我把键盘左下角的control键改为了Esc键，这样从insert进入normal只需要用手掌的肉按一下，更快。

我把空格键右边的menu改成了control，方便用手掌使用快捷键。

把左下角的control改成了Esc,方便回到normal mode(正常模式)

输入大写字母时，按住shift 再按想打的字母，会输入大写，松开shift又会变小写。

Alt + Tab：在打开的窗口之间切换.按住alt,点击tab可以切换不同窗口.

在vscode的下面可以看见vim的状态，点击可以disabled。

normal模式下，按x会删除字符，按.会继续重复操作，即删除，按u可以撤销。

按dd会删除一整行，按.会重复删除一整行。

>G 命令会增加从当前行到文件末尾的缩进。如果我们在此命令之后使用点命令，那么“重复上一次更改”会指示 Vim 从当前位置到文件末尾增加缩进级别。

给下面的代码每行行末加上;

the\_vim\_way/2\_foo\_bar.js

var foo = 1

var bar = 'a'

var foobar = foo + bar

这样做需要将光标移动到行末，然后切换到插入模式进行修改。$ 命令将为我们处理光标移动(移动到行末)，然后我们可以运行 a;<Esc> 来进行更改(a会光标后插入，并且进入插入模式，esc键会返回到normal模式)

为了完成这项工作，我们可以在接下来的两行中运行完全相同的按键序列，但那样做会错过一个技巧。点命令会重复最后一次更改，所以我们可以只运行 j$.. 两次。一键（.）省去了三个键（a;<Esc>）。虽然这只是一个小小的节省，但当重复执行时，这些效率会积累起来。

还可以继续改进：

让我用中文来分析这段关于Vim编辑器的文本。

这段内容在讲解Vim中一个常见的编辑任务：在多行JavaScript代码末尾添加分号。文中展示了几种渐进式的解决方案：

1. 基础方法：

- 用 `$` 移动到行尾

- 用 `a;<Esc>` 添加分号并返回普通模式

2. 改进方法：

- 使用点命令（`.`）重复之前的修改操作

- 操作序列变成：`j$.`（向下移动，到行尾，重复修改）

3. 文中暗示还有更好的解决方案，指出仅仅为了定位光标就使用 `j$` 似乎不够高效。

这段文字实际上是在引导读者认识Vim的 `A` 命令，它能将 `$` 和 `a` 这两个操作合并成一个按键。使用 `A` 命令后，操作流程可以简化为：

1. 第一行使用 `A;<Esc>`

2. 后续行只需 `j.` 即可（向下移动并重复上次修改）

这种方式不仅减少了按键次数，还省去了单独的 `$` 移动操作，使整个编辑过程更加高效。这很好地体现了Vim的设计理念：将常用操作组合成更高效的单个命令。

这里体现了一个重要的Vim使用原则：始终寻找最简洁的操作方式。通过使用更智能的命令（如 `A` 而不是 `$a`），我们可以显著提高编辑效率。

您想了解更多关于这种Vim模式优化的细节吗？

第一种 $ a ;< Esc>

j .可以

第二种

A ；< Esc>

j .可以

第三种

$ a ; < Esc>

j .

为什么第三种不行呢

这里有些误解需要澄清 - 实际上第三种方式和第一种是完全一样的，它们都是可行的！

让我们详细分析这三种方法：

1. 第一种：`$ a ; <Esc>`, 然后 `j .`

- `$` 移动到行尾

- `a` 进入插入模式并将光标置于当前字符后

- 输入 `;`

- `<Esc>` 返回普通模式

- `j .` 移动到下一行并重复上次修改

2. 第二种：`A ; <Esc>`, 然后 `j .`

- `A` 相当于 `$a` 的组合命令，直接移动到行尾并进入插入模式

- 输入 `;`

- `<Esc>` 返回普通模式

- `j .` 移动到下一行并重复上次修改

3. 第三种：`$ a ; <Esc>`, 然后 `j .`

- 这与第一种完全相同！

所以第三种方式是可以工作的！它和第一种方式完全一样。可能造成你认为它"不行"的原因是：

1. 也许是操作时的某些细节出现了问题

2. 或者是理解上的混淆

三种方法的主要区别是：

- 第一种和第三种完全相同，使用了分解动作（`$` 然后 `a`）

- 第二种使用了复合命令 `A`，效果相同但按键更少

第二种方法（使用 `A`）是最推荐的，因为：

1. 按键更少（`A` 代替了 `$a`）

2. 操作更流畅

3. 更符合Vim的设计理念

如果你在实践第三种方法时遇到了具体问题，可以告诉我具体是什么情况，我很乐意帮你分析原因。

这是第三种的结果

23;

457;

547e;d

r8e5d;

firtu5;75

tgvyir8;46

gii;

tcyu;ir5

啊！现在我明白问题所在了。看起来在第三种方法中，你可能是这样操作的：

$ a ; <Esc>  // 移动到行尾，进入插入模式，添加分号，返回普通模式

j .          // 向下移动，重复上次"在当前位置插入分号"

问题出在 . 命令会重复的内容：

. 命令会重复的是上一次的修改操作，即 a;（在当前位置后面插入分号）

而不是重复整个 $a; 的组合（移动到行尾并插入分号）

这就导致：

第一行：正确添加分号在行尾

后续行：分号被插入在光标所在位置，而不是行尾

正确的做法应该是：

第一种方法：

$ a ; <Esc>  // 第一行

j $ .        // 后续行（必须先移动到行尾，再重复插入操作）

第二种方法（最优）：

A ; <Esc>    // 第一行

j .          // 后续行（因为A命令包含了移动到行尾的操作）

所以关键区别在于：

A 命令将"移动到行尾"和"进入插入模式"组合成了一个可重复的操作

而 $ 是一个光标移动命令，不会被 . 命令重复

这就是为什么使用 A 命令是更好的选择 - 它把定位和编辑合并成了一个可重复的操作。希望这个解释能帮助你理解问题所在！

而 $ 是一个光标移动命令，不会被 . 命令重复。为什么

这是 Vim 中一个重要的概念：. 命令（点命令）只重复上次的"修改"，而不重复"移动"。

让我们深入理解：

Vim 中的操作可以分为两类：

移动操作（Motion）：如 h, j, k, l, w, b, $, 0 等

修改操作（Change）：如 i, a, c, d, y 等

. 命令只会重复最后一次的"修改"操作，包括：

插入模式下输入的内容

删除操作

替换操作

等其他改变文本的操作

让我们用例子说明：

$ a ; <Esc>   // 这里包含了移动($)和修改(a;)

j .           // .只重复了 a; 这个修改操作，不重复$移动

而当我们使用 A 命令时：

A ; <Esc>     // A是一个组合命令，包含"移动到行尾并进入插入模式"

j .           // .重复整个A;操作，因为A是一个修改命令

这样设计的原因：

保持命令的一致性：移动命令专注于导航，修改命令专注于改变文本

提高效率：如果 . 重复所有操作（包括移动），可能会导致意外的结果

更好的可预测性：用户可以清楚地知道哪些操作会被重复

这就是为什么 A 命令比 $a 更适合这种场景 - 它把移动和修改合并成了一个单一的修改命令，使得整个操作可以被 . 完整地重复。

换个方式理解：

$ + a = 移动 + 修改

A = 移动和修改的组合命令

. 命令只记住最后一次"修改"，所以：

$a; 后用 . = 只重复 a;

A; 后用 . = 重复整个 "移动到行尾并插入分号" 的操作

<Esc> Press the Escape key

<CR> Press the carriage return key (also known as <Enter>)

<Ctrl> Press the Control key

<Tab> Press the Tab key

<Shift> Press the Shift key

<S-Tab>Press the <Shift> and <Tab> keys at the same time

<Up> Press the up arrow key

<Down> Press the down arrow key

␣Press the space bar

<S-Tab>Press the <Shift> and <Tab> keys at the same time。怎么区分是按S还是shift

在 Vim 的按键表示法中：

在 Vim 的按键表示法中：

<S-Tab> 和 <Shift-Tab> 是一样的，这里的 S 就是代表 Shift 键。

区分方法：

当看到 <S-something> 这种格式时：

S 是 Shift 的简写

例如：<S-Tab> = Shift+Tab

例如：<S-a> = Shift+a（即大写字母 A）//在我的电脑上，是按住shift,再点击a，效果更好。

而单独的 S 命令（不在 <> 中）：

就是 Vim 的 S 命令

等同于 cc 或 ^C

功能是删除整行并进入插入模式

例子：

S = Vim 的替换整行命令

<S-Tab> = 按下 Shift 和 Tab 键

<S-a> = 按下 Shift 和 a 键（输入大写 A）

简单记忆：

在 <> 中的 S- 就是 Shift

单独的 S 就是 Vim 命令

这是 Vim 的惯例表示法，在其他场景也是这样使用的：

<C-a> = Control+a

<M-a> = Meta(Alt)+a

<S-a> = Shift+a

vim的复合命令C s S I

让我详细解释这些 Vim 的复合命令，每个都是"Two for the Price of One"的典范：

C 命令

等同于 c$

功能：删除从光标到行尾的内容并进入插入模式

拆解动作：

c：修改操作

$：到行尾

使用场景：想替换从当前位置到行尾的所有内容

s 命令

等同于 cl

功能：删除光标下的字符并进入插入模式

拆解动作：

c：修改操作

l：一个字符

使用场景：想替换单个字符，但需要输入多个字符

S 命令

等同于 ^C 或 cc

功能：删除整行内容并进入插入模式

拆解动作：

^：移动到行首非空白字符

C：删除到行尾并进入插入模式

使用场景：想完全重写一整行

I 命令

等同于 ^i

功能：移动到行首第一个非空白字符并进入插入模式

拆解动作：

^：移动到行首非空白字符

i：进入插入模式

使用场景：想在行首非空白字符处插入内容

使用建议：

不要使用：c$ 而应该用 C

不要使用：cl 而应该用 s

不要使用：^C 或 cc 而应该用 S

不要使用：^i 而应该用 I

这些复合命令的好处：

减少按键次数

使操作更流畅

提高编辑效率

代码更易读和维护

记忆技巧：

C：Change to end of line

s：Substitute character

S：Substitute line

I：Insert at first non-blank character

这些命令都体现了 Vim 的设计智慧：用最少的按键实现最常用的操作组合。

A o O在vim是什么意思

让我详细解释这三个 Vim 复合命令：

A 命令

等同于 $a

功能：移动到行尾并进入插入模式

拆解动作：

$：移动到行尾

a：在当前字符后进入插入模式

使用场景：想在行尾添加内容

Hello world|     (|表示光标位置)

A 后:

