
通过设置这些选项我们可以告诉 vim 编辑器键盘映射是 Farsi 还是 Hebrew
CTRL-_ 在 Farsi 和正常模式下进行切换
<F9> 在标准的 ISIP-3342 编码和扩展的 ISIR-3342 编码之间进行切换
Hebrew 是由右到左的另一种语言.采用 Hebrew 模式进行编辑可以使用下面的命令:
$ vim -H file.txt
:help hebrew 可以使得我们得到更多的帮助信息.
与 Hebrew 相关的一些命令:
:set hk
:set hkmap
:set nohk
:set nohkmap
通过这些选项我们可以打开或是关闭 Hebrew 键盘映射
:set hkp
:set hkmappp
:set nohkp
:set nohkmapp
通过这些选项我们可以告诉 Vim 编辑器我们正在使用 Hebrew 键盘还是标准的英语键盘(默认为 nohkmapp,即标准的英语键盘)
CTRL-_ 这个命令可以使得我们在 Hebrew 或是正常插入状态下进行切换
:set akm
:set altkeymap
:set noakm
:set noaltkeymap
如果设置了 altkeymap 选项,那么与其交换的键盘映射为 Farsi.如果设置了 noaltkeymap 选项,那么则是 Hebrew 键盘映射.(默认为 noaltkeymap)
Vim 编辑器还可以支持日文,与日文相关的一些命令如下:
:set fe=japan
:set fileencoding-japan
告诉 Vim 编辑器当前文件采用日文编码.
我们可以通过命令:help hangul 得到更多的韩文帮助信息.与韩文相关的命令如下:
:set fe=korea
:set fileencoding=korea
告诉 Vim 编辑器当前的文本采用韩文的编码.

我们还可以使用 Vim 编辑器来编辑二进制文件,相关的命令如下:
:set bin
:set binary
:set nobin
:set nobianary
如果我们设置了 insertmode 选项,那么 Vim 默认的便为插入模式.我们可以通过命令 CTRL-O 切换到正常模式.相关的命令如下:
:set im
:set insertmode
:set noim
:set noinsertmode
CTRL-L　　如果设置了 insertmode 选项,则保留这种设置

我们在 Vim 编辑器的学习使用(二)曾学过一些基本的编辑命令,有了这样的一些基本的编辑命令,我们可以完成绝大多数的编辑任务.在这一次的学习中我们将会接触到更多的更深一些的东西.
Vim 编辑器有着各种不同的命令可以使得我们任意的移到一个单词的开头或是结尾.但是有我们却可以通过 Vim 的一些选项的内容来自定义一个单词的定义.w 命令可以使得光标向前移到一个单词.而命令 e 也是向前移到一个单词,但是这个命令是将光标定位在一个单词的结尾处.
而命令 ge 则是向后移到一个单词到达前一个单词的结尾处.
那么怎么样来定义一个单词呢?我们也许都知道单词只是一系列的字母的组合.然而在 C 程序中 size56 却会被认为是一个单词,因为在 C 程序中我们是通过字母,数字和下划线来定义一个单词的.但是在 LISP 程序中我们可以在变量名中使用-,这时他们会认为 total-size 是一个单词,而在 C 程序中这却会被认为是两个单词.我们如何来解决这样的冲突呢?Vim 的解决办法是产生一个选项来定义哪些是一个单词中的,而哪些不是.例如下面的命令定义了属于一个单词中的字母:
:set iskeyword=specification
查看当前选项的值我们可以使用下面的命令:
:set iskeyword?
下面的是一般的值:
iskeyword=@,48-57,_,192_255
在这些值中间中用逗号来分隔的.
如果我们想要单词中的字母是专一的元音,我们可以使用下面的命令:
:set iskeyword=a,e,i,o,u
我们还可以使用横线来指定字母的范围.如果要指定所有的小写字母,我们可以用下面的命令:
:set iskeyword=a-z
对于那些不能直接指定的字符我们可以使用十进制的数字来表示.如果我们要指定小写字母和下划线为一个单词,我们可以使用下面的命令:
:set iskeyword=a-z,45
(短横线的十进制数字表示为 45)
字母@指代 C 函数 isalpha()返回值为真是的所有的字符.这个要取决于我们所使用的 C 编译器和我们编译 Vim 所使用的操作系统)
排除某一个字符我们可以在这个字符前加上一个前缀^.例如我们可以使用下面的命令来定义一个单词,这个单词可以由除了 q 以外的小写字符组成,也就是说在由空格分格的字符串组成,q 为一

个单词,而其他的字符串的组合为一个单词:
:set iskeyword=@,^q
而字符@则是由@-@所指代.
iskeyword 选项一些特殊字符如下:
a 字符 a
a-z 所有由 a 到 z 的字符
45 十进制数字 45(短横线-)
@ 由函数 isalpha()所定义的所有字符
@-@ 字符@
^x 除了 x 以外的字符
^a-c 除了 a 以处的到 c 的字符,即 b 和 c
命令 iskeyword 可以简记为 isk
iskeyword 选项可以控制哪些字符可以是一个单词而哪些不是.下列的一些相似的命令可以控制其他类型的字符:
isfname 文件名
isident 定义
isprint 打印字符
选项 isfname 在使用命令 gf 时会用到,这个命令将会编辑以光标下的单词为文件名的文件.
选项 isident 在使用命令[d 时会用到,这个命令将会查找以光标下的单词为名称的宏定义.
选项 isprint 定义了哪些字符可以显示在屏幕上.但是对这个我们要小心,如果我们弄错了,屏幕就会被弄得一团糟.这个选项还可以用特定的查找命令 p,这代表可打印的字符.
也许到了现在我们已经明白了什么是单词(words).而 Vim 编辑还有一些命令影响到 WORDS.虽然这只是大小的不同,但是他们却代到了两种不同的事物.word 是指由 iskeyword 选项定义的字符串,而 WORD 则是指没有空白符的字符.在这样的观点上,that-all 是两个单词(word),但是却是一个 WORD.
W 命令是向前移动 WORDS,而 B 命令是向后移动 WORDS.
与 WORD 相关的一些命令如下:
[count]B 向后移动 count 个 WORDS
[count]E 向前移动 count 个 WORDS,并且将光标置于 WORD 的末尾.
[count]gE 向后移动 count 个 WORDS,并且将光标置于 WORD 的末尾.
[count]W 向前移动 count 个 OWRDS.
我们可以使用命令将光标移动到一个句子中第一个没有空白符的字符处.如果我们要到达一行的开始处,我们可以命令用 0 命令.
我们可以使用命令 fx 在当前行光标的后面查找字符 x.如果我们要重复这个查找操作可以使用命令;.就像大多数的 Vim 命令一样这个命令可以用数字来指明查找的次数.;命令是按照前一次 f 或是 F 的查找方向继续查找.如果我们要向相反的方向查找我们可以使用命令,.
命令-可以向上移动到第一个没有空白符的前一行处.我们可以指定参数来移到几行.
命令+可以向下移到到第一个没有空白符的后一行处,我们可以指定参数来移到几行.
命令_可以移动到当前行第一个没有空白符的字符处.
还有一些命令可以使得我们在屏幕的不同地方进行移动.H 命令可以将光标移到屏幕的顶端.如果指定的数字参数则可以移动到从屏幕顶端算起的由数字所指定的行处.与 H 命令相类似是 L 命令,所不同的只是这个命令移动到屏幕的底端.M 命令可以将光标移到屏幕的中间位置.
Vim 编辑器还可以记录你曾经到过的地方,并且可以使得我们回到前一次到过的地方.例如我们在编辑一个文件的时候执行了下面的命令从而到过不同的行处:

1G 到第一行
10G 到第十行
20G 第第二十行
现在我们执行下面的命令:
:jumps
这样我们就会看到一个我们曾到过的行数的列表.在这个列表中>指的是这个列表中的当前项.这样在我们处在命令模式下我们就可以使用命令 CTRL-O 跳回一行.这样>所指的就会上移一项.而命令 CTRL-I 或是<TAB>可以使得我们在这个列表中向下跳转.应用这样的命令我们就可以实现在文件中进行快速的浏览和跳转.
在通常的情况下,当光标位于一行的开头是结尾的时候 Vim 是不可以移动光标的.但是我们可以通过设置 whichwrap 选项来控制光标是否可以越过一行的结尾并且 Vim 处于哪种模式.whichwrap 选项一些可能的值如下表所示:

Character	Command	Mode(s)
b	<BS>	正常模式或是可视模式
s	<Space>	正常模式或是可视模式
h	h	正常模式或是可视模式
l	l	正常模式或是可视模式
<	左移	正常模式或是可视模式
>	右移	正常模式或是可视模式
~	~	正常模式

我们可以通过命令 CTRL-G 来使用 Vim 在屏幕的下端显示出我们所在的位置的一些信息.然而只要我们来发问我们就可以得到一些更详细的信息.为了得到更详细的信息,我们可以在 CTRL-G 命令加上一个数字参数.这个数字越大我们得到的信息就越详细.例如命令 1CTRL-G 会告诉我们文件的全路径.而命令 2CTRL-G 会同时显示缓冲区的数字标号.而命令 gCTRL-G 则可以显示当前光标所在的行号,列号以及文章的字符数等一些详细的信息.
我们在前面曾讨论过,CTRL-U 命令可以使得 vim 编辑器向上翻滚半屏,但是我们可通过设置 scroll 选项来控制这个命令翻滚的行数.例如下面的命令可以使得 Vim 一次翻滚 10 行:
:set scroll=10
我们也可以通过防变 CTRL-U 命令的参数来改变翻滚的行数.例如命令 2CTRL-U 可以使得 Vim 编辑器一次向上翻滚两行,直到有命令来翻滚的大小为止.
如果我们要一次翻滚一行我们可以使用 CTRL-Y 命令.当然这个命令也可以在前面设置参数来控制翻滚的行数.而 CTRL-B 命令则是一次翻滚一屏.
与向上翻滚的命令相对是向下翻滚的命令,这样的命令有如下的一些:
CTRL-D 向下翻滚.这个数值我们可以通过 scroll 的值来控制.
CTRL-E 一次向下翻滚一行.
CTRL-F 一次向下翻滚一屏.
当光标到达窗口的上端或是下端的时候窗口要发生滚动.我们可以通过设置 scrolljump 选项来控制这个翻滚数值的大小.默认情况下为 1,当然了我们也可以将其改我们希望的样子.如下面的命令将翻滚量设为 5:
:set scrolljump=5
与其相类似的就是 sidescroll 选项,所不同是后者来控制水平的翻滚.
通常情况下,窗口的翻滚是当光标到达窗口的顶部或是底部时才发生的,我们也可以通过 scrolloff 选项来控制光标与顶部或是底部有多少距离时发生:
:set scrolloff=3

这个命令将其为 3,当光标与顶部距离为三行时发生翻滚,且翻滚后光标与底部相距三行.
也许我们在编辑文件的过程中有时希望将指定的一行放在屏幕顶端.当然了这样的问题我们可以使用 CTRL-E 或是 CTRL-Y 命令来一行一行的移动直支满期足要求为止.我们还可以将光标放在指定的行上,然后输入 z<Enter>.我们这一行就会出现在屏幕的顶端了.
这个命令在没有任何参数的情况是将当前行置于屏幕的顶端,我们还可以指定参数,这样就可以将指定的行置于屏幕顶端了.例如命令 8z<Enter>就是将第八行置于屏幕顶端.这个命令不仅可以将指定的行置于顶端,还可以将光标移动到本行第一个没有空白符(non-blank)的字符处.如果我们要将光标保持在一行的某一位置不变我们可以使用命令 zt,这样在这一行的位置发生变化,光标的位置也会保持不变.
如果我们要将指定的一行放在屏幕的底部,我们可以使用命令 z-或是 zb.所不同的只是前者是将光标放在这一行中第一个没有空白符的字符,而后者是保持光标的位置不变.
命令 zz 或是 z.可以将指定的行放在屏幕的中部.这两个命令的不同就是前者保持光标的位置不变,而后者是将光标置于第一个没有空白符的字符处.
D 命令可以将光标所在处到这一行的结尾的文字全部删掉.我们也可以在这个命令前面加上数字做为前缀,这样在执行这个命令以后,Vim 不仅会将光标到光标所在行结尾处的字符全部删除,而且会再删除这一行以下的 count(数字前缀)-1 行文本.
与 D 命令相类似的是 C 命令.所不同的仅是 C 命令在删除文本以后会进入插入模式使得我们可以接着进行文本的编辑.
s 命令可以删除单个的字符并进入插入状态.如果我们在前面加上数字做为前缀,那么 Vim 就会删除数字所指定的那么多的字符,然后进入插入状态.
而 S 命令是删除当前行然后进入插入状态.我们可以指定数字 count 做为前缀,这样 Vim 就会删除 count 个行,然后进入插入状态.这个命令与 C 命令的不同之处只是 S 命令作用整个行,而 C 命令仅是光标所处的位置到行末.
x 命令是删除当前光标下的字符,如果指定 count 作为参数,则是向右查找 count 个字符并删除,而 X 命令是删除当前光标前的一个字行,如果指定 count 作为参数,则是向左查找 count 个字符并删除.
在进入插入模式时我们可以使用 i 或是 I 命令.i 是在当前光标处开始插入字符,而 I 则是一行的开头部开始插入字符.(所谓的开头是指第一个没有空白符的字符处).如果我们要在一行中的第一个字符处开始插入我们可以使用 gI 命令(不论有没有空白符).a 命令是在当前光标的后开始插入,而 A 命令与 a 命令相类似,只是他是在一行的后面开始插入状态.
Vim 还可以对于文本进行简单的算术运算.命令 CTRL-A 可以将当前光标下的数字加 1,我们还可以在前面指定参数,这样就可以指定的数字加在光标下的数字上了.如果这个数字是以 0 开头,那么 Vim 就会认为这是一个八进制的数.例如 0177 在我们执行这个命令后就会变为 0200.如果一个是以 0x 或是 0X 开头的 Vim 就会这是一个十六进制的数,例如 0x1E 在我们执行这个命令后就会成为 0x1F.与 CTRL-A 命令相类似的是 CTRL-X 命令.所不同的只是这个命令会使得当前光标下面的数字减 1.Vim 是一个精巧的编辑器,他可以很好的来处理十进制,八进制,十六进制的数字计算问题.
在默认的情况下,Vim 可以识别出八进制和十六进制的数字.我们可以通过 nrformats 来控制 Vim 所识别的数字形式.如果要使得 Vim 识别出十进制和八进制的数字,我们可以使用下面的命令:
:set nrformats=""
如果我们要使 Vim 只识别出八进制数,我们可使用下面的命令:
:set nrformats=octal
默认的情况下我们 Vim 可以识别出十进制,八进制,十六进制数字:
:set nrformats=octal,hex
我们可以使用 J 命令将当前行和下一行合并为一行.在合并后 Vim 会加入一空格来分隔这两行.如

果我们不希望用空格来分隔,我们可以使用 gJ 命令.这个命令与 J 命令类似,只是他不会加入空格来分隔这两行.
R 命令可以使得 Vim 进入替换模式.在这种模式下,我们输入的每一个字符都会替换光标下面的字符,直到我们按<ESC>退出为止.我们还可以指定数字作为参数来指明这条命令所要执行的次数.(注:这里我做的结果是 r 命令可以指定参数来指明执行次数,而 R 命令则不成)
当我们在替换的文本中有<Tab>键时,替换命令就会出现问题.因为他也会将<Tab>替换为相应的字符,这样就影响了我们文本的缩进.在这样的情况下我们可以使用 gr 命令来进行替换.如果光标下的字符是 Tab 的一部分,那么就会跳过而替换别的字符,这样就不会影响我们文本的缩进了.我们还可以使用 gR 命令进入虚替换模式(virtual replace mode),这时我们输入的字符就会替换屏幕空白处的一个字符.
我们可以通过执行命令 CTRL-K character1 character2 来插入一个特殊字符.我们也可以用下面的命令为定义们的自己的特殊字符:
:digraphs character1 character2 number
这就是告诉 Vim 编辑器当我们输入 CTRL-K character1 character2 命令以后要插入数字表示为 number 的字符.
如果我们的工作要我们插入很多的特殊字符,我们可以用下面的命令打开 digraph 选项:
:set digraph
关闭这个选项的命令为:
:set nodigraph
(注:这部分内容部是看不明白,也做不出结果)
命令~可以改变字母的大小写.~命令的执行结果是由选项 tildeop 的值来控制的.如果我们没有设置这个选项,那个这个命令就像正常一样的动作:
:set notildeop
但是如果我们设置了 tildeop 选项,那么这个命令的语法就变成了~motion 的形式:
:set tildeop
例如下面的句子:
this is a test
如果我们没设置这个选项,那么在我们执行命令时就会实单个字符的大小转换.但是在我们设置了这个选项以后,我们将光标放在第一 t 上并执行~ft 的结果则为
THIS IS A Test
这个命令会使得当前光标以后第一 t 和光标间的字符全部转换为大写.如果在这个句子中还有小写的字符,那么这个命令还可以执行第二次.直到句子中的字符全部为大写为止.
这时的命令还可以指定字符转换的个数和方向.例如命令 3~l 是从当前字符开始向右 3 个字符进行大小写的转换.与这个命令方式相类似的是 g~motion 格式的命令,所不同的仅是后者不依赖于 tildeop 选项的设置.命令 g~g~或是 g~~是进行整个一行的转换.
其他的一些大小写转换的命令还有:
gUmotion 命令是使得当前光标处到 motino 所指处的字符全部变为大写.而 gUU 或是 gUgU 则是作用在整个行上.如果指定了参数 count 则所指定的行都会发生变化.而 gumotion,guu,gugu 则是和上面所说的命令相类似,所不同的只是他们是将字符变为小写.
我们在用 Vim 进行编辑的时候可以进行多次的撤销命令,这个次数是由选项 undolevels 来指定的.如果我们要将这个值设为 5000,那么我们可以用下面命令来做到:
:set undolevels=5000
命令 ZQ 是命令:q!的另一种说法.这个会放弃我们所做更改然后退出.
命令:write 则是保存文件.命令:quit 是退出 Vim.我们可以使用一个缩写来达到保存退出的目的::wq

这个命令可以用参数来指定保存退出时使用的文件名,如:
:wq filename
如果这个文件存在且是一个只读文件时我们会得到一些错误信息,当然了我们可以强制执行这个命令:
:wq! filename
我们还可以将指定的行进行保存:
:1,10 wq filename
与命令:wq 相类似的命令是:xit

Vim 编辑器强大的搜索引擎可以使得我们快速的执行各种类型的查找,从而大的方便了我们的编辑工作,使得我们的编辑工作更加快速和高效.
我们在进行查找的过程中可以打开高亮显示选项,这样在我们找到我们想要的字符后,Vim 就会将字符串进行高亮显示,我们也可以很方便的知道我们要找的字符串在哪里.我们可以使用下面的命令来打开高亮显示选项:
:set hlsearch
关闭这个选项的命令为:
:set nohlsearch
在默认的情况下 Vim 编辑器是很敏感的,也就是 Vim 编辑器可以很好的来区分一个单词的大小写,从而可以准确的查找得到我们想要的字符串.例如如果一个文件中有这样的几个字符串:
include,INCLUDE,Include.当我们使用命令/include 来查找字符串时只有 include 字符会被高亮显示.
但是如果我们打开 ignorecase 选项后再执行这个命令结果就不一了,这时所有的类似的字符都会被高亮显示.打开这个选项的命令为:
:set ignorecase
但是这样的查找结果并不是我们想要的,也不是我们所希望发生的,因而我们常用下面的命令选项设置:
:set noignorecase
这样 Vim 就可以正确的查找我们想要的字符串.
如果我们设置了 ignorecase 选项后,我们想要查找字符串 word,而匹配的则可能是 word,Word,WORD.如果我们要查找字符串 WORD,匹配的结里也是一样的.但是如果我们设置了以下的两项后的执行结果就会变得不一样了:
;set ignorecase
:set smartcase
经过这样的设置以后,如果我们输入的是小写字符,那么 Vim 就会匹配各种可能的组合,这时与设置了 ignorecase 的情况相同,但是如果我们在输入中有一个大写字符,那么这时的查找就变成了精确查找了,与设置了 noignorecase 的情况相同.
在默认的情下,我们输入要查找的字符串,vim 是从当前光标处向前查找,直到文件的结尾,如果没有打到,那么就会从文件的开头开始查找,直到光标所处的位置.我们可以通过设置选项 nowrapscan 来禁止这种循环查找的方式,这个命令如下:
:set nowrapscan
这样以后如果已经查找到文件的底部时就会在 Vim 底部显示出一条错误信息.如果我们想要回到正常的状态,我们可以使用下面的命令;
:set wrapscan

如果我们正处在一个很长的查找过短中而我们想要停止这一查找开始我们新的工作,这时我们可以使用 CTRL-C 命令,如果是在 Windows 系统上则要使用 CTRL-BREAK 命令.
我们在编辑文件的过程中还可以使用立时查找的命令.如果我们想快速查找当前光标下的字符串,我们可以使用命令*,这个命令可以向前查找与当前光标下的单词精确匹配的字符串.而命令#则向前查找与当前光标下的字符精确匹配的字符串.换句话说如果当前光标下的字符串为 word,在执行*命令查找时并不会与 Word 相匹配.与这个立时查找命令相类似的就是 g*命令.这也是一个立时查找的命令,只不过他不会进行严格的整字匹配,如果用这个命令来查找 word 那么就有可能和 Word 相匹配.而 g#命令与其相同,只不过他是向相反的方向进行查找匹配.
在默认的情况下,在查找时 Vim 会将光标放在第一个匹配的结果的开始处.我们也可以指定查找结束后光标所处的位置.对于向前查找的命令我们可以斜线后用数字来指明光标所处的位置,如下面的命令:
/set/2
这个命令会在查找结束后将光标入在第一个 set 字符串后第二行的开始处.
在这个命令中这个数字可以是正数也可以是负数.如果仅是一个简单的数字,光标会被放在第一个匹配字符串处后或是前的数字所指定的行的开始处.正是向后,负数是向前.如果斜线后是 b 和数字的,那么在查找结束后,光标将会被置于第一个匹配字符串的开始处,然后向左或是右移动 n 个字符,这里的 n 即为数字所指定的数.如果为正数则是向右移动,如果是负数,则是向左移动.如下面的命令:
/set/b2
这个命令会使用 Vim 在查找结束后将光标放在第一个匹配字符的开始处,然后向右移动两个字符,也就是说最后光标会位于第一个匹配字符串中的 t 的位置.将 b 改为 s 也是一样的效果.
与参数 b 或是 s 相类似是 e 参数,这个参数会使得光标放在第一个匹配字符串的结尾处.同样我们也可以指定数来指是向右还是向左移动光标以及移动的字符数.如下面的命令:
/set/e
这个命令会使光标放在第一个匹配字符处的结尾处,在没有指定数字时是这个样子的.而下面命令:
/set/e2
这个命令是会将光标放在第一个匹配字符串的结尾处,然后向右移动 2 个字符.这里的数字如果是正数则向右移,如果为负数,则向左移.
例如下面的命令:
/set/e+2
这个命令是告诉 Vim 在查找 set 字符串结束后将光标放在第一个匹配字符串的结尾处,然后向右移动两个字符.在这里我们将这个数字称为偏移量.如果我们要重复上一次的查找但是需要不同的偏移量我们可以用下面的命令:
//5
不使用偏移量时我们可以指明一个空的偏移量,如:
//
例如下面的一些例子:
/set/e+2
向前搜索字符串 2,并将光标放在第一个匹配字符串的结尾处,然后向右移动 2 个字符
/ 重复前一次的查找,使用相同的偏移量.
// 重复前一次的查找,但是不使用偏移量.
我们还可以使用查找命令?来进行类似的查找,例如:
?set?b5

这个命令是告诉 Vimu 将光标放在最后一个匹配字符串的开头部分,然后向右移动 5 个字符
??-2 命令则继续前一次的查找命令,但是使用新的偏移量.
??命令是继续前一次的查找命令,但是不使用偏移量.
但是有一件事我们要清楚的就是我们在用偏移量进行光标定位时的查找是从当前光标所在处开始的,这会给我们带来一些麻烦.例如我们在执行命令/set/-2 时 Vim 会将光标放在第一匹配的字符串处,然后上移两行.当我们用 n 命令来重复上一次的查找时,我们会找到我们刚才找到过的那个字符串,然后再向上偏移两行.这个结果就是不论我们输入了多少的次 n 命令,我们仍是在原地踏步的,哪里也去不了.
Vim 使用通用的表达式(regular expressions)来进行逻辑查找.我们在以前讨论过用简单的字符串进行查找,但是这里我们将要看到的通用字符串查找要简单字符查找的功能强大得多.通过在我们的命令中使用通用表达式,我们可以查找任何一种字符类型,例如我们可以查找以 t 开头而以 ing 结尾的字尾串(通用表达式为<t[^]*ing>).然而这种强大的功能也是要付出一定的代价的.通用表达式是神秘的和简洁的.也许我们要花上很上的一段时间才会习惯这种查找方式,然后才能掌握这个强大的查找工具.
在学习这些通用表达式的查找的过程中我们最好将高亮显示这个选项打开,这样就可以使 Vim 高亮显示最后一次查找的匹配结果.打开高亮显示的命令为:
:set hlsearch
一个通用表达式是由一些元素组成的.这些元素是通用表达式中最小的匹配单位.一个元素可以是一个字符,例如 a,与字符 a 相匹配,或者是一个特殊字符,例如$,匹配一行的结束.还可以是其他的字符,例如<,是指一个单词的开始.
在通用表达式中,我们用<来匹配一个单词的开始,用>来匹配一个单词的结束.也就是说要将我们想要查找的字符串放在这两个中间.这样我们就可以精确的来查找我们想要查找的字符串,而不会有其他的一些匹配情况.而如果我们用简单字符串形式来查找,我们就会得到许多的匹配情况,甚至在一个单词中的组成部分也可以成为匹配情况.例如在文件中有 Californian,Unfortunately.如果用命令/for 来查找,那么就会找到这两个单词.而如果我们用通用表达式<for>来进行查找,则只会精确的查找到 for,而不会用其他的匹配情况.这时的命令形式如下:
/<for>
我们在进行查找的时候还可以使用一些修饰符来进行表达式的组合.例如修饰符*就可以表示一个字符可以匹配 0 次或是多次.换句话说,Vim 编辑器会进行尽可能多的匹配.所以通用表达式 te*可以匹配 te,tee,teee 等等.基于还可以匹配 t,为什么呢?因为在这里 e 可以匹配 0 次,所以就可以匹配 t.而修饰符+则表明这个字符可以匹配一次或是多次.所以表达式 te+可以匹配 te,tee,teee 等等.但是这一次这个表达式不可以再匹配 t 了,因为这里 e 最少要匹配一次.
最后一个修饰符=表示一个字符匹配 0 次或是一次.这就是说表达 te=可以匹配 t,te,但是不可以是 tee,虽然这个命令会匹配 tee 的前两个字符.
还有一些特殊的字符可以来匹配一定范围的字符.如 a 匹配一个字符,而 d 匹配任何数字.所以表达式 aaa 可以匹配任意三个字符.例如下面的命令可以查找任意四个数字:
/dddd
我们还可以用下面的命令来查找任意后带一个下划线的三个字符:
/aaa_
a 可以匹配所有的字符(小写的或是大写的).但是假如我们现在只要匹配元音字符又该如何来做呢?这时我们可以使用范围作用符[].范围作用符可以匹配一系字符中的一个.例如[aeiou]只匹配一个小写元音字符.所以表达式 t[aeiou]n 可以匹配 tan,ten,tin,ton,tun.
我们还可以通过短横线来在括号内指明字符的范围.例如[0-9]可以匹配 0 到 9 中的任一字符.
我们还可以组合其他的字符.例如[0-9aeiou]可以匹配任意一个数字或是小写的元音字符.

