missing semester

看完课程后补上的笔记。首页链接：计算机教育中缺失的一课 . lecture0.pdf。
视频链接：[自制双语字幕] 计算机教育缺失的一课(2020) - 第1讲 - 课程概览与 shell_ 哔哩哔哩 _bilibili。个人认为胜过youtube版本。

目录

1 Lecture1 CourseOverview and Shell

本课程包含 11 个时长在一小时左右的讲座，每一个讲座都会关注一个特定的主题。
教学大部分是面向Linux的。课程会使用bash，最被广泛使用的shell。

当您打开终端时，您会看到一个提示符，它看起来一般是这个样子的：`missing:~$`

这是 shell 最主要的文本接口。它告诉你，你的主机名是 missing 并且您当前的工作目录 (cwd)或者说您当前所在的位置是 ~ (表示home)。 $符号表示您现在的身份不是 root 用户。

shell 基于空格分割命令并进行解析，然后执行第一个单词代表的程序，并将后续的单词作为程序可以访问的参数。如果您希望传递的参数中包含空格（例如一个名为 My Photos 的文件夹），您要么用使用单引号，双引号将其包裹起来，要么使用转义符号 \ 进行处理（ My\ Photos ）。

1.1 shell基本命令

shell 是一个编程环境，所以它具备变量、条件、循环和函数。
环境变量是在shell启动是设置的东西。其中包含你的主目录路径以及用户名。
当你执行程序时，shell会在环境变量$PATH中寻找，直到找到相应的程序。
可以通过`which [arg]`命令来寻找指定程序的路径。

shell 中的路径是一组被分割的目录，在Linux和macOS上使用/分割，而在Windows上是\。
绝对路径是能完全确定文件位置的路径。。如果某个路径以/开头，那么它是一个绝对路径。
相对路径是指相对于当 前工作目录的路径。
当前工作目录可以通过`pwd(present working directory)`获取。
`cd -` 会前往上一个目录。

`ls -l` 可以更加详细地打印出目录下文件或文件夹的信息。
首先，行中第一个字符d表示该文件是一个目录。然后接下来的九个字符，每三个字符构成一组。它们分别代表了文件所有者， 用户组以及其他所有人具有的权限。 其中- 表示该用户不具备相应的权限。
注意，/bin目录下的程序在最后一组，即表示所有人的用户组中，均包含 x 权限，也就是说任何人都可以执行这些程序。
如果你对于一个文件有写权限，对于目录却没有写权限，那么你只能清空此文件，不能删除此文件。
简单来说：

表格 1. rwx

r	是否被允许看到这个目录中的文件
w	是否被允许在该目录中重命名，创建或修改文件
x	是否被允许进入该目录或是执行文件

`mv`命令可以用于重命名或是移动文件。
eg：`mv aaa bbb`将文件 aaa 改名为 bbb。 `mv aaa /bbb` 将文件aaa移入bbb目录。

`cp [origin] [target]`命令可以用于复制文件。 其需要两个参数， 一个是要复制的文件路径，一个是目标路径。

1.2 程序间的流

在 shell 中，程序有两个主要的“流”：它们的输入流和输出流。当程序尝试读取信息时，它们会从输入流中进行读取，当程序打印信息时，它们会将信息输出到输出流中。通常输入是键盘，输出则是屏幕终端，但是我们可以重定向这些流。

最简单的方法是使用<以及>符号。
例如使用`cat < aaa.txt > bbb.txt`。此命令会将aaa文件作为cat的输入，同时将cat的输出覆写到bbb文件中。
>>的作用则是追加内容，而非覆写。
|管道(pipes)则可以将一个程序的输出与另一个程序的输入连接起来。

`sudo`命令可以以superuser的身份执行操作。

`tee`命令从标准输入中复制到一个文件，并输出到标准输出。
eg: `ping google.com | tee output.txt`
输出内容不仅会被写入文件，也会被显示在终端中。