Hello world|\_    (\_表示插入模式)

o 命令

等同于 $a<CR>

功能：在当前行下方新建一行并进入插入模式

拆解动作：

$：移到行尾

a：进入插入模式

<CR>：回车键

使用场景：想在当前行下方新增一行

First line|

o 后:

First line

|\_

O 命令

等同于 ko

功能：在当前行上方新建一行并进入插入模式

拆解动作：

k：向上移动

o：在下方新建一行

使用场景：想在当前行上方新增一行

First line|

O 后:

|\_

First line

使用建议：

不要使用：$a 而应该用 A

不要使用：$a<CR> 而应该用 o

不要使用：ko 而应该用 O

记忆技巧：

A：Append at end of line

o：Open line below

O：Open line above

这些命令的共同特点：

都会进入插入模式

都结合了移动和编辑操作

都是高频使用的操作

都优化了常见的编辑流程

这些复合命令再次体现了 Vim 的设计理念：

将常用的操作组合简化为单个按键

让编辑过程更加流畅和高效

"Two for the Price of One" 的思想

vim中的c l ^分别是什么意思

让我详细解释 Vim 中这三个基础命令：

c 命令（change）

基本功能：删除指定范围的文本并进入插入模式

需要配合移动命令或文本对象使用

常见组合：

cw：删除从光标到单词尾并进入插入模式

c$ 或 C：删除从光标到行尾并进入插入模式

cc 或 S：删除整行并进入插入模式

Hello World| (|是光标位置)

cw 后:

Hello |\_ (\_表示插入模式)

l 命令

基本功能：向右移动一个字符

等同于按右箭头键

是 Vim 基本移动命令 hjkl 中的一个

h：左

j：下

k：上

l：右

He|llo World

l 后:

Hel|lo World

^ 命令

基本功能：移动到行首第一个非空白字符

不同于 0（移动到行首第一个字符）

常与其他命令组合使用

···Hello| World    (·代表空格)

^ 后:

···|Hello World

对比示例：

Hello World     (前面有空格)

0 → |    Hello World  (移到真正的行首)

^ → |Hello World      (移到第一个非空白字符)

常见用法：

c 命令组合：

cw：修改单词

c$ 或 C：修改到行尾

cc：修改整行

l 命令用途：

精确移动光标

配合数字使用：3l 向右移动3个字符

^ 命令用途：

快速移动到代码实际开始的位置

与 I 命令相关：I = ^i

这些都是 Vim 中的基础命令，熟练掌握它们对提高编辑效率很重要。

vim中w是什么意思

w 在vim中是一个移动命令，表示移动光标到下一个单词的开头(word的首字母)。具体来说：

如果光标在单词中间，w会移动到下一个单词的开头

如果光标在单词开头，w会移动到再下一个单词的开头

它会把标点符号也视为单词的分隔符

相关的移动命令还有：

b - 向前移动到上一个单词的开头(backward)

e - 移动到当前单词的结尾(end)

W - 移动到下一个以空格分隔的单词(不把标点符号视为分隔符)

vim中i是什么意思

i 在vim中表示"insert"(插入)，是最基本的进入插入模式的命令之一。具体功能是：

在当前光标位置之前开始插入文本

按下 i 后，vim从命令(普通)模式切换到插入模式

在插入模式下，你输入的任何字符都会直接插入到文本中

按 ESC 键可以退出插入模式，回到命令模式

相关的插入命令包括：

a (append) - 在光标后插入

I - 在当前行的行首插入

A - 在当前行的行尾插入

o - 在当前行下方新建一行并插入

O - 在当前行上方新建一行并插入

{start}

var foo = "method("+argument1+","+argument2+")";

f+

var foo = "method("+argument1+","+argument2+")";上面的内容来自vim，其中{start}和f+是什么意思

这是一个vim操作的示例，让我解释这两个标记：

{start} - 这是一个标记，表示这是初始状态或者说是起始位置。它告诉你这是操作开始前的代码状态。

f+ - 这是一个vim命令：

f 是"find"（查找）命令

是要查找的具体字符

f+ 组合起来的意思是："向前查找并跳转到第一个出现的 + 符号处"

所以这个例子在展示：

第一行显示了初始代码

中间的 f+ 是你要输入的命令

第三行再次显示相同的代码，是为了说明光标会移动到第一个 + 符号的位置

实际操作效果是：当你在命令模式下输入 f+ 时，光标会从当前位置跳转到该行中的第一个 + 符号处。这是一个非常有用的快速导航命令。

后续如果要继续查找下一个 + 符号，可以：

再次使用 f+

或者使用 ; 命令重复上一次的查找

vim的;是什么

vim中的 ; (分号)命令是用来重复上一次的字符查找命令（f、F、t、T）的。

具体功能：

重复最近的 f 命令（向右移动光标到查找的字符上）

重复最近的 F 命令（向左移动光标到查找的字符上）

重复最近的 t 命令（向右移动光标到查找的字符前）

重复最近的 T 命令（向左移动光标到查找的字符后）

举例说明：

The cat and cat and cat

     ^ (光标在第一个 c 处)

f t  -> 跳到第一个 t

;    -> 跳到第二个 t

;    -> 跳到第三个 t

vim中的t是干嘛的

vim中的t命令用于光标移动，是"till"的缩写，表示"移动到指定字符之前"。它的具体用法是：

t{char} - 向右移动光标到指定字符的前一个位置

举些具体例子：

Hello World

^

如果光标在开头（^位置），输入tw，光标会移动到：

Hello World

    ^

光标停在"w"的前面。

相关的字符移动命令还有：

f{char} - 向右移动到指定字符上（find）

T{char} - 向左移动到指定字符后面（backwards till）

F{char} - 向左移动到指定字符上（backwards find）

这些命令都可以：

用分号(;)重复上一次操作

用逗号(,)反向重复上一次操作

可以配合数字使用，如2tw表示移动到第二个w的前面

每当Vim让重复一个动作或移动变得容易时，它总是提供某种方式让我们在不小心走得太远时能够退回。对于点命令，我们总是可以按u键来撤销最后的修改。如果在使用f{char}命令后按了太多次;键，我们会错过目标。但我们可以通过按,键来返回，它会在相反方向重复最后的f{char}搜索（参见第120页的技巧50"按字符查找"）。

/pattern<CR>怎么用

/pattern<CR> 是vim中的搜索命令，具体用法如下：

在普通模式下按 / 键，这会在窗口底部打开搜索提示符

输入你要搜索的内容（pattern）

按回车键（<CR> 就表示按回车键）

假设有这样一段文本：

Hello world

Hello vim

Hello pattern

如果你想搜索"Hello"：

1. 按 /

2. 输入 Hello

3. 按回车键

一些实用的pattern模式：

/hello - 搜索"hello"

/Hello\c - 搜索"Hello"，忽略大小写

/^Hello - 搜索行首的"Hello"

/world$ - 搜索行尾的"world"

/he.\*lo - 搜索以"he"开头"lo"结尾的单词

搜索后：

按 n 跳到下一个匹配

按 N 跳到上一个匹配

按 ctrl+o 跳回你开始搜索的位置

为什么这个搜索命令加了pattern

/pattern 中的 "pattern" 只是一个占位符(placeholder)，表示"在这里输入你想搜索的内容"。

它不是要你真的输入"pattern"这个词，而是提示你：

/ 后面应该跟着你要搜索的具体内容

举例说明：

如果你要搜索"hello"，就输入 /hello<CR>

如果你要搜索"vim"，就输入 /vim<CR>

如果你要搜索"test"，就输入 /test<CR>

这就像在写命令格式时：

cp file1 file2 中的 file1 和 file2 是占位符，表示"在这里输入实际的文件名"

/pattern 中的 pattern 也是占位符，表示"在这里输入实际要搜索的内容"

所以在文档中看到 /pattern<CR> 时，应该理解为：按/键，输入你要搜索的内容，然后按回车键。

命令中的pattern是刚刚解释时说的“一些实用的pattern模式：”吗

是的，你理解得对！文档中说的"pattern"正是指这些搜索模式（search patterns）。

这些pattern（模式）包括：

普通文本匹配：

/hello - 直接匹配文本

特殊的模式语法：

\c - 忽略大小写

^ - 匹配行首

$ - 匹配行尾

. - 匹配任意单个字符

\* - 匹配前一个字符零次或多次

等等...

所以当文档用"pattern"这个词时，它不仅指简单的文本搜索，还包括这些具有特殊含义的搜索模式。这就是为什么在vim的搜索命令格式中用"pattern"这个词 - 它暗示你可以使用这些强大的模式匹配功能。

这有点像正则表达式，如果你熟悉的话。vim的搜索模式实际上就是一种简化的正则表达式系统。

?pattern<CR>怎么用？

?pattern<CR> 和 /pattern<CR> 很像，但搜索方向相反：

* /pattern<CR> 是从光标位置向下（向后）搜索
* ?pattern<CR> 是从光标位置向上（向前）搜索

具体用法：

1. 在普通模式下按 ? 键，底部会打开搜索提示符
2. 输入要搜索的内容（pattern）
3. 按回车键

例子：

Hello Pattern 1

Hello Pattern 2   <- 假设光标在这行

Hello Pattern 3

- 用 `/Pattern` 会找到 Pattern 3

- 用 `?Pattern` 会找到 Pattern 1

搜索后：

n 继续按相同方向搜索

对于 / 搜索，n 向下找

对于 ? 搜索，n 向上找

N 按相反方向搜索

对于 / 搜索，N 向上找

对于 ? 搜索，N 向下找

所以 ? 就是 / 的反向版本，可以用来向上搜索文档。

n 继续按相同方向搜索。这里的n是什么的缩写，是new吗

n 在vim中代表"next"（下一个），不是"new"。

这个命令的设计逻辑是：

n (next) - 找下一个匹配，按照最后一次搜索的方向

N (previous/Next in reverse) - 找上一个匹配，与最后一次搜索方向相反

所以：

如果你用 /pattern 搜索（向下）：

n 继续向下找下一个

N 向上找上一个

如果你用 ?pattern 搜索（向上）：

n 继续向上找下一个

N 向下找上一个

记忆方法：

小写n是"next"的缩写，表示"继续按相同方向找下一个"