而修饰符^可以指代除本身以外的所有字符.
下面列出一些匹配的情况:
表达式　　匹配结果
one[-]way　　one-way
2^4　　2^4
2[^*]4　　2^4,2*4
如果我们要指找所有的大写字符又应如何来做呢?一个办法就是使用表达式[A-Z].还有一个办法就是我们可以使用预先定义的字符类.[:upper:]可以匹配大写字符.所以我们要指找大写字符也可以这样的来写:[[:upper:]].我们可以使用字符类来写出所有的字符:
[[:upper:][:lower:]].
在 Vim 中还有许多不同的字符类定义
我们还可以通过表达来指出一个字符重复的次数.这个的一般格式如下:
{minimum,maximum}
例如表达式 a{3,5}可以匹配 3 到 5 个 a.在默认的情况下 Vim 将会尽可能多的进行匹配.所以表达 a{3,5}最多可以匹配到 5 个 a.
在这个命令格式中最小次数可以省略,Vim 默认的情况下最小次数为 0.所以表达式 a{,5}可以匹配 0 到 5 个 a.最大次数也可以省略,在这种情况下 Vim 默认匹配无穷大.所以表达式 a{3,}最少可以匹配 3 个 a,最多是尽可能的多.
如果我们只指定一个数字,那么 Vim 就会精确的匹配相应的次数.例如 a{5}只会精确的匹配 5 次.
如果我们在数字前加了一个负号(-),那么 Vim 在查找时就会尽可能少的进行匹配.
例如 a{-3,5}匹配 3 到 5 个 a,但是会尽可能少的进行匹配.事实上这个表达式仅会匹配 3 个 a,哪怕是我们的文件中用 aaaaa,Vim 也只会尽可能少的进行匹配.
表达式 a{-3,}匹配三个或是更多个 a,并且尽可以少地的进行匹配.而表达式 a{-,5}可以匹配 0 到五个字符.表达式 a{-}可以匹配 0 到无穷大个字符.在通常情况下这个表达式匹配 0 个符,除非在他的后面还有字符或是表达式.例如[a-z]{-}x 将会匹配.cxcx 中的 cx.而表达式[a-z]*x 将会匹配整个 cxcx.最后表达式 a{-5}将会精确的匹配 5 个字符.
我们还可以使用运算符(和)定义一个组.例如表达式 a*b 可以匹配 b,ab,aab,aaab 等等.而表达式 a(XY)*b 可以匹配 ab,aXYb,aXYXYb,aXYXYXYb 等等.
我们还可以用或运算运符|来查找两个或是多个可能的匹配.例如通用表达式 foo|bar 可以查找 foo 或是 bar.
现在我们知道了这样多的表达式的表示方法,那么我们如何来应用他们呢?例如我们现在要查的内容为 1MGU103.这个字符串是由 1 个数字,3 个大写字符,3 个数字组成的.可以有几种方法来表示:
一是先表示前面的一个数字:[0-9],然后加入大写字符后就成为了:[0-9][A-Z].因为有三个大写字符我们可以加入精确匹配的数字:[0-9][A-Z]{3}.最后我们再加入最后面的三个数字.所以最后的结果就成了:[0-9][A-Z]{3}[0-9]{3}
另一种利用 d 指任何的数字,而 u 指任何的大写字符.所以用这样的方法写成的表达示就为:
du{3}d{3}.
从这里我们可以看到用这样的方式来查找要比第一种方法快得多.如果我们编辑的文件在采用这两种方法进行查找时会看出差别,那么我们的文件也许就是太大了.
我们还可以用这样的表达示:duuuddd,这个也可以找到我们想要的内容.
最后我们还可以用字符类来写出我们的表达式:
[[:digit:]][[:upper:]]{3}[[:digit:]]{3}
这四种方法都可以很好的来完成我们的工作,我们只要记住一种我们容易记住的就可以了.毕竟我

们可以记住的简单的方法要比我们不能记住的精妙的方法快得多啊.
到现在我们所有的讨论和方法都是在认为我们已经打开 magic 选项的基础上来做的.如果这个选项被关闭了,那么我们在通用表达式中的许多的特殊字符就失去了他们神奇的魔力,我们只有通过字符转义才可以正常的来使用.
关闭 magic 的选项命令为:
:set nomagic
这时*,.,[,]就都被认为只是平常的字符了.如果我们还想用*来指 0 次或是更多次就要用到转义:*.
所以我们应保证 magic 选项是打开的,这样我们也才可以使用 Vim 强大的搜索能力,而这也正是 Vim 默认情况下所做的.
一些常用的偏移(Offset)定义:
+[num]　　光标置于第一个匹配字符下第 num 行的开始处.
-[num]　　光标置于第一个匹配字符上第 num 行的开始处.
e　　匹配字符串的结尾处.
e[num]
光标置于第一个匹配字符串的结尾处,然后移动 num 个字符,如果为正,向右移,为负,向左移.
b s　　第一个匹配字符串的开始处.
b[num]
s[num]
光标置于第一个匹配字符串的开始处,然后移动 num 个字符,如果为正,向右移,为负,向左移.
常用的通用表达式如下:(认为 magic 选项打开)
简单的元素:
x　字符 x
^　一行的开始处
$　一行的结尾处.
.　单一的字符
<　查找字符串的开始标记
>　查找字符串的结束标记.
范围运算符:
[abc]　匹配 a,b 或是 c
[^abc]　匹配除 abc 以处的字符
[a-z]　匹配从 a 到 z 的所有小写字符
[a-zA-Z]　匹配所有字符,包括大小写.
字符类:
[:alnum:]　匹配所有的字符和数字
[:alpha:]　匹配所有的字符
[:ascii:]　匹配所有的 ASCII 字符
[:backspace:]　匹配退格符<BS>
[:blank:]　匹配空格和 Tab
[:cntrl:]　匹配所有的控制字符
[:digit:]　匹配所有的数字
[:escape:]　匹配 Esc
[:graph:]　匹配所打印的字符,不包括空格
[:lower:]　匹配所有的小写字符
[:print:]　匹配所有的要打印字符,包括空格

[:return:] 匹配所有的行末符号(包括<Enter>,<CR>,<NL>).
[:punct:] 匹配所有的功能符号
[:space:] 匹配所有的空白符
[:tab:] 匹配 Tab
[:upper:] 匹配所有的大写字符
[:xdigit:] 匹配十六进制数字.
类型:
(pattern) 标记一个类型以后使用
1 与第一个在()中的子表达式匹配的字符串匹配相同的字符串
例如表达式([a-z])1 可以匹配 aa,bb 或是类似的.
2 与 1 相类似,但是是使用第二个子表达式
9 与 1 相类似,但是是使用第九个子表达式
特殊字符:
a 大小写字母字符
A 除了 a-zA-Z 以外的字母字符
b <BS>
d 数字字符
D 非数字字符
e <ESC>
f 由 isfname 选项定义的文件名字符
F 文件名字符,但是不包含数字
h 单词的头字符(A-Za-z)
H 不是单词的头字符(A-Za-z)
i 由 isdent 选项定义的字符
I 定义的字符,但是不包括数字
k 由 iskeyword 选项定义的关键字字符
K 关键字字符,但是不包括数字
l 小字字符(a-z)
L 非小写字符(除了 a-z 以外的字符)
o 八进制数字
O 非八进制数字
p 由 isprint 选项定义的可打印字符
P 可打印字符,但是不包括数字
r <CR>
s 空白符<Space>和<Tab>
S 非空白符
t <Tab>
u 大写字母字符(A-Z)
U 非大写字母字符
w 单词字符(0-9A-Za-z)
W 非单词字符
x 十六进制数字
X 非十六进制数字
~ 匹配最后指定的字符串

修饰符:
*　匹配 0 次或是多次,尽可能多的匹配
+　匹配 1 次或是多次,尽可能多的匹配
=　匹配 0 次或是 1 次
{}　匹配 0 次或是多次
{n}
{-n}　匹配 n 次
{n,m}　匹配 n 次到 m 次
{n,}　匹配 n 次到多次
{,m}　匹配 0 次到 m 次
{-n,m}　匹配 n 次到 m 次,尽可能少的进行匹配
{-n,}　至少匹配 n 次,尽可能少的进行匹配
{-,m}　匹配到 m 次,尽可能少的进行匹配
{-}　匹配 0 次到多次,尽可能少的进行匹配
str1|str2　匹配 str1 或是 str2

Vim 编辑器有不同的方法来处理各类事物.我们在 Vim 编辑器的学习使用(四)已经讨论过文本块和多文件的处理方法.有了这些命令,我们就可以很好的来完成我们的工作.在这一次的学习中我们会讨论一些更多的内容.从而使得我们的 Vim 编辑工作来得更完美一些.
当我们插入文本行的时候可以使用 p 命令或是 P 命令.所不同的是 p 命令是在当前行的下一行进行插入,插入后光标移动到新行的开头处,而 P 命令是在当前的上一行进行插入,插入后光标移到新行的下一行的开头处.而我们还可以使用 gp 或是 gP 命令.不同的是 gp 命令是将光标移动到新行的结尾处,也就是新行的下一行的开头处.gP 命令与此相类似,是在当前的上一行进行插入,插入后,光标移动新行的结尾处,也就是下一行的开头处.
在 Vim 编辑器中还有一些特殊的标记符,如单引号'是指光标上一次的位置,但是这个位置不包括由方向键移动的位置.其他的一些特殊的标记包括:
]　上一次插入的文本的开头
[　上一次插入的文本的结尾
"　当我们离开文件时光标所处的位置
(注:这个地方看书是这样写的,但是自己做时却只可以是',[,],而且是要按两次键才行,不解:().
到现在为止我们所做的所有的复制和删除文本时我们并没有指明我们要使用哪一个寄存器.如果我们没有指明要用哪一个寄存器,Vim 就会使用默认的没有命名的寄存器.用来指明这个寄存器的标记符是两个双引号("").在这里前一个双引号用来指示一个寄存器,而后一个双引号则是这个寄存器的名字.这样"a 就是指我们使用 a 寄存器.
我们可以在我们的复制文本或是删除文本这前来指明我们的所复制或是删除的文本要放在哪一个寄存器中,指明寄存器的命令格式如下:
"register
在这里 register 是一个小写字母,是我们所指定的寄存器的名字,这样我们就可以有 26 个寄存器可以使用.
在通常的情况下我们使用 yy 所复制的文本是被放在没有命名的那个寄存器中,我们可以使用命令"ayy 将我们所复制的文本放在指定的寄存器中.当然如果我们使用这样的命令,我们所复制的文本也会被放入未命名的寄存中的.

如果我们想知道寄存中都包含有哪些内容,我们可以使用下面的命令:
:registers
一般我们复制或是粘贴的文本会被入以字母命名的寄存器中,当然我们也可以使用一些特殊的寄存器.
我们可以通过给命令:registers 一个参数,我们可以来查看特定寄存器中的内容,例如我们可以用下面的命令来查看寄存器 a 和 x 中的内容:
:registers ax
当我们使用命令"ayy 我们是将当前文本行的内容放入寄存器中,而当我们再使用相应的大写字母来指定寄存器时,如"Ayy 我们是将当前行内容追加到寄存器"a 中,这时候在这个寄存器中就存两行文本,在寄存器中^J 是指一行的结束.
在 Vim 中还有一些特殊的寄存器,第一个就是我们已经知道的未命名寄存器,他的名字是一个双引号".其他的还有 1 到 9 寄存器,寄存器 1 中含有我们上一次删除的文本,依次类推.
在古老时代的 Vi 中,Vi 只可以撤销三次.如果我们将 dd 命令执行了三次,也许我们就没有太多的好运来使用 u 命令将我们删掉的文本再恢复过来.但是幸运的是这三个文本被分别存放在寄存器 1,2,3 中.我们可以通过命令"1P,"2P,"3P 将这些文本再粘贴回来,或者是我们可以使用下面的命令来达到同样的结果:
""P..
其他的一些特殊的寄存器:

寄存器	描述	可复写
0	上一次复制的文本	是
-	上一次删除的文本	否
.	上一次插入的文本	否
%	当前文件的名字	否
#	交替文件的名字	否
/	上一次查找的字符串	否
:	上一次":"命令	否
_	黑洞(black hole)	是
=	表达式	否
*	由鼠标选中的文本	是

黑洞寄存器(_)(The Black Hole Register)
黑洞寄存器是一个特殊的寄存器,我们放入其中的任何文本都不复存在.当然我们也可以使用 p 命令来粘贴这个寄存器中的文本,但是这样做是没有意义的,因为在这个寄存器中根本就不存在任何内容.这样的寄存器也有着相当重要的作用.如果我们想永久删除某一文本而不是将他放入 1-9 中某个寄存中,我们可以使用这个寄存器.例如命令 dd 删除一行文本并将这一行文本存在寄存器 1 中,而我们用命令"_dd 则是将这行文本放入黑洞寄存器中,这些文本也就会永久地消失了,而寄存器 1 中的文本会保持不变.
表达式寄存器(=)(The Expression Register)
我们可以使用表达式寄存器来在文本中输入表达式.当我们输入表达式寄存器开始的命令时,就会在 Vim 的下部显示一个提示行,这就是给我们一个机会,我们就可以在这里来输入我们的表达式了.然后我们可以使用命令 p 将表达式的结果粘贴到文本.
例如我们要在文本中插入 38*56 的值,我们可以这样来做:
进入命令模式,输入表达式寄存器开始的命令"=,这时就会在 Vim 的下端显示出=来等待我们的输入,我们可以输入 38*56,回车,然后输入命令 p,这样就可以将我们的计算结果插入文本中了.
在表达式寄存器中我们不仅可以使用通常的算术运算符,还可以使用 Vim 特定的函数和运算符.

如果我们不仅仅是用常用的算术运算符来完成我们的工作,也许我们就需要来查阅表达式文档了.例如我们可以通过表达式寄存器来得到环境变量的值.如果我们输入"=$HOME 我们就可以得到HOME 变量的值了.

剪切板寄存器(*)(The Clipboard Register)

剪切板寄存器可以使得我们通过系统的剪切板来读写数据.如果是在 UNIX 系统上,这个要在图形界面下使用.这样我们就可以通过剪切板寄存器来在不同的 Vim 编辑器或者是其他的程序包之间进行文本的剪切和复制.

我们在 UNIX 或是 Linux 系统中可以使用 Vim 编辑器和 grep 命令处理包含某一个指定词的所有文件.这对于我们有着极大有用处,因为我们可以用这个组合来查看或是处理包含某一个变量的程序文件.例如现在我们要编辑包含有变量 frame_counter 的 C 程序文件.这时我们就可以使用下面的命令:

$ vim `grep -l 'frame_counter' *.c`

(注:在这里最外面的是反外号,即数字键 1 旁边的,而里面的是单引号)

在这个命令中,grep 命令在一系列文件查找包含有指定单词的文件.在这里我们指定-l 选项,这样这个命令就会列出包含有这个单词的文件,而不会打印找到的字符串.在这里我们要查找的字符串 frame_counter,我们可以使用这个命令来查找其他的字符串或是表达式.而整个的命令由反引号包括起来.这就可以告诉 UNIX Shell 来运行这个命令,并假定认为这个命令的执行已由命令打印输出.所以这个命令的执行结果就是运行 grep 命令并产生一个文件的列表,这些文件名放在 Vim 的命令行中,这样 Vim 就可以来编辑这些文件了.

也许有的人会说为什么要在这里展现这个命令呢?这是 UNIX Shell 的特征而不是 Vim 作品的一部分.而这正是使得 Vim 更加完美.

我们可以用下面的命令来设置参数列表:

:arg `grep -l 'frame_counter' *.c`

在这个参数列表中我们可以用命令来指定我们想要编辑的文件,例如如果我们要编辑第三个文件我们可以用下面的命令:

:argument 3

这个命令可以使得我们用文件在参数列表中的位置来编辑这个文件.如果我们是用下面的命令来启动 Vim 的:

$ gvim one.c two.c three.c four.c five.c

这时如果我们要编辑第四个文件我们可以用下面的命令:

:argument 4

当我们在命令行中用命令来指定一系列文件时,我们完成文件列表的初始化工作.但是我们可以用命令:args 来指定新的文件列表.例如下面的命令:

:args alpha.c beta.c gamma.c

执行完毕这个命令以后我们开始编辑文件 alpha.c,然后是文件 beta.c.

有时我们在编辑文件时希望文件在打开后光标能定位我们希望在的行,这我们该如何来做到呢?例如我们要第 12 行开始编辑文件,我们可以用命令打开这个文件,然后执行命令 12G 来使得光标到指定的行.我们也可以在 vim 启动时在命令后面加上行号来指明光标要到的地方.如下面的命令:

$ gvim +12 file.c

我们还可以用这样的命令形式+/string 来使得文件在装入以后光标放在指定的字符串.例如我们希望文件在打开后光标放在第一个包含字符#include 处,我们可以用下面这样的命令:

$ gvim +/#include file.c

当然我们可以在+后面放上任何命令模式的命令.

我们可以用多种多样的命令来指明+cmd 的参数.通常:vi 命令格式如下:
:vi [+cmd] {file}
下面的这些命令也可以带上+cmd 的形式:
:next [+cmd]
:wnext [+cmd]
:previous[+cmd]
:wprevious [+cmd]
:Next[+cmd]
:wNext[+cmd]
:rewind[+cmd]
:last[+cmd]
标记符 a-z 只是当前文件的标记.换句话说,我们可以在一个文件标记 a,也可以在另外一个文件中标记 a.如果我们执行命令'a 我们就可以跳转到当前文件中的 a 标记处.而大写字母(A-Z)的标记符则就不一样了.他不仅标记了当前文件中的某一行而且也标记当前文件.例如我们正在编辑文件 one.c 而且在其中用 A 做了一处标记.然后当我们在编辑文件 two.c 时我们就可以执行命令'A,这样就可以跳转到 one.c 文件中的标记处
当我们用插入模式进入文本时我们也可以执行种不同的命令.例如<BS><BackSpace>可以清除光标前面的字符,而 CTRL-U 会清除整个一行或者至少是你刚输入的内容,CTRL-W 会清除光标前面的字符.当我们在插入模式时我们也可以来移动光标,这时我们不可以再用传统的 h,j,k,l,但是这时我们可以使用小方向键来移动光标.<HOME>可以移动到一行的开头,<END>可以移动到一行结尾.<PageUp>可以向上翻屏,<PageDown>可以向下翻屏.
在插入模式下如果我们输入 CTRL-A,Vim 编辑器就会插入我们上一次在 Vim 插入状态时所输入的文本.例如如果我们在文件中一行里输入#include,然后我们用命令 j 向下移动一行并插入新的一行,这时如果我们想输入#include,可以用命令 CTRL-A 来完成.CTRL-@命令与其相类似,所不同的只是在执行完这个命令后会退出插入模式.我们还可以用命令 CTRL-V 来引用下一个字符,换句话说任何有着特殊意义的字符他都将被忽略掉.例如如果我们输入 CTRL-V<Esc>会在文本中插入空白符.我们也可以用命令 CTRL-Vdigits 来插入这个数字所指的字符,例如如果我们输入 CTRL-V64 就会在文本中插入@.在默认的情况下 CTRL-V 是使用十进制的数字,但是我们也可以插入十六进制的数字.
CTRL-Y 命令可以输入光标上面的字会,当我们想要重复上一行时这个命令就会显得尤为有用.
与 CTRL-Y 命令相似的是 CTRL-E 命令,不同的是这个命令可能重复光标下面的字符.
命令 CTRL-Rregister 可以使得我们寄存器中的内容插入文本中.如果寄存器的内容含有类似于<BS>这样的特殊字符时,这些特殊会被重样解释,就像我们从键盘中输入这些字符一样.如果我们想这样,只是想着将这些字符作为普通字符业对待,我们可以用这样的命令:CTRL-R
CTRL-R register
例如我们输入下面的文本:
This is a test.
然后我们用命令"ayy 将这些文本复制到寄存器 a 中,完成以后我们进入插入模式,输入命令 CTRL-Ra,我们就会刚才我们复制的内容就会出现在文本行了.
如果我们不希望特殊字符被重新解释我们可以用命令 CTRL-R CTRL-R register.
命令 CTRL-CTRL-N 可以离开插入模式而回到正常模式下.换句话这个命令与<ESC>相类似,但这个命令可以在任何模式下使用,就这一点而似乎是要比<ESC>命令强大一些啊.
最后要介绍的就是 CTRL-O 命令,这个命令可以回到命令模式执行 Vim 命令在执行完以后又回到了插入模式.例如我们执行命令 CTRL-Odw,Vim 就会回到命令模式执行命令 dw,在执行完毕后又

回到了插入模式.
我们在使用全局标记时遇到的一个问题就是当我们退出 Vim 以后我们所设定的那些全局变量就不在存在.而我们希望能将这些保存下来.文件 viminfo 就是设计成为一个用保存标记信息以及下面的一些信息的文件:
命令行历史
查找历史
输入历史
寄存器
标记
缓冲区列表
全局变量
保存某一项的办法是我们要打开这个保存的选项,我们可以下面的命令来做到:
:set viminfo=string
这里的 string 表明了我们要保存的内容.
这里的 string 的语法是一个选项字符跟上一个参数.中间用逗号来分隔.首先'option 用来表明我们要为少文件保存局部标记,我们可以选一个合适的数字,例如我们要选 1000,那么这时的命令形式如下:
:set viminfo-'1000
而 f 选项则用来控制是否要保存全局标记.如果这个选项的值为 0,则不保存,如果值为 1 或是我们没有指明 f 选项,那么 vim 都会保存这些全局标记.如果我们要用这一特征,我们可以用下面的命令:
:set viminfo='1000,f1
选项 r 会告诉 Vim 一些关于可移动介质的情况.在 Vim 中为可移动介质上的文件所做标记是不会被保存的,原因就是我们在 Vim 中要跳转到软盘文件的标记处是一个很难的操作.我们多次来表明 r 选项.如果我们在 Windows 系统上就可以用下面的命令来告诉 Vim 磁盘 A 和 B 是可移动介质:
:set viminfo='1000,f1,rA:,rB:
在 UNIX 系统并没有标准的软盘名称.在一般的情况下我们总是将软盘挂载在/mnt/floppy 文件下,所以我们可以用下面的命令:
:set viminfo='1000,f1,r/mnt/floppy
"选项用来控制每一个寄存器可以保存多少行.在默认的情况下,所有的行都会被保存.如果是 0 就不会保存.一般情况默认的设置已经可以很好的来满足我们的要求了,所以我们可以不必加入我们自己的设置.
:选项可以用来控制:历史行的记录.我想 100 应是可以满足我们的要求了吧:-):
:set viminfo='1000,f1,r/mnt/floppy,:100
/选项可以用来定义查找历史记录的大小,100 也已是很充足的了:
:set viminfo='1000,f1,r/mnt/floppy,:100,/100
(注:Vim 不会保存那些超过他的记录能力的内容,这是在选项 history 中设置的)
通常,当我们启动 Vim 以后如果我们已经设置了 hlsearch 选项,编辑器就会高亮显示上一次查找的内容.要关闭这个选项,我们可以 viminfo 的选项列表中加入 h 标记,或者是我们在 Vim 启动后执行命令:nohlsearch 来关闭这个选项.
@选项用来控制输入历史记录的行数.输入历史会记住我们输入的所有内容.
如果我们设置了%选项,我们就可以保存缓冲区列表记录.如果我们没有在命令行指定一个文件来编辑,缓冲区列表就会被重新保存:
:set viminfo='1000,f1,r/mnt/floppy,:100,/100,%
!选项用保存全局变量,这些全局是名字全部为大写字母的变量:

:set viminfo='1000,f1,r/mnt/floppy,:100,/100,%,!
最后 n 选项用来指明 viminfo 的文件名.在 UNIX 系统中默认的为$HOME/.viminfo
我们可以将这些命令以及一些其他的初始化命令放在初始文件 vimrc 中.这样 viminfo 文件就会在 Vim 退出时自动保存,启动时来读取初始化.
如果我们要更清楚一些的来保存和读取这个文件,我们可以用下面的命令来以另外的文件名保存:
:wviminfo[file]
这样我们的这一些设置信息就会被写入这个文件中了.
我们可以用下面的命令来读取这个文件:
:rviminfo [file]
这样就会从这个文件中读入设置信息.如果这些信息与当前的设置有冲突,那么我们的这些设置信息就不再起作用了.我们可以用下面的命令来强制这些设置信息起作用:
:rviminfo![file]
有时我们会编辑一些文件中一行的宽度要超过屏幕的宽度.当行宽超过屏幕的宽度会发生什么事情呢?这时 vim 会将这一行进行回折以适应屏幕的宽度.如果我们设置了 nowrap 选项,则 Vim 会用单行来处理文件的第一行文本.这时超出屏幕的部分就会不再出现,从而从屏幕消失.
在默认的情况下,Vim 并不会显示水平滚动条,我们可以用下面的命令来使得 Gvim 显示小平滚动条:
:set guioptions+=b
这样 Gvim 就可以水平滚动了.
当我们设置了 nowrap 选项后,命令^将光标移动当前行的第一个非空字符处.g^命令可以将光标移动到当前屏幕的第一个非空字符处.在执行这们执行这样的命令时如果在窗口的其他部分有文本,那么这一部分的文本将会被忽略.类似的一些命令如下:

命令	命令	含义
^	g^	向左移动当前屏幕的第一个非空字符处
<Home>	g<Home>	
0	g0	向左移动当前屏幕的第一个字符处
<End>	g<End>	
$	g$	向右移动当前屏幕的结尾处
	gm	移动到屏幕的中间

命令 count|可将光标移动屏幕中指定的列.
命令 countzh 可以向左移动屏幕,移动的量度为 count 个字符.而命令 countzl 与其相类似,只是这个命令是向右移动屏幕.
命令 zH 可以向左移动个屏幕,而命令 zL 可以向右移到半个屏幕.命令 j 或是<Down>可以下移一行.在这里我们要知道的就是如果我们设置了 wrap 选项,那么下移一行在屏幕上显示也许就会是几行,这时我们要清楚,此时的几行正是设置了 nowrap 时的一行.
当我们设置了 wrap 选项,一个很长的一行 Vim 就会折成几行显示在屏幕上,这时我们可以用命令 gj 或是 g<Down>来下移屏幕屏幕中显示的一行,这时也许我们移动的并非是真正的一行.而命令 gk 或是 g<Up>命令与其相类似.
在默认的情况下,Vim 编辑时会折回很长的一行,这时他首先是尽可能多的在屏幕的第一行放置文本,如果这一行的文本超出了屏幕的范围,Vim 就会将其打断,然后在屏幕中的下一行显示其余的部分.我们也可以通过下面的命令来并闭这个选项:
:set nowrap
这时一个很长的句子超出屏幕的部分就在从屏幕上消失.我们可以通过沿着这个句子来移动光标,这样屏幕就会进行水平滚动,我们就可以看到这一行的其余部分了.