`touch`命令可以修改文件的参数，或是创建一个不存在的文件。

2 Shell Tools and Scripting

2.1 Shell 脚本

一般来说，shell脚本中的空格是用来作为参数分隔的。
在shell中，$1表示的是脚本的第一个参数。具体如下：

表格 2. bash特殊符

$0	脚本名
$1-$9	脚本第一到第九个参数
$@	所有参数
$#	参数的数量
$?	上一条指令的返回值
$$	现在脚本进程的pid
!!	上一条指令(包括参数)，用来sudo !!执行上一条没执行成功的指令时很有用
$_	上一条指令的最后一个参数

命令会通过STDOUT返回值，错误则通过STDERR。
0表示执行正确，其他值则是错误。

short-circuting[1]逻辑短路指使用计算机中的与或运算来跳过语句。
eg: `ture || echo "will not be printed"` and `false && echo "will not be printed"`

在shell中，我们可以将一个命令的输出作为一个变量处理。例如 `for file in $(ls)`。
我们也可以通过 `<(cmd)`的形式将命令的输出放入一个临时文件中。因为有些命令是从文件中获取输入而非STDIN标准输入，所以有时这很有效。`eg: diff <(ls aaa) <(ls bbb)`。

当在bash中进行逻辑判断时，我们推荐使用[[ ]]而非[ ][2]。尽管其不兼容与sh，但这更安全。。

在bash中我们可以通过通配符来简化操作和参数。
通配符*与？。*匹配多位，？只匹配一位。
`eg: foo1,foo2,foo10. foo?-->foo1,foo2. foo*-->foo1,foo2,foo10`
花括号则可以用来拓展。
`eg: mv *{.py,.sh} will move all *.py and *.sh。 touch {foo,bar}/{a..h} will create files foo/a,foo/b...foo/h,bar/a...bar/h`。

你也可以通过其他语言写脚本。

我们可以通过文件顶部的shebang[3]来指定执行脚本的解释器。
我们可以在shebang中使用env[4]命令来解析系统中相关命令的位置，从而提高可移植性。

1. Short-circuit evaluation - Wikipedia。
2. BashFAQ/031 - Greg's Wiki (wooledge.org)
3. Shebang (Unix) - Wikipedia
4. env(1) - Linux manual page (man7.org)

`eg: #!/usr/bin/env python`

2.2 shell工具

man手册可以帮助我们了解命令的使用方法，但有时候man手册提供的内容过多了。这时我们可以通过tldr[5]来了解相关命令的例子从而更好的使用它们。

`find`命令可以用来寻找文件，同时也可以对这些文件进行操作。
`eg: find . -name '*.tmp' -exec rm {} \; delete all files with .tmp extension`
我们也可以应用更现代且语法更便捷的fd[6]来搜索文件。
`locate`命令。。感觉没啥用。

搜索文件中的内容可以使用grep命令。你也可以使用更现代的ripgrep。
`eg: rg -u --files-without-match "^#!"`
`it will find all files (including hidden files by using -u) without a shebang(by using --files.. to print those who don't match the pattern we give to command)`

可以通过`history`命令来快速的回归命令历史。也可以通过<c-r>来反向搜索命令历史。
fzf[7]是更高效的反向搜索工具。

快速的显现目录结构可以通过`ls -R` 或是 `tree`来实现。

3 Editors(Vim)

3.1 Vim模式

Vim是基于模式的。具体转换如下:

$$\text{normal} \overset{i}{\longleftrightarrow}{}^{\text{esc}} \text{insert}$$
$$\text{normal} \overset{r}{\longleftrightarrow}{}^{\text{esc}} \text{replace}$$
$$\text{normal} \overset{v}{\longleftrightarrow}{}^{\text{esc}} \text{visual}$$
$$\text{normal} \overset{\text{shift}+v}{\longleftrightarrow}{}^{\text{esc}} \text{visual-line}$$
$$\text{normal} \overset{<c-v>}{\longleftrightarrow}{}^{\text{esc}} \text{viusal-block}$$
$$\text{normal} \overset{:}{\longleftrightarrow}{}^{\text{esc}} \text{command}$$

