1. echom

Using :echom will save the output and let you run :messages to view it later.

2. Toggling Boolean Options

```
:set number!
```

3. Checking Options

```
:set number?
```

Tips: viw “ select a word

4. Strict Mapping

```
:nmap dd O<esc>jddk    “ recurtion
```

Nonrecursive Mapping (always use this way to mapping)

```
:nmap x dd
:nnoremap \ x
```

Tips: <nop> in maps, :setlocal and :map <buffer>

5. Autocommands

```
:autocmd BufNewFile *.txt :write
```

```
:autocmd BufWritePre *.html :normal gg=G
```

```
:autocmd BufWritePre,BufRead *.html :normal gg=G
```

```
:autocmd BufNewFile,BufRead *.html setlocal nowrap
```

```
:autocmd FileType javascript nnoremap <buffer> <localleader>c I//<esc>
:autocmd FileType python     nnoremap <buffer> <localleader>c I#<esc>
```

```
:autocmd FileType python     :iabbrev <buffer> iff if:<left>
```

```
:autocmd FileType javascript :iabbrev <buffer> iff if ()<left>
```

see :help <left>

list of events:
:help autocmd-events
or
http://vimdoc.sourceforge.net/htmldoc/autocmd.html#{event}

vim 在处理 autocmd 命令时会有一个问题：如果重复加载两条相同的 autocmd 命令，vim 会执行这条命令两次，这在 source .vimrc 时会带来性能上的问题．解决办法如下：

```
augroup filetype_html
    autocmd!
    autocmd FileType html nnoremap <buffer> <localleader>f Vatzf
augroup END
```

使用 augroup，并用 autocmd!清除之前的 autocmd 命令．

Related links:

http://learnvimscriptthehardway.stevelosh.com/chapters/12.html

http://learnvimscriptthehardway.stevelosh.com/chapters/14.html

6. Operator-Pending Mappings

非常有意思的一章，以前忽视了这个用法，很有意思，先查阅 :help text-objects.

```
:onoremap p i(
```

这样执行 dp 命令就相当于执行 di(.

```
:onoremap b /return<cr>
```

非常强大，调用搜索命令，移动到 return 这里．

Tips:
f(移动到下一个 (处，也可以把 (换成其他字符，就是移动到下一个该字符处
F) 移动 到上一个) 处

```
:onoremap in( :<c-u>normal! f(vi(<cr>
```

查看:help omap-info

考虑下面这个命令映射:

```
:onoremap ih :<c-u>execute "normal! ?^==\\+$\r:nohlsearch\rkvg_"<cr>
```

关于:normal 命令:

```
:normal! gg “ 这个命令会执行 normal 模式下的 gg 的命令，也就是 到文件第一行
```

加 ! 表示不使用 mapping.

normal 无法识别特殊字符 <cr>，所以 :normal! gg/a<cr> 只会到第一行，不会执行搜索，解决办法是使用下面的 execute.

关于 execute 命令:

```
execute “write” 这个命令就相当于 :write<cr>
```

execute 会将后面字符串中的特殊字符在执行前替换掉，\\ 变为 \，\r 变为 <cr>. 所以上面的字符串被替换为:

```
normal! ?^==\+$<cr>:nohlsearch<cr>kvg_
```

`?^==\+$:` 向上搜索“开头为==，且有 2 个以上=字符的只含=字符的行，^ 行首，$ 行尾”

kvg_:
g_: 移动到该行的最后一个非空字符，用$的话会把行尾的换行符也选中

```
:onoremap ah :<c-u>execute "normal! ?^==\\+$\r:nohlsearch\rg_vk0"<cr>
```

0: 移动到该行第一个字符，^ 是第一个非空字符

总结: visual selection 就是 onoremap 的操作区域. 若没有 visual selection 那么 cursor 的移动区域就是操作区域.

Tips: vim 的 / ? 搜索命令可以用正则式，查看 :help pattern-overview 有更多非常有用的匹配规则.