大写N就是小写n的反向操作

这个设计遵循了vim的一般模式：小写命令和大写命令常常是相反的操作。

:s/target/replacement怎么用

:s/target/replacement 是vim的替换命令（substitute），用于替换文本。

基本语法：:s/要查找的内容/替换成的内容

具体例子：

原文：Hello world

命令：:s/world/vim

结果：Hello vim

重要说明：

默认情况下，:s 命令只替换当前行的第一个匹配

常用的修饰符：

g（global）：替换该行所有匹配

用了这个命令之后怎么用&重复

有几种方法可以用&重复上一次替换命令：

& - 重复上一次替换操作（只在当前行，不带任何修饰符）

:s/hello/world     # 第一次替换

&                  # 重复上次替换（只在当前行）

g& - 重复上一次替换操作（在整个文件中，保留原来的修饰符）

:%s/hello/world/g  # 在整个文件中替换

g&                 # 用相同的pattern在整个文件中重复替换

:&& - 重复上一次替换操作（在当前行，保留原来的修饰符）

:s/hello/world/g   # 替换当前行所有匹配

:&&                # 重复上次替换，保留/g标志

:s/hello/world/g   # 替换当前行所有匹配

:&&                # 重复上次替换，保留/g标志

需要注意：

单独的&只重复基本替换，不会保留之前命令中的修饰符（如g）

g&会保留修饰符并应用到整个文件

:&&会保留修饰符但只应用到当前行

所以如果你最初用了:%s/hello/world/g，想要完全重复这个操作，最好用g&。

:s/hello/world     # 第一次替换

&                  # 重复上次替换（只在当前行）。写完:s/hello/world要按enter再写&吗

是的，操作顺序是：

输入 :s/hello/world

按回车键（Enter）执行这个替换命令

然后在normal模式下（不需要输入:）直接按 & 键重复上次替换

所以完整的操作流程是：

:s/hello/world<Enter>    # 第一次替换

&                        # 直接按&键（不需要:）重复上次替换

这里要注意：

& 是在normal模式下直接按的键，不需要输入:

而 :&& 是在命令模式下输入的，需要先按:

总结来说：

&：在normal模式下直接按

:&&：需要输入:进入命令模式，然后输入&&再回车

qx{changes}q怎么用

qx{changes}q 是vim中的宏录制命令，让我分解说明：

q - 开始录制宏

x - 选择一个寄存器来存储宏（可以用a-z中的任何字母）

{changes} - 你要录制的操作序列

q - 结束录制

具体使用步骤：

1. 按 q 后接一个字母（比如 qa）开始录制

2. 执行你想要录制的操作

3. 再按一次 q 结束录制

实际例子：

假设要在每行开头添加 "Hello "

1. qa               # 开始录制到寄存器 a

2. I Hello <Esc>    # 在行首插入"Hello "

3. j               # 移动到下一行

4. q               # 结束录制

录制完成后：

用 @a 执行一次宏

用 5@a 执行5次宏

用 @@ 重复上次执行的宏

记住：

x可以是a-z中的任何字母

在录制过程中你可以执行任何vim命令

录制时要考虑命令的可重复性

从claude复制文本粘贴到wps时，如果出现背景颜色，可以先粘贴到vscode上面，再剪切到wps上面。注意此时要让vim处于disabled，而不是insert。

qx{changes}q，其中q是什么意思，是一个缩写吗

q 在这里来自单词 "quote" 或 "record"，表示引用或记录的意思。

不过关于这个说法我需要说明：我不能完全确定 Vim 创建者选择 q 的确切原因。不同的资料可能会给出不同的解释。

我们确切知道的是：

q 用来开始宏录制

q 再次按下时用来结束宏录制

这种命令选择可能考虑了：

q 键在键盘上容易按到

r (record) 已经被用作"替换字符"的命令

其他更多可能的历史原因

如果你对这个命令的历史来源感兴趣，我建议查看 Vim 的官方文档或历史资料来获取更准确的信息。这里我只能提供操作层面的说明，而不是确切的命名由来。

:%s/content/copy/g 是什么意思

这个命令是 Vim 的 :substitute 命令，用于查找和替换文本。让我们逐步解析这个命令：

解释

:：进入 Vim 的命令行模式。

%：表示整个文件（而不是当前行）。

s：表示替换（substitute）命令。

/content/：要查找的模式，即你希望替换的文本“content”。

/copy/：替换模式，即用来替换“content”的文本“copy”。

g：表示全局（global），即在每一行中替换所有匹配的实例，而不仅仅是每行的第一个。

整体命令

:%s/content/copy/g 将在整个文件中查找所有“content”的实例，并将它们替换为“copy”。

示例

如果你的文件包含以下内容：

We need to replace the word content.

Another line with the word content.

运行 :%s/content/copy/g 后，文件将变为：

We need to replace the word copy.

Another line with the word copy.

vim中\* command是什么

在 Vim 中，\* 命令是一个非常强大的搜索命令。它的功能是在当前行中查找并高亮显示光标所在单词的所有匹配项，然后将光标移动到下一个匹配项上。

使用方法

将光标放在你要搜索的单词上。

按 \* 键。

Vim 会自动搜索下一个出现的匹配项，并高亮所有匹配项。

示例

假设你有以下文本：

The cat and the hat sat on the mat.

将光标放在第一个 the 上，然后按 \*，结果如下：

The cat and the hat sat on the mat.

^^^

光标会跳到下一个 the，并且所有 the 都会被高亮显示。

相关命令

#：与 \* 类似，但搜索光标下单词的上一个匹配项。

n：跳到下一个搜索结果。

N：跳到上一个搜索结果。

这个命令非常有用，可以快速查找并浏览文档中的所有匹配项，而无需手动输入搜索字符串。

我们使用一个按键来移动，另一个按键来执行。真的是最理想的解决方案了，对吧？这种编辑策略会一再出现，所以为了方便起见，我们将这种模式称为点公式。

Normal Mode

vim中x是什么，b是什么

让我来解释一下vim中这两个常用的基本命令：

x - 在普通模式下用于删除光标所在位置的单个字符。它相当于 Delete 键的功能。

b - 在普通模式下用于向后（左）移动光标到当前单词的开头。如果光标已经在单词开头，则会移动到前一个单词的开头。

x和s的有区别吗

x 和 s 在vim中虽然都可以用来删除字符，但有以下区别：

x - 删除光标所在的单个字符，删除后光标停留在下一个字符。保持在普通模式。

s - 删除光标所在的单个字符，并立即切换到插入模式。相当于执行了 x 后紧接着按 i 进入插入模式。

简单记忆：

x (delete) = 仅删除

s (substitute) = 删除并替换(进入插入模式)

你可以这样理解：当你只想删除一个字符时用 x，当你想删除一个字符并输入新的内容时用 s。

既然x是delete,为什么不用d表示，x是什么的缩写 吗，或者说有别的原因

这是个很好的问题。vim中选择用x而不是d来表示delete主要有几个历史和实用原因：

历史渊源：

x键继承自更早的vi编辑器，而vi又继承自更早的ex编辑器

在老式终端上，x键通常被用作删除/擦除字符的标准按键

它的位置在键盘上比较容易按到（比d更靠右）

d已有其他用途：

vim中d被设计为更强大的"delete operator"，可以配合动作命令使用

比如：dw（删除单词）, dd（删除整行）等

d更像是一个"前缀"操作符，而x是一个独立的单字符操作

符号联想：

x在视觉上像一个叉号，暗示"删除"的动作

这种视觉联想可能也是设计考虑之一

所以x并不是某个单词的缩写，而是基于历史传统和实用性考虑的设计选择。这也体现了vim继承了很多早期文本编辑器的设计理念。

b - 在普通模式下用于向后（左）移动光标到当前单词的开头。如果光标已经在单词开头，则会移动到前一个单词的开头。b是什么的缩写吗

b 是 "back" 或 "backward" 的缩写。这个选择很符合直觉，因为它确实是向后(backward)移动到单词开头。

vim 中的移动命令有一组相关的设计：

b (back/backward) - 向左移动到单词开头

w (word) - 向右移动到单词开头

e (end) - 向右移动到单词结尾

这些命令都可以和数字配合使用：

2b - 向后移动2个单词

3w - 向前移动3个单词

4e - 向前移动到第4个单词的结尾

而且它们都可以和其他操作符（如d）配合：

db - 向后删除到单词开头

dw - 向前删除到单词开头

de - 删除到单词结尾

这种设计体现了vim命令的助记性和系统性，让用户更容易记住和使用这些命令。

vim中d 是什么

vim中的 d 是一个强大的删除操作符(operator)，它需要配合一个动作命令(motion)来使用。以下是一些常见用法：

基本组合：

dd - 删除整行

dw - 从光标删除到下一个单词开头

de - 从光标删除到当前单词结尾

d$ 或 D - 从光标删除到行尾

d0 - 从光标删除到行首

dj - 删除当前行和下一行

dk - 删除当前行和上一行

数字组合：

2dd - 删除2行

d2w - 删除2个单词

高级用法：

di( - 删除括号内的内容

da( - 删除括号和括号内的内容

dit - 删除 HTML/XML 标签内的内容

删除的内容会被存入 vim 的寄存器中，可以用 p 命令粘贴。

d 作为操作符的特点是它总是需要一个动作命令来指定操作范围，这也是它与 x（删除单个字符）的主要区别。d 实际上是 "delete" 的缩写。

vim中dw是什么

记住一个关键点：dw 会删除从光标位置到下一个单词开头的所有内容（包括空格）。这和 de（删除到当前单词结尾，不包括后面的空格）是不同的。

那如果光标已经在最后一个单词上面，后面没有下一个单词，按dw会怎么用

如果光标在最后一个单词上，dw 会删除到行尾。我用例子来说明：

光标在最后单词的开头：

hello world |next

dw后：

hello world

光标在最后单词的中间：

hello world ne|xt

dw后：

hello world ne

即使最后有空格也是一样：

如果光标在行尾的空格上：

hello world next|

dw后：

hello world next

这时候 dw 的行为其实和 D（删除到行尾）或者 d$ 很类似。因为对vim来说，既然后面没有"下一个单词"了，就会删除到行尾。

vim的daw是什么

vim中的 daw 是一个组合命令：

* d (delete) - 删除操作符
* aw (a word) - 表示"一个单词"这个文本对象（包括后面的空格）

daw 会删除整个单词，不管光标在单词的任何位置（开头、中间或结尾），并且会删除单词后的空格。让我用例子来说明：

1. 光标在单词任何位置都能删除整个单词：

The c|at runs fast

daw后：

The runs fast

即使光标在单词开头：

即使光标在单词开头：

The |cat runs fast

daw后：

The runs fast

或者单词结尾：

The cat| runs fast

daw后：

The runs fast

daw 和 dw 的主要区别：

dw 只删除从光标位置到下一个单词开头

daw 删除整个单词（不管光标位置）加后面的空格

所以 daw 通常更方便，因为：

不需要考虑光标的具体位置

删除更干净（包括后面的空格）

更容易重复使用（用 . 命令）

vim中的b是什么

vim中的 b 命令是移动光标的命令：

b (back/backward) - 向后（左）移动光标到当前单词的开头

具体行为：

如果光标在单词中间，移动到该单词的开头

如果光标已经在单词开头，则移动到前一个单词的开头

vim在normal模式下光标是一个长方形，那怎么知道是在字母的左边还是右边呢

在vim的normal模式下，长方形光标的实际位置确实可能让人困惑。实际上，这个长方形光标总是停在字符的上面，但光标的"逻辑位置"是在字符的左侧。

插入模式验证

按 i 进入插入模式

观察新字符插入的位置

这个位置就是光标的逻辑位置

vim中的yyp是什么

vim中的 yyp 是两个连续的操作：

yy - 复制（拷贝）整行

p - 在当前行下方粘贴复制的内容

例如：

hello world   (光标在这一行)

yy            (复制整行)

p             (在下方粘贴)