3.2 Vim基础

表格 3. vim :

:q	退出当前窗口
:w	保存
:wq	保存并退出
:help {topic}	open help
:e {filename}	切换文件
:ls	显示打开的buffers

5. tldr pages
6. sharkdp/fd: A simple, fast and user-friendly alternative to 'find' (github.com)
7. Configuring shell key bindings · junegunn/fzf Wiki (github.com)

Vim会维持一组打开的文件，即*buffer*。
Vim具有很多的*tab*,每一个都对应一个或多个*window*。
每一个*window*显示出一个*buffer*。
一个*buffer*可以被多个*window*显示，即使这些*window*在同一个*tab*中。

表格 4. Vim movement

w	下一个词 next word
b	单词开头(向前) begining of word
e	单词尾部(向后) end of word
0	行开头
^	此行第一个非空字符
$	行尾
H	窗口顶部
M	窗口中部
L	窗口底部
<C-u>	向上滚动(up)
<C-d>	向下滚动(down)
gg	文件开头
G	文件尾部
{number}G	跳到相应行
%	在{，[，(上使用会跳转到对应的括号
f{character}	向后跳转到此字符
t{character}	向后跳转到此字符前
F{character}	向前跳转到此字符
T{character}	向前跳转到此字符后
/(word)	使用/进行单词搜索，按下回车后会跳转到相应处

表格 5. Vim modify

~	翻转大小写
ci[	修改[ ]中的内容
da(	删除包括( ) 在内的所有内容

3.3 Vim自定义

使用~/.vimrc文件自定义你的Vim。
你可以在自己的shell中启用vim模式。
bash使用set -o vi。zsh使用bindkey -v。fish使用fish_vi_key_bindings。
你也可以不论shell类型直接使用export EDITOR=vim。但这也会改变其他程序的editor，比如git。Readline也可以使用 set editing-mode vi来进入Vim模式。例如python repl便会收到影响。

使用Vimium - Chrome Web Store (google.com)在chorme中启用Vim模式。
宏就不介绍了，一时半会说不清楚。
使用最少的操作完成文件处理:VimGolf - real Vim ninjas count every keystroke!。

4 Data Wrangling

将一种格式的数据变为另一种格式的都可以称为数据整理。
`less`命令可以用来分页查看文本。
使用sed这个流编辑器来处理文本。`eg: cat ssh.log | sed 's/.*Disconnected from //'`

s替换命令的格式是 s/REGEX/SUBSTITUTION/
sed除了搜索和替换之外，其他的功能并不算好。

4.1 正则表达式

正则表达式在默认情况下每行只会匹配一次，替换一次。默认情况下不会跨行匹配。
想要全部匹配，请使用s/REGEX/SUBSTITUTION/g。
sed中的特殊字符需要使用\来转义。你也可以使用sed -E来使用更现代d1正则表达式。

表格 6.

字符	匹配模式
.	任意字符
*	前一个字符的zero or more
+	前一个字符的one or more
[abc]	a,b,c中的任意一个
(regex1|regex2)	rx1,rx2中的任意一个
^	行开头
$	行尾

通常来说，*和+都是greddy贪婪的，他们会尽可能多的匹配文本。若要改为非贪婪的，则需要在两者后面加上？。
你可以使用regex debugger来调试表达式。
你也可以使用捕获组 capture group来取得正则表达式中匹配上的内容，只需要用()括起来即可。
`eg: sed -E 's/.*Disconnected from  (invalid |authenticating )?user (.*) [^ ]+ port [0-9]+( \[preauth\])?$/\2/'`
其中有三个括号，我们想要的user后的用户名便是\2。我们将这一整句内容替换为了用户名。