当然了我们也可以来自定义我们自己的句子回折形式:
首先我们可以告诉 Vim 在合适的地方来打断一个句子.我们可以用下面的命令来实现:
:set linebreak
那么又如何来定义一个合适的地方呢?这个是由 breakat 选项中的字符来确定的.在默认的情况下这些默认的字符是^I!@*-+_;:,./?如果我们不希望在下划线_处打断句子,我们可以用下面的命令来将_从这个列表移除就可以了:
:set breakat-=_
在通常的情况下,如果一个句子被打断 Vim 是不会在句子的连接的地方显示任何内容的.我们可以通过设置 showbreak 选项来显示我们所希望显示的内容信息:
:set showbreak="->"
我们最后要讨论的一个问题就当我们要在屏幕的结尾处打断一个句子我们应如何来做呢?这时我们可以用两个选择:一是我们可以不显示半行.这时 Vim 编辑器会在屏幕的底部来显示@以表时这是一个长句子,我们不能把他全部放在屏幕内.二是我们可以显示半行.
Vim 默认的是采用第一种方法.如果我们要采用第二种方法我们可以用下面的命令来实现:
:set display=lastline

我们在 Vim 编辑器的学习使用(五)中曾结讨论了一些基本的窗口使用命令.这些命令可以使得我们在不同的窗口内进行编辑工作,从而使得我们编辑多个文件成为可能.而在这里我们将会讨论一些更多的与窗口相关的命令操作.
当我们使用多个窗口进行文件编辑时,我们如何来进行窗口的切换操作呢?我们可以使用命令 CTRL-Wj 回到下一个窗口,而使用命令 CTRL-Wk 回到上一个窗口.我们还可以使用下面的命令来进行窗口的切换操作:
CTRL-Wt 切换到顶部的窗口
CTRL-Wb 切换到底部的窗口
CTRL-Wp 切换到我们进行切换操作以前我们所在的窗口
countCTRL-Ww 向下切换一个窗口.如果是在底部,则进行回环.如果指明了数字,则切换到数字所指定的窗口.
countCTRL-WW 向下切换一个窗口,如果是在顶部,则进行回环,如果指蝗了数字,则切换到数字所指定的窗口
我们在使用多个窗口进行文本编辑的时候,我们还可以进行窗口的移动,命令 CTRL-Wr 命令可以使得窗口向下进行循环移动.这个命令可以带一个数字作为参数,可以指明向下循环移动所执行的次数.与其相类似的命令是 CTRL-WR 命令,这个命令可以使得窗口向上循环移动.
命令 CTRL-Wx 可以使得我们将当前窗口与下一个窗口进行位置的对换.如果当前是一个底部,则没有下一个窗口,这时执行这个命令时是将当前窗口与上一个窗口进行位置对换.
当我们在用多窗口进行多文件的编辑时我们可以用命令对这些文件进行共同的操作.:write 命令可以保存当前文件.我们可以用下面的命令来实现对所有已经修改过的文件,包括隐藏缓冲区中的文件,进行保存操作:
:wall
命令:quit 可以退出当前文件.如果这是一个文件的最后一个窗口,那么这个文件将会被关闭.如果我们同时打开了多个窗口进文本的编辑,我们可以用下面的命令来退出所有的文件:
:qall
如果在这些文件中有文件进行了修改但是没有保存,在执行这个命令时会给出警告信息,这样我们就可以保存那些我们没有保存的修改了,但是如果我们想要放弃我们所做修改工作而强行退出我

们可以用下面的命令:
:qall!
我们还可以将这个两个命令进行组合来实现对所有文件的保存退出的命令:
:wqall
命令 CTRL-Wo 可以使得当前窗口成为屏幕上的唯一的一个窗口,而其他的窗口全部关闭.系统会认为我们在其他的每一个窗口中都执行了命令:quit.
如果我们通过命令:args file-list 指定了一个文件列表或是在启动 vim 时指定了一个文件列表,那么:all 命令就会为每一个文件打开一个窗口,这样我们就可以进行多文件的编辑工作了.
下面的命令由命令:all 变化而来的,这个命令可以每一个隐藏的缓冲区打开一个窗口:
:unhide
这个命令还可以带一个参数,用来指明一次打开的窗口数.例如如果我们要打开所有的缓冲区但是在屏幕上显示不超过 5 个窗口,我们可以用下面的命令:
:unhide 5
我们还可以用 CTRL-W CTRL-^命令来分裂窗口来编辑交替文件.这个命令是新打开一窗口,并在这个窗口中装入交替文本并进编辑.而命令 CTRL-^则是通过切换窗口来编辑交替文件.
命令 CTRL-W CTRL-I 会分裂当前窗口,然在查找当前光标下的单词第一次出现的地方.这样的查找不不仅是在当前文件中查找,也会在由#include 所包含进来的文件中进行查找.
在 Vim 中还有许多缩写的命令可以来快的完成工作,如下面的一些命令:
:countsnext :split 与:countnext 的组合
:countsprevious :split 与:countprevious 的组合
:countsNext :split 与:countNext 的组合
:srewind :split 与:rewind 的组合
:slast :split 与:last 的组合
:sargument :split 与:argument 的组合
CTRL-WCTRL-D :split 与]CTRL-D 的组合
CTRL-Wf :split 与:find 的组合
CTRL-Wg] :split 与 CTRL-]的组合
在这些命令一个算是优点的地方就是如果命令执行失败那么是不会打开一个新的窗口的.
我们在用 Vim 编辑器进行文件编辑的时候可以用不同的缓冲区装入不同的文件,我们可以在启动 Vim 时指定要编辑的文件列表,我们也可以在编辑的过程中用下面的命令新增一个缓冲区:
:badd filename
这样这个指定的文件就会被加到缓冲区的列表中.这个文件的编辑过程只有我们切换到那个缓冲区时才会开始.这个命令还可以带参数,来指明当我们为这个缓冲区打开窗口时,光标所处的位置:
:badd +lnum filename
我们可以用下面的命令来删除一个缓冲区:
:bdelete filename
或者是也可以用下面的命令:
:bdelete 3
:3 bdelete
我们还可以用下面的命令来删除指定范围的缓冲区:
:1,3 bdelete
如果我们使用了!选项,那么我们在缓冲区所有的所有的改动都会被放弃:
:bdelete! filename
命令:bunload 会卸载一个缓冲区,这样这个缓冲区就会从内存中卸载,所有为这个缓冲区打开的窗

口也会关闭.但是这个文件名文件名仍然会存在于这个缓冲区列表中.:bunload 命令与:bdelete 命令的用法相类似.
我们可以用下面的命令来为每一个缓冲区打开一个窗口:
:ball
我们可以用 laststatus 选项来控制最后一个窗口是否显示状态行,这个选项的值如下:
0　　最后一个窗口从不显示状态行
1
如果在屏幕上只有一个窗口,那么不显示状态行.如果有两个或更多个,则要在最后一个窗口显示状态行.
2　　在窗口上总是显示状态行,哪怕屏幕中只有一个窗口.
我们可以用 winheight 选项来控制一个窗口最小的行数.但是这个并没有一个硬性的限制,如果窗口显得太拥挤了,Vim 会减少窗口的尺寸.
当我们打开 equalalways 选项后,Vim 会以相同的尺寸来分裂窗口,而这也正是 Vim 编辑器默认的情况,但是如果我们设置了 noeqaulalways 选项后我们就可以用不同的尺寸来分裂一个窗口.
winheight 选项用来控制当前窗口的最小高度.而 winminheight 选项则用来控制其他窗口显示的高度.
在通常的情况下,:split 命令是在当前窗口的上方打开一个新窗口.而 splitbelow 选项可以使得 Vim 在当前窗口的下方打开一新窗口.
假如我们正在编辑一个很长的文件,而现在天已经晚了,我们想着退出工作并在第二天接着做.这时我们可以将我们正在编辑的文件信息存成一个文件,在我们要第二天要接着编辑这个文件时只要读入这个文件就可以了.这样的一个文件包含了所有我们正在编辑的文件信息,例如文件列表,窗口,标记,寄存器以及其他的一些信息等等.
我们可以用下面的命令来产生一个程序文件:
:mksession filename
例如我们存储的文件是:
:mksession vimbook.vim
如果我们要接着工作,想要装入这个程序文件时只要用下面的命令:
:source vimbook.vim
我们也可以在启动 Vim 时指明要读入的程序信息文件:
$ vim -c ":source vimbook.vim"
我们可以使用 sessionoption 选项来控制我们在这样的文件存入什么样的内容.他是由逗号分开的一系列的关键的字符串组成的.例如默认的设置是这样的:
:set sessionoptions=buffers,winsize,options,help,blank
可能的关键字的值如下:
buffers　　保存所有的缓冲区.包括在屏幕上显示的以及隐藏的和卸载的缓冲区.
globals　　保存全局变量.这些全局变量是由大写和至少一个小写字母组成的.
help　　帮助窗口
blank　　屏幕上的空窗口
options　　所有的选项和键盘映射
winpos　　GUI 窗口的位置
resize　　屏幕的尺寸
winsize　　窗口的尺寸
slash　　在文件名中用斜线来代替空格.
unix　　用 UNIX 的行结尾格式来保存程序信息文件.

我们在 Vi 编辑器的学习使用(六)学习了基本的可视化模式,这时我们可以执行简单的可视化命令.在这里我们将会讨论更多的与可视化相关的命令.这些命令中的许多只有很少的观众,如果我们可以看这一次的学习,也许这很少的观众中就会包括我们.

我们在 Vi 编辑器的学习使用(四)知道了如何来用寄存器实现复制,粘贴和删除的工作.我们也可以在可视化模式中来实现这些操作.例如要删除一个文本我们可以这样的来做:在可视化模式中高亮显示这些文本,然后执行 d 命令.如果要将这些文本删除后放入寄存器中,我们可以用下面的命令来实现:"register d.要复制文本到寄存器中我们可以使用 y 命令.而 D 和 Y 命令与其相对应的小写字母的命令相类似,只是他们作用在一整行,而 d 和 y 命令是作用于高亮显示的部分.

在块可视化模式中,$命令可以使得选中的文本扩展到所有的选中行的结尾处.当我们上下移动光标时,可以使得选中的文本扩展到这一行的结尾处.如果新行要比当前行长得多,这样的扩展也是会发生的.

gv 命令可以重复前一次可视化模式时选中的文本.如果我们已经在可视化模式状态下,执行这个命令时会选中前一次选中的文本.如果我们重复执 gv 命令,就会在当前选中的文本和前一次选中的文本之间进行切换.

在 Vim 编辑器的可视化模式下的许多命令都是用来帮助我们高亮显示我们想要的文本的.例如命令 aw 高亮显示下一个单词.事实他不仅高亮显示这个单词,而且也包括这个单词后的空格.在一开始也许我们会认为这个命令没有太大的用处.因为 w 命令可以向前移动一个单词,我们为什么不用这个命令呢?

这是因为当我们执行选择文本的操作时,选中的是从老的光标所在处到新的光标所在处之间的文本.当我们使用命令 w 来移动文本时,结果是光标置于下一个单词的第一个字符上.如果这时我们要执行删除操作,我们命令的执行是不仅删掉了我们要删掉的单词,也同时删除了下一个单词的第一个字符.

而 aw 命令是将光标放在下一个单词的第一个字符的前面.换句话说,我们选中的是下一个单词前面的单词以及空格,而不是选中的下一个单词.

而另外一个使用 aw 命令而不使用 w 命令的原因就是不论光标置于一个单词的哪一个字符上,aw 命令都可以选中整个单词,而 w 命令只是选中当前光标处和这个单词结尾之间的字符.

如果我们仅仅是想选中一个单词我们可以使用 iw 命令.

我们还可以使用下面的命令来选择文本:

countaw	选中一个单词以及其后的空格.
countiw	仅仅是选中一个单词.
countaW	选中一个 WORD 以及其后的空格.
countiW	仅仅是选中一个 WORD
countas	选中一个句子以及其后的空格.
countis	仅仅选中一个句子.
countap	选中一个段落以及后面的空格.
countip	仅仅是选中一个段落.
counta(	在括号所包括的文本内,选择直到括号的文本并包括括号.
counti(	与上面的命令相类似,只是不包括括号.
counta<	选择<>内的文本,包括<>
counti<	选择<>内的文本,不包括<>
counta[	选择[]内的文本,包括[]

counti[选择[]内的文本,不包括[]
counta{ 选择{}内的文本,包括{}
counti{ 选择{}内的文本,不包括{}
在可视化模式下,当我们选中一些文本以后,我们可以用命令 o 来使用光标移动选中的文本的另一个结尾处.然后我们可能再次执行 o 命令,来使得光标移动选中文本的另一个结尾处,也就我们来的地方.
而 O 命令可以在块可视化模式下将光标移动选中文本的另一角.换句话说,O 命令是将光标移动选中文本中的同一行的结尾处.
在可视化模式下选中的文本,我们可以用命令~来实现大小写的转换.而 U 命令是使得选中的文本变成大写的形式,而 u 命令是将选中的文本变为小写的形式.
我们可以在可视化模式下选中文本,然后用命令 J 将这些选中的行合并为一行,并用空格来分隔这些行.如果我们希望在合并以并没有空格来分隔,我们可以用命令 gJ.
我们可以用命令 gq 来格式化可视化模式下选中的文本.
我们还可以用 g?命令来加密高亮显示的文本,在这个命令中我们采用的是 Vim 中所采用的 rot 13 加密算法.如果我们对同一个文本进行两次加密操作,就相当我们进行了解密操作.
在可视化模式下我们还可以用命令:来对指定的范围进行命令行操作.例如我们要将文本块写入一个文件我们可以这样的来做:
在可视化模式下选中我们要写入文件的文本,然后执行下面的命令:
:write block.txt
这样就可以将指定的文本块写入文件了.
命令!是使用外部的命令来对我们所要编辑的文件中的文本进行操作.例如我们可以使用!sort 来使用 UNIX 下的 sort 程序进行文本的排序.我们可以这样的来做:
在可视化模式下选中我们要进行操作的文本,然后执行下面的命令:
:!sort
这样就可以对这些我们选中的文本进行排序操作了.
选择模式是另一种的可视化模式,他可以允许我们对选中的文本进行的快速的删除作替换的操作.我们使用选择模式也是很简单的操作.我们可以高亮显示文本,然后用<BS>来删除这段文本.我们也可以高亮显示文本,然后用我们所输入的内容来替换这些文本.
那么选择模式和可视化模式相比较又如何呢?在可视化模式下,我们可以高亮显示我们选中的文本,然后执行命令操作.换句话我们要用命令来结束可视化模式.而在选择模式下,命令仅限于<BS>(用于操作删除操作)和可打印的字符(用于替换操作).这样就会使得我们的操作变得更为简单,因为我们不需要来输入命令了,然而与可视化模式相比较他也有着太多的限制.
我们可以用下面命令来开始一种选择模式:
gh 进入字符选择模式
gH 进入行选择模式
gCTRL-H 进入块选择模式
在选择模式下移动光标比在正常模式下要显得困难一些.因为如果我们输入任何的可打印字符,Vim 就会删掉我们选中的文本并进插入状态开始我们的输入.所以要选择文本我们只好使用小方向键,CTRL 以及功能键.
如果我们进行了如下的设置我们还可以用鼠标来选择文本:
:set selectmode=mouse
(注:如果没有设置这个选项,可以在可视模式下执行鼠标操作而不可以在选择模式下执行鼠标操作)
在选择模式下,我们可以用命令<BS>或是 CTRL-H 来删除我们选中的文本.如果我们输入可打印

的字符 Vim 编辑就会删除我们选中的文本然后进入插入模式.
我们可以用命令 CTRL-O 来从选择模式切换到可视化模式.如果我们要可视化模式和选择模式中进行切换,我们可以使用 CTRL-G 命令.
在通常情况下,当我们选择文本后,这些文本仍会保持选中的状态.有时即使是我们在执行了命令以后,这些文本仍然保持选中的状态.gV 命令可以选得选中的文本在命令执行过后消失选中状态.这个在我们使用宏时显得更为有用.我们用他来一些工作,工作完成以后,我们就希望他能消失.

Vim 编辑器由一群需要一个好的文本编辑器的程序员们所写出来的.正因为是这样,Vim 中包含了许多的命令,我们可以用这些命令来自定义并且使我们的程序编辑工作变得更为简单.
例如如果我们现在正在编辑我们的程序文件.我们设置了 autoindent 选项,并且现在正处在第三层次的缩进上.而现在我们要加入一个注释块.这是一个很大的注释块,我们希望能将这个注释块放在文本的第一列.这时我们就需要禁止所有的自动缩进形式.为了这样的目的,一种方法是可以输入几次 CTRL-D 命令,或者是使用 0CTRL-D 命令.
命令 0CTRL-D 是在插入模式下移除所有的自动缩进的设置,并将光标放在第一列(在这里我们要注意的,当我们输入 0 时,我们所输入的 0 会显示在屏幕上,这时 Vim 会认为我们要在文本中插入一个 0,然后我们执行命令 CTRL-D,这时 Vim 就会意识到我们要做的是执行命令 0CTRL-D,并且这时 0 会就会从屏幕上消失.
当我们使用 0CTRL-D 命令以后,光标会回到第一列,而且下一行也是从第一列开始的.
然后如果我们正在输入一个标签或是我们要输入#ifdef 时我们只需要一行中的光标移动第一列就可以了.这时我们可以使用命令^CTRL-D.这个命令只会将当前行的光标放在第一列.当我们开始下一行时,这种缩进形式又会自动的执行.
CTRL-T 命令与<Tab>命令相类似,只是前者在文本中插入一个缩进,而这个缩进的大小由 shiftwidth 选项来控制的.如果我们设置了 shiftwidth 选项的值为 4,我们输入<Tab>,光标会移动以后第八列处,这是因为<Tab>缩进的大小是由 tabstop 选项来控制的,而在默认的情况下这个值为 8.但是如果我们输入了 CTRL-T 命令就会将光标移动以后的第四列处.这两个命令在一行的任何一点都是可以正常工作的.所以我们可以输入一些文本然后执行 CTRL-T 命令将其缩进,然后用 CTRL-D 来取消这样的缩进形式.

在通常的情况下当我们用命令 CTRL-R 来插入寄存器中的内容时,这些内容是自动缩进的.如果我们不希望这样的事情发生,我们可以用 CTRL-RCTRL-Orgister 命令.或者是如果我们要插入寄存器中的内容,并希望 Vim 编辑能正确的完成我们的要求我们可以用下面的命令 CTRL-R CTRL-P register.这样的命令在我们的程序文件中进行剪切和复制时显得尤为有用,因为在这时的程序文件中一般是设置了自动缩进选项的,而如果我们剪切或是复制后再粘贴时就会用两次的缩进操作,而这不是我们所希望的,这时我们就要用这些命令了.
在通常的情况下,命令"registerp 命令是将指定寄存器中的文本插入到缓冲区中,而命令"register]p 命令与此相类似的,所不同的是这个命令在插入时会有自动缩进的设置.而与此相类似的是命令"register]P 与"registerP.
在计算机产生以前的时代,存在着一个交流的工具被叫作打印机.打印机的一些模块可以执行 tabs.但是很不幸的是这些 tab 的大小被设置成为八个空格.而在计算机产后以后,他的第一个终端就是打印机.后来有一些更为现代的设备代替了打印机,但是古老的八个空格大小的 tab 被保留了下来,以保持与以前的相兼容.
但是这样的设计为我们以后的程序设计工作带来无尽的麻烦.因为有研究可以显示最易读的程序

是四个字符的缩进,而 Tab 是八个字符的缩进.我们如何来调合这样的事实.人们想出了许多解决的办法,如下:

1 在我们输入代码时结合使用空格和 Tab.如果我们需要 12 个字符的缩进,我们可以使用一个 Tab 和四个空格.

2 告诉机器将 Tab 设为四个字符的大小,然后在任何的地方使用 Tab.

3 放开我们的双手并且认为 Tab 是一个工作的魔鬼,而且我们总是使用空格.

幸运的是 Vim 编辑器支持以上的三种方法.

如果我们使用空格和 Tab 的组合,我们可以像正常一样的进行编辑.在默认的 Vim 设置情况下就可以很好的来工作.

但是我们可以通过设置 softtabstop 选项的值的来使我们的工作变得再简单一些.这个选项告诉 Vim 编辑使得 Tab 键看上去和感觉上是使用 softtabstop 选项贩值,但是实际上我们是使用 Tab 和空格的结合在做事情.

例如我们可以用下面的命令来使得以后每一次我们在按下 Tab 键时光标位于以后的第四列处::set softtabstop=4

当我们第一次按下 Tab 时,Vim 会在文本中插入四个空格.当我们第二次按下 Tab 键时,Vim 就会去掉刚才的四个空格,然后在文本中插入一个 Tab.也就是说当有八个空格相当于一个 Tab 时,Vim 就会用一个 Tab 来代替这八个空格.

另一个相关的选项就是 smarttab 选项.我们可以用下面的合谋来设置这个选项:

:set smarttab

当我们设置了这个选项以后,插入一行开头的 Tab 就会被看作是软 Tab.在这种情况下,Tab 的大小所使用的值就是 shiftwidth 选项所设定的值.但是在其他的地方插入的 Tab 就会像是正常的 Tab 一样的.在这里我们要注意的就是当我们要让这个选项来正常工作时,一定要使软 Tab 关掉(:set softtabstop=0).

精巧缩进(smart indent)是软 Tab 和正常 Tab 的组合.当我们执行下面的命令,Vim 编辑器就会区别对待一行开头的 Tab:

:set smarttab

例如如果我们有下面的设置:

:set shiftwidth=4

:set tabstop=8

:set smarttab

在这些设置中我们是设置 Tab 是八个空格而缩进是四个空格.当我们在一行的开始输入 Tab 时,光标就会移动一个缩进的大小,也就是四个空格.当我们输入两个 Tab 时,光标就会移动两个缩进的大小,也就是八个空格.

下面的内容则显示出了我们在一行的开始输入特定的内容时的显示:

<Tab> 四个空格

<Tab><Tab> 一个 Tab

<Tab><Tab><Tab> 一个 Tab,四个空格

<Tab><Tab><Tab><Tab> 两个 Tab

但是当我们在一行的其他地方输入 Tab 时,就会显得像正常的一样.

我们可以用下面的命令来设置一个 Tab 为四个空格:

:set tabstop=4

事实上我们可以用这个命令来将 Tab 设置成任何我们想要的值.

如果我们在我们的文件中含有 Tab 字符,我们可以通过设置 expandtab 选项来控制.当我们设置了这个选项,一个 Tab 键就会插入一系列的空格.在这里我们要知道的就是 expandtab 选项并不会影

响文章中的其他的 Tab 键值.换句说文档中的 Tab 值仍然会保持.如果我们要将 Tab 转换为空格,我们可以使用:retab 命令.

我们可以使用:retab 命令来实现在文章中不同 Tab 设置之间的转换.我们可以用这个命令来使得文章中一系列的空格转换为 Tab,或者是将一个 Tab 转换为一系列的空格.例如现在我们有一个文件使用是四个空格的 Tab 设置.但是这并不是标准的设置,而我们希望将他转换为八个空格的设置.我们希望这两个文件看起来是一样的,只是 Tab 的值不同.我们可以按照下面的方法来做:

:set tabstop=4

:%retab 8

这样以后我们的文件看起来就像是没有做过修改一样的,因为 Vim 将空白符与 tabstop 的新值相匹配.

再比如我们需要一个没有 Tab 的文件.首先我们执行命令设置 expandtab 选项.这样就会使得我们输入的新文本中的 Tab 成为空格.但是老文本中的 Tab 依然存在.要将这些 Tab 转换为空格,我们可以用下面的命令:

:%retab

因为我们没有指定一个新的 Tab 值,Vim 就会使用现在的 tabstop 的值.但是因为我们又设置了 expandtab 这个选项,所以所有的 Tab 就会用空格来代替.

在这些 Tab 设置中存在的问题就是会有不同的人用不同的习惯来使用 Vim 处理文件.所以说如果我们可以很好的处理三种不同类型的人写的文件,我们就可能容易的确处理各种不同的 Tab 设置.这个问题的一个解决办法就是我们在写文件的时候在文件的开头或是文件的结尾下加上一个注释块,在其中标出我们所使用的 Tab 设置.例如:

/*vim:tabstop=8:expandtab:shiftwidth=8*/

当我们知道这些以后我们可以建立我们自己所喜欢的格式.但是 Vim 是一个精巧的编辑器,他可以理解类似这样的注释并且为我们配置这些设置.但是有一点是要严格执行,那就是注释必须是这种形式的,而且是必须是在一个程序文件的前五行或是后五行才可以.这种类型的注释就叫做模式行(modeline).

假如我们设置了 shiftwidth 的值为四,而我们在一行的开始输入了三个空格.那么当我们执行命令>>时会是什么样呢?会在前面插入四个空格的缩进还是移动到最近的一个缩进处呢?答案取决于我们所设置的 shiftround 选项的值.

以通常的情况下,这个选项是没有设置的,所以会新增四个空格的缩进.但是如果我们执行了下面的命令,那么光标就会移动到最近的一个缩进处:

:set shiftround

当我们执行命令=时我们可以使用通过 equalprg 选项所指定的程序来进行文件的格式化工作.如果没有设置这个选项或者是我们没有编辑一个 Lisp 程序,那么 Vim 就会使用他自己的缩进程序来缩进 C 或者是 C++的程序文件.如果我们想要使用 GNU 的缩进程序我们可以执行下面的命令:

:set euqalprg=/usr/local/bin/indent

我们还可以要求 Vim 来格式化注释而且他可以很好的来完成我们的要求:

例如我们有下面的一段注释:

```
/*
 *This is a test.
 *Of the text formatting.
 */
```

我们可以通过下面的命令来格式化这段注释:

1 将光标放在这段注释开始的地方.