结果变为：

hello world

hello world

这是一个快速复制并粘贴整行的方法。p 会在当前行的下方粘贴，如果想在上方粘贴，可以用大写的 P。

vim的w是什么

vim中的 w 是移动光标的命令：

w (word) - 向前（右）移动光标到下一个单词的开头

具体行为：

如果光标在单词中间，移动到下一个单词的开头

如果光标已经在单词开头，则移动到下一个单词的开头

例子：

hello wo|rld next   (|表示光标位置)

按w后：

hello world |next

<C-a> and <C-x> commands是什么

<C-a> 和 <C-x> 是vim中用于数字增减的命令：

<C-a> 是按住Ctrl键和a键，用于增加数字

<C-x> 是按住Ctrl键和x键，用于减少数字

基本用法：

不带数字前缀：

|5     (|表示光标位置)

<C-a>  (数字加1)

6

|5

<C-x>  (数字减1)

4

带数字前缀：

|5

10<C-a>  (数字加10)

15

|5

10<C-x>  (数字减10)

-5

特点：

如果光标不在数字上，会自动找到当前行后面的第一个数字

可以处理各种进制的数字（十进制、十六进制等）

可以对数字做批量增减

例子：

.blog { background-position: 0px 0px }

光标在任意位置

180<C-x>

结果：

.blog { background-position: -180px 0px }

<C-a> and <C-x> commands中的a  x是什么的缩写吗

<C-a> 中的 a 是 "add"（添加）的缩写。

<C-x> 中的 x 可能来自 "decrement"（递减）的 "x" 或者是因为 x 键在键盘上与 a 的位置对称。

这个设计遵循了一些常见模式：

a 用于增加（add/augment）

x 用于减少（这更像是一个设计选择，因为 x 键位容易按到，且与 a 形成对比）

这个命令组合的设计可能也受到了其他Unix工具和编辑器的影响，比如在 GNU Screen 和 tmux 这样的终端复用器中，<C-a> 也是一个常用的命令前缀。

不过需要注意的是，这些解释部分是推测，因为vim的一些命令设计选择可以追溯到很早的vi编辑器时期。

我在vscode上面按<C-a>，数字没有加1，而是选中全文怎么办

这是因为VSCode默认的快捷键<C-a>是"全选"(Select All)。要在VSCode中使用vim的数字增减功能，你需要：

确保已安装VSCode的Vim插件（VSCodeVim）

修改VSCode的键盘快捷键设置：

打开设置（Settings）

搜索"keyboard shortcuts"或直接按Ctrl+K Ctrl+S

搜索<C-a>

找到Select All这一项

禁用它或改成其他快捷键

在VSCode的settings.json中添加vim快捷键映射：

"vim.normalModeKeyBindingsNonRecursive": [

    {

        "before": ["<C-a>"],

        "after": ["<C-a>"]

    },

    {

        "before": ["<C-x>"],

        "after": ["<C-x>"]

    }

]

这样设置后，<C-a>和<C-x>就会按vim的方式工作了。

如果你不想修改VSCode的默认行为，也可以在vim配置中设置其他快捷键来实现相同功能。

我还没有去修改。

Vim 的语法还有一条规则：当一个操作命令被重复调用时，它会作用于当前行。例如，dd 删除当前行，而 >> 缩进当前行。gU 命令是一个特殊情况。我们可以通过运行 gUgU 或简写形式 gUU 来使其作用于当前行。

vim的文本对象是什么

在 Vim 中，**文本对象** 是一组特殊的操作目标，它们定义了可以选择的一块逻辑文本（例如一个单词、段落、括号中的内容等）。文本对象是 Vim 强大编辑能力的核心之一，主要用于选择或操作特定范围的文本。

### ****文本对象的特点****

* **范围明确**：文本对象的边界通常由语法或结构决定，例如句子、段落、括号等。
* **操作灵活**：可以与操作符（如 d 删除、y 复制、c 修改等）配合使用。
* **逻辑性强**：选择的范围是根据内容逻辑，而不是物理上的光标位置。

### ****常见的文本对象****

#### ****单词相关****

* aw（a word）：包括当前单词及其后面的空格。
* iw（inner word）：只包括当前单词，不包含空格。

#### ****句子和段落****

* as（a sentence）：包括当前句子及其后面的空格。
* is（inner sentence）：只包括当前句子，不包含多余的空格。
* ap（a paragraph）：包括当前段落及段落之间的空行。
* ip（inner paragraph）：只包括当前段落的内容，不包含空行。

#### ****括号和引号****

* a( 或 ab（a block）：包括括号 () 和括号中的内容。
* i( 或 ib（inner block）：只包括括号中的内容，不包含括号本身。
* 类似的还有：
  + {}：a{ 和 i{。
  + []：a[ 和 i[。
  + ""：a" 和 i"。
  + ''：a' 和 i'。

#### ****标签（如 HTML/XML）****

* at（a tag）：包括当前 HTML/XML 标签及其内容。
* it（inner tag）：只包括标签的内容，不包含标签本身。

### ****使用示例****

#### 配合操作符

* daw：删除当前单词及其后面的空格。
* ci(：修改括号中的内容（进入插入模式）。
* yas：复制当前句子。
* vip：选择当前段落（可视模式）。

#### 配合可视模式

* 在普通模式下按 v 或 V 进入可视模式，然后键入文本对象，例如 iw 或 ip，以选中对应范围的文本。

### ****扩展文本对象****

通过插件或自定义，可以增加更多的文本对象。例如：

* kana/vim-textobj-entire 插件：增加 ie 和 ae，操作整个文件。
* vim-textobj-indent 插件：增加基于缩进的文本对象（如 ii 和 ai）。

### ****总结****

Vim 的文本对象让编辑文本变得更加直观和高效。通过学习和熟练使用文本对象，配合操作符和插件，可以极大提升 Vim 的编辑能力，满足各种复杂场景的需求。

Vim的切换大小写为什么是g~,g是什么的缩写吗？

g **作为扩展命令的前缀**  
g 是 Vim 中的一组扩展命令的前缀，用来引导一系列操作。例如：

* gg 跳到文件的开头。
* gq 用于文本的重新格式化。
* gu 转换为小写。
* gU 转换为大写。(Upcase)
* g~ 切换大小写（toggle case）。

~ 表示大小写切换  
在 Vim 中，~本身表示切换当前字符的大小写（在普通模式下操作当前字符）。因此，g~ 可以被看作是对大小写切换功能的扩展。

组合逻辑  
g~ 作为前缀，后接一个“动作”或“范围”，例如：

g~w：切换从光标位置到下一个单词的大小写。

g~$：切换光标位置到行尾的大小写。

g~G：切换从光标位置到文件末尾的大小写。

所以，g 并不是一个单词的缩写，而是 Vim 中用来引导扩展功能的一部分。g~ 的设计符合 Vim 的直观逻辑，通过前缀扩展基本命令实现更灵活的操作。

Insert Mode

在插入模式下，退格键的功能和你预期的一样：它会删除光标前面的字符。以下这些组合键也可供我们使用：

<C-h> Delete back one character (backspace)

<C-w> Delete back one word

<C-u> Delete back to start of line

这些组合键确实有一些便于记忆的方法：

<C-h> (删除一个字符):

这个最容易记，因为 h 就相当于退格键(backspace)

在很多终端环境中，Ctrl-h 历史上就是退格键的表示方式

你可以把 h 想象成 "hit back"（往回删）

<C-w> (删除一个词):

w 在 Vim 的普通模式下本来就表示 "word"（词）

可以记成 "wipe word"（擦除词）

<C-u> (删除到行首):

u 可以理解为 "up"（向上）

也可以记成 "undo line"（撤销光标同一行前面部分）

在 Unix 系统中，Ctrl-u 也是一个常用的清除当前行的快捷键

实用记忆技巧：

这些组合键的删除范围逐渐增大：字符(h) < 单词(w) < 整行(u)

它们在 Unix/Linux 命令行中也很常用，掌握后可以一举多得

可以把这三个命令想象成不同级别的"后悔"：轻微后悔(删一字)、中度后悔(删一词)、深度后悔(删一行)

yt,     Practical Vim, by Drew Neil

Read Drew Neil's

jA␣ Practical Vim, by Drew Neil

Read Drew Neil's

<C-r>0 Practical Vim, by Drew Neil

Read Drew Neil's Practical Vim

.<Esc> Practical Vim, by Drew Neil

Read Drew Neil's Practical Vim.

对于字符类寄存器使用 <C-r>{register}

<C-r>{register} 命令在插入模式下粘贴几个词时很方便。但如果寄存器包含大量文本，你可能会注意到屏幕更新时有轻微的延迟。这是因为 Vim 会像逐个字符输入一样插入寄存器中的文本。如果启用了 'textwidth' 或 'autoindent' 选项，你可能会得到不想要的换行或额外的缩进。

<C-r><C-p>{register} 命令更智能。它会原样插入文本并修复任何意外的缩进（参见 :h i\_CTRL-R\_CTRL-P）。但这个命令组合有点复杂！如果我想粘贴包含多行文本的寄存器内容，我更倾向于切换到普通模式，使用其中一个粘贴命令（参见第155页的技巧63"从寄存器粘贴"）。

注意：

<C-r> = Ctrl + r

<C-p> = Ctrl + p

{register} 表示寄存器地址（如0、1、2等）

vim的yt,是什么

vim的yt,命令是一个复合命令，让我来拆解说明：

y - 复制（yank）命令

t - till的缩写，表示"直到"

, - 目标字符（在这个例子中是逗号）

所以 yt, 的完整含义是：复制从当前光标位置到下一个逗号之前的所有文本（不包括逗号）

例如：

The|command yt, yanks the words...

