



4 Git

操作系统 计算机网络 计算机硬件 数据库 编译原理
数据结构与算法 程序设计 软件工程 编程语言

都不是今天要学的东西.....

Vim 文本编辑

```
1 OS default
+
""" Map leader to space -----
let mapleader=" "

""" Plugins -----
set surround
set multiple-cursors
set commentary
set argtextobj
set easymotion
set textobj-entire
set ReplaceWithRegister
set keep-english-in-normal-and-restore-in-insert

""" Plugin settings -----
let g:argtextobj_pairs="[:],(:),<:;>"

""" Common settings -----
set showmode
set so=5
set incsearch
set nu

" Show a few lines of context around the cursor. Note that this makes the
" text scroll if you mouse-click near the start or end of the window.
set scrolloff=5

""" Idea specific settings -----
set ideajoin
set ideastatusicon=gray
set idearefactormode=keep

""" Handler settings -----
".ideavimrc" [noeol][dos] 53L, 1424B
```


4 Git

- Shell 是用户与操作系统内核之间的接口，通过 Shell 可以运行命令、启动程序、管理文件系统等
- Shell 既可以作为交互式解释器（处理用户输入的命令），也可以用于执行脚本文件（自动化任务）
- 常见的 Shell 类型：
 - Bourne Shell (sh)
 - Bourne Again Shell (bash)
 - Z Shell (zsh)
 - C Shell (csh)
- Shell 有时和“终端”混用，但实际有区别

Linux

- 最常用的 Shell 是 bash, 默认安装在大多数 Linux 发行版中
- 其他 Shell 包括 zsh 和 fish

```
1 pwd # 显示当前目录
2 ls -l # 列出文件
```

不同平台下的 Shell

Windows

- 传统命令行工具是 cmd，现代 Windows 推荐使用 PowerShell 和 Windows Terminal
- Windows 10 及以上版本支持 WSL (Windows Subsystem for Linux)，可以运行 Linux 命令

```
1 # 在 PowerShell 中列出文件
2 Get-ChildItem
```

macOS

- macOS 基于 Unix, 默认使用 zsh, 之前的版本使用 bash
- 支持大多数 Unix/Linux 命令, 可以通过 Homebrew 安装额外工具

```
1 # 安装软件包
2 brew install wget
```

环境配置

基于类 Unix 命令行进行讲解

- Linux 和 macOS：均为类 Unix 系统，支持 POSIX 规范的命令行
- Windows：需要安装 MSYS2 开发环境

<https://www.msys2.org>

单个命令

最简单的结构，仅由一个命令组成

- ls (list)
- pwd (print working directory)
- whoami (Who am I?)

可以直接运行，不需要额外参数

参数对象命令

命令 + [参数] + [操作对象] + [...] (方括号表示可选)

参数用于改变命令的行为，通常以短格式（如 -l）或长格式（如 -long）表示

```
ls -l
```

```
grep --ignore -case "error" fake.log
```

```
cp file1.txt file2.txt
```

很多短格式参数是长格式参数的缩写，等价于后者

```
1 ls -l
2 ls -list # 等价长格式参数
```

```
1 grep --ignore-case "error" fake.log
2 grep -i "error" fake.log # 等价短格式参数
```


类 Unix 文件权限系统

在类 Unix 系统（笼统来说就是 Unix 和 Linux）中，每个文件和目录都有一组权限，决定用户对文件或目录的操作方式。权限分为三类：

- **所有者 (User)**：文件或目录的拥有者
- **群组 (Group)**：与文件所有者同属一个群组的用户
- **其他人 (Other)**：所有其他用户

类 Unix 用户和用户组



文件类型

共七类，其中最常见的三类为：

- **普通文件** (regular file) 用 - 表示
- **目录** (directory) 用 d 表示
- **符号链接** (symlink, symbolic link) 用 l 表示

此外还有块设备 (b 驱动器)、字符设备 (c 输入输出设备)、命名管道 (p 进程单向通信) 和套接字 (s 网络通信)

权限符号解释

权限由十个字符表示，例如：-rw-r-r-。第一个字符表示文件类型，其后的九个字符分为三组，依次为：

- r：读权限（read）
- w：写权限（write）
- x：执行权限（execute）

三组分别代表**所有者**、**群组**、**其他人的权限**

Example

-rw-r-- 的含义如下：

- -: 文件类型 (- 表示普通文件，d 表示目录)
- rw-: 所有者有读写权限，但没有执行权限
- r-: 群组只有读权限
- r-: 其他人也只有读权限