2 用命令 v 进入可视化模式.

3 将光标移动到这段注释的结尾处.
4 用命令 gq 来格式化这段注释.
结果如下:
/*
 *This is a test.Of the text formatting.
 */
(没有达到预期的效果,想来应处理程序文件中才有用)
我们可以通过 comments 选项来定义哪些是注释而哪些不是.我们还可以用命令 gq{motion}来完成注释的格式化工作.
我们可以使用 comments 选项来定义哪些文本是注释而哪些文本不是注释.这个选项由成对出现的标记:字符串格式组成(flag:string).
可以使用的标记(flags)如下:
b
后面必须跟上空格.这就是说如果一个字符后面跟上空格或是其他的空白符,那么这个字符开始了一个注释.
f 只有第一行有注释字符串.在下一行不要重复这个字符串,但是要保持缩进格式
l
当使用在三段注释的情况下,必须保证中间一行要与注释的开始和结束相对应.而且必须使用 s 或是 e 标记.
n 指明了嵌套注释
r 与 l 相类似,所不同的只是右对齐
x
告诉 Vim 在三段注释的情况下我们可以在下面的三种情况下仅输入最后一个字符就可以结束注注释:
1 我们已经在注释的开头输入了.
2 注释有中间部分.
3 结束字符串的第一个字符是这一行的第一个字符.
对于三段注释的情况,下面的一些标记适用:
s 开始三段注释
m 三段注释的中间部分
e 三段注释的结尾
number 在三段注释的中间部分的缩进中添加指定的空格
一个 C 程序的注释用/*开始,有中间部分*,以*/结尾.就像下面的一样:
/*
  * This is a comment
  */
这样的注释结果是由 comments 选项所指定的:
sl:/*,mb:*,ex:*/
在这个设置中 sl 表明这是一个三段注释的起始处并且在这个命令中的其他行需要缩进一个额外的空格.这个注释是以/*开始的.
这个注释的中间部分是由 mb:*来定义的.m 表明了一个中间部分,而 b 则是说在我们输入的任何内容后必须有一个空格.这段文本以*开始注释.
注释的结束是以 ex:*/来指定的.e 表明注释的结束,而 x 则是表明了我们只需要输入结束标记的最后一个字符来完成注释.而结束的定界符是*/.

那么我们如何来使这样的定义工作呢?首先我们要设置下面的选项:
:set formatoptions=qro
下面的一些选项在我们要格式化文本时会显得更为有用:
q　　允许使用 gq 来格式化化注释
r　　在我们输入回车后自动的加入注释的中间部分
o　　我们用 o 或是 O 命令来开始一个注释行时自动的添加注释行的中间部分
下面让我们看一下这样的设置会如何的工作:
我们要开始一段注释,我们在这一行输入注释的开始标记/*,然后我们打下回车,因为在格式选项中我们设置了 r,所以在下一行中会自动的添加注释的中间部分并且会在后面添加一个空格.当我们在再输入回来也会出现同样的情况,但是这时我们要结束我们的注释输入了,我们就如何结束呢?Vim 是一个精巧的编辑器,当我们在这种情况下只输入/时,光标就会向后移动一格插入/,这样就正确的结束我们的注释输入了.结果如下:
/*
* This is a test
*/
我们还可以使用各种不同的格式命令来格式化文本或是注释.
C 程序文件的缩进过程是由下面的一些选项来控制的:
cinkeys　　定义了引发缩进事件的关键字
cinoptions　　定义了如何缩进
cinwords　　定义了 C 和 C++的关键字
cinkeys 选项定义了哪些字符会引起缩进的变化.这个选项事实上是一对输入字符和关键字字符的组合(type-chars key-chars).
输入字符如下:
!　　他后面的字符不会被插入.这在我们要定义一个字符来格式化一行使之重新缩进时会显得更为有用.在默认的情况下,CTRL-F 就被定义为重新缩进.
*　　这一行会在这个字符输入之前进行重新缩进.
0　　如果这是一行中第一个输入的字符就会影响这一行的缩进发生变化.(这并不是说他是这一行的第一个字符,因为这一行会进行自动缩进.他只是指第一个输入的字符)
关键字符如下:
^X　　控制字符 CTRL-X
o　　告诉 Vim 在我们用命令 o 开始一个新行时要缩进该行.
O　　与 o 相同
e　　当我们输入最后一个 else 中的 e 时重新缩进此行.
:　　当我们在一个标签或是事件的描述后输入:时会重新缩进此行.
<^>,<<>,<>>,<o>,<e>,<O>　　在尖括号中的精确的字符
而 cinkeys 选项默认的值如下:
0{,0},:,0#,!^F,o,O,e
cinoptions 选项来控制每一行缩进的大小.这个选项是由一系列的关键字和缩进(key indent)所组成的.这里子关键字是一个单一的字符,用来指明影响程序的哪一部分.而缩进(indent)则是告诉程序要用多大的缩进.这个缩进可以是几个空格(如 8 个),或者是负数量的空格(如-8 个).还可以是由 s 所指定的 shiftwidth 选项值的倍数.例如 1s 是指一个 shiftwidth,而 0.5s 是指半个 shiftwidth,而-1s 则没有缩进一个 shiftwidth.
关键字　　默认　　描述
>　　s　　没有被其他字符所覆盖,正常缩进.

e 0 以花括号为结束标记的行后面的行的额外缩进.
n 0 在 if,while 后的没有在花括号内的单一行的额外缩进.
f 0 添加到函数体的额外缩进.包括定义函数的{}.
{ 0 添加到开始{的空格
} 0 添加到结束}的空格
^ 0 添加到开始于第一列的{}内的文本的空格.
: s 在 switch 语句中的自动缩进
= s 在 case 语句后的额外缩进
g s 对于 C++中的保护关键字(public,private,protected)的缩进
h s 保护关键字语句后的语句的缩进
p s K&R 风格的缩进
t s 在单一行进行函数类型声明的缩进
+ s 连续行的缩进
c 3 多行注释中间部分的缩进(如果没有指定*)
(2s 表达式中间部分一行的缩进,事实上是指一对括号内部的缩进.
u 2s 嵌套括号内的一行的缩进,与(相类似,只是这个要更深一层.
) 20 指定用来查找一对闭括号()的行数.
* 30 指定用来查找没有结束的注释的行数

选项 cinwords 用来定义哪一些单词可以使得下一个 C 语句在精巧缩进(smartindent mode)模式下和在 C 缩进模式下(Cindent mode)缩进一个层次.默认的选项值如下:

:set cinwords=if,else,while,do,for,switch

假如我们要用尽量少的编辑动作来比较两个文件的不同,那么我们要怎么样的来做呢?这时我们可以打开两个窗口,分别在两个窗口中进行编辑.然后我们在每一个窗口中执行下面的命令:

:set scrollbind

这样以后如果有一个窗口发生滚动,那么另一个窗口也就会有相同的动作.

就你是我们要在两个文件窗口中进行动作一样,也许有的时候我们会需要移动一个窗口而不要移动另一个窗口,这时我们可以我们要移动的窗口内执行下面的命令就可了:

:set noscrollbind

如果我们要同步滚动,我们可以执行下面的命令:

:set scrollbind

选项 scrollopt 可以用来控制 scrollbind 如何的来工作.我们可以将其设为如下的值:

ver 垂直同步滚动
hor 水平同步滚动
jump 当我们在两个窗口中进行切换时,一定要使用偏移(offset)为 0

最后如果我们要使两个窗口同步,我们可以使用下面的命令:

:syncbind

假如我们正在看一个文件的两个版本,我们需要在这两个文件中进行滚动查看.也许我们为了查看一些东西而关掉了 scrollbind.这样这两个文件就会停在不同的地方.而我们还希望他们可以再一次同步滚动.这时我们可以分别到这两个文件所在的窗口然后将他们移动相同的地方.而事实上我们可以叫 Vim 来完这样的工作.在这样的情况下,我们可以分别在两个文件中设置 scrollbind,然后执行下面的命令:

:syncbind

这样 Vim 就使得两个文件同步了.

假如我们正在看一个程序文件,但是却碰到一个我们并不理解的函数调用.我们可以用命令 CTRL-]

跳转到函数定义的地方.但是这样做却有一个问题,那就当前的文件被函数定义的内容所替代,我们也就不可以从屏幕上看到这个文件的内容了.
这个问题的一个解决办法就是我们使用被称为 preview 的特殊窗口.我们可以通过执行下面的命令来打开 preview 的特殊窗口,在这个窗口中显示函数定义的内容:
:ptag function
如果我们已经打开了一个 preview 窗口,那么他就会切换到当前正在查看的函数定义内容.如果我们要关闭这个窗口,我们可以执行下面的命令:
:pclose
命令 CTRL-Wz 和 ZZ 也可以有关闭这个窗口的作用.
在 preview 窗口中我们执行下面的命令来完成我们的工作:
:ppop 在这个窗口中执行一个:pop 命令.
:ptselect identifier 打开一个新的 preview 窗口并执行:tselect 命令
:ptjump identifier 打开一个新的 preview 窗口并执行:ptjump 命令
:count ptnext 在这个窗口中执行:count tnext 命令
:count ptprevious 在这个窗口中执行:count ptprevious 命令
:count ptrewind 在这个窗口中执行:count ptrewind 命令
:ptlast 在这个窗口中执行:ptlast 命令.
CTRL-W} 以当前光标下的内容执行一个:ptag 命令
CTRL-Wg} 以当前光标下的内容执行一个:ptjump 命令
matchpairs 选项可以用来控制哪些字符可以用%命令来进行匹配.这个选项默认的值如下:
:set matchpairs=(:),{:},[:]
这就告诉 Vim 要匹配(),{},[]
匹配<>,我们可以用下面的命令:
:set matchpairs=<:>
这个命令在我们要编写 HTML 文件时会显得更为有用.
这个命令仅仅是匹配<>.如果我们要同时匹配其他的字符,我们可以用下面的命令:
:set matchpairs=(:),{:},[:],<:>
这样的命令显得似乎是有一些太长了,我们可以用+=命令来达到同样的目的.例如上面的命令我们可以用下面的命令来作到:
:set matchpairs+=<:>
如果我们希望我们在输入括号时,光标会跳转到与其匹配的地方进行显示,我们可以执行下面的命令来做到:
:set showmatch
通常情况下这个跳转持续的时间为 0.5 秒(半秒),我们可以用 matchtime 选项来控制这个时间.例如如果我们希望这个时间持续 1.5 秒,我们可以用下面的命令:
:set matchtime=15
在这里是以 0.1 秒为单位的.
我们在 Vim 编辑器还可以用命令来查找未匹配的括号.如[{查找前一个未匹配的{,]{查找后一个未匹配的{.[}查找前一个未匹配的},而]}查找后一个未匹配的}.
])查找后一个未匹配的),而[(查找前一个未匹配的(.[#查找前一个未匹配的#if 或者是#else.而]#查找后一个未匹配的同类情况.
下面的命令可以移动一个 Java 方法的开头或结尾:
[m 向后查找一个方法的开头
[M 向后查找一个方法的结尾

]m　　向前查找一个方法的开头
]M　　向前查找一个方法的结尾
在 Vim 编辑器还提供了许多的移动命令来帮助程序人员在他们的程序文件中进行浏览.下面的一些命令可以找到位于第一列的{和}:
count[[　　向后查找位于第一列的前一个{
count[]　　向后查找位于第一列的前一个}
count]]　　向前查找位于第一列的后一个{
count][　　向前查找位于第一列的后一个}
命令[/和[*可向后移动他可以找到的第一个 C 注释的开始处,而]/和]*可以向前移动他可以找到下一个 C 注释的结束处.
随着一个程序文件变得越来越大,我们会将这个程序文件分在不同的目录中,这样就会大大的方便我们的管理.让我们来看一下一个小的工作,我们有一个包含有 main.c 和 main.h 的主目录,其余的目录就是含有 lib.c 和 lib.h 的 lib 目录(库文件).
我们从主目录中开始我们的编辑工作.第一件事就是告诉 Vim 关于是我们的新目录的情况.我们可以用:set ^=将这个目录放在查找路径的上部:
:set ^=../lib
假如我们正在编辑文件 main.c,而这个文件的内容类似于下面的:
#include "main.h"
#include "lib.h"

int main(int argc,char*argv[])
现在我们要查看一下 lib.h 中一个子程序的声明.一个办法是我们可以执行下面的命令:
:vi ../lib/lib.h
这个命令是假定我们知道 lib.h 的位置所在.但是在这里我们会有一个更好的方法.首先我们将光标放在下面一行中的文件名上:
#include "lib.h"
然后我们执行命令 gf.这就告诉 Vim 编辑试着编辑以光标下的内容为文件名的文件.编辑器就会在 path(路径)变量的每一个目录中进行查找.
假如我们要编辑文件 lib.c,而这个文件名并没有出现在现在的文本内容,我们就没有办法来用 gf 命令,这时我们可以用下面的命令:
:find lib.c
这个命令类似于:vi 命令,所不同的只是他要在路径中进行查找.下面的命令与其相类似,只是他是分裂当前窗口进行查找:
:sfind lib.c
gf 命令与:find 命令相类似,只是这个命令认为当前光标下的内容是我们要编辑的文件.如果在 path 中有不只一个文件与指定的文件相匹配,这时我们可以通过给定 gf 命令一个参数来选择我们要编辑的文件.
换句话说如果我们将光标放在 param.h 上然后执行命令 2gf,Vim 就会编辑通过 path 选项指定的路径目录中查找的文件列表中的第二个文件.
path 选项用来告诉 Vim 在哪里查找被当前文件包含进来的文件,这个选项的格式如下:
:set path=directory,directory,...
在这里的 directory 是指我们要查找的目录,如:
:set path=/usr/include,/usr/X11R6/include
我们还可以在这个命令中使用通配符来进行匹配,如:

:set path=/usr/include,/usr/include/*
下面是一些特殊的目录:
**　　匹配整个目录树,如:
　　:set path=/usr/include/**
　　这个命令查找目录/usr/include 及其所有的子目录.下面的命令指定了在所有以/ho
me/oualline/progs 开始以 include 结束的目录内的文件
　　:set path=/home/oualline/progs/**/include
""　　空字符串指当前目录
.　　指我们正在编辑的文件所在的目录
例如下面的命令是告诉 Vim 查找的目录包括/usr/include 及其所的子目录,我们正编辑的文件所在的目录(.)以及当前目录(,,):
:set path=/usr/include/**,.,,
如果我们想确定一下我们可以查找到所有的#include 文件,我们可以使用下面的命令:
:checkpath
这个命令的作用范围不仅仅是我们正在编辑的#include 目录,而且包括任何他们#include 的目录,结果就是要查看所有的#include 文件.
在这种情况下,有许多的文件要包含文件 stddef.h 和 stdarg.h.但是 Vim 却不能找到这些文件.如果我们要告诉 VimLinux 特殊的 include 目录,我们要以执行下面的命令:
:set path+=/usr/include/linux
但是:checkpath 命令只是列出所以不能找到的文件,如果我们要列出所以的#include 文件,我们可以用下面的命令:
:checkpath!
Vim 编辑器知道 C 和 C++的宏定义.但是如果是其他的语言又会怎么样呢?选项 define 包含了一个长规的表达式,Vim 编辑器可以通过他来查找一个定义.例如如果我们要使 Vim 查找以字符串 function 开头的宏,我们可以使用下面的命令:
:set define=function
选项 include 定义一个包含(include)的目录是什么样子的.这个选项可以用来为我们使用命令]CTRL-I,[CTRL-I,]d,[d 在这些我们所包含进来的文件中进行查找.这个选项也可以为命令:checkpath 所用.正如 define 选项一样,这个选项的值也是一个长规的表达式.
命令[i 用来查找光标下的内容第一次出现的地方.注释的文本会被忽略掉.
命令]j 用来查找光标下的内容下一次出现的地方.注释的文本会被忽略掉.
命令[I 会列出所有包含当前光标下的内容的句子,命令]I 与其相类似,只是这个命令是从当前光标处开始.
:make 命令会产生一个错误列表.Vim 编辑器会记住我们前 10 次:make 命令和:grep 命令的执行结果.如果我们要到前一次的错误列表,我们可以用下面的命令:
:colder
如果我们要到一个新的错误列表,我们可以用下面的命令:
:cnewer
当我们执行:make 命令时所要执行的程序名称是由 makeprg 选项来定义的.在通常的情况下会设为 make,但是 Visual C++的用户可以通过下面的命令将其设为 nmake:
:set makeprg=nmake
:make 命令会重定向 Make 的输出到一个错误文件.这个文件的名字是由 makeef 选项来控制的.如果这个选项包含有字符##,字符##就会被专一的数字所代替.这个选项默认的值取决于我们正在使用的操作系统.默认的值如下:

Amiga t:vim##.Err
UNIX /tmp/vim##.err
Windows vim##.err
我们可以在命令中包含指定的 Vim 关键字.%字符可以扩展当前文件的名字,所以我们执行下面的命令:
:set makeprg=make%
然后我们执行命令:
:make
他就会执行下面的命令:
$ make file.c
file.c 就是我们正在编辑的文件的名字.这个并没有太大的用处,所以我们可以重新定义这个命令并使用:r(root)的权限:
:set makeprg=make%:r.o
这样我们就会执行下面的命令:
$ make file.o
选项 errorformat 可以用来控制 Vim 如何来组织错误文件以使得他可以知道文件名以及错误发生的地方.这个选项的格式如下:
:set errorformat={string},{string},{string}
这里的字符是由特殊字符%所指出的典型的错误信息用来指明特殊的操作(与标准 C 函数 scanf 很相像).这些特殊的字符如下:
%f 文件名
%l 行号
%c 列号
%t 错误类型(单一字符)
%n 错误行号
%m 错误信息
%r 匹配一行中的剩余
%*{char}匹配并跳过由{char}所指定的 scanf 转换
%% 字符%
当我们在编译一个程序的时候,我们也许要在几个目录中进行遍历.GNU make 程序会在当我们进入一个目录或是离开一个目录时打印出相应的信息.
如果要正确的得到文件名,Vim 就要清楚的知道这些目录的变化.下面的一些错误格式用来在目录发生变化时告诉 Vim 一些相关的信息:
%D 当进入一个目录时打印出指定的信息.字符串的%f 指明我们所进入的目录
%X 指定离开目录时的信息.字符串中的%f 指明了 make 已用毕的目录.
一些编译器,例如 GNU GCC 编译器,会输入一些冗长的错误信息.如果我们正在使用默认的 errorformat 就会导致三种错误信息.这实在是够讨厌的.但是幸运的是 Vim 编辑器可以识别出不同的错误信息.处理不同信息的模式代码如下:
%A 开始多行信息
%E 开始多行错误信息
%W 开始多行警告信息
%C 连续多行信息
%Z 结束多行信息
%G 全局.只有在+或是-连接时才有用

%O 单行文件信息:重新读入匹配的部分
%P 单行文件信息:将%f 文件压入栈
%Q 单行文件信息:将最后一个文件压出栈
+或是-可以放在任何字符的前面,从而组成下面的内容:
%-letter 不要包含输出中的匹配行
%+letter 包含%m 错误字符串的整个匹配行
在通常的情况下,我们执行:make 命令并有错误发生,Vim 会在当前的窗口中显示错误文件.如果我们通过设置 switchbuf 选项来进行窗口的分裂,Vim 就会在一个新的窗口来显示错误文件.:grep 命令会运行由选项 grepprg 所指定的程序.这个选项包含了我们要用的命令行.#和%字符会扩展到当前文件名和交替文件名.而字符串$*将会被:grep 命令的参数所代替.
在这里我们要注意的就是在 UNIX 系统上,grepprg 默认是指 grep -n.在 Windows 系统上,默认是指 findstr/s
:grep 命令使用 grepformat 选项来告诉 Vim 如何来组织 Grep 的输出文件.
在通常的情况下,Vim 是使用二分法进行查找指定的标记名字.如果一个标记文件是按序排列的,这样的方法可以是很快速的.否则的话我们可以使用线性查找的方法.如果要强制进行线性查找,我们可以用下面的命令;
:set notagbsearch
这个选项会在我们的标记文件不是有序的时有用.
有一些系统会限制我们在函数名中所使用的字符数.如果我们要在 Vim 中加入这样的限制,我们可以通过设置 taglength 选项来限制我们函数名的最大长度.
我们可以用 tags 选项来指定标记文件名.这个可以用来指其他目录中的文件.如:
:set tags+=/home/oualline/tools/vim/tags
但是这会带来一些令人费解的地方.是我们在当前目录中启动并告诉 ctags 将标记文件放在目录/home/oualline/tools/vim 还是我们在当前的目录执行了 ctags 呢?现在的 Vim 编辑器已经用其他的选项来解决了这个问题了.如果我们进行下面的设置,所有的标记都会和含有标记文件的目录有关系:
:set tagrelative
否则的话,他们会和当前目录有关系.
如果我们设置了 tagstack 选项,那么:tag 命令和:tjump 命令就会建立一个标记栈.否则是不会保持栈的.
Vim 编辑器允许我们自定义用来进行语法高亮显示的颜色.Vim 编辑器识别下面三种不同的终端:
term 平常的白色背景色,黑色前景色的终端(没有颜色)
cterm 彩色终端,例如 xterm 或者是 Windows 的 DOS
gui Gvim 所产生的窗口
要改变通常的终端的高亮显示的颜色,我们可以用下面的命令:
:highlight group-name term=attribute
这里的 group-name 是我们要高亮显示的语法组.这是我们要设置的语法匹配的规则用来告诉 Vim 程序中的哪一部分要进行高亮显示.而 attribute 是终端的属性.对于平常的终端如下:
bold underline reverse italic standout
我们可以用逗号来组合这些属性,如:
:highlight Keyword term=reverse,bold
假如我们有一个不是通常代码的终端.我们可以通过 start 和 stop 高亮显示的选项来定义我们自己的属性.这些定义一个用来发送的字符串来开始一个颜色和结束一个颜色.例如:
:highlight Keyword start=<Esc>X stop=<Esc>Y

有了这样的定义,当 Vim 要显示关键字时,例如显示 if 就会显示为<Esc>Xif<Esc>Y
如果我们对 UNIX 的终端定义的文件较为熟悉,我们可以使用终端代码.us 定义了开始下划线的代码,而 ue 则是退出下划线模式的字符串.要指定这种高亮显示的方法,我们可执行下面的命令:
:highlight Keyword start=t_us stop=t_ue
颜色是由 cterm 的设置来定义的.我们可以使用 cterm=attribute 的方式来进行我们的设置.
但是对于一个彩色终端来说还有许多其他的选项.ctermfg=color-number 可以用来设置前景色.ctermbg=color-number 用来设置后景色.Vim 可以识别出颜色的名称.例如下面的命令可以告诉 Vim 在显示注释时后景色为蓝色,而前景色为红色,并且有下划线:
:highlight Comment cterm=underline ctermfg=red ctermbg=blue
GUI 终端可以使用选项 gui=attribute 的方式来在图形窗口下显示语法元素的属性.选项 guifg 和 guibg 定义了前景和后景的颜色.这些颜色是名称来进行区别的.如果名称中包含空格,那么这个颜色的名称就要用单引号括起来.为了事物的可移动性,Vim 建议我们只使用以下的颜色:Black Blue Brown Cyan DarkBlue DarkCyan DarkGray DarkGreen DarkMagenta DarkRed Gray Green LightBlue LightCyan LightGray LightGreen LightMagenta LightRed LightYellow Magenta Orange Purple Red SeaGreen SlateBlue Violet White Yellow
我们可以使用 X11 的颜色数字来定义我们的颜色.这就可以在所有的系统上正确的显示,而不论是否使用 X11 系统.这种模式为#rrggbb,在这里 rr 是红色的数量,bb 是蓝色的数量,gg 是绿色的数量.
我们还可以在一个高亮显示行来定义几种终端的颜色,如:
:highlight Error term=reverse cterm=blod ctermfg=7 ctermbg=1
语法元素是由$VIMRUNTIME/syntax 中的宏来定义的.然而为了使得事情变得更为简单,我们常会用下面的一些名字:
Boolean Character Comment Conditional
Constant Debug Define Delimiter
Error Exception Float Function
Identifier Include Keyword Label
Macro Number Operator PreCondit
PreProc Repeat Special SpecialChar
Structure Tag Todo Type
Typedef
除了这些语法元素,Vim 还定义了下面的许多事物:
Cursor 光标下的字符
Directory 目录名称以及其他列出的特殊名称
ErrorMsg 在最底行显示出的错误信息
IncSearch 增长(Incremental)查找的查找结果
ModeMsg 在左下角显示的模式名称
MoreMsg 当 Vim 在显示一个很长的信息并且要显示更多的信息时的提示
NonText
Vim 编辑器会在超出文件结尾时显示~.我们可以用@来表明一行不会在显示一屏上.这些语法元素可以用来定义用哪些颜色来显示语法元素
Question 当 Vim 询问问题时.
SpecialKey 命令:map 列出键盘的映射.这个选项定义了用特殊键来进行高亮显示
StatusLine 当前窗口的状态行.
StatusLineNC 其他窗口的状态行

Title　　命令:set all,:autocmd 的输出标题
Visual　　这个颜色用来高亮显示可文本块
syntax 选项包含有用于当有语法高亮显示的语言.我们可以用下面的命令来关闭语法显示:
:set syntax=off
如果想要打开,我们就用下面的命令:
:set syntax=on

在这一次的学习中我们会介绍一些更多的关于缩写和键盘映射的问题.
我们在编辑的过程中可以用:abbreviate 命令来设置一个缩写,那么我们如何来移除一个缩写呢?我们可以用命令:unabbreviate 来移除一个缩写.例如我们用下面的命令来设置一个缩写:
:abbreviate @a fresh
如果我们要移除这个缩写我们可以用下面的命令:
:unabbreviate fresh
如果我们要清除所有的缩写,我们可以用下面的命令:
:abclear
我们用上面的命令定义的缩写可以正常的工作在插入模式和命令行模式两种状态下.如果我们是在文本中输入@a,他就会扩展为 fresh,而如果我们在:命令行中输入@a,他也可以扩展成为 fresh.如果我们要定义一个只工作在插模式下的缩写,我们可以用这样的命令:
:iabbreviate @a fresh
这也就是说如果我们在命令行输入@a,那么他仅是@a,而不会扩展为 fresh.如果要取消一个插入模式的缩定义,我们可以用下面的命令:
:iunabbreviate @a
同样的我们可以用下面的命令来清除所有的插入模式的缩写定义:
:iabclear
相类似的,如果我们要定义一个只在命令行模式下工作的缩写,我们可以用命令:cabbreviate 来完成,而取消这个定义的命令为:cunabbreviate,如果要清除所有的缩写列表,我们可以用下面的命令:
:cabclear
如果我们要列出所有的缩定,我们可以用下面的命令:
:abbreviate
在这个命令的执行结果第一列显示出缩写的类型,标记如下:
c　　命令行模式
i　　插入模式
!　　两种模式均可
CTRL-C 命令可以使得 Vim 离开插模式.这个命令与<Esc>命令的不同之处就在于在回到正常状态的过程中并不会检查一个缩写.
map 命令可以使得我们将一定的模式与键盘对应起来.例如如果我们要使用 F5 来复制来选中的文本到寄存器 v 中,我们可以用下面的命令来定义:
:map <F5>"vy
这样的定义的F5在正常以及可视模式下都可以使用.但是也许我们真正希望的是只在可视模式下来使用这个命令,这时我们可以用下面的命令来进行定义:
:vmap <F5>"vy
这里的 v 是告诉 Vim 这样定义的命令是只在可视模式下使用.