（|表示光标位置）

执行 yt, 后，会复制"command"这个词，因为它位于光标和第一个逗号之间。

其他类似的例子：

yt. - 复制到下一个句号之前的内容

yt) - 复制到下一个右括号之前的内容

yt空格 - 复制到下一个空格之前的内容

这是一个非常实用的命令，因为它让你能够：

快速复制到特定字符之前的内容

不需要移动光标就能完成复制

非常精确地控制要复制的文本范围

记忆技巧：

y = yank（复制）

t = till（直到...之前）

最后的字符是你想要停止的位置

命令 yt, 会将文字 Practical Vim 复制到复制寄存器中（我们将在第120页的技巧50"按字符查找"中学习 t{char} 动作）。在插入模式下，我们可以按 <C-r>0 来将刚才复制的文本粘贴到当前光标位置。我们将在第145页的第10章"复制和粘贴"中更详细地讨论寄存器和复制操作。

<C-r>=6\*35<CR>是什么

让我解释这个命令序列：

<C-r>=6\*35<CR> 是在插入模式下进行快速计算的命令，它分为几个步骤：

<C-r> (按下 Ctrl+r)：

在插入模式下准备访问寄存器

= (按下等号键)：

表示要使用表达式寄存器

这时会在Vim窗口底部出现一个计算提示符

6\*35 (输入计算表达式)：

在提示符中输入要计算的表达式

这个例子是计算 6 乘以 35

<CR> (按下回车键)：

执行计算

将结果（210）直接插入到当前光标位置

实际使用例子：

价格是: $|   (光标位置用|表示)

按下 <C-r>=6\*35<CR> 后：

价格是: $210|

这个功能很实用，因为：

可以直接在文档中进行计算

不需要使用外部计算器

计算结果直接插入到文本中

可以进行各种数学运算（不仅仅是乘法）

其他计算例子：

<C-r>=10+5<CR> → 15

<C-r>=100/4<CR> → 25

<C-r>=2\*\*3<CR> → 8 (2的3次方)

表达式寄存器允许我们执行计算，然后直接将结果插入到文档中。在这个技巧中，我们将看到这个强大功能的一个应用。

Vim的大多数寄存器要么包含作为字符串的文本，要么包含完整的文本行。删除和复制命令允许我们设置寄存器的内容，而粘贴命令则允许我们通过将内容插入文档来获取寄存器中的内容。

表达式寄存器与众不同。它可以计算一段Vim脚本代码并返回结果。在这里，我们可以把它当作计算器使用。给它传递一个简单的算术表达式，比如1+1，就会得到结果2。我们可以像使用保存在普通寄存器中的文本一样使用表达式寄存器的返回值。

表达式寄存器用=符号来表示。在插入模式下，我们可以通过输入<C-r>=来访问它。这会在屏幕底部打开一个提示符，我们可以在那里输入想要计算的表达式。完成后，按回车键<CR>，Vim就会在文档中的当前位置插入结果。

为了让系统能够识别Vim命令，您需要将Vim的安装目录添加到系统的环境变量中。打开Windows系统的环境变量配置窗口，将Vim的安装目录路径添加到系统变量“Path”中，然后保存设置。

我把D:\Vim\vim91添加到系统变量path了。

vim的r是什么

在Vim中，r是一个非常有用的快捷键，主要功能是：

替换单个字符

在普通模式（Normal Mode）下

光标停留在想要替换的字符上

按r，然后立即输入新字符

这会直接用新字符替换光标下的原字符

例子：

原文: hello world

光标在h上时：

ra  →  aello world  (h被a替换)

使用场景：

快速修正单个错误字符

不需要进入插入模式就能替换字符

比删除再插入更高效

相关的还有：

R（大写R）：进入替换模式，可以连续替换多个字符

<C-r>：插入寄存器内容（我们之前讨论过的）

Visual Mode

一些Visual模式的命令与普通模式下的基本功能相同，但有细微的差别。例如，c命令在两种模式下都是一致的，都是删除指定的文本然后切换到插入模式。不同之处在于我们指定操作范围的方式。在普通模式下，我们先触发更改命令，然后通过动作指定范围。如果你还记得第24页的技巧12（结合与征服），这被称为操作符命令。而在Visual模式下，我们先进行选择，然后再触发更改命令。这种控制的反转可以推广到所有操作符命令（参见第25页的表2，Vim的操作符命令）。对大多数人来说，Visual模式的方法感觉更直观。

vim 里面的operator motion action commad分别是什么

1. Operator (操作符)

定义: 表示对文本执行的某种操作。

例子: d (删除)、y (复制/yank)、c (更改/change) 等。

特点:

Operator 本身并不完成操作，必须与后续的 motion 或 范围 搭配使用。

示例:

dw：删除当前单词。

d$：删除光标到行尾。

yG：复制到文件末尾。

2. Motion (动作/移动)

定义: 描述光标移动的范围，用于指定 Operator 作用的区域。

例子: w (下一个单词)、e (单词结尾)、$ (行尾)、gg (文件开头) 等。

特点:

Motion 定义了文本操作的范围。

可以独立使用来移动光标，或与 operator 结合使用。

示例:

w：光标移动到下一个单词开头。

5j：向下移动 5 行。

3. Action (动作命令)

定义: 指独立的命令，可以直接完成某种功能，无需搭配其他部分。

例子: x (删除当前字符)、p (粘贴)、u (撤销) 等。

特点:

不依赖 Motion 或 Operator，可以独立完成动作。

示例:

x：删除光标所在的字符。

u：撤销上一步操作。

vim的command是什么

在 Vim 中，command 通常指 命令模式 下用于执行各种任务的命令。这些命令提供了强大的功能，用于文件操作、文本编辑、设置配置等。

1. Command 的定义

Command 是在命令行模式（即按 : 键进入的模式）下输入的命令，用于执行特定任务。

用途广泛，例如：保存文件、退出 Vim、替换文本、调整设置等。

2. Command 的基本分类

(1) 文件操作命令

用于处理文件的保存、打开、关闭等操作。

:w：保存当前文件（write）。

:q：退出当前窗口（quit）。

:wq：保存并退出。

:e filename：打开指定文件。

:x：保存并退出，等价于 :wq。

(2) 文本编辑命令

执行查找、替换、删除等编辑操作。

:s/foo/bar/：在当前行，将第一个 foo 替换为 bar。

:%s/foo/bar/g：在整个文件中，将所有 foo 替换为 bar。

:g/pattern/d：删除所有匹配 pattern 的行。

:m10：将当前行移动到第 10 行之后。

(3) 设置命令

用于配置 Vim 的行为。

:set number：显示行号。

:set ignorecase：搜索时忽略大小写。

:set tabstop=4：设置 Tab 的宽度为 4。

:set wrap：启用自动换行。

(4) 帮助命令

查找 Vim 的内置帮助文档。

:help：打开帮助页面。

:help :w：查看 :w 命令的帮助信息。

:help motion：查看与 Motion 相关的帮助。

(5) 窗口/标签操作命令

管理多个窗口或标签。

:split：水平分割窗口。

:vsplit：垂直分割窗口。

:tabnew：打开一个新的标签页。

:tabclose：关闭当前标签页。

3. Command 的执行流程

按下 : 进入命令行模式。

输入具体的命令（如 :w、:q! 等）。

按 Enter 执行。

4. Command 和其他模式的区别

Command 模式（:command）：输入 : 开始，用于执行各种文件级别的操作。

Normal 模式：用于日常的文本移动与操作，通常是单字符命令（如 dd、yy）。

Insert 模式：用于直接输入文本。

Visual 模式：用于选中文本。

示例：常用 Command

:w                   " 保存文件

:q!                  " 强制退出（不保存）

:split               " 水平分割窗口

:%s/foo/bar/g        " 全局替换所有 foo 为 bar

:set number          " 显示行号

:help :q             " 查看 :q 命令的帮助

We can even

use the search command (and n/N) to jump to pattern matches.是什么意思

1. Search Command（搜索命令）

搜索命令的作用：

在 Vim 中，用来查找特定的文本模式。

用法：

/pattern：从光标所在位置向下查找 pattern。

?pattern：从光标所在位置向上查找 pattern。

示例：

输入 /hello，Vim 会查找当前文档中下一个包含 hello 的位置。

输入 ?error，Vim 会向上查找包含 error 的位置。

2. n 和 N 的作用

n：重复上一次的搜索命令，保持方向。

如果上次是 /pattern（向下搜索），按 n 会继续向下查找。

如果上次是 ?pattern（向上搜索），按 n 会继续向上查找。

N：重复上一次的搜索命令，但反转方向。

如果上次是 /pattern（向下搜索），按 N 会向上查找。

如果上次是 ?pattern（向上搜索），按 N 会向下查找。

3. 完整操作流程

按 /，然后输入需要搜索的模式，比如 error，再按回车。

光标会跳转到下一个匹配的 error。

按 n，光标会跳转到下一个 error。

按 N，光标会跳转回上一个 error。

4. 翻译

"We can even use the search command (and n/N) to jump to pattern matches." 可以翻译为：

我们甚至可以使用搜索命令（以及 n/N）来跳转到与模式匹配的位置。

这说明了 Vim 的搜索功能和快速跳转的高效性，非常适合定位和浏览文件中的特定内容。

vim怎么粘贴

从系统剪贴板粘贴

在插入模式下粘贴：

按 i 进入插入模式。

按 Ctrl+Shift+V 粘贴内容（在许多终端模拟器中）。

或者，右键单击终端窗口并选择“粘贴”。

在正常模式下粘贴：

使用 "+p 命令从系统剪贴板粘贴内容。

从 Vim 寄存器粘贴

使用寄存器粘贴：

使用 p 命令在光标后面粘贴内容（默认情况下，p 从无名寄存器中粘贴内容）。

使用 P 命令在光标前面粘贴内容。

从指定寄存器粘贴：

使用 "{register}p 从指定寄存器粘贴内容。例如，从寄存器 a 粘贴内容，可以使用 "ap。

示例

假设你在 Vim 中复制了一段文本，并希望粘贴到另一个位置：

在正常模式下，移动光标到你想粘贴的位置。

按 p 在光标后粘贴，或按 P 在光标前粘贴。

插入模式下的系统剪贴板粘贴

如果你在插入模式下，并想从系统剪贴板粘贴，可以按 Ctrl+Shift+V 或右键单击选择“粘贴”。

一些Visual模式的命令与普通模式下的基本功能相同，但有细微的差别。例如，c命令在两种模式下都是一致的，都是删除指定的文本然后切换到插入模式。不同之处在于我们指定操作范围的方式。在普通模式下，我们先触发更改命令，然后通过动作指定范围。如果你还记得第24页的技巧12（结合与征服），这被称为操作符命令。而在Visual模式下，我们先进行选择，然后再触发更改命令。这种控制的反转可以推广到所有操作符命令（参见第25页的表2，Vim的操作符命令）。对大多数人来说，Visual模式的方法感觉更直观。  
  
o Go to other end of highlighted text是什么意思

"Go to other end of highlighted text" 的意思是"跳转到高亮文本的另一端"。

让我详细解释一下：

在Vim的Visual模式下，当你选中一段文本时，会有两个端点：

起始点（你开始选择的位置）

结束点（当前光标所在的位置）

按下o键可以让光标在这两个端点之间来回跳转

这个功能非常有用，因为它允许你：

快速调整选区的两端

在不取消选择的情况下修改选区的范围

从另一个方向重新定义选区

举个例子：

假设你用v进入Visual模式并向右选择了一些文本

发现起始点选错了

按o跳回起始点

调整起始点位置

再按o回到结束点继续选择

这是一个非常实用的命令，可以让你更灵活地控制选区的范围。

vim的vbb是什么意思

vim中的vbb是一个组合命令，让我来解释其含义：

v - 进入字符模式的Visual模式（可视模式）

bb - 向后移动两个词（b命令向后移动一个词，所以bb就是向后移动两个词）

所以当你输入vbb时：

首先进入Visual模式

然后光标向后移动两个词

在这个过程中，从起始位置到光标当前位置的文本会被高亮选中

vim的e是什么意思

vim中的e命令表示"跳转到当前单词的结尾"（end）。

详细说明：

e - 移动到当前或下一个单词的结尾。光标在单词上面就移动到当前单词的结尾，光标在空格，就移动到下一个单词的结尾。

E - 移动到当前或下一个字串（以空格分隔的字符串）的结尾

vim的J是什么

在 Vim 中，J 命令用于将光标所在行与下一行合并。这是一个非常有用的命令，特别是在编辑和整理文本时。使用 J 命令时，当前行的内容会与下一行的内容连接在一起，并在它们之间插入一个空格。