可以通过 `chmod` 命令进行修改

常用权限命令

- `ls -l`: 列出文件的详细信息，包括权限
- `chmod`: 改变文件或目录的权限
- `chown`: 改变文件的所有者和群组

```
1 chmod u+x filename # 为文件所有者添加执行权限
2 chmod g-w filename # 为文件删除群组写权限
3 chmod a+x filename # 为所有用户添加执行权限
4 chmod u=rw,go=r filename # 将所有者的权限设置为读、
   写，组和其他用户的权限设置为只读
```

八进制权限

每个权限都对应一个数值：

- 4：读 r，即 2^2
- 2：写 w，即 2^1
- 1：执行 x，即 2^0

```
chmod 755 filename # 代表什么？
```

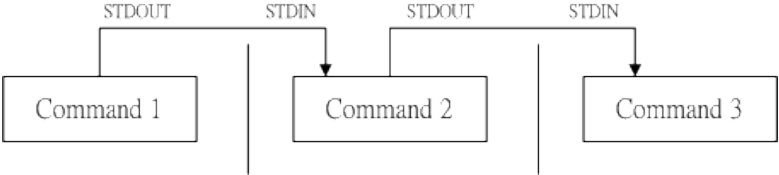
管道命令

使用管道符 | 将一个命令的输出作为下一个命令的输入

```
1 cat file.txt | grep "pattern"
2 cat file.txt | grep "pattern" | head -n 10
```

管道命令用于将多个命令组合在一起，形成复杂操作

管道命令



后台执行命令

在命令后加上 `&`，可以将命令放到后台执行

```
ping -c 4 google.com &
```

```
sleep 8 &
```

后台执行允许命令运行时继续进行其他操作

重定向命令

重定向命令将命令的输出重定向到文件，或者将文件作为命令的输入

```
1 echo "Hello" > output.txt # 输出重定向
2 sort < input.txt          # 输入重定向
```

> 覆盖写入文件，» 追加写入文件

有没有 « ？

文件结束符 EOF

EOF (End Of File) 在计算机中是表示文件或数据流结束的标志

在不同数据流中, EOF 的含义和处理略有不同

- 文件操作: EOF 标志表示文件的结束
- 命令行输入: Ctrl+D or Ctrl+Z

```
1 cat << EOF
2 This is a line of text.
3 This is another line of text.
4 EOF
```

此处的 EOF 可以用其他字符代替

复杂重定向

假设 nonexistent_dir 不存在：

```
{ ls -l /nonexistent_dir 2>&1 && echo "Directory  
listed successfully" ; } >output.log 2>error.log
```

???

标准输入输出

在命令行环境中，通常有三种标准输入输出流，它们通过文件描述符来表示：

- **标准输入 (stdin)**: 对应文件描述符 0，用于接收来自键盘或其他输入设备的数据
- **标准输出 (stdout)**: 对应文件描述符 1，用于输出正常的命令结果，默认显示在终端中
- **标准错误 (stderr)**: 对应文件描述符 2，用于输出错误消息，也显示在终端中，但与标准输出分开

```
command > output.txt 2> error.txt < input.txt
```

- `> output.txt` 将标准输出重定向到 `output.txt` 文件
- `2> error.txt` 将标准错误输出重定向到 `error.txt` 文件
- `< input.txt` 将标准输入从 `input.txt` 文件中读取数据

逻辑运算符 && 和 ||

- &&: 逻辑与 (AND), 表示前一个命令成功时才执行下一个命令。
- ||: 逻辑或 (OR), 表示前一个命令失败时才执行下一个命令

```
1 # 只有当第一个命令成功时, 才会执行第二个命令
2 command1 && command2
3
4 # 如果第一个命令失败, 则执行第二个命令
5 command1 || command2
```

&& 和 || 均采用断路判断

其他逻辑运算符

- `;`: 无论前一个命令是否成功, 都会执行后续命令
- `!`: 逻辑非, 反转命令的退出状态
- `()`: 在子 shell 中执行一组命令
- `{}`: 在当前 shell 中执行一组命令

```
1 # command1 和 command2 无论成功与否都会依次执行
2 command1; command2
3
4 # 如果 command 失败, 则返回成功的退出状态; 反之亦然
5 ! command
6
7 # 在子 shell 中执行 command1 和 command2
8 (command1 && command2)
9
10 # 在当前 shell 中执行 command1 和 command2
11 { command1; command2; }
```


注意!!!