关于 execute 中使用特殊字符查看 :help expr-quote

B 移动到上个字符串开头
E 移动到下个字符串结尾

下面是电子邮件的 inside text object:

```
onoremap in@ :<c-u>execute "normal! /[0-9a-zA-Z_-]\\+@[0-9a-zA-Z]\\
+\\.\r:nohlsearch\rvE"<cr>
```

7. Variables

```
:let foo = "bar"
:echo foo
```

选项作为变量, 前面加&:

```
:set textwidth=80
:echo &textwidth

:set wrap
:echo &wrap

:let &textwidth = &textwidth + 10
:set textwidth?

“ set the local value of an option as a variable, instead of the global value
:let &l:number = 1

“ 寄存器作变量, @a 表示寄存器 a
:let @a = "hello!"
:echo @a

“ 寄存器 / 中保存上次 /pattern 搜索的 pattern 字符串
:echo @/

:help registers “ 查看寄存器
```

Variable Scoping

```
:let b:hello = "world"
```

b:hello 表示该变量的作用域是 buffer.

查看 :help internal-variables 还有更多的变量作用域.

Tips: 用 | 分隔命令

```
:echom "foo" | echom "bar"
```

相当于单独执行两个 echom 命令.

8. Conditionals

```
if 0
   echom “if”
elseif “nope!”
   echom “else if”
else
   echom “else”
endif
```

在 if 语句中出现字符串的，会把字符串转为整数，转换规则为“some”转为 0，“10some”为 10，“some10”为 0.

if 中可以用比较表达式，>，<，,>=，<=，==，字符串之间也可以进行比较，对大小写的敏感依赖于用户设置.

```
:set noignorecase
:if "foo" == "bar"
:    echom "one"
:elseif "foo" == "foo"
:    echom "two"
:endif
```

```
:set ignorecase
:if "foo" == "FOO"
:    echom "no, it couldn't be"
:elseif "foo" == "foo"
:    echom "this must be the one"
:endif
```

因为 == 的这个特性，一定不要在 vimscripts 中使用 ==，就像不用 map 用 noremap 一样.

记住：无论何时使用比较符，一定要用显示的 #，？版本，对于整数比较也是，方便以后改为字符串.

==? 不区分大小写
==# 区分大小写

```
:set noignorecase
:if "foo" ==? "FOO"
:    echom "first"
:elseif "foo" ==? "foo"
:    echom "second"
:endif
```

查看 :help expr4 还有更多的比较符，注意 =~ 是关于正则式的匹配. 而且每个版本都有 ?，#版本，如>?，<#，!=?.

9. Functions

vim 的函数有一个很奇怪的地方：函数名一定要是以大写字母开头.

```
:function Meow()

:  echom "Meow!"

:endfunction
```

```
:call Meow()
```

也可以有返回值

```
:function GetMeow()
:  return "Meow String!"
:endfunction
```

```
:echom GetMeow()
```

若不显示写成 return，则函数默认返回 0

```
:echom Meow()
```

Function Arguments

```
:function DisplayName(name)
:  echom "Hello!  My name is:"
:  echom a:name
:endfunction
```

必须要加 a: 作用域，表示函数参数，否则会提示找不到变量 name.

Varargs

变长参数

```
:function Varg(...)
:  echom a:0
:  echom a:1
:  echo a:000
:endfunction

:call Varg("a", "b")
```

a:0 参数个数
a:1 第一个参数
a:000 一个 list，包含所有参数，格式为[var1, var2]，list 不能用 echom，所以用了 echo

```
:function Varg2(foo, ...)
:  echom a:foo
:  echom a:0
:  echom a:1
:  echo a:000
:endfunction

:call Varg2("a", "b", "c")
```

Assignment

```
:function Assign(foo)
:  let a:foo = "Nope"   " Error
:  echom a:foo
:endfunction

:call Assign("test")
```

会出错，a:foo 是只读变量，不能赋值.

```
:function AssignGood(foo)
:  let foo_tmp = a:foo
:  let foo_tmp = "Yep"
:  echom foo_tmp
:endfunction

:call AssignGood("test")
```

此时 foo_tmp 是临时变量，只在函数内有效，若换成 g:foo_tmp 则是全局变量，函数外也有效.

10. Numbers

vim 中有整数和浮点数. 整数为 32bits，有 16 进制和 8 进制形式，0xff 为 16 进制，015 为 8 进制.

浮点数也有多种表示形式，100.1，5.45e+3，15.2e-4，15,3e9，4e10 等.

整数与浮点数运算类似 C 语言，1 + 2.0 结果为浮点数，3 / 2 结果为 1，3 / 2.0 结果为 1.5.

```
:help Float
:help floating-point-precision
```

11. strings

不像 python，vimscript 中的 strings 不能用 + 连接，“hello ” + “world” 会变成 0，“10ss”

+ “20aa” 为 30. “10.5” + “10” 是 20，不支持浮点数.