尽可能使用操作符而不是可视命令

虽然可视模式可能比 Vim 的普通模式操作更直观，但它有一个弱点：它与点命令的配合并不总是很好。我们可以通过在适当时候使用普通模式操作符来规避这个弱点。

假设我们想要转换以下链接列表使其变成大写：

<a href="#">one</a>

<a href="#">two</a>

<a href="#">three</a>

我们可以通过运行 vit 来选择标签的内部内容，这可以理解为：可视化选择标签的内部(visually select inside the tag)。it 命令是一种特殊的动作，称为文本对象，我们将在第 126 页的技巧 52 使用精确的文本对象追踪你的选择 中详细介绍。

使用可视模式操作符

在可视模式下，我们先做出选择，然后对其进行操作。在这种情况下，我们可以使用 U 命令，它可以将选中的字符转换为大写（:h v\_U）。参见第 46 页的表 4，可视模式下的大写转换。

在转换完第一行后，我们现在想要对接下来的两行执行相同的更改。我们试试使用点命令公式如何？

运行 j. 会将光标移动到下一行，然后重复最后的更改。在第二行上可以正常工作，但如果我们再次尝试，就会得到这个奇怪的结果。

<a href="#">ONE</a>

<a href="#">TWO</a>

<a href="#">THRee</a>

你看到了吗？当一个可视模式命令重复时，它会影响相同范围的文本（参见 :h visual-repeat）。在这种情况下，原始命令影响了由三个字母组成的单词。这对第二行来说没问题，因为它也包含一个三个字母的单词，但当我们尝试对包含五个字母的单词重复命令时，就不合适了。

使用普通操作符

Visual模式下的U命令在普通模式下有等效操作：gU{motion}（:h gU）。如果我们使用这个来进行第一次更改，就可以使用点公式完成后续编辑。

讨论

这两种技术都只需要四个按键：vitU与gUit，但底层语义完全不同。在Visual模式方法中，四个按键可以被视为两个独立命令：vit选择文本，U转换选择。相比之下，gUit可以被视为由操作符(gU)和动作(it)组成的单一命令。

如果我们希望设置dot命令重复有用的操作，那么最好避免使用Visual模式。作为一般规则，在处理重复性更改时，我们应该优先使用操作符命令而不是其Visual模式等价物。

这并不意味着Visual模式毫无用处。它仍然有其存在的意义。并非每个编辑任务都需要重复，对于一次性更改，Visual模式完全适用。尽管Vim的动作允许精确操作，但有时我们需要修改结构难以追踪的文本范围。在这些情况下，Visual模式正是最佳工具。

启用Visual模式

v键是我们进入Visual模式的入口。在普通模式下，我们可以单独按v键来启用基于字符的Visual模式。通过按V（使用Shift键）启用基于行的Visual模式，通过按<C-v>（使用Control键）启用基于块的Visual模式。

提示25：更改文本列

我们可以使用Visual-Block模式同时在多行文本中插入文本。

Visual-Block模式不仅在处理表格数据时有用。在处理代码时也经常能受益。例如，考虑这段（不太理想的）CSS：

li.one   a{ background-image: url('/images/sprite.png'); }

li.two   a{ background-image: url('/images/sprite.png'); }

li.three a{ background-image: url('/images/sprite.png'); }

假设sprite.png文件已从images/目录移动到components/目录。我们需要更改每行以引用文件的新位置。我们可以使用Visual-Block模式完成这个操作。

这个过程应该看起来很熟悉。我们首先定义要操作的选择，恰好是一个矩形Visual-Block。当我们按c键时，所有选定的文本都会消失，并且我们进入插入模式。

当我们在插入模式下输入"components"时，它只出现在最顶部的行。下面两行没有任何变化。只有当我们按<Esc>返回普通模式时，我们才能在那些行中看到输入的文本。(在visual studio code中使用vim插件不会这样，会三行同时改变。但是在终端用vim会这样)

在不规则的Visual Block后追加

Visual-Block模式非常适合操作代码中的矩形块，如行和列，但并不局限于矩形文本区域。

我们已经见过这段JavaScript代码：

var foo = 1

var bar = 'a'

var foobar = foo + bar

三行连续的代码，每行长度不同。我们想在每行末尾追加分号。在提示2"不要重复自己"中，我们使用点命令解决了这个问题，但同样可以使用Visual-Block模式：

进入Visual-Block模式后，通过按$扩展选择到每行末尾。乍看之下，由于每行长度不同，这可能会造成困难。但在这种情况下，Vim理解我们想扩展选择到所有选定行的末尾。这使我们摆脱了矩形的约束，创建了一个沿文本右边缘不规则的选择。

定义选择后，我们可以使用A命令在每行末尾追加（参见"Vim的'i'和'a'键约定"）。这会在选择的最顶部行进入插入模式。在插入模式期间，我们输入的内容只会出现在这一行，但一旦返回普通模式，更改就会传播到我们选择的其他行。(vscode使用vim插件时，会同时更改)

**Vim的"i"和"a"键约定**

Vim在从普通模式切换到插入模式时有几个约定。i和a命令都能实现这一点，分别将光标定位在当前字符的前面或后面。I和A命令行为类似，但它们将光标定位在当前行的开头或末尾。

Vim在从Visual-Block模式切换到插入模式时遵循类似的约定。I和A命令都能做到这一点，分别将光标放在选择的开头或末尾。那么i和a命令呢？它们在Visual模式下做什么？

在Visual和Operator-Pending模式下，i和a键遵循不同的约定：它们构成文本对象的第一部分。如果你在Visual-Block模式下选择了内容，按i键后发现没有进入插入模式，请尝试使用I键。

Command-Line Mode

Part IV

Registers

字符交换

我经常拼写错误。有时候，我会注意到自己经常拼错某个词，这时我可以通过练习来改正。然而，有些拼写错误则比较随机。我最常犯的打字错误是把两个字符的顺序搞错。Vim 让修正这种错误变得非常简单。

假设我们在输入本书标题时犯了这样的字符交换错误：

Practica lvim

F␣ Practica lvim

x Practicalvim

p Practical vim

这里我们把空格打得太早，但这个错误很容易纠正。F␣ 命令将光标移到我们想要交换的两个字符中的第一个字符（参见第 120 页的技巧 50，“按字符查找”）。x 命令会删除光标下的字符，并将它复制到未命名寄存器中。然后，p 命令将未命名寄存器中的内容粘贴到光标位置之后。将这两个命令组合起来，xp 就可以理解为“交换接下来的两个字符”。

vim的yiw是什么

在Vim中，yiw是一个由两部分组成的命令：

y：代表"yank"（复制）

iw：代表"inner word"（内部单词）

具体含义是：

复制光标所在的整个单词，不包括周围的空格

不移动光标位置

例子：

假设光标位于单词 "hello" 的任何位置，yiw会复制整个 "hello" 到未命名寄存器，可以随后用p粘贴。

相关命令：

diw：删除整个单词

ciw：删除整个单词并进入插入模式

Vim 的术语与其他软件的术语对比

“剪切(cut)”、“复制(copy)”和“粘贴(paste)”是全球通用的术语，这些操作在大多数桌面软件和操作系统中都有提供。Vim 也提供了这些功能，但使用了不同的术语：删除（delete）、 yank（拷贝）和 put（粘贴）。

put：Vim 的 put 命令实际上与粘贴操作完全相同。幸运的是，这两个词都以字母 p 开头，所以无论我们使用哪种术语，记忆命令键的助记符都是一样的。

yank：Vim 的 yank 命令相当于复制操作。历史上，c 命令已经被分配给更改操作，因此 Vi 的作者不得不想出一个替代名称。y 键是可用的，所以复制操作就成为了 yank 命令。

delete：Vim 的 delete 命令相当于标准的剪切操作。也就是说，它将指定的文本复制到寄存器中，然后从文档中删除。理解这一点是避免犯常见错误的关键，详见《Oops! I Clobbered My Yank》一节（第 147 页）。

你可能会想知道，Vim 中用什么命令来实现“真正的删除文本”——也就是如何从文档中删除文本而不将其复制到任何寄存器中。Vim 的解决方案是使用一个特殊的寄存器，称为 黑洞寄存器，从中删除的内容不会返回。黑洞寄存器由 \_ 符号表示（见 :h quote\_），因此，"\_d{motion} 就是执行真正删除的命令。

**寻址寄存器**

删除、复制和粘贴命令都与Vim的某个寄存器交互。我们可以通过在命令前加上 "{寄存器} 来指定要使用的寄存器。如果不指定寄存器，Vim将使用未命名寄存器。

例如，如果我们想将当前单词复制到寄存器a，可以运行 "ayiw。或者如果想将当前行剪切到寄存器b，可以运行 "bdd。然后可以通过输入 "ap 粘贴来自寄存器a的单词，或通过输入 "bp 粘贴来自寄存器b的行。

除了普通模式命令，Vim还提供用于删除、复制和粘贴操作的Ex命令。我们可以通过运行 :delete c 将当前行剪切到寄存器c，然后使用 :put c 命令将其粘贴到当前行下方。与普通模式命令相比，这些可能看起来很冗长，但在与其他Ex命令和Vim脚本结合使用时很有用。例如，第244页的技巧100，在寄存器中收集TODO项，展示了 :yank 如何与 :global 命令一起使用。