区分 & 和 &&; | 和 ||

```
1 # command1 后台运行, command 2 前台执行
2 command1 & command2
```

```
1 # 管道操作: 将 command1 的输出作为 command2 的输入
2 command1 | command2
```

回过头来

假设 nonexistent_dir 不存在：

```
{ ls -l /nonexistent_dir 2>&1 && echo "Directory  
listed successfully" ; } >output.log 2>error.log
```

!!!

SSH 远程主机

SSH 连接的基本命令

```
1 # 通过 SSH 连接到远程主机
2 ssh username@remote_host
3
4 # 连接时指定自定义端口
5 ssh -p 2222 username@remote_host
6
7 # 使用指定的私钥文件连接 (i->identity)
8 ssh -i /path/to/private_key username@remote_host
```

端口转发

本地转发 (local forwarding)

- 创建一个本地端口，将发往该端口的所有通信都通过 SSH 服务器，转发到指定的远程服务器的端口
- -C：启用数据压缩
- -N：不执行远程命令，仅创建隧道，不打开远程 shell
- -g：允许远程主机连接到本地转发的端口
- -L lPort:127.0.0.1:rPort：设置本地端口转发，将本地 lPort 端口映射到远程服务器上的 rPort 端口

```
1 # 连接时指定自定义端口
2 ssh -CNg -L lPort:127.0.0.1:rPort user@remote -p
   sPort
```

为什么要这样干？

POSIX 可移植操作系统接口

- Portable Operating System Interface of UNIX
- 1974 年，贝尔实验室正式对外发布 Unix
- Unix-like OS: BSD, Sun Solaris
- 20 世纪 80 年代中期，Unix 厂商试图通过加入新的、不兼容的特性促进商业竞争
- IEEE & ISO

KISS

Unix 设计哲学：Keep It Simple, Stupid

命令行程序设计

如何设计命令行程序？

```
./test trash -n name [-option...] [parameter...]
```

C++ 实现

```
1  void executeCommand(const string& command, const vector<
string>& options, const vector<string>& parameters);
2
3  int main(int argc, char* argv[]) {
4      if (argc < 2)
5          return 1;
6
7      string command = argv[1];
8      vector<string> options;
9      vector<string> parameters;
10
11     for (int i = 2; i < argc; ++i) {
12         string arg = argv[i];
13         if (arg[0] == '-')
14             options.push_back(arg); // It's an option
15         else
16             parameters.push_back(arg); // It's a parameter
17     }
18
19     executeCommand(command, options, parameters);
20     return 0;
21 }
```


Python 实现

```
1 def execute_command(command, options, parameters):
2     # Logic of processing command, options, and parameters
3     pass
4
5 if __name__ == "__main__":
6     import sys
7     if len(sys.argv) < 2:
8         sys.exit(1)
9
10    command = sys.argv[1]
11    options = []
12    parameters = []
13
14    for arg in sys.argv[2:]:
15        if arg.startswith('-'):
16            options.append(arg) # It's an option
17        else:
18            parameters.append(arg) # It's a parameter
19
20    execute_command(command, options, parameters)
```

4 Git

赛博战争

无聊的：浏览器大战、操作系统之争、编程语言之争、代码缩进风格之战...

-> 编辑器之战

GUI: VSCode

CMD: Vim v.s. Emacs

编辑器的选择其实通常只是个人问题，但 Vim 通常是无可避免的选择，因为它与命令行高度集成

模式

Vim 的设计以大多数时间都花在阅读、浏览和进行少量编辑改动为基础，因此它具有多种操作模式：

- 正常模式 (NORMAL): 在文件中四处移动光标进行修改
- 插入模式 (INSERT): 插入文本
- 替换模式 (REPLACE): 替换文本
- 可视模式 (VISUAL): 选中文本块
- 命令模式 (:): 用于执行命令

模式快捷键

- Escape: 正常模式
- i: 插入模式
- R: 替换模式
- v: 可视模式
- V: 可视（行）模式
- C-v: 可视（块）模式
- :: 命令模式

