进行有效编辑的七种习惯

Bram Moolenaar

如果你的很多时间是用来敲纯文本，写程序或HTML，那么有效地使用一个好的编 辑器能节省你不少时间。这篇文章里的指导和提示将有助于你更快工作，更少犯 错误。

文中采用开源文本编辑器Vim(Vi IMproved)说明有效编辑的思想，但这些思想也适用于其他编辑器。择合适的编辑器只是有效编辑的第一步，对于哪个编辑器更好的讨论将占很大地方，这里就不提了。如果你不知道该用哪个编辑器，或者对现在所使用的不太满意，不妨试试Vim；你是不会失望的。

第一部分：编辑一个文件

快速定位

编辑中大部分时间是花费在阅读、查错和寻找应该进行编辑的地方上，而不是插入新文字或进行修改。在文件中不断定位(navigate)是经常要做的，所以最好学会如何快速地进行。

你常会搜寻文档中的一些文字。或者找出包含特定词或词组的行。你当然可以使用搜寻命令 /pattern，不过还有更聪明的方法：

* 如果你看到一个特定词，想看看其他地方是不是出现过同样的词，可以使用‘*’命令。它将对光标所指的词进行搜寻。

* 如果设置了 ' incsearch' 选项，Vim将在你正在输入搜寻模式的时候就显示搜寻的结果（而不是等到你敲了回车之后）。这能够使你更快地找出拼写错误。

* 如果设置了 ' hlsearch' 选项，Vim将使用黄色背景对搜寻结果进行高亮显示。你可以对搜寻的结果一目了然。应用在程序代码中可以显示变量的所有引用。你甚至不需要移动鼠标就能看到所有的搜寻结果。

对于结构化的文档，快速定位的办法就更多了。Vim提供专门针对C程序（以及C++、Java等等）的特殊命令：

* 使用 %可以从开始括号跳到对应的关闭括号。或者从 ``#if'' 跳到对应的 ``#endif''。事实上， % 可以完成许多对应项之间的跳转。可以用来检查if()和{}结构是否平衡。

* 使用 [{可以在代码段(block)中跳回到段起始的 ``{``。

* 使用 gb 可以从引用某个变量的地方跳转到它的局部声明。

定位的方法当然不止这些。关键是你需要知道有这些命令。你也许会说不可能学会所有命令 — Vim里有成百个定位命令，有的很简单，有的很聪明 — 这需要 几星期的学习。不过，你不必如此；你只需要了解自己的编辑特点，然后掌握相关的定位命令就可以了。

可以采取三个基本步骤：

1. 在你进行编辑的时候，注意那些重复进行的操作。

2. 找出能快速进行这些操作的编辑命令。阅读文档，问问朋友，或者看看其他人是如何做的。

3. 进行练习，知道熟练为止。

让我们通过以下这个例子说明一下：

1. 你发现在写C程序时，经常要查找函数定义。你目前使用 * 命令对函数名进行搜寻，但得到的往往是函数的引用而不是函数定义。你觉得一定会有更好的办法。

2. 读过一篇快速参考以后，你发现关于定位标记的说明，里面说明了如何定位函数定义，这正是你要找的！

3. 你试着生成了一个标记文件，使用Vim自带的ctags程序。你学会了使用CTRL-] 命令，发现这省了不少事。为了更方便，你在 Makefile 里加入了几行以自动生成标记文件。

当你使用以上三个步骤时，有几点需要注意的地方：

* ``我只想完成任务，不想去读那些文档来找新的命令。''。如果你真的是这么想的，那么你将永远停留在计算的石器时代。有些人编写什么都用Notepad，却总不明白为什么其他人总能用他一半的时间成任务。

* 不要过分。如果你总为一点小事也要去找完美的命令，你就没法集中精力到你本要完成的任务上了。只要找出那些耗费过多时间的操作，然后使用相关的命令直到熟练就可以了。这以后你就能集中精力到自己的文档上了。

下面这些章节给出了大多数人遇到的操作。你仿照它们在实际工作中使用 三个基本步骤。

不要敲两次

我们所使用的字词集合是有限的。既使是词组和句子也不过是有限的几个。对于程序来说更是如此。很明显，你不想把同样的东西敲上两遍。

你经常会想把一个词替换成另一个。如果是全文件替换，你可以使用 :s (substitute)命令。如果只是几个位置需要被替换，一个快速办法是使 用 * 命令找出下一个词，使用 cw 来进行替换。然后敲 n 找到下个词，再用 . 重复 cw 命令。

. 命令重复上一个改变。这里的改变是插入、删除或替换操作。能够重复 进行操作是个极为强大的机制。如果好好使用它，那么你大部分的编辑工作可能 只不过是敲几下 . 的事。小心不要在两次重复之间做其他修改，因为这 将改变你要重复的操作。如果确实需要如此，可以使用 m 命令记住要修 改的位置，等重复操作进行完毕之后再回过头来修改它。