### 未命名寄存器 ("")

如果我们没有指定要操作的寄存器，Vim 会使用 **未命名寄存器**，该寄存器由 " 符号表示（参见 :h quote\_quote）。

要显式访问该寄存器，我们需要使用两个双引号。例如，""p，这实际上等价于单独使用 p 命令。

命令如 x、s、d{motion}、c{motion}、y{motion}（及其大写版本）都会设置未命名寄存器的内容。在每种情况下，我们可以使用 "{register} 前缀来指定其他寄存器，但未命名寄存器是默认的。如果我们不小心，很容易覆盖未命名寄存器的内容，这可能会引发问题。

请再次参考《Oops! I Clobbered My Yank》一节（第 147 页）。我们首先通过 yank 命令获取一些文本（单词 "collection"），打算在其他地方粘贴。然后，为了清理空间，我们删除了一些阻碍粘贴的文本，这导致未命名寄存器的内容被覆盖。当我们使用 p 命令时，返回的是我们刚刚删除的文本，而不是我们先前获取的文本。

Vim 的术语选择不太合适。x 和 d{motion} 命令通常被称为“删除”命令，但这其实是个误称。更好的说法是将它们视为“剪切”命令。未命名寄存器中往往不包含我预期找到的文本，幸运的是，接下来我们将了解的 **yank 寄存器** 更加可靠。

### yank 寄存器 (“0)

当我们使用 y{motion} 命令时，指定的文本不仅会被复制到未命名寄存器，还会被复制到 **yank 寄存器**，该寄存器由 0 符号表示（参见 :h quote0）。

顾名思义，yank 寄存器只有在我们使用 y{motion} 命令时才会被设置。换句话说：x、s、c{motion} 和 d{motion} 命令不会设置 yank 寄存器。如果我们 yank 了一些文本，我们可以确定它会一直保留在寄存器 0 中，直到我们明确地通过 yank 另一些文本来覆盖它。

yank 寄存器是可靠的，而未命名寄存器是易变的。

我们可以使用 yank 寄存器来解决《Oops! I Clobbered My Yank》一节中的问题（第 147 页）：

yiw collection = getCollection();//normal模式，光标在collection的字母上面

process(somethingInTheWay, target);

jww collection = getCollection();//移动光标到something上

process(somethingInTheWay, target);

diw collection = getCollection();

process(, target);

“0P collection = getCollection();//注意P是大写

process(collection, target);

diw 命令仍然会覆盖未命名寄存器，但它不会影响 yank 寄存器。我们可以通过按下 "0P 来安全地从 yank 寄存器粘贴，Vim 会给我们我们想要的文本。

vim的yiw是什么

在Vim中，yiw是一个由两部分组成的命令：

y：代表"yank"（复制）

iw：代表"inner word"（内部单词）

具体含义是：

复制光标所在的整个单词，不包括周围的空格

不移动光标位置

例子：

假设光标位于单词 "hello" 的任何位置，yiw会复制整个 "hello" 到未命名寄存器，可以随后用p粘贴。

相关命令：

diw：删除整个单词//其实是剪切整个单词

ciw：删除整个单词并进入插入模式

这是Vim中非常实用的文本编辑命令之一。

### 系统剪贴板寄存器 ("+ 和 "\*)

我们到目前为止讨论的所有寄存器都是 Vim 内部的寄存器。如果我们想将 Vim 内部的文本复制到外部程序中（或反之），则必须使用系统剪贴板。

Vim 的 **加号寄存器**（"+）引用了系统剪贴板，使用 + 符号来访问（参见 :h quote+）。

* 如果我们使用外部应用程序的剪切或复制命令捕获文本，我们可以通过 "+p 命令（或在插入模式下使用 <C-r>+）将其粘贴到 Vim 中。
* 反过来，如果我们在 Vim 的 yank 或 delete 命令前加上 "+，指定的文本将被复制到系统剪贴板中。这样，我们就可以轻松地将其粘贴到其他应用程序中。

X11 窗口系统有第二种剪贴板，称为 **主剪贴板**（primary clipboard）。它表示最近选择的文本，我们可以使用鼠标中键（如果有的话）从中粘贴。Vim 的 **星号寄存器**（"\*）映射到主剪贴板，并使用 \* 符号来访问（参见 :h quotestar）。

在 Windows 和 Mac OS X 系统中，没有主剪贴板，因此我们可以互换使用 "+ 和 "\* 寄存器：它们都代表系统剪贴板。

### 技巧 62

#### 用寄存器替换视觉模式的选区

在视觉模式下使用 Vim 的 p 命令时，它具有一些不寻常的特性。我们将在此技巧中探索如何利用这些特性。

当我们在视觉模式下使用 p 命令时，Vim 会用指定寄存器的内容替换当前的选区（参见 :h v\_p）。我们可以利用这个功能来解决《Oops! I Clobbered My Yank》一节中的问题（第 147 页）。

yiw collection = getCollection();

process(somethingInTheWay, target);

jww collection = getCollection();

process(somethingInTheWay, target);

ve collection = getCollection();

process(somethingInTheWay, target);

p collection = getCollection();

process(collection, target);

当我们在 Vim 的视觉模式下使用 p 命令时，它实际上做了两件事：既获取未命名寄存器的内容，又设置未命名寄存器的内容。

可以这样理解：文档中的视觉选区与寄存器中的文本交换位置。这是一个功能？还是一个 bug？你来决定！

vim的mm是什么

在 Vim 中，mm 是一个 **标记命令**，用于设置一个名为 **m** 的标记（mark）。具体来说，m 后面跟一个字母（如 a、b 等）表示创建一个标记，并将其应用到当前行的位置。

例如，ma 命令会在当前行的位置设置一个标记 a，然后你可以使用 `a (反引号加标记名)快速跳转到这个标记的位置。(`是键盘左上角，数字1左边的按键，不是单引号’)

相关命令：

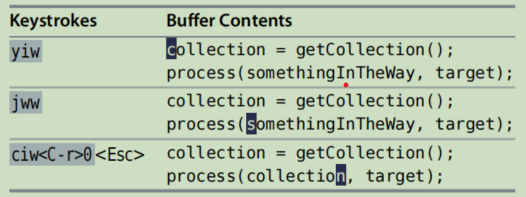
* 'm ：跳转到标记 'm' 所在行的行首//单引号
* `m：跳转到标记 'm' 的精确光标位置//反引号

collection = getCollection();

process(, target);//复制collection后，光标在(用p粘贴，光标在,用P粘贴。

我不喜欢在按字符区域的文本是否需要放在光标前面或后面的问题上思考。为了避免这个问题，我有时更喜欢在插入模式下使用 <C-r>{register} 映射来粘贴字符区域的文本，而不是使用普通模式的 p 和 P 命令。使用这种技巧时，寄存器中的文本总是插入到光标位置之前，就像我们在插入模式下输入文本一样。

在插入模式下，我们可以通过按 <C-r>" 来插入未命名寄存器的内容，或者按 <C-r>0 来插入 yank 寄存器的内容（有关更多细节，请参见第31页的提示 15：“在不离开插入模式的情况下从寄存器粘贴”）。我们可以使用这种方法来解决第147页“糟糕！我覆盖了我的 yank”中的问题。



ciw<C-r>0<Esc>是什么意思

ciw<C-r>0<Esc> 是一条在 Vim 中的命令组合，具体步骤如下：

1. ciw：删除当前光标所在的单词，并进入插入模式。这个命令会删除光标所在的整个单词，并准备插入新内容。
2. <C-r>0：在插入模式下，从 yank 寄存器（即最近一次复制的文本）中粘贴内容。0 是 yank 寄存器的标识符，表示你最近一次用 yy、dw 等命令复制的文本。
3. <Esc>：退出插入模式，返回到普通模式。

Macros

Vim 提供了多种重复更改的方法。我们已经学过了点命令（dot command），它对于重复小的更改非常有用。但是当我们想重复一些更大的更改时，我们应该使用 Vim 的宏。

使用宏，我们可以将任何一系列按键记录到一个寄存器中，然后将它们播放出来。

**录制和执行宏**

宏允许我们录制一系列更改，然后再回放它们。本提示展示了如何操作。

许多重复性任务涉及多个更改。如果我们想自动化这些任务，可以录制一个宏并执行它。

**通过录制宏捕获命令序列**

q 键既是“录制”按钮，也是“停止”按钮。要开始录制我们的按键，我们输入 q{register}，指定我们希望将宏保存到的寄存器地址。如果我们操作正确，状态行会显示“recording”。此后，我们执行的每个命令都会被捕获，直到我们再次按下 q 停止录制。

让我们来看一下实际操作：

按下 qa 开始录制，并将宏保存到寄存器 a。然后，我们对第一行进行两项更改：在末尾添加一个分号，并在开头加上单词 var。完成这两个更改后，我们按下 q 停止录制宏（参考 :h q 了解更多）。

我们可以通过输入以下命令来查看寄存器 a 的内容：

**通过执行宏回放命令序列**

@{register} 命令执行指定寄存器中的内容（参见 :h @）。我们还可以使用 @@，它会重复最近一次调用的宏。

通过搜索进行导航，使用文本对象，充分利用 Vim 的各种移动命令，使你的宏更加灵活和可重复。别忘了：录制宏时，禁止使用鼠标！

vim的r是什么意思

在 Vim 中，r 是一个命令，用来替换当前光标所在位置的字符。使用 r 命令时，可以指定一个字符，它会将光标所在位置的字符替换为指定的字符。

r{char}

其中，{char} 是你想替换成的字符。

### 示例：

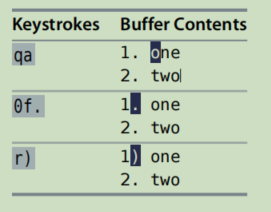
* ra：将光标当前位置的字符替换为字母 a。
* r1：将光标当前位置的字符替换为数字 1。

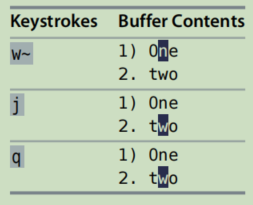
vim中的~是什么

### ~ 命令：

* ~ 是一个用于切换字符大小写的命令。它会将光标所在位置的字符从小写转换为大写，或者从大写转换为小写。

注意这个宏中使用的移动命令。我们从 0 命令开始，这个命令通过将光标放置在行首来规范化光标位置。这意味着我们接下来的每个移动命令都会从同一个位置开始，使得宏的执行更加可重复。





有人可能会觉得下一个命令 f. 是多余的，因为它仅仅是将光标向右移动一步，和 l 命令一样。为什么要使用两个按键而不是一个呢？

再次强调，这是为了可重复性。在我们的示例中，行号只有 1 到 4，但假设行号是两位数呢？

1. one

2. two

...