4.2 其他数据工具

sort工具可以对其的输入进行排序。
sort -n将会以第一列的数字大小排序而非字典序。 -k则以选择输入中以空格为分隔符的列来执行排序。sort -nk1,1即从第一列开始，在第一列结束。sort -r可以倒序输出。

uniq工具会查看一个排序后的行列表，然后其会去除那些重复的行。多个相同的行只会打印一次。uniq -c将计算任何重复行的次数并消除他们，并将次数加到行前缀上。

可以使用tail -n5，head -n10类似的工具获取输出的头部和尾部。

paste命令可以将多个行合并在一起，使其合为一行。-s会按输入将输出的一行分隔，-d则会指定分隔符(`eg:paste -sd,`)。

可以使用wc -l计算行的数量。使用bc来计算表达式。
比如说将多行数字输出使用`|paste -sd+ | bc -l` 将多行数字输出使用+连接后使用bc计算。
使用R进行数据统计，使用gnuplot画图。

xargs可以将一系列输入转化为参数,很有用。比如你有很多旧版本的工具链需要删除时。
`eg: xxx toolchain |grep 2019| xargs xxx toolchain uninstall`

4.3 awk

awk也是处理文件流的，但其更专注与处理列。其默认将输入解析为以空格为分隔符的列。

在awk参数中，$0指全部内容，$1-n则指1-n列。
eg: awk '$1 ==1 && $2 ~ /^c.*e$/ {print $2}'
输出所有第一列为1且第二列匹配模式的行的第二列。

awk编程,输出有多少行匹配条件的：
```
BEGIN {rows = 0}
$1 ==1 && $2 ~ /^c.*e$/ {rows +=1 }
END {print rows}
```

begin匹配输入开头，end匹配输入尾。

5 Command-line Environment

5.1 job control

<C-c>停止当前进程是因为终端向程序发送了一个SIGINT信号。该信号表示程序中断。
我们也可以使用<C-\>来发送SIGQUIT信号来终止程序。中断和终止是不同的。
SIGKILL无法被程序捕获，如果发送SIGKILL，它无论如何都将终止进程的执行。

通过<C-z>发送SIGSTOP信号可以将程序暂停。
使已经暂停的在后台进程再次运行可以使用bg %n。n指其在Jobs中的序号。
在命令后加上&会使命令在后台运行。

我们可以使用kill命令发送任何信号。eg:kill -STOP %1停止选定进程。
nohup命令将所执行的命令封装起来，忽略任何挂起信号，并使其继续运行。

5.2 Terminal Multiplexers

推荐将<C-b>重映射为<C-a>，反正你自己已经在dotfiles中设置好了。

tmux的几个核心概念：
-sessions： 有多个windows

tmux ls会显示当前所有tmux sessions
tmux new -s NAME 创建带名字的session
可以使用<C-b> d 脱离此session
tmux a 重新回到上一个session，可以使用-t指定回归的session

-windows：类似于编辑器和浏览器中的标签。

<C-b> c可以创建一个新窗口。关闭只需用<C-d>即可
<C-b> p前往上一个窗口，<C-b> n前往下一个窗口。
<C-b> ,可以更改窗口名。<C-b> w显现出所有现在的窗口。
<C-b> N可以跳转至N号窗口。

-panes：分裂窗口

<C-b> "垂直分裂窗口。<C-b> %水平分裂窗口。
<C-b> 箭头用来在panes间按方向跳动。
<C-b> 扩张当前窗口。
<C-b> space 改变当前panes的排列。

<C-b> x 关闭当前pane

5.3 dotfiles

alisa别名可以省去很多时候输入命令参数的麻烦。
eg: alisa ll="ls -lh" 由于alias是shell命令，所以=旁边不能有空格。
mv=mv -i mkdir=mkdir -p都是可以尝试的。