如下面的列表:

Command	Normal	Visual	Operator Pending	Insert	Command Line
命令	正常	可视	运算符延伸	插入	命令行
:map	y	y	y		
:nmap	y				
:vmap		y			
:omap			y		
:map!				y	y
:imap				y	
:cmap					y

现在假如我们要定义<F7>以使得命令 d<F7>可以删除 C 程序的文本块.与此相类似的,y<F7>可以将程序块复制到未命名寄存器中.所以我们要做就是要傅 F7 来选择当前的文本块.我们可以使用下面的命令:

:omap <F7>a{

这个命令会使得<F7>在 operator-pending 模式下选择文本块.有了这样的映射,当我们输入 d<F7>命令中的 d 时,我们就进入了 operator-pending 模式.然后执行命令<F7>就可以地命令 a{了,这样我就可以选择文本块了.因为我们执行了 d 命令,所以这个文本块被删除了.

其他的一些映射命令如下:

:map lhs rhs

这个是将 lsh 映射到 rhs,所以当我们按下 lhs 时我们实际上执行的是 rhs

如下面的映射命令:

:map ^A dd

:map ^B ^A

执行了这样的命令以后,当我们输入 CTRL-A 时 Vim 会删除一行.而 CTRL-B 也会是同样的作用.当我们使用控制字符时,我们必须用 CTRL-V 来引用他.换句话说如果我们要达到:map ^A dd 的目的,我们就可以用下面的命令来完成:

:map CTRL-VCTRL-A dd

(似乎这个命令这样的做是不成的)

如果我们要重新映射,我们可以使用命令:noremap,例如:

:noremap lhs rhs

如果我们要取消一个映射,可以使用:unmap 命令,如:

:unmap lhs

如果我们取消所有的映射,我们可以使用命令:

:mapclear

但是我们在使用这个命令时要注意,因为这个命令也会移除所有我们自定的默认映射.

如果我们要列出所有的映射,我们可以用下面的命令:

:map

第一列的标记指明了这样的映射可以在哪一种模式下工作:

字符	模式
<space>	正常,可视,运算符(operator-pending)
n	正常
v	可视
o	operator-pending
!	插入和命令模式

i 插入模式
c 命令模式
第二列指出各种 lhs 的任何映射.第三列是 rhs 的映射值.如果 rhs 是以*开头的,那么这个 rhs 是不可以重新映射的.
:map 命令可以列出所有的映射,而:map! 只列出插入和命令行模式的映射.而:imap,:vmap,:omap,:nmap 命令只是列出指定模式的映射.在默认的情况下,Vim 允许循环映射,要关掉这个特征,可以执行下面的命令:
:set noremap
如果我们执行下面的命令:
:abbreviate @a ad
:imap ad adder
这样当我们输入@a 时,字符 ad 会被插入,然而 ad 又映射到插入模式的字符串 adder,所以字符串 adder 会被插入到文本中.如果我们使用用命令:noreabbrev 就可以避免这样的问题.

虽然 Vim 编辑器在可视的情况下可以极好的完成我们的工作,但是有时我们是也需要使用命令行命令的.例如在脚本中命令行命令的使用会更容易,同时有许多特别的命令是只在命令模式下才可以实现的.
:delete 命令可以删除一个范围内的文本行.例如我们要删除 1 到 5 行,我们可以用下面的命令来做:
:1,5 delete
:delete 命令的一般格式如下:
:range delete register count
在这个命令中的 register 是我们的删除的文本要放入的寄存器.这个可以是我们用 a-z 命名的寄存器中的一个.如果我们使用大写的字符做为寄存器的名字,那么这些文本就会被添加到已经存在文本的寄存器中.如果没有指定这个参数,那么就会使用未命名的寄存器.而 count 则指出要删除的行数.range 则是指明要使用的行.
我们还可以使用命令来删除含有指定字符串的行.如我们可以用下面的命令来删除从第一个包含字符串 hello 到第一个包含字符串 goodbye 的行之间所有的行:
:/hello/,/goodbye/ delete
在这里我们要注意的就是如果 goodbye 在 hello 之前出现,那么这个命令就不会正常的工作了.我们还可以使用偏移量(offset)来重新定义要查找的字符串.例如/hello/+1 是指含有 hello 字符串行的下一行,因而我们还可以使用下面的命令来进行删除的操作:
:/beach/+1,/seashore/-1 delete
我们还可以使用下面的一些简写的运算符:
/ 向前查找上一次使用的模式
? 向后查找上一次使用的模式
& 向前查找上一次使用的子模式
(注:这个地方不懂,也不晓如何来用)
我们还可以使用链式的模式,例如下面的命令在找到字符串 first 以后要查找 second 字符串:
/first//second
所以我们还可以使用下面这样的删除命令:
:/hello//goodbye/ delete
我们还可以指定行号,用来指明在第几行进行查找,如果我们要从第七行开始查找,我们可以使用下面的命令:

7/first/
如果我们只是执行一个:命令,那么 Vim 编辑器就会进行命令行模式然后允许我们指定一个要删除的范围.如果我们在这个命令之前指定了一个数字,例如 5:,那么要删除的范围就是这个数字所指定的范围(包括当前行).事实上这样指定的范围如下:
:...+count-1
例如我们在一段文本中执行下面的命令:
3:delete
这实际上是执行下面的命令:
:...+2 delete
这个命令可以删除当前行以及当前行以下的两行,总计三行的文本.
删除命令的另一个形式如下:
:line delete count
在这种情况下,:delete 命令回到 line 所指定的行(默认为当前行),然后删除 count 行文本(包括当前行).
:copy 命令是将几行的文本从一个地方复制到另一个地方.这个命令的一般格式如下:
:range copy address
如果没有特别指定,range 默认是指当前行.这个命令是拷贝 range 所指定的范围行的文本到 address 指定行的后面.
与:copy 命令相类似的是:move 命令,所不同的只是这个命令是移动而不复制.
假如我们要在我们正编辑的文本中插入一些行,而由于某些原因,我们要使用命令行的方式来完成这样的工作.这时我们就要将光标移动我们希望新行出现的上一行.换句话说,我们是希望将插入的文本出现在当前行的后面.然后我们可以执行命令:append 来开始我们的插入过程,我们输入我们要插入的内容并用句号(.)来结束一行的输入.例如我们在一个测试文本中执行下面的命令:
:1append
这个命令是要第一行的后面插入我们新的文本行.执行这个命令后,我们可以在 Vi 的底部输入我们要插入的内容.当我们要结束我们的输入时,只要在一新行输入.就可以了.这样我们输入的内容就会出在第一行的后面了.
:append 命令的一般格式如下:
:line append
我们输入的新文本将会插入在 line 所指定的行后面.
:insert 命令也可以插入文本,这个命令的一般形式如下:
:line insert
这个命令与:append 命令相似,所不同的是后者是当前行的后面插入文本,而前者是在当前的前面插入文本.
我们不必打开 number 选项也可以实现在打印一行文本时打印此行的行号.命令:#与:print 命令相类似,所不同的只是前者可以打印出行号.
例如在我们的测试文本中执行下面的命令:
:1 print
执行结果如下:
A UNIX sales lady,Lenore,
而我们执行下面的命令:
:1#
执行结果如下:
1 A UNIX sales lady,Lenore,

选项 list 可以使得不可见的字符成为可见字符.命令:list 可以列出指定的行,而且这个命令全认为 list 选项已经打开.如下面的命令:

:1,5 list

这个命令就可以列出 1-5 行的内容.

这个命令与:print 命令不同的地方只是这个命令也可以打印出回车,Tab 等不可见的字符.

:z 命令可以打印出一个范围内的文本行(默认情况下为当前行)以及这一行附近的行.例如我们执行下面的命令:

:100 z

这个命令会打印出从第 100 行开始的直到当前屏幕满屏的所有的文本行.

这个命令可以指定一个数字,用来表示除了打印指定的行以外要额外打印的行.例如下面的命令:

:100 z 3

这个就使得 Vim 除了打印第 100 行还要另打印额外的三行.

:z 命令以后还可以再跟上一个代码用来表示要显示多少的文本.可用的代码如下:

代码	起始行	结束行	当前行
+	当前行	向前一个屏幕	向前一个屏幕的下一行
-	向后一个屏幕	当前行	当前行
^	向后两个屏幕	向后一个屏幕	向后一个屏幕
.	向后半屏	向前半屏	向前半屏
=	向后半屏	向前半屏	当前行

基本的 substitute 命令格式如下:

:range substitute /from/to/flags count

这里的定义符可以是除了字母,数字,反斜线,双引号或是竖线以外的任何字符.在 Vim 编辑器中 Vim 还使用一些特殊的字符来表示特殊的事物.例如*表示重复 0 次或是多次.如果我们设置了 nomagic 选项,那么这些字符的特殊意义就会被关掉了.

命令:smgic 可以执行一个替换操作,但是这个命令要求我们设置了 magic 选项.

例如我们可以使用只有一行的文件来测试这些命令.我们可以用命令将整个文件打印出来:

:%print

Test aaa* aa* a*

然后我们设置了 magic 选项并且执行替换命令 .p 标记告诉编辑器打印出他所改变的行:

:set magic

:1 substitute /a*/b/p

命令的执行结果如下:

bTest aaa* aa* a*

这个命令只是改变了一行开始的部分.为什么会将 Test 变为 b*Test 而且并没有 a 呢?这就是因为*可以匹配 0 次或是多次,而 Test 正是以 0 个 a 开始的.但是为什么只是替换了一次呢?这是因为:substitute 命令中是改变第一个出现的地方.如果我们使用 g 标记就可以替换全部的匹配项了,我们撤销刚才的命令并执行下面的命令:

:undo

:1 substitute /a*/b/pg

这个命令的执行结果如下:

bTest b*b b*b b*

现在我们在关闭 magic 选项的情况下再做一次:

:undo

:set nomagic

:1 substitute /a*/b/pg
这个命令的执行结果如下:
Test aab ab b
在没有设置 magic 的情况下,*仅是一个*.
而:smagic 命令则是在执行替换命令时强制转换*以及其他一些字符的意义,例如我们执行下面的命令:
:undo
:smagic /a*/b/pg
这个命令的执行结果如下:
bTest b*b b*b b*
而相类似的是命令:snomagic 选项强行关掉 magic 选项:
:undo
:snomagic /a*/b/pg
这个命令的执行结果如下:
Test aab ab b
&命令可以重复执行替换.这个命令可以保存旧的 from 和 to 的字符串,但是允许我们使用不同的范围(range)和标记(flags).这个命令的一般形式如下:
:range & flags count
例如我们执行下面的命令:
:1 substitute /a+/b/p
这个命令的执行结果如下:
Test b* aa* a*
这个命令可以改变第一个出现 from 所指的字符的地方.但是我们希望是的整个一行都要发生相应的替换,这时我们可以重复这一次替换命令:
:&g
这一次在命令的执行过程中就不会打印执行结果,因为我们在这里指定的标记是 g,而不是 p.这个命令的执行结果如下:
Test b* b* b*
命令:&和命令:substitute 在没有指定替换字符串的情况下作用相同,都可以执行上一次的替换命令.
在正常的命令状态下命令&可以重复上一次的:substitute 命令.例如如果我们执行下面的命令就会将第五行中的字符 manager 变为 idiot:
:5 substitute /manager/idiot/
这时如果在正常的命令模式下我们执行命令&,那么这一行中的下一个 manager 字符串也会发生变化.如果我们下移一行然后执行命令&,那么这一行也会发生相应的变化.如果我们在这个命令中指定了 count,那么这个命令就可以作用多行.
:~命令与命令&g 相类似,所不同的是前者使用的字符是上一次使用/或是?查找时使用的字符串,而不是上一次:substitute 命令中的字符串.这个命令的一般格式如下:
:range~ flags count
在一般的情况下,:substitute 命令只是改变一行中第一个出现指定的字符串处,除非我们使用了 g 标记.如果我们希望 g 标记能成为默认的设置,我们可以使用下面的命令:
:set gdefault
但是这里我们要注意的就是也许这样的设置会打断我们的一些脚本.
到了现在我们所说过的一些命令都是有一个限制的,那就是我们所要执行的命令只作用在相邻的

行上.然而有时我们所希望是改变含有特定类型的行,这时我们就要用到:global 命令了.这个命令的一般形式如下:

:range global /pattern/command

这个命令可以告诉 Vim 编辑器对在指定的范围内包含有指定的类型的所有行执地指定的命令.例如如果我们要打印出一个文件中所有包含单词 Professor 的行,我们可以使用下面的命令:

:%global /Professor/ print

而命令:global!将对所有的行执行指定的命令,但是却不匹配指定的类型.与其相类似的是:vglobal 命令.

命令:ijump 查找指定的类型,并且会跳转到指定的范围内第一个出现的单词处.这个命令不仅会在当前文件中进行查找,而且会在由#include 所包含进来的文件中进行查找.这个命令的一般格式如下:

:range ijump count [/]pattern[/]

如果我们在这个命令中指定了 count,那么就会跳转到第 count 个类型出现处.这个类型会被看作是精确的文本,除非是由斜线括起来的.

例如下面的一个 Hello.c 的文件:

```
#include <stdio.h>
int main()
{
    printf("Hello Worldn");
    return (0);
}
```

如果我们执行下面的命令就会跳转到第一个含有 define EOF 的行处:

:ijump /defines*EOF/

在我们目录的情况,他是在包含进来的文件 stdio.h 中

与这个命令相类似的是命令:ilist,所不同的只是这个命令是列出相应的行而不是跳转到相应的行.

命令:isearch 与:ilist 命令相类似,只是这个命令列出第一个出现指定内容的行.

最后命令:isplit 是命令:ijump 和命令:split 的组合.

我们已经知道用命令[CTRL-D 来跳转到当前光标下的宏定义处.下面的命令也可以起到同样的作用:

:djump name

例如我们要跳转到宏 MAX 处,我们可以用下面的命令:

:djump MAX

执行这个命令我们并不必需要知道我们定义的宏的全名,如果我们只是知道宏名的一部分,我们就可以使用由斜线括起来的常规表达式来进行相应的查找.如下面的命令:

:djump /MAX/

我们还可以指定参数来控制这个命令的查找范围,如下面的命令:

:50,100 djump /MAX/

这个命令只在第 50 到第 100 行内进行相应的查找.

如果我们希望查找到是第二个而不是第一个宏定义,我们可以通过指定 count 参数来做到.例如我们要查找第二个 MAX 的定义处,我们可以用下面的命令:

:djump 2 MAX

下面的命令会在一个新窗口内显示宏定义:

:range dsplit count [/]pattern[/]

这个命令是命令:split 和命令:djump 的组合.

命令:dlist 与命令:dsplit 相类似,只是前者并不会移动到一个宏的定义处,而是列出匹配的宏定义.
而命令:dsearch 只是显示第一个匹配的宏定义.如果命令:ilist,ijump,:djump,:dlist,:dsearch 带上!选项,那么在注释内的定义也可以进行查找.
如果我们要改变当前 Vim 的工作目录,我们可以用下面的命令:
:cd dir
这个命令与系统的 cd 命令相类似.如果没有指定目录,那么就会回到我们的用户主目录.
要查看当前 Vim 的工作在哪一个目录下,我们可以有用下面的命令:
:pwd
要回到前一个工作目录,我们可以用下面的命令:
:cd -
下面的命令可以打印出当前文件的名字以及一些相关的行信息:
:file
如果我们要改变当前文件的名字,我们可以使用下面的命令:
:file name
例如我们正在编辑的文件名为 complete.txt,我们可以使用:write 命令进行保存.现在我们要缩短文件并且存为 summary.txt,我们可以执行下面的命令:
:file summary.txt
如果我们继续进行编辑,那么所的改动都会被存入文件 summary.txt 中.
与:file 命令相似,命令:=可以打印出当前行的行号.
:write 命令可以将缓冲区或是一个文件中指定范围内的文本存入.他还有一些其他的选项.例如下面的命令就可以将我们正编辑的文件内容追加到文件 collect.txt 中:
:write >> collect.txt
如果这个文件并不存在,那么这个命令就会给出错误信息.如果我们要强行追加,可以使用!选项:
:write!>>collect.txt
:write 命令不仅可以保存文件,而且可以将文件导入其他的程序.在 Linux 系统中,我们可以用下面的命令将文件发送到打印机:
:write !lpr
(注::write! lpr 与:write !lpr 这两个命令的不同,前才是强行保存文件而后者则是将文件发送到打印机)
命令:uddate 与命令:write 相类似,所不同的只是如果缓冲区没有被修改,那么这个命令就不会起作用了.
:read 命令将会读入一个文件.这个命令的一般格式如下:
:line read file
这个命令会将名为 file 的文件读入并且插入在 line 后面.如果没有指定文件,那么就会使用当前的文件,如果没有指定要插入的行,那么就会使用当前的行.
与:write 命令相类似,:read 命令可以使用一个命令而不是一个文件.如果要读入一个命令的输出并插入到当前行的后面,我们可以使用下面的命令:
:line read !command
我们在以前的学习曾学过如何在寄存器中录制宏.如果我们要在命令行中使用这些宏,我们可以用下面的命令来执行寄存器中的宏内容:
:line@register
这个命令会将光标移动到指定的行,然后执行寄存器中的内容.这就意味着下面的命令执行上一次的命令行:
:@:

如果要执行上一次的:@register 命令,我们可以用下面的命令:
:line@@
命令:>使得文本右缩进,:<命令使文本向左缩进.例如下面的命令将会使第五行到第十行向右缩进:
:5,10>
:change 命令与:delete 命令相类似,所不同的只是他还同时执行:insert 命令,也就是有我们可以同时输入我们要插入的文本.
命令:startinsert 命令可以开始插入模式,就像是我们在正常模式下执行 i 命令一样.
如果我们要将几行合并为一行,我们可以使用命令:join,在这个命令的执行中将会使用空格来分隔这几行.如果我们不希望加入空格,我们可以用下面的命令来合并:
:join!
下面的命令可以将指定的行的文本复制到寄存器中:
:range yank register
如果没有指定寄存器,将会使用未命名寄存器.
:put 命令会将寄存中的内容粘贴到指定的文本行的后面.例如要将寄存器中的内容粘贴到第五行的后面,我们可以用下面的命令:
:5put a
如果要将文本放在这一行的前面,我们可以用下面的命令:
:5put! a
:undo 命令会撤销上一次的命令操作,与 u 命令相类似.而:redo 命令会重做撤销的操作,与命令 CTRL-R 命令相类似.
如果要标记一行的开始,可以用下面的命令:
:mark {register}
如果指定了行,那么那一行将会被标记.命令:k 将会起到同样的作用,只是我们不需要在寄存器名前加上一个空格.例如下面的命令:
:100 mark x
:100 ka
命令:preserve 可以将整个文件写入 swap 文件.这就使得我们可以在没有原始文件的情况下来修复我们的编辑部分.
如果我们要执行单一的 Shell 命令,我们可以用下面的命令:
:!cmd
cmd 就是我们要执行的系统命令.
例如要查看当前的日期,我们可以用下面的命令:
:!date
如果我们要重复上一次的 Shell 命令,我们可以用下面的命令来做:
:!!
最后下面的命令可以挂起 Vim 而进入命令提示行:
:shell
现在我们就可以执行各种的系统的命令了.在我们完成我们的工作以后,我们可以用 exit 命令回到 Vim 编辑器.
下面的一些选项可以控制命令的执行:
shell 我们要执行的命令名
shellcmdflag 跟在命令后的标记
shellquote 在命令中的引用字符
shellxquote 命令中的引用字符和重定向

shellpipe　　使用管道的字符串
shellredir　　重定向输出的字符串
shellslash　　在文件名中使用向前的斜线(只在 DOS 中使用)
命令:history 可以当前命令模式下的命令历史
Vim 编辑器可以记录各种命令的历史,下面的标记指出所记录的历史类型;
c　　cmd　　:　　命令行历史
s　　search　　/　　查找字符串历史
e　　expr　　=　　表达式寄存器历史
i　　input　　@　　输入行历史
a　　all　　所有的历史
如果我们列出所有的历史,我们可以用下面的命令:
:history all
:history 命令的一般格式如下:
:history code first,last
如果没有指定 first 和 last,那么就会列出所有的命令.first 参数默认是指历史第一个输入的,而 last 就是指最后一个.负数是指由历史的结束处向前的数的第几个.例如-2 是指最后个命令的输入.
例如下面的命令列出第一个到第五个的命令行历史:
:history c 1,5
而下面的命令则是列出了上五次的查找历史:
:history s -5
history 选项可以用来控制记录的历史命令数.例如我们要将记录的历史命令数增加为 50,我们可以用下面的命令:
:set history=50
Vim 编辑器可以记录上几次的错误以及在屏幕最后一行显示的信息.要查看这些信息,我们可以用下面的命令:
:messages
下面的命令可以使得信息的输出在显示在屏幕的同时会拷贝到一个文件中:
:redir > file
如果要停止拷贝,可以使用下面的命令:
:redir END
这个命令在保存调试信息时会显得更为有用.
我们还可以用:redir 命令将输出追加到文件中:
:redir >> file
:normal 命令可以使我们执行一个正常模式下的命令.例如下面的命令是将光标下的单词改为 DONE:
:normal cwDONE
在行这些命令时要求命令必须是一个完整的命令.如果我们已经挂起 Vim 然后执行命令时,Vim 显示全直到命令完全时也会发生变化.
下面的命令会将当前的文本写入文件,然后退出:
:exit
如果我们使用!选项,即时这个文件被标记为只读,那么 Vim 也会强行保存.
我们还可以在命令行指定一个文件名,那么当前的内容就会在退出以前写入我们所指定的文件,例如下面的命令:
:exit save.txt

如果我们只是想着将文件中的一部分保存到另一个文件,我们可以指定一个范围来保存.例如要保存 100 行,我们可以用下面的命令:

:1,100 exit save.txt

下面的命令与:exit 命令相类似,不同的是这个命令总是会保存文件:

:range wq! file

而:exit 命令只是在文件发生改变时才会保存.

Vim 编辑器是一个可定制性很强的编辑器.在这一次的学习中我们将会看到如何来定制我们的 Vim 编辑器的图形界面.

假如我们现在正在终端窗口下使用 Vim 编辑器,而现在我们要切换到图形界面下,我们可以用下面的命令来做到:

:gui

(注:但是这个命令要求我们的在编译 Vim 时加入了这个选项)

当我们的启动 gvim 时,这个窗口的位置是由窗口系统来控制的.在 UNIX 下,这个窗口的大小就是我们的启动编辑时终端窗口的大小.换句话说,如果我们有一个 24X80 的终端窗口,那么我们的在启动 Vim 编辑器时就会得到一个 24X80 的编辑窗口.如果有一个比较大的终端窗口,例如是 50x132 的,那么我们就会得到一个这样大小的编辑窗口.在 UNIX 系统下,我们可以使用-geometry 标记来指定启动的 vim 的位置和大小.这个选项的一般形式如下:

-geometry width+x heightx_offset-y_offset

这里的 width 和 height 以字符数指定了窗口的大小.而 x_offset 和 y_offset 则是指定了窗口的位置.x_offset 以像素数指定了屏幕的左边和窗口的左边的距离.如果这个值是一个负数,则是指定编辑器的左边和屏幕的右边的距离.与此相类似,y_offset 则是指定了上边缘,如果为负数,则是指定了下边缘的空白大小.所以如果我们要在屏幕的左上角启动 gvim,可以用下面的命令:

$ gvim -geometry +0+0

参数 width 和 height 则是指定了所启动的窗口的大小,也就是所具有的行数和列数.例如如果我们需要一个 80x24 的编辑窗口我们可以用下面的命令:

$ gvim -geometry 80x24

我们在图形界面下可以用下面的命令来得到当前窗口所在的位置:

:winpos(以左上角为标准)

如果我们要移动到指定的位置,我们可以用下面的命令形式:

:winpos X Y

例如如果我们要将当前的窗口移动到(20,30)处,我们可以用下面的命令:

:winpos 20 30

下面的命令可以显示当前编辑窗口的行数:

:set lines?

要改变这个值,我们可以用下面的命令形式来做到:

:set lines=lines

这里的 lines 则是我们希望在新的编辑窗口中所具有的行数.

与此相类似的如果我们要改变当前窗口的列数,可以用下面的命令形式:

:set columns=columns

在比较老一些的 Vim 版本中还会有:winsize 这个命令.这个命令并不推荐大家使用,因为我们可以用:set lines 和:set columns 命令来代替了.