10. ten

11. eleven

在前九行，0l 会将光标移动到行的第二个字符（也就是句点）。但是从第十行开始，0l 就无法达到目标了，而 f. 命令在这些行上都能正常工作，甚至能够一直工作到三位数的行号。

使用 f. 移动命令还增加了一个安全防护。如果当前行没有 . 字符，f. 命令会报错并中止宏的执行。我们稍后会利用这个特性，因此请记住这个思路。

我们可以通过按 @a 执行我们刚才录制的宏。这会执行以下步骤：跳转到行中的第一个 . 字符，将其更改为 )，将下一个单词的首字母大写，然后最后跳到下一行。

我们可以通过执行 @a 三次来完成这个任务，但运行 3@a 更快捷：

现在让我们加入一个新的障碍。假设我们的文件包含注释：

1. one

2. two

// break up the monotony

3. three

4. four

现在看看，如果我们尝试在这个文件上重新播放相同的宏，会发生什么。

宏在第三行——即包含注释的那一行——停滞了。当执行 f. 命令时，它找不到 . 字符，宏就中止了。我们触发了安全机制，这也是好事。如果宏在这一行成功执行，它可能会做出一些不需要的更改。

但我们面临一个问题。我们要求 Vim 执行宏五次，而它在第三次重复时中止了。所以我们必须再次在接下来的行上执行宏，以完成任务。让我们看一个替代方法。

vim的Vjj是什么意思

在Vim中，"Vjj"是一个普通模式（Normal mode）下的命令序列：

"V"：进入可视行模式（Visual Line mode）

"jj"：向下移动两行

具体操作：

"V"：

激活可视行模式

选中当前整行

光标变成横线，表示整行选择

"jj"：

连续按两次"j"键

向下移动两行

同时扩大选择范围，包括当前行和下面两行

实际效果是：选中当前行和下面两行的全部内容。这个命令常用于快速选择多行文本，为后续的复制、删除或其他操作做准备。

例如，如果你想复制三行，可以使用"Vjjy"（选择三行并复制）。

V和v有什么区别

在Vim中，"V"和"v"都是进入可视模式（Visual mode），但有一些关键区别：

"v"（可视模式）：

按字符选择

可以精确选择部分文本

光标呈现小方块状

可以选择文本的一部分

非常适合选择不完整的行或部分单词

"V"（可视行模式）：

按行选择

整行整行地选择

光标呈现横线状

始终选择完整的行

光标所在的行会被完整选中

vim的G是什么

在Vim中，"G"（大写G）是一个快速跳转到文件末尾的命令。

具体操作：

在普通模式（Normal mode）下按"G"

光标会立即移动到文件的最后一行

如果文件很长，可以快速到达末尾

相关的导航命令：

"gg"：跳转到文件开头

"数字G"：跳转到特定行号

例如 "10G" 表示跳转到第10行

这是Vim中非常实用的快速导航命令，可以帮助你快速浏览和定位文件内容。

'<,'>是什么

'<,'> 是 Vim 中的一个特殊范围表示符，用于表示 **可视选择的文本**。具体来说：

* '< 表示 **可视选择的开始位置**。
* '> 表示 **可视选择的结束位置**。

因此，'<,'> 是指 **当前可视模式中选中的文本范围**。当你在 Vim 中进入可视模式（使用 v、V 或 Ctrl-V）并选中一部分文本后，'<,'> 就会自动代表该选中区域。这个范围可以在 Ex 命令中使用，比如 :m（移动）、:d（删除）、:y（复制）等命令。

例如：

* :'<,'>m$：这条命令会将当前可视选择的文本移动到文件的最后一行。
* :'<,'>d：删除当前可视选择的文本。

使用 '<,'> 使得这些操作可以直接作用于当前选中的文本，而不需要显式指定行号。

$和G的区别是什么

在Vim中，"$"和"G"都是移动命令，但它们有明显的区别：

"$"（美元符号）：

在当前行的末尾停止

光标移动到行的最后一个字符

仅在当前行有效

可以在任何模式下使用

"G"：

跳转到文件的最后一行

移动光标到整个文件的末尾

是全文导航命令

只在普通模式下有效

举例说明：

如果你在一个多行文本中

"$"会让光标移到当前行的行尾

"G"会让光标直接跳到文件的最后一行

实际应用：

"$"：适合行内编辑

"G"：适合快速定位到文件末尾

简单记忆：

"$" = 行末

"G" = 文件末

### 使用 :m 命令移动行

:move 命令与 :copy 命令类似（请参阅 :h :move）：

css

复制代码

:[range]move {address}

我们可以将其简化为单个字母：:m。假设我们想将“Hardware Store”部分移动到“Beauty Parlor”部分之后。我们可以使用 :move 命令来完成，如表格 8 所示（第 63 页，“使用 :m 命令移动一组行”）。



输入:后'<,'>会自动出来，只需要再输入m$就可以了。

$会移动到当前行的最后一个字符，那为什么m$将当前行（光标所在行）移动到文件的最后一行

在 Vim 中，$ 是一个 **地址标识符**，用于表示 **当前行的最后一个字符**。然而，:m$ 命令并不是把光标移动到行的最后一个字符，而是将 **当前行** 移动到 **文件的最后一行**。

这看起来可能有点混淆，但本质上，:m$ 中的 $ 并不是用来定位到当前行的最后一个字符，而是表示 **文件的末尾**。在 Vim 中，很多命令和操作中的 $ 都有类似的语法：它表示 **目标是文件的最后部分**，而不仅仅是当前行的最后字符。例如：

* :$：指文件的最后一行。
* :m$：将选中的行移动到文件的最后一行。

### 解释：

* :m 是移动命令。
* :$ 或 $ 在这里指文件的最后一行，而不是当前行的最后一个字符。

所以，:m$ 将 **当前行** 移动到 **文件的末尾**，而不是当前行的最后一个字符。

### 运行普通模式命令覆盖一个范围

如果我们想要在一系列连续的行上运行一个普通模式命令，可以使用 :normal 命令。当与点命令（.）或宏结合使用时，我们可以非常轻松地执行重复任务。

考虑我们在技巧 2 “不要重复自己” 中遇到的示例。我们希望在一系列行的末尾添加分号。使用点公式可以快速完成任务，但在那个例子中我们只需要对三行进行更改。如果我们必须做同样的更改五十次怎么办？使用点公式，我们需要按 j. 五十次，总共一百次按键！

有一个更好的方法。为了演示，我们将在这个文件的每一行末尾添加分号。为了节省空间，我只包含了五行，但如果你能想象这里有五十行，那么这个技巧将显得更加强大：

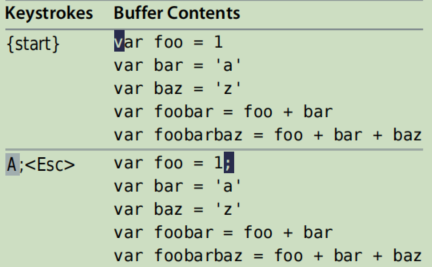
var foo = 1

var bar = 'a'

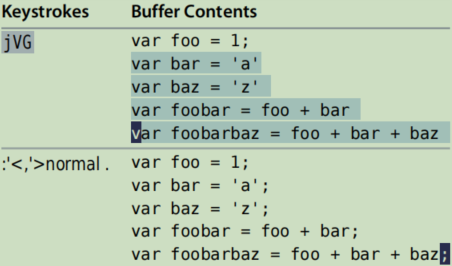
var baz = 'z'

var foobar = foo + bar

var foobarbaz = foo + bar + baz



我们希望避免在每一行逐个执行 . 命令。相反，我们可以使用 :normal Ex 命令在一系列行上执行点命令：



:'<,'>normal .

这条命令的意思是：“对于可视选择中的每一行，执行普通模式的 . 命令。”这个技巧无论是在五行还是五十行上操作都一样有效。它的真正美妙之处在于我们甚至不需要数行数，只需在可视模式下选择它们。

在这个例子中，我们使用 :normal 来执行点命令，但我们也可以用同样的方法执行任何普通模式命令。例如，我们可以用这个单一命令解决上面的问题：

:%normal A;

% 符号用作代表整个文件的范围。所以 :%normal A; 指示 Vim 在文件的每一行末尾添加分号。这个更改涉及切换到插入模式，但 Vim 随后会自动返回到普通模式。

在 “Ex 命令影响广泛” 中，我们提到 Ex 命令可以同时更改多行。:normal 命令允许我们将 Vim 普通模式命令的表达性与 Ex 命令的范围结合起来。这是一个强大的组合！

### 提示 31：重复上一个 Ex 命令

虽然可以使用 . 命令来重复我们最近的普通模式命令，但如果我们想要重复上一个 Ex 命令，则必须使用 @:。了解如何撤销最后一个命令总是很有用，所以我们也会在讨论中考虑这一点。

在第 1 章《Vim 的方式》中，我们看到 . 命令可以用来重复最后一次的更改。但是 . 命令不会重放从 Vim 的命令行进行的更改。相反，我们可以通过按 @: 来重复最后一个 Ex 命令（参见 :h @:）。

### 提示 32：Tab 补全你的 Ex 命令

就像在 shell 中一样，我们可以使用 <Tab> 键在提示符下自动补全命令。

Vim 在选择用于 Tab 补全的建议时非常智能。它会查看命令行上已输入的内容，并构建一个适合的建议列表。例如，我们可以输入：

:col<C-d>

会显示：

colder colorscheme

<C-d> 命令会让 Vim 显示可能的补全列表（参见 :h c\_CTRL-D）。如果我们按 <Tab> 键，提示符会在 colder、colorscheme 和原始的 col 之间循环。我们可以通过按 <S-Tab> 键向后滚动建议。//vim里面才有，vscode里的vim插件里面没有，但是可以按Tab滚动。

vim怎么选择全文

在 Vim 中选择全文非常简单。你可以按下以下步骤：

进入正常模式：先确保你在正常模式下，可以按下 Esc 键来进入。

选择模式：按下 V 键，进入可视模式。

选择全文：然后按下 G 键，这会将光标移动到文件末尾，从而选择整个文档。

这样，你就选择了整个文档的内容。如果你有其他问题或需要进一步的帮助，请告诉我！

这个方法只能让当前行和最后一行被选中，怎么把前面的也选中呢

好的，我们可以使用另一种方法来选择全文。你可以使用以下步骤：

进入正常模式：按 Esc 确保你在正常模式下。

进入可视行模式：按 ggVG：

gg 移动光标到文件的第一行。

V 进入可视行模式。

G 选择直到文件末尾。

这样可以确保整个文件的所有行都被选中。

希望这对你有帮助！如果你有其他问题或需要进一步的帮助，请告诉我。