Vimscript 中用 . 作为字符串连接符.

```
“hello ” . “world”  生成 “hello world”,
```

```
10 . “foo” 生成 10foo
```

```
10.1 . “foo”  Error
```

转义字符 \

```
echo “foo\nbar” 正常换行
```

```
echom “foo\nbar” 换行符变为 ^@
```

使用单引号，保留原有字符串，不转义

```
echom 'foo\nbar'
```

有一个例外，单引号中的两个连续单引号生成一个单引号.

```
:echom 'That''s enough.'
```

That's enough.

:help expr-quote

12. String Functions

vimscript 有很多内置函数处理字符串，

```
:echom strlen("foo")
```

```
:echom len("foo")
```

分割

```
:echo split("one two three")
```

指定分隔符

```
:echo split("one,two,three", ",")
```

返回的是一个 list，['one', 'two', 'three']

连接

```
:echo join(["foo", "bar"], "…")
```

foo...bar

Lower and Upper Case

```
:echom tolower("Foo")
:echom toupper("Foo")
```

:help split() 可以发现，分隔符可以用正则式来匹配，非常强大
:help functions 查看内置函数

13. Execute

Tips:
:help <cword>
<cWORD> 扩展字符串

使用<cWORD>时用注意

```
:nnoremap <leader>g :grep -R <cWORD> .<cr>
```

若光标下为 foo;ls，那么会执行 ls 命令.

加上单引号

```
:nnoremap <leader>g :grep -R '<cWORD>' .<cr>
```

但此时对 that's 又失效了.

解决办法是使用 shellescape() 函数:

```
:nnoremap <leader>g :execute "grep -R " . shellescape("<cWORD>") . " ."<cr>
```

但还是不行，shellescape("<cWORD>") 会返回 '<cWORD>'. 应该先使用 expand():

```
:nnoremap <leader>g :exe "grep -R " . shellescape(expand("<cWORD>")) . " ."<cr>
```

Tips:

```
vnoremap <leader>g :<c-u>call GrepOperator(visualmode())<cr>

function! GrepOperator(type)
```

```
    echom "Test"
endfunction
```

在 visual 模式下按 :, 会自动插入 `<, `>, 这表示 visual selection 的范围, <c-u>的作用是删除光标到开头的所有内容.

Tips:

当 yank 或 delete 不指明寄存器时, yank 和 delete 的内容都被放到 unnamed @ 寄存器中了.

```
nnoremap <leader>g :set operatorfunc=<SID>GrepOperator<cr>g@
vnoremap <leader>g :<c-u>call <SID>GrepOperator(visualmode())<cr>

function! s:GrepOperator(type)
    let saved_unnamed_register = @@

    if a:type ==# 'v'
        normal! `<v`>y
    elseif a:type ==# 'char'
        normal! `[v`]y
    else
        return
    endif

    silent execute "grep! -R " . shellescape(@@) . " ."copen

    let @@ = saved_unnamed_register
endfunction
```

红色部分是恢复使用的寄存器.

使用 s: 使函数作用域只在本 script 中, <SID>是使得在 map 中可以找到对应的函数, :help <SID>

:help map-operator

`[ 是 motion 的起始字符位置, `]是终止字符位置. visual mode 下不属于 motion.

`<v`>可以换成 gv, gv 表示上一次的 visual selection 区域.

14. list

```
:echo ['foo', 3, 'bar']
```

```
:echo ['foo', [3, 'bar']]
```

index

```
:echo [0, [1, 2]][1]
:echo [0, [1, 2]][-2]
```

slicing

```
:echo ['a', 'b', 'c', 'd', 'e'][0:2]
:echo ['a', 'b', 'c', 'd', 'e'][-2:-1]

:echo ['a', 'b', 'c', 'd', 'e'][:1]
:echo ['a', 'b', 'c', 'd', 'e'][3:]
```

对字符串也有类似的操作

```
:echo "abcd"[0:2]
```

连接

```
:echo ['a', 'b'] + ['c']
```

List Functions

```
:let foo = ['a']
:call add(foo, 'b')
:echo foo
```

```
:echo get(foo, 0, 'default')
:echo get(foo, 100, 'default')
```

```
:echo index(foo, 'b')
:echo index(foo, 'nope')
```

```
:echo join(foo)
:echo join(foo, '---')
:echo join([1, 2, 3], '')
```

```
:call reverse(foo)
:echo foo
:call reverse(foo)
:echo foo
```

```
:help List
```