有些函数名和变量名可能很难敲。你能准确无误地输入XpmCreatePixmapFromData''么？Vim的自动补齐机制能给你省不少事。它查看 你正在编辑的文件以及#include文件，你可以只敲入XpmCr''，然后使用 CTRL-N 命令让Vim把它补齐为``XpmCreatePixmapFromData''。这不但节省 了输入时间，而且减少了输入的错误。

如果你有同样的词组或句子需要输入多次，还有个更简单的办法。Vim可以进行录制宏。使用 qa 命令开始在'a'寄存器里录制宏。然后正常地输入编 辑命令，最后用 q 退出录制状态。如果你想重复所录制的命令，只需 执行 @a 命令。Vim总共提供26个这样的宏寄存器。

使用宏录制功能可以记录各种操作，不只限于插入操作。如果你想重复一些东西， 不妨一试。

需要注意的是记录的命令会被原封不动地重复执行。在进行定位时简单的重复宏操作可能不是你想要的结果。比如对于一个词这里可能需要左移4个字符，在下个地方可能就要左移5个字符。所以必须定位到合适的位置再重复进行宏操作。

如果你要重复的命令很复杂，把它们一次敲进去会很困难。这时你可以写一个脚本或宏。这常被用于建立代码模板；比如，一个函数头。你想做得多聪明就可以做得多聪明。

知错就改

编辑时经常会出错。无人能免。关键是快速发现并进行改正。编辑器应该提供这方面的支持，不过你必须告诉它什么是对什么是错。

你可能常常会重复同样的错误，你的手指所做的并非是你要它做的。可以使用缩写(abbreviation)进行修正。下面是一些例子：

```
* :abbr Lunix Linux
* :abbr accross across
* :abbr hte the
```

这些词会在编辑时被自动改正。

同样的机制也可以用于对很长的词语进行缩写。特别适用于输入那些你觉得很难敲的词，它可以避免出错。比如：

```
* :abbr pn pinguin
* :abbr MS Mandrake Software
```

但有时候你想要的正是那些缩写，比如想插入``MS''。所以缩写中最好使用那些不会出现在文中的词。

Vim提供了一个很聪明的高亮机制，一般用于程序的语法高亮，不过也可以用来查错。

语法高亮会使用颜色显示注释。这听上去不是什么特别重要的功能，不过一旦用

起来就会发现这其实很有用。你能够快速地发现那些没有高亮却本应作为注释的文字（可能是因为忘了敲注释符）。也可以发现一些被错误当成注释的代码（可能是因为忘了敲``*/''）。这些错误在黑白方式下是很难被发现的，浪费了不少 调试时间。

语法高亮也可以用来查找不匹配的括号。一个未被匹配的)''会被亮红色背景加以标识。你可以使用 % 命令他们是被如何匹配的，然后把(''或 ``)''插入到合适的位置。

另一类常犯的错误也很容易发现，比如把 #include <stdio.h>''敲成了#included <stdio.h>''。在黑白方式下这是很难发现的，但在语法高亮下 则能很快发现include''能被高亮而included''没有。

再看一个更复杂的例子：对于英文文本你可以定义一个所要使用的词的长列表。所有未在表中出现的词都可能是错误，并进行高亮显示。可以定义几个用于编辑词表的宏。这正是字处理器的拼写检查功能。Vim中是靠一些脚本来实现的，你也可以对它进行定制：比如，只对注释中的文字进行拼写检查。

第二部分：编辑多个文件

文件总是成帮结伙

人们很少只编辑一个文件。一般需要顺序或同时编辑一些相关的文件。你应该利用编辑器使多文件编辑工作更为高效地。

上面提到的标识(tag)机制也支持跨文件搜寻。一般做法是为项目的所有文件生成标识文件，然后在项目的所有文件中搜寻函数、结构、类型(typedef)等的定义。这比手工搜寻要快捷的多；我浏览一个程序要做的第一件事便是建立标识文件。

另一个强大的功能是使用 :grep 命令对一组文件进行模式搜寻。Vim把 搜寻结果做成一个列表，然后跳到第一个结果。使用 :cn 命令跳到下 一个结果。如果你想改变一个函数调用的、参数个数，那么这个功能会很有用。

头文件里有很多有用的信息。然而要知道一个声明出现在哪个头文件中却需要花不少时间。Vim能够理解头文件，能够从中找到你需要的东西。把光标移动到函数名下，然后敲 [I：Vim就会显示出一个头文件中该函数名的所有匹配。 如果你想得到更详细的结果，可以直接跳到声明中。一个类似的命令可以用于检 查你所使用的头文件是否正确。

你可以把Vim的编辑区域进行分隔，用来编辑不同的文件。你可以对两个或多个文件进行比较，或者进行拷贝/粘贴。有许多命令用于打开关闭窗口，文件间跳转，暂时隐藏文件等等。可以再使用上面提到的三个基本步骤选择合适的命令进行学习。