我们还要可以用 guioptions 选项来控制许多的 GUI 的基本特征.这个命令的一般形式如下:
:set guioptions=options
options 是一个字母的集合,每一个是一个选项.
下面的是一些 Vim 定义的选项:
a Autoselect
如果我们设定了这个选项,当我们在的可视化模式下选择了文本,Vim 会试着将我们的所选择的文本放在系统的全局寄存器中.这就意味着我们可以在这个 Vim 中选择文本然后用命令"*p 将这些文本粘贴到另一个 Vim 中.如果没有这个选项,那么我们就要用命令"*y 来将所选择的文本复制到系统寄存器中.这也意味着在系统寄存中的文本也可以被其他的程序所使用.例如在 UNIX 系统上我们可以在可视化模式下选择文本然后用鼠标中键将其他复制到系统的终端窗口中.如果我们在 Windows 平台下,我们在可视化模式下选择的文本会被自动放到系统寄存器中.这就意味着我们可以在 Vim 编辑中选择文本然后将其粘贴到 Word 文档中.
f Foreground
在 UNIX 系统上,gvim 命令可以执行 fork()命令,这样编辑器就可以在后台运行.我们可以通过设置这样的选项来防止这样的事情发生.如果我们要编辑一个脚本程序需要执行 gvim 命令使得用户可以编辑文件而且要等待到编辑工作结束时,这样的选项就会显得更为有用.如果我们要调试一个程序,我们就会发现这时的 f 标记也是相当有用的.(注:这里我们的注意的就是这个要在初始文件中进行设置)
i Icon
如果我们设置了这个选项,gvim 就会在 X Windows 系统上运行而且最小化时会显示一个图标.如果没有设置这个选项,这时只会显示我们正在编辑的文件的名称而不会显示图标.
m Menu 可以显示菜单栏
M Nomenu
如果在初始时设置了这个选项,那么系统菜单的定义文件$VIMRUNTIME/menu.vim 就不会被读入.(注:这个选项要在 vimrc 文件中进行设置)
g Gray
这个选项可以将那些不可用的菜单内容显示为灰色.如果没有设置这个选项,那些不可用的菜单内容就会从菜单栏或是工具栏中移除.
t Tear off 打开 tear off 菜单
T Tool bar 包括工具栏
r Right scrollbar 在编辑器右侧放置滚动条
l Left scrollbar 在编辑器在侧放置滚动条
b Bottom scrollbar 在编辑器底部放置滚动条
v Vertical dialog boxes
在对话框中采用垂直的排列顺序
p Pointer callback fix
我们可以用这个选项来处理当 X11 窗口管理器发生问题时的情况.这样可以使用程序可以产生定点回溯.我们必顺在 gvimrc 文件中进行设置.
我们可以通过 toolbar 选项来控制 toolbar 的外观.他的一些值如下:
icon 显示工具栏图标
text 显示文本
tooltips 当光标位于图标上时显示的文字
Vim 编辑器默认的设置如下:
:set toolbar=icons,tooltips

如果我们要关掉工具栏,我们不可以将这个选项设为空字符串,而是用下面的命令来做到:
:set guioptions-=T
如果我们使用终端窗口进行编辑,一些终端可以允许我们更改当前窗口的标题和图标.
例如如果我们要将当前窗口的标题改为我们正在编辑的文件名,我们可以用下面的命令:
:set title
有时文件的全名要比我们所有的标题的空间长.我们可以用下面的命令来改变文件可以占用的空间数量:
:set titlelen=85
在这种情况下,标题的文本可以占用标题栏 85%的空间.
如果我们不喜欢 Vim 为我们设置的标题,我们可以用下面的命令形式来进行更改:
:set titlestring=Hello World!
当我们要退出 Vim 时,Vim 会试着重新载入先前的标题.如果他不能成功载入(在这一点上 Vim 并没有记忆功能),编辑器就会将标题设置为有由 titleold 选项所指定的字符串.例如:
:set titleold=vim was here!
如果窗口设置了图标(iconified),icon 选项就会告诉 Vim 是否试着将文件名放入图标标题.如果设置了这个选项,Vim 就会试着更改图标的文本.如果我们不喜欢 Vim 默认的设置,我们可以通过 iconstring 选项来指定图标的文本.如果我们设置了 icon 选项使得图标下面的字符串包含用我们当前正在编辑的文件名(或者是我们通过设置 iconstring 达到同样的作用).如果关闭了这个选项,光标就会有一个一般的 Vim 标题.
Vim 编辑器是 UNIX 系统上一个可以使用鼠标进行操作的一个文本编辑器.这就意味着我们可以使用鼠标进行许多的编辑操作.我们可以用下面的一些我们讨论的内容定义我们的鼠标操作.在一般的情况下,当我们要从一个 Vim 编辑窗口移动到另一个 Vim 编辑窗口时,我们要使用一些窗口切换的命令,如 CTRL-Wj 或是 CTRL-Wk.如果我们执行了下面的命令,那么光标所在的窗口就是我们的当前窗口:
:set mousefocus
mousemodel 选项可以定义鼠标可以做什么.有三种可能的模式:extend,popup,popup_setpos.设置鼠标的模式可以用下面的命令形式:
:set mousemodel=mode
在所有的模式中,鼠标左键移动光标,拉动左键可以选择文本.
在 extend 模式中,右键可以扩展文本而右键可以粘贴文本.这样的操作与 xterm 中使用鼠标相类似.
在 popup 模式中,右键可以显示弹出菜单.这种模式也大多数的 Windows 程序相类似.
popup_setpos 模式与 popup 模式相类似,所不同的只是当我们按下鼠标右键时文本的光标会移动到鼠标点击处,然后显示弹出菜单.
mouse 选项可指定鼠标对应某一模式.可能的模式如下:
n 正常(Normal)
v 可视(Visual)
i 插入(Insert)
c 命令行(Command-line)
h 除了 hit-return 以外的帮助文件中的所有模式
a 除了 hit-return 以外的所有模式
r more-prompt 和 hit-return 提示
鼠标左键将文本中的光标移动到鼠标点击处,鼠标右键可以使得 Vim 编辑器进入可视化模式,并且文本光标处和鼠标右键点击处之间的文本会被选中.鼠标中键的作用类似于 P 命令,可以将未命名寄存器中的文本粘贴到文件中.如果我们在点击鼠标中键之间指定了寄存器,Vim 就会将这个寄存

器中的内容粘贴到文件中.如果我们的鼠标有一个滚轮,向上滚动时就会向上移动三行,与此相类似当向下滚动时就会向下滚动三行.如果我们同时按下 Shift 键就会以一个屏幕的单位量进行滚动.也就是说 shift+鼠标可以向上移动一个屏幕,而 shift+鼠标可以向下移动一个屏幕.
选项 mousetime 可以定义双击之间的最大时间间隔.这个命令的一般形式如下:
:set mousetime=time
这里的时间是以毫秒为单位的,在默认的情况下为半秒(500ms)
当我们在图形界面下进行文本编辑时,我们要同时处理文本光标和鼠标光标.如果我们认为鼠标这样的存在方式使得我们不舒服,我们可以告诉 Vim 编辑器当我们没有鼠标时隐藏鼠标光标.我们可以用下面的命令来做到:
:set mousehide
这样以后,当我们开始输入时鼠标就会隐藏.而当我们移动光标时鼠标就会再一次出现.
selectmode 选项可以定义编辑器开始选择模式(selectmode)而不是可视模式(visualmode).下面的事件可以引发选择模式:
mouse 移动光标
key 使用特殊键
cmd v,V,CTRL-V 命令
这个命令的一般形式如下:
:set selectmode=mode
这里的 Mode 是由逗号分隔的事件列表(mode,key,cmd)
选项 keymodel 可以使得,,,,,,做一些特殊的事情.
例如如果我们设置了下面的命令,Shift+Key 就可以选择文本:
:set keymodel=startsel
如果我们执行了下面的命令,没有移位的键就会结束选择状态:
:set keymodel=stopsel
我们还可以将这两个命令组合到一起:
:set keymodel=startsel,stopsel
Vim 编辑器所使用的菜单是由文件$VIMRUNTIME/menu.vim 定义的.如果我们要定义我们自己的菜单,我们首先要看一下这个文件.
定义一个菜单内容,可以使用:menu 命令.这个命令的基本形式如下:
:menu menu-item command-string
这个命令与:map 命令相类似
这里的 menu-item 描述了在哪里放置菜单内容.我们比较熟悉的经典菜单内容为 File.Save,这就表明了 Save 菜单在 File 菜单下.而&号则是表明这是一个快捷键.例如下在 gvim 中,我们可以用 Alt-F 来选择 File 菜单而用 s 来选择保存菜单(这里没有看明白).所以这里的 menu-item 看起来就是这样的&File.&Save.
而事实上 File.Save 的定义如下:
:menu 10.340 &File.&Save:w :confirm w
这里的数字被称为优先级(priority)数.他可以用来决定编辑器如何放置菜单内容.第一个数字 10 表明了在菜单栏上的位置.小数字靠近左侧,而大数字靠近右侧.第二个数字则是决定了下拉菜单的位置.小数字在上部,而大数字在下部.
在这个例子中我们还可以看出很重要的一点就是 menu-item 必须是一个词.如果我们要在名字中加入空格或是 Tab,我们就要用到<>或是.例如:
:menu 10.340 &File.&Dolt:exit
最后我们还可以为一定的模式来定义菜单.这个命令的一般形式如下:

:[mode]menu [priority]menu-item command-string
mode 是下列中的一种:
a 正常(Normal),可视(Visual),运算符操作(Operator-pending)
n 正常(Normal)
v 可视(Visual)
o 运算符操作(Operator-pending)
i 插入(Insert)
c 命令行
还有一些特殊的菜单名称:
工具栏(ToolBar) 菜单下面的图标
弹出菜单(PopUp) 在一定的模式下在编辑窗口中点击右键弹出的菜单
工具栏是使用图标而不是使用文本来表明菜单的作用.例如名为 ToolBar.New 的 menu-item 是使得 New 图标显示在工具栏上.在 Vim 编辑器中有 28 个基本的图标.每一个图标都有两个名字,例如 New 图标可以由 ToolBar.New 或是 ToolBar.builtin00 来表示.builtin00-builtin27 所指代的图标分别为 New,Open,Save,Undo,Redo,Cut,Copy,Paste,Print,Help,
Find,SaveAll,SaveSesn,NewSesn,LoadSesn,RunScript,Replace,
WinClose,WinMax,WinMin,WinSplit,Shell,FindPrev,FindNext,FindHelp,
Make,TagJump,RunCtags.
如果图标并不与基本的图标相匹配,编辑器就会在$VIMRUNTIME/bitmaps 目录中进行查找.在 UNIX 系统中图标大小为 20x20 像素.
当我们将光标放在图标上时工具栏会显示一个小提示.要定义这个提示,我们可使用下面的命令:
:tmenu menu-itme tip
例如下面的命令可以使得当光标放在打开图标上时会显示 Open file 字样:
:tmenu ToolBar.Open Open file
我们可以用下面的命令列出所有的菜单映射:
:menu
我们在使用这个命令时遇到的一个问题就是我们会得到 51 个屏幕的数据输出.这对于我们来说实在是太多了.如果我们要只显示特定的菜单内容,我们可以用下面的命令:
:menu menu
例如我们要列出 File 的菜单内容,我们可以用下面的命令:
:menu File
而下面的命令只是列出 File.Save 的菜单内容:
:menu File.Save
在这个命令的执行结果的每一行的首个字母指明了这个命令的模式.这里的字符是与我们前面提到的 mode 参数相一致的.
下面的命令可以执行一个菜单的内容就像我们的从菜单中选中一样:
:emenu menu-item
:menu 命令定义了菜单内容.如果我们定义了一个菜单内容而且想要使得在他的右侧没有映射(no mapping),可以使用:noremenu 命令.
下面的命令可以从菜单中移除菜单项:
:[mode]unmenu menu-item
如果我们使用*,那么整个菜单都会被清除.
要移除工具栏的提示,可以用下面的命令:
:tunmenu menu-itme

(注:这里不懂:(
Tearing Off a Menu
You can tear off a menu by using the dotted tear-off line on the GUI. Another way to do this is to execute the following command:
:tearoff menu-name)
Vim 编辑器中的一些命令是专为图形界面的 Vim 而设计的.这些命令都是关于对话框的.
命令:browse 可以打开一个文件浏览器,然后会对选中的文件执行命令.例如下面的命令可以使得我们打开文件浏览器然后选择要打开的文件:
:browse edit
编辑器然后会执行:edit file 命令.
:browse 命令的一般形式如下:
:browse command [directory]
这里的 command 是以文件名为参数的编辑器命令.例如:
:read,:write,:edit
如果我们指定了 directory 参数,我们就指定了文件浏览器开始的目录.如果没有指定这个参数,那么浏览器就会选择 browsedir 选项所指定的目录.这个选项可以用下面的三个值:
last 使用上一次浏览的目录(默认)
buffer 使用与当前缓冲区相同的目录
current 总是使用当前目录
所以如果我们要总是从当前目录开始,我们就要将下面的命令放在我们的初始化文件中:
:set browsedir=current
下面的命令会显示一个查找对话框:
:promptfind [string]
如果我们指定了 string 的值,那么这个值就会作为查找区的初始值.然后当我们按下查找下一个的按键时,Vim 编辑器就会查找指定的字符串.
与此相类似,下面的命令可以打开一个替换对话框:
:promptrepl string
如果我们指定了 string 的值,他就会被作为查找的内容.
下面的命令会使我们打开一个对话框,我们可以在这个对话框中输入我们想要得到的帮助内容:
:helpfind
:confirm 选项会在例如执行:quit 命令会破坏数据时执行命令.如果命令的执行会破坏数据,那么就会显示一个确认窗口.
例如下面的命令会显示出一个确认窗口:
:confirm :quit
下面的命令会打开一个窗口并且充许我们在其中查看选项:
:browse set
这个窗口可以使得我们访问所有的选项.开如是一个简短的内容提示.我们可以用光标命令定位内容然后回车就会得到一个详细的列表信息.
clipboard 选项可以控制编辑器如何来处理由鼠标选择的文本.如果我们要将所有的鼠标选择的文本放在未命名寄存器中和剪切板寄存器中,我们可执行下面的命令:
:set clipboard=unnamed
这就意味着我们可以文本粘贴到其他的程序中.
另一个选项如下:
:set clipboard=autoselect

如果我们设置了这个选项,可视模式下选择的文本就会放在系统剪切板中.
autoselect 选项可以在图形界面和文本下工作.
当我们启动图形界面的 Vim 时,Vim 会试着确定我们的背景是浅色(light)还是深色(dark),并且执行下面的命令来设置正确的值:
:set background=value
语法文件就会由这个值来确定使用哪种颜色.
如果我们不喜欢 Vim 当前使用的图形界面下面的字体,我们可以用下面的命令来进行更改:
:set guifont=font
这里的 Font 是字体的名称.在 X Windows 系统中我们可以用命令 xlsfonts 列出所有可用的字体.在 Windows 系统中我们可以由控制台来得到一个字体列表.
我们还可以使用下面的命令:
:set guifont=*
这个命令就会打开一个字体选择列表,我们可以从中选择字体.
selection 选项可以定义如何来处理选择的文本.可用的值如下:
.old
不充许选中的内容超过一行的最后一个字符.选中部分的最后一个字符包括在操作之内.
.inclusive
超过一行的最后一个字符会被包含进来.选中部分的最后一个字符在操作之内
.exclusive
越过一行结尾的字任会被包含进来.选中部分的最后一个字符不在操作之内.
选项 guicursor 可以定义图形界面下的光标显示.这个命令的形式如下:
:set guicursor=mode:style[-highlight],mode:style[-highlight],...
我们可以设置的 mode 如下:
n 正常模式
v 可视模式
ve 可视模式但不包括 selection
o 运算符操作模式
i 插入模式
r 替换模式
c 命令行正常(追加)模式
ci 命令行插入模式
cr 命令行替换模式
sm 在插入模式下 showmatch
a 所有模式
我们还可以将这些模式进行组合,如下:
n-v-c
这里的 style 如下:
horN 水平栏,字符高度的 N 个百分比
verN 垂直栏,字符宽度的 N 个百分比
block 光标块,覆盖整个字符
blinkwaitN
blinkonN
blinkoffN
当我们指定了这些选项,系统等待 blinkwait 毫秒,然后为 blinkoff 关闭光标,为 blinkon 打开光标.并

进行循环.
在 X Windows 系统中,窗口管理器负责窗口的边框和装饰.选项 guiheadroom 可以告诉 Vim 编辑器在窗口的周边(上部和下部)有多大的空白空间,这样当他进入全屏幕模式时会为边框留下空间.
我们在图形界面下使用 Vim 时要想执行一个 Shell 命令时又应如何来做呢?通常系统会使用 UNIX 设备 pty 来处理命令接口.
如果我们要使用管道进行连接,可以用下面的命令:
:set noguipty
否则在默认的情况下使用 pty 进行 Shell 和 GUI 的连接:
:set guipty

Vim 编辑器有着丰富的脚本语言.当我们要为特殊的任务而定制我们的编辑器时,这些命令语言就会给我们极大的灵活性.
Vim 编辑器允许我们定义,设置和使用自己的变量.为变量设定一个值,我们可以使用:let 命令.这个命令的一般形式如下:
:let {variable}={expression}
Vim 编辑器采用大多数程序设计语言的变量命名方式,也就是在 Vim 中的变量是以字符或是下划线开头,由一系列的字符,数字或是下划线组成的.
例如要定义变量 line_size,我们可以用下面的命令:
:let line_size=30
要查看变量的内容,我们可以使用:echo 命令.如:
:echo "line_size is"line_size
当我们执行了这样的命令以后,Vim 就会在最后一行显示如下的内容:
line_size is 30
变量也可以包含数字和字符串,如:
:let my_name="mylxiaoyi"
Vim 编辑器使用特殊的前缀来指明不同有变量类型.这些前缀如下:
大写字母,数字,下划线　　可以存放在 viminfo 文件中的变量.如果 viminfo 选项中含有!标记,变量可以由:makesession 命令保存.
小写字母,数字,下划线　　不会存在任何保存文件中的变量.
$environment　　环境变量
@register　　文本寄存器
&option　　选项名字
b:name　　当前缓冲区的变量.每一个缓冲区有这个变量值不同
w:name　　当前窗口的变量
g:name　　全局变量(用于函数内部表明全局变量)
a:name　　函数参数
v:name　　Vim 内部变量
如下面的一些例子:
环境变量$PAGE 包含用页查看命令:
:let $PAGE="/usr/local/bin/less"
显示上一次查找的类型:
:echo "Last search was"@/
下面的两个命令有着同样的作用:
:let &autoindent=1

:set autoindent
为当前缓冲区定义语法:
:let b:current_syntax=c
内部变量(v:name)用于存放信息.如下面的内部变量列表:
v:count 为上一次正常模式命令所指定的数量(count)
v:count1 与 v:count 相类似,所不同的只是如果没有指定数量则默认值为 1
v:errmsg 上一次的错误信息
v:warningmsg 上一次的警告信息
v:statusmsg 上一次的状态信息
v:shell_error 上一次 Shell 命令的结果.如果为 0,则命令正常执行,若为非 0,则失败
v:this_session 上一次装入或是保存的文件的命名
v:version Vim 编辑器的版本号
Vim 编辑器还要使用如下的一些常量:
123 简单整数
0123 十进制整数
0xAC 十六进制整数
如下的字符串常量
"string" 简单字符串
'string' 精确字符
这两种字符串的不同在于前者可以用反斜线进行转义字符的扩展,而后者则不成,在后者的字符串反斜线只是原样输出.例如下面的命令:
:echo ">100<"
:echo '>100<'
其输出结果分别为:
>@<
>100<
在 Vim 编辑器中我们还可以用表达式进行整数的操作.这些操作包括如下的算术运算:
int+int 加
int-int 减
int*int 乘
int/int 除
int%int 取余
-int 取负
另外逻辑运算符可以作用于字符串和整数.如果比较成功则返回 1,否则则返回 0.如下面的比较:
var == var 检查是否相等
var != var 不等
var < var 小于
var > var 大于
var <= var 小于等于
var >= var 大于等于
另外比较运算符可以进行字符串和表达式的比较.例如下面进行指定的字符串("word")和表过式"w*"比较,如果表达式匹配则返回 1.
"word"=~"w*"
例如下面的两个常规表达式的比较:

string =~ regexp　　相匹配的常规表达式
string !~ regexp　　两个表达式不匹配
另外字符串还有下面的特殊比较:
string ==? string　　字符串相等,忽略大小写
string ==# string　　字符串相等,大小写必须匹配
string !=? string　　字符串不相等,忽略大小写
string !=# string　　字符串不相等,大小写必须匹配
string <?　string　　小于,忽略大小写
string <#　string　　小于,大小写必须匹配
string <=? string　　小于等于,忽略大小写
string <=# string　　小于等于,大小写必须匹配
string >?　string　　大于,忽略大小写
string >#　string　　大于,大小写必须匹配
string >=? string　　大于等于,忽略大小写
string >=# string　　大于等于,大小写必须匹配
从这里我们可以看到每一个运算符有三种形式.基本形式(==)对应 ignorecase 选项.?(==?)忽略大小写的不同而#(==#)从不忽略这样的区别.
如果我们要删除一个变量,我们可以用下面的命令:
:unlet[!] {name}
在通常的情况下如果我们删除一个不存大的变量,Vim 编辑器就会显示错误.而如果我们使用!,则不会显示错误信息.
当我们要输入文件名时,我们可以使用下面的一些特殊的单词或是符号:
%　　当前文件名
#　　交换文件名
<cword>　　光标下的单词
<cWORD>　　光标下的 WORD
<cfile>　　光标下的文件名
<afile>　　当执行相关的自动命令(autocommand)正读入或是写入的文件名.
<abuf>　　在一个自动命令中的当前缓冲区标号
<amatch>与<abuf>相类似.但是当在 FileType 或是 Syntax 事件中使用时并不是文件名,而文件类型或是语法名.
<sfile>　　当前正用于:sourced 的文件名.
我们可以用下面所列出的内容来修改这些单词或是符号.例如:p 可以将文件名变为全名.例如光标下的文件名为 test.c,<cfile>就将是 test.c,而同是<cfile:p>就将成为/home/oualline/examples/test.c
我们可以用下面的内容进行修改:
:p
将文件名变成全路径文件名.但是我们要注意的是当我们用多个修饰符时,我们要个放在第一个.
:~
将全路径名/home/oualline/examples/test.c 变为用~标记的文件名为,如~oualline/examples/test.c
:.　　如果可能将成为当前目录相关的目录
:h　　文件名的头部.例如../path/test.c 就会为../path
:t　　文件名的尾部.例如../path/test.c 就会为 test.c
:r　　无扩展名的文件名.例如../path/test 就会成为 test
:e　　扩展名

:s?from?to?　　将第一次出现的 form 字符串改变为 to 字符串
:gs?from?to?　　将所有的字符串 form 改变为 to 字符串
我们可以来看一下这些修饰符是如何作用在文件名上的.首先我们要先创建一个文件,其内容为我们运行实验的文件名.我们将光标放在这个文件名上,使用下面的命令来设置修饰符:
:echo expand("<cword>:p")
我们可将这里的:p 换成我们可以试验的任何修饰符.
下面的内容我们将会更详细的说明一个:echo 和 expand 功能
:echo 的功能只是重复他的参数.例如:
:echo "Hello world"
Hello world
我们还可以用他来显示变量的值:
:let flag=1
:echo flag
1
:echon 命令也只是重复他的参数,但是不会输出新行.例如:
:echo "aa" | echo "bb"
aa
bb
:echon "aa" | echon "bb"
aabb
(注:这里的|用来分隔同一行的两个命令)
我们可以使用:echohl 命令来改变:echo 的输出的指定高亮颜色组.例如:
:echohl ErroMsg
:echo "A mistake has been make"
:echohl None
一个好的程序习惯表明我们应该总是在我们的输出信息之后重设高亮显示为 None.这样就不会影响其他的:echo 命令了.
如果我们要查看所定义的高亮显示组,我们可以用下面的命令:
:highlight
Vim 编辑器中有许多的控制语句可以使我们改变宏的功能.通过这些功能,我们可以更好的使用 Vim 编辑器的脚本语言.
:if 语句
:if 语句的一般形式如下:
:if {condition}
:　　"Statment
:　　"Statment
:endif
如果条件(condition)为非 0,if 语句块内的语句将会被执行.在其中的四个空格的缩进是可选的,但是却是推荐使用,这样可以使得程序易读.
:if 语句还可以有 else 的子句:
:if {condition}
:　　"Statment
:　　"Statment
:else

```
:       "Statment
:       "Statment
:endif
```
最后关键字:elseif 是:if 和:else 的组合.使用这个可以减少使用额外的:endif 的需要:
```
:if &term == "xterm"
:       "Do xterm suff
:elseif &term == "vt100"
:       "Do vt100 suff
:else
:       "Do non xterm and vt100 stuff
:endif
```
循环

:while 命令开始一个循环.这个循环是由命令:endwhile 命令结束的:
```
:while counter<30
:    let counter=counter+1
:       "Do something
:endwhile
```
:continue 命令回到程序的顶部开始执行下一次循环,而:break 命令则退出循环:
```
:while conter <: 30
:       if skip_flag
:               continue
:       endif
:       if exit_flag
:               break
:       endif
:    "Do something
:endwhile
```
:execute 命令:

:execute 命令像正常的命令模式一样执行一参数:
```
:let    command = " echo 'Hello world!'"
:execute command
```
Vim 编辑器还允许我们定义自己的函数.函数定义的一般形式如下:
```
:function {name}({var1},{var2},...)
```
(注:函数的名称要以大写字母开始)

结束定义用下面的命令:
```
:endfunction
```
下面我们来定义一个小函数,用这个函数来返回两个数中较小的一个.我们这样开始定义:
```
:function Min(num1,num2)
```
这个命令是告诉 Vim 我们定义的函数名为 Min,他有两个参数.我们要做的第一件事就是我们要比较两个数中哪一个要小一些:
```
:       if a:num1 < a:num2
```
这里的前缀 a:是告诉 Vim 这个变量是一个函数参数.我们将最小的参数赋值给 smaller 变量:
```
:       if a:num1 < a:num2
:               let smaller = a:num1
```

```
:       else
            let smaller = a:num2
:       endif
```
这里的 smaller 是局部变量.在这个函数中使用的所有变量均为局部变量,除非我们使用了 g:作为前缀.例如在函数定义外我们定义了变量 var.在函数内部我们要使用时要用 g:var 来调用.所以说一个变量依据其内容有三个不同的名字.
现在我们可以用:return 语句来返回最小的那个数.最后我们结束函数定义:
```
:       return smaller
:endfunction
```
完整的函数定义如下:
```
:function Min(num1,num2)
:       if a:num1 < a:num2
:           let smaller = a:num1
:       else
:           let smaller = a:num2
:       endif
:       return smaller
:endfunction
```
这样我们就可以用表达式来使用我们的函数了,如:
```
:let tiny = Min(10,20)
```
我们还可以用:call 命令用函数名来显示调用函数功能:
```
:[range]call {function}([parameters])
```
如果指定了[range]则每一行都要调用函数,除非这个函数是一特殊的 range 风格函数.
如果我们要试着定义一个已经存在的函数,我们就会得到一个错误信息.我们可以用!来强制 Vim 替换以前所定义的同名的函数.
如果我们将 range 关键字放在函数定义的后面,这个函数就会被认为是一个范围(range)函数.例如:
```
:function Count_words() range
```
当在一个范围的行内运行这个程序时,变量 a:firstline,a:lastline 就会设置成为这个范围内的第一行和最后一行.
如果在函数的定义后面有 abort,那么这个函数就会在第一个错误时退出,如:
```
:function Do_It() abort
```
最后 Vim 允许我们在函数中使用个数不定的参数.例如下面的命令定义了一个函数,这个函数必须有一个参数,但是可以用至多 20 个参数:
```
:function Show(start,...)
```
变量 a:1 包含第一个可选的参数,a:2 为第二个,依次类推.变量 a:0 包含多余的参数.例如:
```
:function Show(start,...)
:       let index = 1
:       echo "Show is"a:start
:
:       while (index <=a:0)
:           echo "Arg" index "is" a:index
:           let index = index + 1
:       endwhile
:endfunction
```