命令模式通常涉及到许多与文件本身有关的操作命令

- :q 退出（关闭窗口）
- :w 保存（写）
- :wq 保存然后退出
- :e 文件名 打开要编辑的文件
- :ls 显示打开的缓存
- :help :w 打开:w 命令的帮助文档
- :help w 打开 w 移动的帮助文档

正常模式

移动 (名词)

- 基本移动: hjkl (左, 下, 上, 右)
- 词: w (下一个词), b (词初), e (词尾)
- 行: 0 (行初), ^ (第一个非空格字符), \$ (行尾)
- 屏幕: H (屏幕首行), M (屏幕中间), L (屏幕底部)
- 文件: gg (文件头), G (文件尾)
- 行数: : + 行数 或者 行数 + G
- 查找: f + 字符, t + 字符, F + 字符, T + 字符

为什么叫名词?

语法规则

verb + noun

动词 + 名词

编辑 (动词)

- i(insert) 进入插入模式
- o(open a new line) 在本行之下插入行
- O 在本行之上插入行
- d(delete) + 移动命令删除
- c(change) + 移动命令改变
- x 删除字符 (等同于 dl)
- s(substitute) 替换字符 (等同于 cl)
- dd 删除一行
- u(undo) 撤销, C-r(redo) 重做
- y(yank) 复制 (其他一些命令比如 d 也会复制)
- p(paste) 粘贴

编辑 (动词) Con't

- 可视化模式 + 操作
- 选中文字后, d(delete) 删除或者 c(change) 改变

计数

你可以用一个计数来结合名词和动词

这会执行指定操作若干次

- 3w: 向后移动三个词
- 5j: 向下移动 5 行
- 7dw: 删除 7 个词

修饰语

你可以用修饰语改变“名词”的意义

- i: 表示“内部”或者“在内”
- a: 表示“周围”

- ci(: 改变当前括号内的内容
- ci[: 改变当前方括号内的内容
- da': 删除一个单引号字符串, 包括周围的单引号

自定义 Vim

.vimrc

Vim 由一个位于 `/.vimrc` 的文本配置文件来配置，可以被重度自定义

扩展 Vim

- ctrlp.vim: 模糊文件查找
- ack.vim: 代码搜索
- nerdtree: 文件浏览器
- vim-easymotion: 魔术操作

1 前言

2 Shell 命令行

3 Vim 编辑器

4 Git

Example

TCP/IP

- 软件开发人员只需使用可靠传输的协议栈就能完成网络通信任务，并不需要了解下层的 IP 协议
- 然而有时网络会越过抽象机制泄漏出底层问题
- 例如：劣化的网络通信情况会导致丢包现象严重
- 从而导致程序员在排查 Bug 时不得不掌握底层知识

Git 数据模型

Git 通过一个精心设计的模型来支持版本控制所需的所有特性

- 快照
- 对象
- 引用
- 仓库

快照 (Continue)

```

1 <root> (tree)
2 |
3 +- foo (tree)
4 |   |
5 |   + bar.txt (blob, contents = "hello world")
6 |
7 +- baz.txt (blob, contents = "git is wonderful")

```

关联快照

提交

在 Git 中，快照被称为“提交”

```
1  o <-- o <-- o <-- o
2
3      ^
4      |
5      --- o <-- o
```

- 其中的 o 表示一次提交
- 箭头指向了当前提交的父辈

対象 (Continue)

```
1 objects = map<string, object>
2
3 def store(object):
4     id = sha1(object)
5     objects[id] = object
6
7 def load(id):
8     return objects[id]
```

给这些哈希值赋予人类可读的名字：引用

引用 (Continue)

```

1 references = map<string, string>
2
3 def update_reference(name, id):
4     references[name] = id
5
6 def read_reference(name):
7     return references[name]
8
9 def load_reference(name_or_id):
10    if name_or_id in references:
11        return load(references[name_or_id])
12    else:
13        return load(name_or_id)

```

- 对象不可变，引用可变：可以被更新，指向新的提交

GitHub Git 文档:

<https://docs.github.com/zh/get-started/using-git/about-git>

Git 游戏: https://learngitbranching.js.org/?locale=zh_CN

计算机协会
Command-Line 命令行教学

Reference

- [1] “Linux 命令大全”. In: <https://www.linuxcool.com> (Nameless).
- [2] “The Missing Semester of Your CS Education”. In: <https://missing-semester-cn.github.io/about> (Anish, Jose, and Jon).