多窗口还有更多的用法。预览标识(preview-tag)就是个很好的例子。它打开一个特殊的预览窗口，光标还保留在你正在编辑的文件中。预览窗口中可以是光标所指函数的声明。如果你移动光标到另一个名字下，停留一两秒，预览窗口中就会显示那个名字的定义。名字还可以是头文件中声明的结构或函数。

让我们一起来工作

编辑器可以编辑文件。e-mail程序可以收发消息。操作系统可以运行程序。每个程序都有它自己的任务，而且应该做好。如果能让程序一同工作，那么就会实现很强大的功能。

举个简单的例子：选择一个列表中的结构化的文字，并对它进行排序： !sort。这将使用外部命令sort''来过滤文件。容易吧？排序功能是可以 添加到编译器中的。不过看一下man sort''就知道它有很多选项。它可能用了 一个极为精巧的排序算法。你还打算把它加到编辑器中么？更何况还有其他不少 过滤程序。编辑器可能会变得很大。

Unix精神的一个体现就是提供独立的程序，各自做好自己的任务，然后组合起来完成更大的任务。不幸的是，许多编辑器不能很好地和其他程序一起工作，比如， 你不能包Netscape的邮件编辑器换成其他编辑器。这样你只能使用那个不顺手的 程序。另一个趋势是在编辑器里提供所有的功能，Emacs就是个代表（有人说 Emacs其实是个操作系统，只是可以用来编辑文件）。

Vim尽力和其他程序集成，但这需要经过斗争。目前Vim已经可以作为 MS-Developer Studio和Sniff的编辑器。一些e-mail程序（比如Mutt）也支持外 部编辑器。和Sun Workshop的集成工作正在进行中。总的来说这个领域还有待提高。将来我们会有一个大于其各部分总和的系统。

文本结构化

你可能经常会遇到有一些结构的文本，这些结构可能同于那些现有命令所支持的结构。这样你不得不利用那些底层的``砖头''创建你自己的宏和脚本。这里说明的就是这类更复杂的东西。

有个简单的办法可以加速编辑-编译-修改这个循环。Vim提供 :make 命 令，用于进行编译，并且获取错误输出，把你带到发生错误的地方进行修正。如 果你使用了另一个编译器，那么错误就无法被Vim获得。如果不想自己动手，可 以修改'errorformat'选项。告诉Vim错误是什么样子，以及如何从中获 得文件名和行号。它支持复杂的gcc错误信息，所以应该也能支持其他编译器。

有时处理一个新的文件类型只需要设置几个选项或写一些宏。比如，为了在man手册中进行跳转，你可以写一个宏获取光标下的词，清除缓冲区，然后读入新的man手册。这是简单而高效的参照(cross-reference)方法。

使用三个基本步骤，你可以更有效地处理各种结构化文件。只需要想想你想对文件采取的操作，然后找到相应的命令去用就是了。就这么简单，你只要去做就成了。

第三部分：磨刀

养成习惯

要学会开车必须下功夫。这是不是你只骑自行车的原因么？当然不是，你会发现你必须花时间来获得所需的技术。文本编辑也不例外。你需要学习新的命令，并使用它直至成为习惯。

另一方面，你不应该试图学习编辑器提供的每个命令。这是彻底的浪费时间。大多数人只需要学习10%到20%的命令就足够工作了。但是每个人所需要的命令都各不相同。你需要不断学习，找出那些可以自动完成的重复操作。如果你只做一次操作，而且以后也不会再去做，那么就不需要进行优化。是如果你发现你在过去的一小时中重复了好几遍同样的操作，那么就有必要查看一下手册，看看能否更快速地完成。或者写一个宏来做。如果是是个不小的任务，比如对一类文本进行对齐，你需要阅读一下新闻组或看看Internet上是不是有人已经解决了同样的问题。

最根本的步骤是最后的那一个。你可能能够找到一个重复性的任务，找到一个不错的作法，可过了一个周末就彻底忘了自己是怎么做的了。这不成。你必须重复你的作法直到烂熟于胸。只有这时你才真正获得了你需要的高效。一次不要学得太多。一次只试一些工作得很好的方法。对于那些不常用的技巧，你可能只需要把它记下来，留待以后查阅。总之，如果抱着这样的目标，你的编辑技能就会更加有效。

最后需要指出的是，如果人们忽略了以上几点会发生什么：我仍然可以看到有人

盯着屏幕看上半天，用两个指头敲几下，然后继续抬头看着屏幕，还抱怨自己太累.. 把十个指头都用上！这不光更快，还不累。每天抽出一个小时练习一下指法，只要几星期就足够了。

后记

书名得益于Stephen R. Covey所著的那本畅销书《高效人的七种习惯》(``The 7 habits of highly effective people'')。

关于作者

Bram Moolenaar是Vim的主要作者。他编写了Vim核心功能，并采纳了许多开发者 提供的代码。他的e-mail地址是：Bram@Moolenaar.net