我们可以用下面的命令列了所有用户定义的函数:
:function
要查看单一的函数,我们可以执行下面的命令:
:function {name}
例如我们要查看函数 Show:
:function Show
要删除一个函数,我们可以用下面的命令:
:delfunction name
Vim 编辑器允许我们定义自己的命令.我们可以像执行其他的命令模式的命令一样来执行我们自己定义的命令.要定义一个命令我们要使用:command 命令,例如:
:command Delete_first :1delete
这样当我们执行命令:Delete_first Vim 就会执行:1delete,从而删除第一行.
如果我们要列出用户定义的命令,我们可以用下面的命令:
:command
要删除用户定义的命令,我们可以用下面的命令:
:delcommand
例如:
:delcommand Delete_one
我们还可以用下面的命令来清除所有的用户定义的命令:
:comclear
用户定义的命令可以指定一系列的参数.参数的个数要由-nargs 选项在命令行中指定.例如,Delete_one 命令没有参数,我们可以像下面的样子来定义:
:command Delete_one -nargs=0 1delete
然而因为在默认的情况下-nargs=0,所以我们不需要指定他.
其他的-nargs 选项值如下:
-nargs=0　　没以参数
-nargs=1　　1 个参数
-nargs=*　　任何个数的参数
-nargs=?　　零个或是一个参数
-nargs=+　　一个或是更多个参数
在命令的定义中,参数是由关键字<args>指定的.例如:
:command -nargs=+ Say :echo "<args>"
然后我们输入:
:Say Hello World
命令的执行结果为:
Hello World
一些命令是指定一个范围作为其参数.告诉 Vim 我们在定义这样的一个命令我们需要指定-range 选项.选项的值如下:
-range　　允许的范围,默认为当前行.
-range=%　　允许的范围,默认为当前文件(while file)
-range=count　　允许的范围,但是他只是一个单一的数字,默认下为 count.
当我们指定了一个范围以后,我们就可以用关键字<line1>和<line2>得到这个范围的第一行和最后一行.
例如下面的命令定义了一个 SaveIt 命令,这个命令可以将指定范围的文件写入文件 save_file:

:command -range=% SaveIt:<line1>,<line2> write! save_file
其他的一些选项和关键字如下:
-count=number
这个命令指定一个数量,默认为 number.数量的结果保存在关键字<count>中.
-bang 我们可以使用!修饰符.如果指定了,!将会被存放在关键字<bang>中.
-register
我们可以指定一个寄存器,默认为未命名寄存器.寄存器的定义放在关键字<reg>中.
关键字<f-args>含有与关键字<args>相同的信息.所不同的只是函数的调用方式不同.例如:
:command -nargs=* DoIt :call AFunction(<f-args>)
:DoIt a b c
执行下面的命令:
:call AFunction("a","b","c")
最后我们还有<lt>关键字,他包含字符<.
基本的功能函数:
Vim 编辑器有许基本的功能函数.这一部分将会列出所有的功能函数:
append({line_number},{string})
作用:在 line_number 行后新增一行加入 string
参数:
line_number
某行的行号,将会在其后插入文本.0 将会使得在文件的开始处插入文本.
string 在指定行后将插入的文本.
返回值:整数标记.或为 0 则没有错误,1 则是由于 line_number 超出范围所产生的错误 argc()
作用:计算参数列表中的参数个数.
返回值:整数.参数个数.
argv({number})
作用:返回参数列表中的参数.
参数:
number: 参数索引.0 则为参数列表中的第一个参数.
返回值:字符串.返回请求的参数.
browse(save,title,initial_directory,default)
作用:显示一个文件查看器,允许用户选择文件.这个只是在 GUI 版本中工作.
参数:
save:一个整数用来表明这个文件是否被读入或是保存.如果 save 为非 0,则查看器选择一个文件写入.如果为 0,则这个文件用于读取.
title:对话框的标题.
initial_directory:开始查看时的目录.
default:默认的文件名.
返回值:字符串.选择的文件名.如果用户选择了关闭或是有错误发生,则会返回一个空字符串.
例如下面的命令:
:call browse(0,"Hello","/home/mayuelong/Documents","Shell.txt")
bufexists(buffer_name)
作用:检查一个缓冲区是否存在.
参数:
buffer_name:要检查的缓冲区的名称.

返回值:整数标记.若存在则返回真(1),否则为假(0)
bufloaded(buffer_name)
作用:查看一个缓冲区是否被装入
参数:
buffer_name:查看是否装入的缓冲区的名称.
返回值:整数标记.若存在则返回真(1),否则为假(0)
bufname(buffer_specification)
作用:查看指定的缓冲区
参数:
buffer_specification:指明缓冲区的标号或是字符串.如果指定缓冲区标号,那么 buffer_specification 返回缓冲区标号.如果指定了字符串,他就会被作为一个常规的表达式并且会列出查找得到并且匹配的缓冲区.这里有三个特殊的缓冲区:%当前缓冲区,#交换缓冲区,$列表中的最后一个缓冲区.
返回值:字符串.包含缓冲区全名的字符串或是有错误发生或是没有匹配时则会返回空串.
bufnr(buffer_expression)
作用:得到缓冲区的标号.
参数:
buffer_expression:与函数 bufname 中的相类似.
返回值:整数.缓冲区的标号.若有错,则返回-1.
bufwinnr(buffer_expression)
作用:得到一个缓冲区的窗口号
参数:
buffer_expression:与 bufname 功能函数的参数相同
返回值:整数.与缓冲区相匹配的第一窗口的标号.或是有错误或是没有匹配的窗口时返回-1
byte2line(byte_index)
作用:将字节索引转换为行号
参数:byte_index 在当前缓冲区的字符索引
返回值:整数.包含由 byte_index 所指定的字符的行号或是 byte_index 超出范围时返回-1
char2nr(character)
作用:将字符转换成相应的数字
参数:
character:转换单一的字符.如果指定了一个长的字符串,只使用第一个字符.
返回值:整数.与字符相对应的数字标号.例如:char2nr("A")的值为 65(即其相对应的 ASCII 码)
col(location)
作用:返回指定位置的列号
参数:
location:标记的描述(如"x)或是"."得到当前光标所在处的列号
返回值:整数.返回标记或是光标的所在的列号.当有错误发生时则返回 0
confirm({message},{choice_list},[default],[type])
作用:显示一个对话框可以使得用户一系列的选择并返回用户的选择.
参数:
{message}在对话框中显示的提示信息.
{choice_list}包含选择列表的字符串.每一个新行("n")来分隔每一个选择.用&来表明加速字符.
[default]表明默认选择的索引.第一个按钮为#1.如果没有指定这个参数,则第一个按钮为默认的.
[type]要显示的对话框的类型.可用的选择如下"Error","Question","Info","Warning","Generic".默认

情况下"Generic"
返回值:整数.选择的数字(从 1 开始).如果用户按<ESC>或是 CTRL-C 来退出则返回 0
如下面的一些例子:
echo confirm("Choose one","&Onen&Twon&Three",2,"Error")
echo confirm("Choose one","&Onen&Twon&Three",1,"Question")
echo confirm("Choose one","&Onen&Twon&Three",0,"Warning")
echo confirm("Choose one","&Onen&Twon&Three",0,"Info")
echo confirm("Choose one","&Onen&Twon&Three",0,"Generic")
delete({file_name})
作用:删除文件
参数:
{file_name}要删除的文件名
返回值:整数.0 表明文件已被删除,非 0 则表示错误.
did_filetype()
作用:检测 FileType 事件是否发生.这个命令在用自动命令时显得更为有用.
返回值:整数.如果自动命令正在执行并且至少有一个事件已发生则返回非 0,否则返回 0
escape({string},{character_list})
作用:将{string}字符串的字符转换成转义字符.{character_list}则为要进行转义的字符列表.
返回值:字符串
例如下面的命令:
:echo escape("This is a 'test'"," '")
其执行结果为:
This is a 'test'
exsits({string})
作用:检测由{string}所指定的内容是否存在.
参数:
{string}要检测的内容.这个可以用来指定一个选项('&autoindent'),环境变量('$VIMRUNTIME'),基本的函数名('*escape')或者是一个简单的变量('var_name')(注:在这里我们要指定定这个引号标记,因为我们是在字符串中传递这些值)
返回值:整数.如果存在则返回 1,否则返回 0
expand({string},[flag])
作用:返回与{string}匹配的文件列表.这个字符串可以包含通配符或是其他的描述.在这里我们要注意的是'suffixes'选项和'wildignore'选项可以影响扩展如何实现.如果一个特殊的单词例如'<cfile>'被扩展,并不会显示更深一些的扩展.如果光标位于字符串'~/.vimrc'上,'expand('<cfile>')的结果为'~/.vimrc'.如果我们要得到全名,我们需要扩展两次.所以'expand(expand('<cfile>'))将会返回'/home/oualline/.vimrc'.选项'wildignore'和'suffixes'将会起作用,除非指定了一个非 0 的[flag]
返回值:字符列表.由新行分隔的与{string}相匹配的文件名列表.如果没有匹配的,则会返回回空字符串.
filereadable({file_name})
作用:检测一个文件是否可读
参数:要检测的文件名
返回值:整数.如果这个文件存在并可读则会返回非 0 值.0 则表明这个文件不存在或是这个文件不可读.
fnamemodify({file_name},{modifiers})

作用:对{file_name}执行{modifiers}并返回结果.
参数:
{file_name}文件名
{modifiers}修改标记,如":r:h"
返回值:字符串.被修改的文件名
getcwd()
作用:得到当前的工作目录
返回值:字符串.当前的工作目录
getftime({file_name})
作用:得到一个文件的修改时间
参数:
{file_name}　　要检测的文件名
近回值:整数.文件的修改时间,若发生错误则返回-1
getline({line_number})
作用:从当前的编辑缓冲区中取出一行
参数:
{line_number}　　要得到的行号或是"."表明光标所在的行
返回值:字符串.一行文本.如果{line_number}超出范围则会返回空字符串
getwinposx()getwinposy()
作用:返回 GUI 窗口的 x 或是 y 坐标
返回值:整数.以像素表示的 GUI 窗口的 x 或是 y 坐标.如果这些信息不可得则返回-1.
glob({file_name})
作用:在文件名中扩展通配符并返回文件列表.
参数:
{file_name}
表明要匹配文件名类型的字符串.我们还可以使用由反引号(`)构成的外部命令.如下面的例子:glob("`find . -name '*.c' -print"')
返回值:字符串.由<NL>分隔的匹配的文件列表.如果没有匹配的则会返回空串
has({feature})
作用:检测某一个特征已被安装
参数:
{feature}　　包含特征名的字符串
返回值:整数标记.如果这个特征已被编译则要返回 1,否则返回 0
histadd({history},{command})
作用:在一个历史列表加入某一个内容
参数:
{history}　　要用到的历史名.如下:
"cmd"　　":"　　命令历史
"search" "/"　　查找历史
"expr"　　"="　　表达式历史
"input"　　"@"　　输入历史
返回值:成功则为 1,错误为 0
histdel({history},[pattern])
作用:从历史中移除命令

参数:
{history}　　要用到的历史列表
[pattern]
定义要移除的内容的表达式.如果没有指定类型,将会从历史中移除所有的内容.
histget({history},[index])
作用:从历史记录中得到某一内容
参数:
{history}　　要用到的历史列表
[index]
要得到的内容索引.最新的输入为-1,次新的为-2,依次类推.最后一个输入的为 1,次后输入的为 2,依次类推.如果没有指定索引,将后返回最后一个输入的命令.
返回值:字符串.历史记录中指定的命令,如果有错误发生时则会返回空串
histnr({history})
作用:返回指定历史列表中的当前输入的标号
参数:
{history}　　要检测的历史
返回值:整数.在这个历史列表中的最后一个内容.如果有错误则返回-1
hlexists({name})
作用:检测一个语法加亮组是否存在
参数:
{name}　　要检测的组名
返回值:整数标记.若存在则为非 0,否则为 0
hlID({name})
作用:指定语法加亮组的名,返回 ID 号
参数:
{name}　　语法加亮组的名
返回值:整数.ID 号
hostname()
作用:得到电脑的主机名
返回值:字符串.电脑的主机名
input({prompt})
作用:问一个问题并会得到答案
参数:
{prompt}　　要显示的提示
返回值:字符串.与用户输入相一致的内容
isdirectory({file_name})
作用:检测{file_name}是否为一个目录
参数:要检测的内容名
返回值:整数标记.如果是为目录则为 1,如果不是目录或不存在则返回 0
libcall({dll_name},{function},{argument})
作用:在 DLL 文件中调用函数(只在 Windows 系统中使用)
参数:
{dll_name}　　共享库的文件名,在其中定义了{function}
{function}　　函数名

{argument}
单一参数.如果这个参数为一个整数,他就会作为整数传递.如果是字符串,就会以"char*"传递.
返回值:字符串.返回函数.
line({position})
作用:指定一个标记或是其他的位置指示,返回行号
参数:
{position}　　位置标记.可以是一个标记,'x,当前光标所在处".",或是文件尾"$"
返回值:整数.行号.如果没有标记或是其他的错误,则会返回 0
line2byte({line_number})
作用:将行号转换成字符索引
参数:
{line_number}
要转换的行号.这个可以是一个标记('x),当前光标处('.')或者是缓冲区的最后一行('$')
返回值:整数.由 1 开始的行中第一个字符的索引.如果有错误发生则会返回-1
localtime()
作用:以 UNIX 格式返回当前时间
返回值:整数.
maparg({name},[mode])
作用:返回映射到哪一个键
参数:
{name}　　{lhs}映射名
[mode]　　字符映射的模式.默认为""
返回值:字符串.映射结果字符串.如果没有映射返回空串
mapcheck({name},[mode])
作用:检测一个映射是否存在
参数:
{name}　　{lhs}映射名
[mode]　　字符映射的模式,默认为""
返回值:字符串.返回匹配{name}的任何映射.这个函数与 maparg 函数有一些不同.前者会在映射中查找相冲突的名字.例如如果我们有一个映射"ax",他就会转换为"axx"
例如下面的一些例子:
:map ax test
:echo maparg("ax")
test
:echo maparg("axx")

:echo mapcheck("ax")
test
:echo mapcheck("axx")
test
match({string},{pattern})
作用:检测{string}是否与{pattern}相匹配.如果要设置了'ignorecase'就会使得编辑忽略大小的情况.
参数:
{string}　　要检测的字符串

{pattern}　　要检测的类型
返回值:整数.{string}中{pattern}出现的第一个字符索引.第一个字符为 0.如果没有相匹配的就会返回-1.
matchend({string},{pattern})
作用:与 match 函数相类似,所不同的是他会返回{pattern}后的字符索引.
matchstr({string},{pattern})
作用:与 match 函数相类似,只是他会返回匹配的字符串
参数:
{string}　　要检测的字符串
{pattern}　　要检测的类型
返回值:字符串.{string}中匹配的部分.如果没有匹配就会返回空串
nr2char({number})
作用:将数转换成字符
参数:字符的 ASCII 码值.
返回值:长度为 1 的字符串.与数字相对应的字符
rename({from},{to})
作用:重命名一个文件.
参数:
{from}　　存在的文件名
{to}　　要重命名为的文件名
返回值:整数标记.如果成功则为 0,否则为非 0
setline({line_number},{line})
作用:用{line}的字符串内容替换{line_number}行的内容.
参数:
{line_number}　　要改变的行号
{line}　　替换行的文本
返回值:整数标记.如果没有错误则为 0,否则为非 0
strftime({format},[time])
作用:通过{format}返回时间格式.在字符串可以放置的转换字符决定于我们系统的 strftime 函数.
参数:字符串.包含时间格式化的时间字符串
strlen({string})
作用:计算字符串的长度
参数:
{string}　　我们要计算长度的字符串
返回值:整数.字符串的长度
strpart({string,{start},{length})
作用:返回由{start}开始的长度为{length}的子串
例如:
:echo strpart("This is a test",0,4)
This
:echo strpart("This is a test",5,2)
is
如果{start}和{length}所指定的为不存在的字符,就会忽略这种情况.例如下面的命令从第一个字符的左边开始.

:echo strpart("This is a test",-2,4)
Th
参数:
{string}我们要从中查找的字符串
{start}我们要选取的字符串的开始地址
{length}我们要选取的字符串的长度
返回值:字符串.我们要选取的子串
strtrans({string})
作用:将字符串不可打印的字符转换成可以打印的字符
参数:
{string}　　包含不要打印字符的字符串
返回值:字符串.不可打印字符的结果子.例如 CTRL-A 就会转换为^A.
substitute({string},{pattern},{replace},{flag})
作用:在字符串,将第一个与{pattern}匹配的字符替换成{replace}.这个命令与 Vim 的编辑命令相同
参数:
{string}　　要替换的字符串
{pattern}　　用来指定替换部分的字符串
{replace}　　要替换成为的文本
{flag}
如果没有指定只是替换第一个出现的,如果为 g,就会可以替换全部的情况.
返回值:字符串.替换后的字符串
synID({line},{column},{transparent_flag})
作用:返回由{line}和{column}所指定的语法 ID.
参数:
{line},{column}　　内容所在的位置
{transparent_flag}If non-0, transparent items are reduced to the items they reveal
返回值:语法 ID.
synIDattr({sytnax_id},{attribute},[mode])
作用:得到语法颜色元素的标记
参数:
{sytnax_id}　　语法描述的标号.
{attribute}　　标记名.可有和的标记名如下:
"name"　　语法内容名
"fg"　　前景色
"bg"　　后景色
"fg#"　　以#RRGGBB 格式的前景色
"bg#"　　以#RRGGBB 格式的后景色
"bold"　　如果内容为粗体则为 1
"italic"如果内容为斜体则为 1
"reverse"如果内容反转则为 1
"inverse"与上面内容相同
"underline"如果内容为下划线则为 1
[mode]
要得到哪种终端标记.这个可以是"gui","cterm","term".默认的终端是我们当前用的.

返回值:字符串.标记值.
synIDtrans({syntax_id})
作用:返回转换的语法 ID
参数:
{syntax_id}　　语法元素的 ID
返回值:整数,转换的语法 ID.
system({command})
作用:执行外部命令并输出.选项'shell'和'shelldir'可以用于这个函数
参数:
{command}　　要执行的命令.
返回值:字符串.输出命令的执行结果.
tempname()
作用:产生一个临时文件名
返回值:字符串.用为临时文件的文件名.
visualmode()
作用:得到最后的可视化模式
返回值:字符串.作为命令字符串得到最后一个可视化模式.这个可以是 v,V,或是 CTRL-V
virtcol({location})
作用:得到给定位置的列号.
参数:位置可以是."(光标处),'a(标记 a),或者是$"(文件最后处)
返回值:整数.
winbufnr({number})
作用:得到窗口内缓冲区的标号.
参数:
{number}　　窗口标号.0 为当前窗口.
返回值:整数.缓冲区标号.若有错则为-1.
winheight({number})
作用:得到窗口高度.
参数:
{number}　　窗口标号.0 为当前窗口.
返回值:整数.以行数表示的窗口高度.若有错,则为-1
winnr()
作用:得到当前窗口号.
返回值:整数.当前窗口标号.最上的窗口为#1

Vim 编辑器是一个高可定制的编辑器.他会提供给我们相当多的选项可用.在这一部分我们会谈到一些如何用这些选项来定制我们的编辑器的外观和行为.
Vim 编辑器是一个高可定制的编辑器.他会提供给我们相当多的选项可用.在这一部分我们会谈到一些如何用这些选项来定制我们的编辑器的外观和行为.
Vim 编辑器有许多设定选项的方法.一般情况下要设定一个选项,我们可以用下面的命令:
:set option=value
这样的设置方法是在大多数的情况下都可以很好的来工作的.真值选项的设置可以用下面的命令:

:set option
如果要重置我们可以用下面的命令:
:set nooptoion
如果我们要查看一个选项的值,我们可以用这样的命令:
:set option?
如果我们要将一个选项设为其默认的值,我们可以用下面的命令来做到:
:set option&
布尔选项:
我们可以对布尔选项执行下面的操作:
:set option　　打开选项
:set nooption　　关闭选项
:set option!　　转换选项
:set invoption　　转换选项
:set option&　　将选项设为默认值
例如:
:set list
:set list?
list
:set nolist
:set list?
nolist
:set list!
:set list?
list
:set list&
:set list?
nolist

数字选项:
我们可以对数字选项执行下面的一些操作:
:set option += value　　在选项中加入一个值
:set option -= value　　在选项中减去一个值
:set option ^= value　　将值乘以选项
:set option&　　将选项设为默认值
例如:
:set shiftwidth=4
:set shiftwidth += 2
:set shiftwidth?
shiftwidth=6
:set shiftwidth-=3
:set shiftwidth
shiftwidth=3
:set shiftwidth ^= 2
:set shiftwidth

shiftwidth=6
:set shiftwidth&
:set shiftwidth
shiftwidth=8

字符串相关的命令:
我们可以对字符串选项执行下面的操作:
:set option += value　　在选项的末尾加上 value
:set option -= value　　从选项中移去 value 或是字符
:set option ^= value　　在选项的开头加入 value
例如:
:set cinwords=test
:set cinwords?
cinwords=test
:set cinwords+=end
:set cinwords?
cinwords=test,end
:set cinwords-=test
:set cinwords?
cinwords=end
:set cinwords^=start
:set cinwords?
cinwords=start,end
下面的命令会设置一个布尔选项(例如 list 或是 nolist),但是他会显示出其他类型选项的值::set option
然而用这样的命令方式来显示一个选项的值并不是很好的方法,因为如果我们不小心这个方法就会导致错误.更好的方法是用下面的命令来显示一个选项的值:
:set option?
相对的下面的命令形式:
:set option = value
查看其值的方法为:
:set option:value
其他的一些:set 参数:
下面的命令会打印出所有的与其默认值不同的选项:
:set
下面的命令会打印出所有的选项:
:set all
下面的命令会打印出终端控制代码:
:set termcap
最后如果我们要将所有的选项都重新置为默认值我们可以用下面的命令:
:set all&
我们可以在一行中放入几个:set 操作.例如要设置三个不同的选项我们可以用下面的命令:
:set list shiftwidth=4 incsearch
在一个文件中自动设置选项:

我们可以在我们的文件中放入 Vim 设置.当我们启动 Vim 编辑一个文件时,他会读入这个文件中的最初几地并查找如下格式的行:

vim:set option-command option-command option-command...

这样的行被称为模式行(modeline)

例如在一个程序中一个模式行如下:

/*vim:set shiftwidth=4 autoindent:*/

相对的格式如下:

Vim:option-command:option-command:....

选项 modeline 可以打开或是关闭这种行为.选项 modeline 可以控制 Vim 在查找设置命令时会读入开始的几行.

例如如果我们设置了下面的选项,Vim 就不会查找模式行了:

:set nomodeline

如果我们设置了下面的选项,Vim 就会在每一个文件的头部或是尾部查找由 modeline 选项所指定的行数:

:set modeline

例如也许我们都会在 Vim 的帮助文件的末尾看到下面的几行:

vim:tw=79:ts=8:sw=8:

在这个设置中是将 tw(textwidth)选项设为 78,ts(tabstop)选项设为 8,sw(shiftwidth)选项设为 8.这样的设置使得我们的帮助文件看起来更好看一些.通过使用模式行,帮助文件的制造者就会确保这些文本可以被正确的格式化而不论我们在其他的文件中使用怎么样的本地设置.

.vimrc 文件:

假如我们要对每一个不同的目录进行不同的设置,一个办法就是在每一个目录中放入.vimrc 或是.gvimrc 文件.然而这样做还是不够的,因为默认的 Vim 会忽略这些文件.

要使得 Vim 读入这些文件,我们必须执行下面的命令:

:set exrc

进行这样的设置却存在着安全问题.毕竟一些不成功的命令是很容易加入到这些文件中的,即使我们是在其他的目录中进行我们的编辑工作,这样的设置也会影响我们的.

为了避免安全问题,我们可以用下面的命令来设置安全选项:

:set secure

这个选项会阻止当前目录下的初始文件中的:autocommand,:write,:shell 命令的执行.

自定义键盘的使用:

大多数运行在 Windows 平台下的程序会使用 Alt 键来选择菜单内容.然后 Vim 希望能用所有的键来完成我们的命令.选项 winaltkeys 可以控制 Alt 键的使用.

例如,如果我们执行了下面的命令,所有的 Alt 键就可以通过:map 命令来使用之成我们命令中的一部分:

:set winaltkeys=no

执行 Alt-F 并不会选择文件菜单而是执行映射的 Alt-F 命令.这个可以映射可以是下面的样子:

:map <M-f> :write

在这里我们要知道的就是在 Vim 中 Alt 会被叫作 M-,Meta 的意思.

如果我们执行了下面的命令,所有的 Alt 键盘就会选择菜单内容而不会执行映射命令了:

:set winaltkeys=yes

第三个选项是 yes 和 no 的组合:

:set winaltkeys=menu

在这种模式下,Alt 键可以用于选择菜单,也可以用来映射:map 命令.所以 Alt-F 可以选择文件菜单,

而我们还可以用 Alt-X 来映射命令.
我们可以有两个选项来控制当我们使用字符界面的 Vim 时如何来读入键盘.下面的选项会告诉 Vim 直接由控制台读入:
:set conskey
如果我们希望 Vim 编辑器从标准的输入读入脚本文件时不要用这个选项.
下面的选项会告诉 Vim 使用 BIOS 来读入键盘:
:set bioskey
在这里我们也要指出的就是如果我们要使用重定向的脚本也不要设置这样的选项.通过指定 Vim 使用 BIOS,我们就会得到更快的 CTRL-C 的响应以及 Break 中断.
自定义键盘映射:
大多数的 Unix 功能键会发出由<Esc>开始的字符串.但是这样却存着一个问题,那就是<Esc>键是用来结束插入模式的.所以我们如何在插入模式下处理功能键呢?
Vim 的一个解决办就是在按下<Esc>以后等待一会看是否还会有其他的字符输入.如果是这样的,Vim 就会知道按下了功能键并会执行相应的操作.要打开这个特征我们可以执行下面的命令:
:set esckeys
但是如何来处理其他键的顺序呢?我们可以通过下面的两个选项来控制区:
:set timeout
:set ttimeout
下面的是这些设置的作用:

timeout	ttimeout	结果
notimeout	nottimeout	没有超时
timeout	N/A	所有的代码键(<F1>...)以及:map 宏超时.
notimeout	ttimeout	只有代码键超时

选项 timeoutlen 决定在按下 Esc 后等待多长时间来决定是否还有输入.默认值为 1 秒:
:set timeoutlen=1000
一般情况下,timeoutlen 选项控制功能键以及键盘映射的等待时间.如果我们希望键盘映射使用不同的时间,我们可以用 ttimeout 选项:
:set ttimeout=500
这两个选项是告诉 Vim 在 Esc 按下后等待 1/2 秒来决定我们是否还会输入功能键或是等待 1 秒来决定是否还有其他的键盘映射输入.
确认:
在一般的情况下,我们会遇到一些 Vim 认为有疑问的事情,如从一个已修改的缓冲区退出,命令失败.如果我们设置 confirm 选项,Vim 在类似的情况下就会显示出一个确认的对话框:
:set confirm
这样当我们试着在一个已经修改的缓冲区中执行:quit 命令时就会显示一个要我们来确认的对话框.
自定义消息:
Vim 通常使用屏幕下部来显示消息.有时这样的消息会超出一行而且我们会得到类似于按回车继续的提示.为了避免这样的提示,我们可以通过设置 cmdheight 选项来增加消息的行数.例如我们可以用下面的命令来使得消息的空间设为 3:
:set cmdheight=3
显示状态:
当我们设置了 showmode 选项,Vim 编辑器就会在屏幕的下部显示出我们当前所处的状态.要设置成这样的模式,我们可能使用下面的命令:

:set showmode
显示部分命令:
如果我们设置了 showcmd 选项,当我们输入命令时就会在屏幕的下部显示出部分命令.例如我们执行下面的命令:
:set showcmd
现在我们输入 fx 命令来查找 x.当我们输入 f 时就会在底部显示 f.
这对于我们输入较为复杂的纵使时就会显示尤为有用.