Tips:

:match Keywork /\clist/

15. loops

for

```
:let c = 0

:for i in [1, 2, 3, 4]
:  let c += i
:endfor

:echom c
```

while

```
:let c = 1
:let total = 0

:while c <= 4
:  let total += c
:  let c += 1
:endwhile

:echom total
```

16. dictionaries

```
:echo {'a': 1, 100: 'foo'}

:echo {'a': 1, 100: 'foo',}
```

keys 都会被转成字符串，这里的 100 会转成'100'，结尾的 ，最好加上，避免以后添加条目时出错.

Index，两种方式

```
:echo {'a': 1, 100: 'foo',}['a']
:echo {'a': 1, 100: 'foo',}[100]
```

```
:echo {'a': 1, 100: 'foo',}.a
:echo {'a': 1, 100: 'foo',}.100
```

赋值，添加条目

```
:let foo = {'a': 1}
:let foo.a = 100
:let foo.b = 200
:echo foo
```

```
foo = {'a': 100, 'b': 200}
```

删除条目

```
:let test = remove(foo, 'a')
:unlet foo.b
:echo foo
:echo test
```

remove 会返回删除 key 的 value. unlet 直接删除条目.

有很多函数操作 dict, 如 get, has_key, items, keys, values.

Tips: foldcolumn
winnr()
wincmd

17. functional programming

deepcopy() 可以用来模拟 Immutable Data Structures.

```
function! Reversed(l)
    let new_list = deepcopy(a:l)
    call reverse(new_list)
    return new_list
endfunction

function! Append(l, val)
    let new_list = deepcopy(a:l)
    call add(new_list, a:val)
    return new_list
endfunction

function! Assoc(l, i, val)
    let new_list = deepcopy(a:l)
    let new_list[a:i] = a:val
    return new_list
endfunction

function! Pop(l, i)
    let new_list = deepcopy(a:l)
```

```
    call remove(new_list, a:i)
    return new_list
endfunction
```

函数作为变量

```
:let Myfunc = function("Append")
:echo Myfunc([1, 2], 3)
```

Append 就是函数名，myfunc 实际上 Append 函数.

函数 list

```
:let funcs = [function("Append"), function("Pop")]
:echo funcs[1](['a', 'b', 'c'], 1)
```

Higher-Order Functions

```
function! Mapped(fn, l)
    let new_list = deepcopy(a:l)
    call map(new_list, string(a:fn) . '(v:val)')
    return new_list
endfunction
```

查看 :help map(), :help v:, v:var 中保存 item

```
function! Filtered(fn, l)
    let new_list = deepcopy(a:l)
    call filter(new_list, string(a:fn) . '(v:val)')
    return new_list
endfunction
```

查看 :help filter()

```
function! Removed(fn, l)
    let new_list = deepcopy(a:l)
    call filter(new_list, '!' . string(a:fn) . '(v:val)')
    return new_list
endfunction
```

Tips:

:help type()

18. Paths

```
:echom expand('%')
:echom expand('%:p')
:echom fnamemodify('foo.txt', ':p')

:help fnamemodify()
```

Listing Files

```
:echo globpath('.', '*')

:echo split(globpath('.', '*'), '\n')

:echo split(globpath('.', '*.txt'), '\n')

:echo split(globpath('.', '**'), '\n')
```

** 表示递归匹配

```
:help filename-modifiers
:help wildcards
```

Tips: 关于 folding，看 vim-potion 那个例子.

```
:help line-continuatio

:help search()

:help ordinary-atom
```

19. autoload 机制

在需要的时候加载函数. 函数的定义有的特别，需要用 # 符号.

```
:call somefile#Hello()

function somefile#Hello()
    " ...
endfunction
```

若调用时 somefile#Hello 函数存在，则直接用，否则在 $RUNTIMEPATH/ 中寻找 autoload/somefile.vim 文件，并 source 加载该文件，然后再调用函数.

可以用多个 # 表示子目录.

```
:call myplugin#somefile#Hello()

function myplugin#somefile#Hello()
    " ...
endfunction
```

注意在相应文件中定义函数时也是要用 # 的形式. 不能只写最后的函数名.