在一般的情况下,当我们执行了:shell 命令时 Vim 就会警告我们文件已修改.如果我们要关闭这个选项,我们可以执行下面的命令:
:set nowarn

当 Vim 捕获一个错误时,他只是会显示出一个错误信息.如果我们要打开声音警告我们可以执行下面的命令:
:set errorbells
但是有时这样的设置会影响我们的其他的同事,在这样的情况下我们可以来设置 visualbell 选项.这样当我们输入错误时屏幕就会闪动然后回到正常状态.要设置这样的选项我们可以执行下面的命令:
:set visualbell
我们还可以自定义状态行.我们可以使用下面的选项来定义我们的状态行:
:set statusline=format
%用来指明一个特殊的区域.例如%f 来告诉 Vim 在状行中包含文件名.如下面的命令:
:set statusline=The file is"%f"
这时的状态行为:
The file is "sample.txt"
我们可以指定一个内容的最大和最小的宽度.例如下面的命令告诉 Vim 编辑器文件至少为 8 个字符但是最多为 19 个字符.这些内容是右对齐的,如果我们希望他们能左对齐,我们可以在%后面加上-.数字的内容是忽略开头的 0 来显示.如果我们需要我们可以在%后加上一个 0.例如我们要显示列数而需要开始的 0,我们可以使用下面的命令:
:set statusline=%05.10c
格式,类型以及描述如下:
%(...%)　　定义一个项目组.如果在这个组中的所有的内容均为空,整个组的内容就不会显示.
%{n}*
%对其余的行使用高亮显示组 Usern,直到看到另一个%n*.格式%0*会使得行成为正常的高亮显示.如果高亮显示组 User1 带有下划线(:set statusline=File:%1*%f%0).这时状态行显示的文件名就会带有下划线.
%<　　定义如果状态行过长在何处换行
%=
定义在一行中部的某处.所有向左的字符串放在这行的左部,而向右的字符串放在一行中靠近右边的部.
%　　字符%
%B　　光标下字符的十六进制形式
%F　　全路径的文件名

%H 如果为帮助缓冲区则为 HLP
%L 缓冲区中的行数.
%M 如果缓冲区修改过则显示为+
%O 以十六进制方式显示文件中的字符偏移
%P 文件中光标前的%
%R 如果缓冲区只读则为 RO
%V 列数.如果与%c 相同则为空字符串
%W 如果窗口为预览窗口则为 PRV
%Y 文件类型
a%(字符串) 如果我们在编辑多行文本,这个字行串就是"({current} of {arguments})".例如:(5 of 18).如果在命令行中只有一行,这个字符串为空.
%b(数字) 光标下的字符的十进制表示形式.
%c(数字) 列号
%f(字符串) 在命令行中指定的文件名
%h(标记) 如果为帮助缓冲区为[Help]
%l(数字) 行号
%m(标记) 如果缓冲区已修改则表示为+
%n(数字) 缓冲区号
%o(数字) 在光标前的字符数,包括光标下的字符
%p(数字) 文件中所在行的百分比
%r(标记) 如果缓冲区为只读则表示为 RO
%t(字符串) 文件名(无路径)
%v(数字) 虚列号
%w(标记) 如果为预览窗口则显示为 Preview
%y(标记) 我们输入的文件类型
%{expr%} 表达式的结果
标记的内容会被特殊的对待.多行标记,例如 RO 和 PRV,则会自动的由逗号来分隔.而例如+和 help 则会由空格来分隔.例如:
:set statusline=%h%m%r
如果我们并不喜欢默认的状态行,我们可以打开标尺选项(ruler option):
:set ruler
这个可以使得 Vim 显示类似于下面的状态行:
help.txt [help][RO] 1,1 Top
在文件名和标记后会显示出当前的列号,虚列号,并且会显示我们在文件中所处的位置.
如果我们要定义我们自己的标尺格式,我们可以用下面的命令:
:set rulerformat=string
这里的 string 就是我们在 statusline 选项用到的.
当我们删除或是修改了一些行的文本,如果这些行数超出了 report 所指定的行数,Vim 编辑器就会告诉这一些情况.所以如果我们要 Vim 报告所有的变化,我们可以用下面的命令:
:set report=0
相反,如果我们不希望 Vim 来告诉我们这些变化,我们可以将这个值设成一个相当大的值就可了.
我们可以用下面的命令来设置帮助窗口的最小尺寸:
:set helpheight={height}
我们这里设置的数值会在打开帮助窗口时用到.

我们也可以用下面的命令来设置预览窗口的大小:
:set previewheight={height}
在一般的情况下,list 命令使用^I 来表示 Tab 而使用$来表示一行的结尾.我们也可以自定义这种形为.我们可以使用 listchars 选项来定义 list 模式如何工作.这个命令的格式如下:
:set listchars=key:string,key:string
而 key:string 可以用到的值如下:
eol:{char}　　定义放在一行结尾处的字符
tab:{char1}{char2}　　一个 Tab 值为 char1 和 char2 的组合来表示.
trail:{char}　　用来表示结尾空格的字符.
extends:{char}　　用来表示一行的结尾和下一行连接的字符
我们可以使用 highlight 选项来改变许多对象的高亮显示.这个选项的格式如下:
:set highlight=key:group,[key:group]...
我们可以使用下面的 key 值:

key	Default	Meaning
8	SpeicalKey	当用:map 命令列出特殊键时用来高亮显示
@	NonText	Vim 用~和@来表示不在缓冲区中的内容时适用.
M	Modemsg	在屏幕下部显示的模式信息.
S	StatusLineNC	除当前窗口外的每一个窗口的状态行.
V	VisualNOS	可视化模式下选择的文本.
w	WildMenu	作为通配符的一部分而显示
d	Directroy	当我们按下 CTRL-D 时显示的目录
e	ErrorMsg	错误信息.
i	IncSearch	作为增量查找的一部分而显示
I	Search	作来查找的一部分而显示
m	MoreMsg	--More--提示
n	LineNr	由命令:number 打印出的行号
r	Question	按下回车的提示及其他的问题
s	StatusLine	当前窗口的状态行
t	Title	输入信息的命令标题.
v	Visual	在可视化模式下选择的文本
w	WaringMsg	警告信息.

我们还可以使用缩写字符:

r	Reverse
i	Italic
b	Bold
s	Standout
u	Underline
n	None
-	None

所以我们可以通过下面的命令来使用 ErrorMsg 组来高亮显示错误信息:
:set highlight=e:ErrorMsg
或者我们可以通过下面的命令来以 reverse,bold,italic 方式来显示错误:
:set highlight=ervb
如果我们设置了 more 选项,当一个命令的显示会超出一屏时就会显示 More 提示.如果没有进行设

置只是会翻滚屏幕.默认如下:
:set more
下面的命令定义了可以由 CTRL-A,CTRL-X 识别的数字格式.
:set nrformats=octal,hex
(注:十进制总是可以识别)
如果我们设置了下面的选项,Vim 就会试着重新装入终端屏幕的内容.
:set restorescreen
换句话说,他会试着使得我们运行这个程序以后的屏幕看起来就像他运行以前的一样.
xterm 可以允许我们按下鼠标左键拉动时选择文本.这个文本可以粘贴到其他的窗口中.然后一些由于兼容的问题会使得我们在粘贴文本时出现问题.为了避免这样的问题,我们可以使用下面的命令来设置粘贴模式:
:set paste
然而有时我们希望是粘贴模式而有时希望不是.pastetoggle 选项可以使得我们定义一个键在这两种模式中进行切换.例如,如果我们使用 F12 键进行切换,我们可以用下面的命令:
:set pastetoggle=<F12>
当关闭粘贴模式时,所以的选项就会恢复到先前的值.
当我们在 ex 模式下输入命令时,我们可以实现文件名的自动完成.例如如果我们想要读入文件 input.txt,我们可以输入下面的命令:
:read input<Tab>
Vim 就会试着猜出我们想要的文件.如果在我们当前的目录下只有文件 input.txt,他就会显示出如下的形式:
:read input.txt
如果在当前目录下有几个带有 input 的文件名,则会显示第一个.如果我们再按下 Tab,就会显示出第二个相匹配的结果,再按下 Tab 就会显示第三个.
我们可以通过下面的命令来定义完成通配符的键:
:set wildchar=character
如果我们是在一个宏内使用文件名,我们要设置 wildcharm.这是我们在宏内完成文件名自动完成的字符.例如下面的命令:
:set wildcharm=<F12>
:map<F11> :read in<F12>
这样当我们按下 F11 时就会读入文件命令.
也许我们并不希望匹配备份文件或是其他类似的文件.我们可以用下面的命令来告诉 Vim 哪些是这样的文件:
:set wildignore=pattern,pattern
这样与指定类型相匹配的文件都会被忽略掉.例如我们要忽略目录文件和备份文件,我们可以用下面的命令:
:set wildignore=*.o,*.bak
suffixes 选项会列出一系列文件名的前缀,当遇到这样的前缀时就会得到一个较低的优先级.换句说就是如果一个文件名有这样的前缀就会在匹配时就会一个来显示.
一般情况下,文件名的完成代码并不会显示一个匹配列表.如果我们设置下面的选项:
:set wildmenu
当我们试着要完成一个文件名时,就会在窗口的状态行显示一个可能的文件菜单.
我们可以通过方向键来完成选择.在一行最末的>表明在右部有更多的选择.下键可以使得编辑器进入一个目录.上键回到父目录,回车选择项目.

我们可以通过使用 wildmode 选项来自定义文件完成功能的行为.下面的命令使得 Vim 只完成第一个的匹配:

:set wildmode=

如果我们一直按下 wildchar 键,则只会显示第一个匹配的结果.

而下面的命令则会用他可以查找到的第一个文件来完成文件名:

:set wildmode=full

这样以后,如果我们一直按下 wildchar 键,其他的匹配文件就会按顺序来显示.

下面的命令则会匹配最长的子串:

:set wildmode=longest

当我们执行下面的命令时会完成同样的功能,但是只是显示那些位于 wildmenu 中的文件:

:set wildmode=longest:full

下面的命令则会显示一个可能的文件列表:

:set wildmode=list

这种模式并不会匹配完成.如果我们希望这样的情况,我们可以使用下面的选项:

:set wildmode=list:full

最后要完成最长的子串并列出文件,我们可以使用下面的选项:

:set wildmode=list:longest

当我们设置了 startofline 选项,屏幕的移动命令与一些光标的移动命令例 H,M,L,G 相同.

如果我们设置了 write 选项,我们可以使 Vim 保存文件.如果我们没有设置这个选项,我们只可以查看文件.如果我们希望 Vim 作为一个安全的查看者,这样的设置是相当有用的.

一般情况下,当我们试着保存一个我们不应保存的文件时,Vim 就会提示我们使用覆盖选项(!).我们可以告诉 Vim 总是认为输入了这个选项:

:set writeany

但是这样的设置并不是一个很好的选择,因为这样就会覆盖掉以前我们已经存在的文件.

如果我们要为一个缓冲区设置最大的内存,我们可以用下面的命令:

:set maxmem={size}

在这里这个大小是千字节计的.

为所有的缓冲区定义一个内存容量,我们可以用下面的命令:

:set maxmemtot={size}

maxfuncdepth 选项可以定义最大的嵌套函数数.相似的,maxmapdepth 参数可以定义最大的嵌套映射数.

下面的内容我们来说一些终端选项:

我们所使用的终端名存放在 term 选项中.一般的情况下我们并不需要设置这个选项,因为这个是由我们所使用的 Shell 和操作的环境来定义的.也许有时我们需要将其读入来允许特定的终端宏.

对于慢终端(slow terminal)我们可以使用 lazyredraw 选项.他可以阻止 Vim 在宏中部重绘屏幕.默认的设置如下:

:set nolazyredraw

这个选项已经被现在的终端技术实现.如果我们设置了这个选项,我们就不会看到正在执行的宏.

在 UNIX 系统中有一个名为 termcap 的终端控制代码数据库.Vim 编辑器同样也有其基本的数据库.

如果我们打开了 ttybuiltin 选项,就会首先搜索这个内在的数据库.

如果我们设置了 ttyfast 选项,Vim 就会认为我们有一个比较快的终端连接,而且会改变输出来产生一个相对平滑的更新,但是这个有着更多的特征.如果我们有一个慢的连接,我们要重置这个选项.

ttymouse 选项控制终端鼠标代码.这个选项对于试着用终端控制代码来做一些有趣的事时是相当有趣的.例如,如果我们要在控制的输入模式下使用鼠标左键<LeftMouse>和右键<RightMouse>,我

们要允许这些选项.
ttyscroll 选项可以控制当屏幕需要更新时要滚动多少行.如果我们在是一个慢的终端上,我们可以将这个选项设为一个较小的值.
下面的部分我们来说一些在 Vim 是会少用到的选项.这些选项可以保持与 Vi 相兼容和支持以前的一些设备.
下面的选项可以使用 Vim 尽可能的像 Vi 一样的运行:
:set compatible
与其相类似下面的命令可以使我们与 Vi 的兼容性相一致:
:set cpoptions={characters}
下面的命令可以使得:substitute 命令中的 g 和 c 选项与 UNIX 编辑器 Ed 的用法相类似:
:set edcompatible
下面的选项可以设置 lisp 模式.这个选项可以设置许多的选项来使 Lisp 编辑更容易.
:set lisp
选项 tildeop 可以使得~与一个运算符相类似.这个与为了 Vi 的兼容性.如果关闭了这个选项,~命令会选择单一字符情况.用下面的命令,~命令的格式为~motion:
:set tildeop
helpfile 选项可以定义主帮助文件的位置.如果我们要重置:help 命令从哪里得到信息时显得有用.例如:
:set helpfile=/usr/sdo/vim/my_help.txt
下面的命令可以提供向后兼容:
:set weirdinvert
下面的命令会在每一个字符的输出时产生延时(以毫秒计):
:set writedelay={time}

语法加亮功能是由位于$VIMRUNTIME/syntax/language.vim 中的语法文件来控制的.如果我们对现在的语法加亮功不满意,我们可以为这些语法文件做一份拷贝,然后按照我们的想法进行修改.在 Vim 编辑器中我们有多种语言可以选择.
汇编语言:
现在有许多不同种类的汇编语言.但是在默认的情况下,Vim 认为我们是使用 GNU 风格的汇编语言.其他的一些汇编语言如下:
asm GNU 汇编(默认)
asmh8300 Hitachi H-8300
masm Microsoft MASM
nasm Netwid 汇编
如果要想叫 Vim 知道我们正在使用一种其他的汇编语言,我们可以用下面的命令:
:let asmsyntax=language
这里的 language 就是我们在前面列出的关键字中的一个.
Basic
Visual Basic 和标准的 Basic 都是使用以.BAS 为扩展名的文件.为了区分这两种文件,Vim 编辑器会读入 5 行并检查字符串 VB_Name(以.FRM 为扩展名的一般为 Visual Basic)
C/C++
对于 C/C++的语法颜色我们做出许多的自定义.如下面的一些内容:

c_comment_string 高亮显示注释中的字符串及颜色
c_space_errors 在<Tab>前显示空白符标记
c_on_trail_space_error 不显示空白符标记
c_no_tab_space_error 不在<Tab>前标记空格
c_no_ansi 不高亮显示 ANSI 类型及常量
c_ansi_typedefs 高亮显示 ANSI 宏定义
c_ansi_constants 高亮显示 ANSI 类型
c_no_utf 在字符串是高亮显示 u 或是 U
c_no_if0 作为注释不要高亮显示#if0/#endif
有时也许我们会遇到在注释中或是#if0/#endif 块中显示高亮错误.这时我们可以用 CTRL-L 命令来重绘屏幕来解决这样的问题.要想永久的来解决这样的问题,我们要使用下面的命令来增加查找语法匹配的行数:
:let c_minlines=number
在这个命令中的 number 是要查找的最小行数.将其设为一个相对较大的数可以帮助我们来解决类似这样的问题.
COBOL
在 Vim 编辑器中有两种 COBOL 的高亮显示:新的开发方式(fresh development)和古老的(legacy).我们可以用下面的命令来使用古老的高亮显示方式:
:let cobol_legacy_code=1
DTD
DTD 常是大小写敏感的.如果我们希望其忽略大小的情况我们可以使用下面的命令:
:let dtd_ignore_case=1
语法加亮会将不可识别的标记识为错误.要关闭这个特征,我们可以用下面的命令:
:let dtd_no_tag_errors=1
存在参数的名字会使用注释组来高亮显示.我们可以用下面的命令来关闭这个特征:
:let dtd_no_parameter_entities=1
Eiffel
Eiffel 并不区分大小写,但是标准的格式要求使用大小写.因而语法加亮的规则希望我们可以用这种标准的风格.我们可以用下面的命令来关闭大小的检查:
:let effiel_ignore_case=1
如果我们希望高亮显示可以正确的检查 Current,Void,Result,Precursor,NONE 我们可以用下面的命令:
:let eiffel_strict=1
如果我们希望不使用标准格式加亮我们可以用下面的命令来做到:
:let eiffel_pedantic=1
我们通过下面的命令可以使用小写形式的 current,void,result,precursor,none:
:let eiffel_lower_case_predef=1
为了处理 ISE 的新语法,我们可以用下面的命令:
:let eiffel_ise=1
为了支持十六进制的常量,我们可以用下面的命令:
:let eiffel_hex_constants=1
ERLANG
ERLANG 代表 ERicsson LANGuage.语法加亮有两种选择:
erlang_keywords 不高亮显示关键字

erlang_characters 不高亮显示特殊字符
FVWM
FVWM 是一个窗口管理器.如果我们要为这个程序编辑配置文件,我们可以用下面的命令来告诉 Vim 颜色文件的位置:
:let rgb_file="/usr/X11/lib/X11/rgb.txt"
这个例子展示了与 Linux 相匹配的 rgb.txt 文件的位置.其他的系统也许会放在/usr/lib 或是其他的地方.
HTML
HTML 语法文件使用下面的高亮标记:
htmlTitle
htmlH1
htmlH2
htmlH3
htmlH4
htmlH5
htmlH6
htmlBold
htmlBoldUnderline
htmlBoldUnderlineItalic
htmlUnderline
htmlUnderlineItalic
htmlItalic
htmlLink
如果我们要关闭语法加亮可以用下面的命令:
:let html_no_rendering=1
如果我们要为这些内容定义我们自己的颜色,我们可以将颜色设置的命令放在我们的 VIMRC 中然后使用下面的命令:
:let html_my_rendering=1
在一些文件中包含<!--和--!>或是<!和!>作为注释.如果我们希望这些注释高亮显示,我们可以用下面的命令:
:let html_wrong_comments=1
Java
Java 语法有下面的一些选项:
java_mark_braces_in_parens_as_errors
如果设置了这个选项,位于括号中的花括号会被认为是错误
java_highlight_java_lang_ids 高亮显示所有位于 java.lang.*中的标识符
java_highlight_functions="indent" 设置函数声明总是缩进
java_highlight_function="style" 设置函数声明不缩进
java_highlight_debug 高亮显示调试语句
java_allow_cpp_keywords
将所有的 C/C++关键字识为错误.这个可以帮助我们避免使用他们,所以我们的代码可以更好的移植到 C/C++.
java_ignore_javadoc 关闭高亮显示 javadoc
java_javascript 打开在 Javadoc 内的 Javascript 加亮显示

java_css 加亮显示在 Javadoc 内的 CSS 样式表
java_vb 为 Vbscript 加亮显示
Lace
好的风格是要区分大小的.如果我们要关闭好风格的特征,我们可以用下面的命令:
:let lace_case_insensitive=1
Lex
Lex 文件被由%%组成的行分隔成几个主要的部分.如果我们在写一个比较长的 Lex 文件,语法加亮也许就不会找到%%.为了解决这个问题,我们可以用下面的命令来增加语法操作的最小行数:
:syntax sync minlines=300
Lite
Lite 使用类似于 SQL 的查询语句.我们可以用下面的命令来加亮字符串的 SQL 语句:
:let lite_sql_query=1
如果我们有很多的命令,我们希望增加语法加亮的行数:
:let lite_minlines=300
Maple
Maple
V,是一种符号语言.他有许多不同的包,使用者可以有选择的来装入.如果我们希望加亮所有的包,我们可以用下面的命令:
:let mvpkg_all=1
Perl
如果我们在我们的文件中加入了 POD 文档,我们可以打开 POD 语法加亮:
:let perl_include_POD=1
下面的选项可以改变 Perl 如何在引用中显示包名:
:let perl_want_scope_in_variables=1
如果我们要使用复杂的变量声明,我们可以用下面的命令:
:let perl_extend_vars=1
下面的命令会将字符串作为语句处理:
:let perl_string_as_statement=1
如果我们在同步方面有一些困难,也许我们要改变一些下面的选项:
:let perl_no_sync_on_sub=1
:let perl_no_sync_on_global=1
:let perl_sync_dist=lines
Php3
下面的选项控制 Php3 的高亮显示:
php3_sql_query 高亮显示字符串的查询
php3_baselib 高亮显示基本的库方法
php3_minlines 语法加亮中的同步行数
Phtml
加亮字符串中的 SQL 语法,我们可以用下面的命令:
:let phtml_sql_query=1
要改变窗口的同步,我们可以用下面的命令:
:let phtml_minlines=lines
PostScript
用于 PostScript 加亮的一些选项如下:

poster_level 设置 PostScript 语言的级别(默认为 2)
poster_display 加亮 PostScript 的特征显示
poster_ghostscript 加亮显示 GhostScript 的语法
poster_fonts 加亮字符
poster_encodings 编码表
poster_andornot_binary 颜色逻辑操作不同
Printcap 和 Termcap
我们可以用下面的命令来增加同步的行数:
:let ptcap_minlines=100
Rexx
我们可以用下面的选项来设置同步的行数:
:let rexx_minlines=lines
Sed
要显示出 Tab 我们可以使用:set list 选项.我们可以用下面的命令来以不同的方式高亮显示他们:
:let highlight_sedtabs=1
如果我们执行下面的命令就可以很容易的算出字符串的 tab 数:
:set tabstop=1
Shell
下面的选项可以改变脚本的高亮显示:
bash_is_sh 加亮 bash 语法
highlight_balanced_quotes 高亮显示双引号中的单引号
highlight_function_name 在声明中的高亮显示函数名
sh_minlines 设置同步的行数
sh_maxlines 限制同步显示的行数
Speedup
用于 Speedup 的一些选项如下:
strict_subsections 只高亮显示每一个子部分中的关键字
highlight_types 高亮显示流类型
oneline_comments=1 允许#注释后的任何数字
oneline_comments=2 以第二个#开始的代码视为错误
oneline_comments=3 如是在一行中有两个或是更多个#,将事先视为错误
Tex
Tex 是一个很复杂的语言.如果编辑器没有找到结尾 texZone,我们可以下面的内容放入我们的文件中:
%stopzone
TinyFugue
我们可以用下面的选项来为 TinyFugue 文件设置同步限制:
:let tf_minlines=lines

vi 编辑器的学习小结.一些关于 vi 想说的话.

在最初接触 LINUX 的时候就知道了在 UNIX 和 LINUX 的系统中还有一个相当牛气的可视化编辑器 VI,那时也有过想好好学一学的想法,不过在找了一些资料后就觉得学起来也真是太麻烦了,光是一个鼠标移动的命令就我记好久,最后终于还是没有学会.因为在那个时候还在一些文本编辑器可以来用,觉得这些文本编辑器也还是不错的.但是终于有一天我要配置 Debian 的 apt 源,因为 Debian 是不允许以 ROOT 用户登陆的,我也就再也没有办法来使用一些其他的文本编辑器.真的是没有办法了,就死记了几个命令,以 root 的身分用 vi 来配置.也就是在这一次也才真正的认识到了 VI 的一些不可替代的作用.所以也就想着好好的学一些关于 VI 的东西.但是在网上搜一些东西总是不太使人满意,也许是我不会搜吧.所以就到了 VI 的主页去下了一个文档回来慢慢的看,顺便写一些 VI 编辑器的学习使用的小文,也算是给其他的一些朋友们提供一些方便吧.

因为这是一个英文的文档,自己的英文也并不是太好,只能是慢慢的来啃,一点一点的来试,同时记下自己的一些作法,也算是对这个文档的一点翻译吧.这些小文断断续续的写了两个多月.在这里真的是要感谢那些来看了我的博客的朋友们,因为正是这些朋友们的点击使我认识到我的这点小小的东西还是有着他的作用的,至少也是为一些朋友提供了方便和帮助,所以也就决心将这个慢慢的写下去.

现在将这个小文写到了第二十九,本也想着来一个完美一点的,要写完第三十的,但是第三十说的是一些 VIM 语法配置方面的事情,我试着写了一些,不过觉得实不懂,如果将这些东西写出来怕是会误导一些朋友啊.索性还是不要写的好啊.如果有感兴趣的朋友可以亲自的去看一下这些文章.

这个文档可以从这里获得:http://www.vim.org/docs.php

另外在这里也还是有一个中文的文档的:http://vimcdoc.sourceforge.org

希望这些对 VI 感兴趣的朋友们能够在以后的学习中取得更大的进步.

我也希望能和更多的有共同兴趣的人成为朋友,我的一些联系方法如下:

QQ:289216073

MSN:darkhorse11@hotmail.com

E-mail:xiaoyi239@163.com

(注:
Harrison,我的一个网友,正是他将这些小东西整理到一起,又方便大家的查看.在这里我要真诚的感谢 Harrison 的热情和工作.
Harrison 的联系方式:
QQ:11658128
再一次谢谢 Harrison)