有许多很关键的配置文件，例如.vimrc，.bashrc可以放置在dotfiles中。
我们使用dotfiles下的安装脚本建立软链接， 从而快捷的安装所有的配置文件。 dotfiles可以通过git获取， 而链接可以通过ln -s创建。可以参见你自己的dotfiles中的install.sh明晰安装过程。
你可以上网学习其他人的dotfiles中的配置并化为己用。

5.4 Remote Machines

ssh为其中关键。
一般来说使用ssh username@ip来进行连接。
每次都要输入密码是有点麻烦的，我们可以使用公钥私钥体系。

可以先使用ssh-keygen -o -a 100 -t ed25519 -f ~/.ssh/id_ed25519创建keys。
当然，你在使用github时可能配置过了。
运行cat .ssh/id_ed25519.pub | ssh username@ip 'cat >> ~/.ssh/authorized_keys
可以使用更简单的ssh-copy-id -i .ssh/id_ed25519.pub username@ip。其会复制密钥。

在远端复制文件不能使用cp，但可以使用scp命令。
使用rsync复制则是更为先进。

你可以在~/.ssh/config文件里配置内容。

```
Host vm
     User shulva
     HostName 111.111.111.111
     Port 2222
     IdentitiyFile ~/.ssh/id_ed25519
```

之后使用ssh vm即可登录。
在config里配置，便利性和可移植性都有所提高。你也可以将自己的ssh文件放在dotfiles中。
但这会对安全性有所影响。

6 Version Control(Git)

Version Control (Git) . lecture6.pdf

6.1 Git's data model

在git数据模型中，一个文件被称之为blob，一堆字节的合集。
而一个文件夹则被称之为tree。tree中可以包含一系列tree和blob。
一个git的快照snapshot本质上是对最顶层的tree(directory)的追踪。

git使用DAG来处理快照间的关系。一般来说git将这些快照称为commit。

git中的commit是不可改变的， 但这并不意味着错误是不可更改的。 但是这种更改会产生新的commit。你可以添加新的东西，但是无法改动图的结构。

git的数据模型伪代码示例如下:

```
type blob = array<byte>
type tree = map<string,tree|blob>
type commit = struct{
    parent:array<commit>
    author:string
    message:string
    snapshot:tree
    }
```

git数据模型实际磁盘存储的数据结构如下,使用hash函数记录:

```
type object = blob|tree|commit
objects = map<string,object>

def store(objects):
    id = sha1(object)
    object[id]=object

def load(objects):
    return objects[id]
```

blob,tree,commit都是Objects。当他们指向其他objects时，他们并不是真的包含Objects。他们只是拥有他们的引用。

所有object都储存于Object表中，git中不同对象之间的引用都是通过它们的id，也即是hash值。

所有的快照都可以被它们的SHA-1 hash值唯一标识。但是hash值是一组长为40的字符串，不好记忆。所以git通过维护references来解决这一问题。

git的引用模型伪代码示例如下:

```
references = map<string1_name,string2_hash>
```

通过自己可记忆的name来映射hash值。

6.2 Staging Area 暂存区

暂存区会告知git在下一次创建快照/commit时应该包含哪些更改。

git将追踪文件的权力交给了用户，从而让用户可以自己定义快照，而非直接快照整个文件。

使用git add <filename>可以直接将文件添加到暂存区。

6.3 Git command-line interface

推荐Git - Book (git-scm.com)了解具体细节。

git commit的信息编写:

How to Write a Git Commit Message (cbea.ms)

tbaggery - A Note About Git Commit Messages

HEAD指向最后一个快照。

`git log --all --graph --decorate` 将提交历史用DAG的方式显现出来（其实也没显示多少）

`git log --all --graph --decorate --oneline` 更精简的图结构

`git checkout <reference>` 更新HEAD到你指向的commit或是分支，改变你当前工作目录

`git diff`可以显示当前目录与上次快照之间的不同

```
git diff <filename>当前文件与上次快照间的不同
git diff <reference> <filename>当前工作目录与指定快照间文件的不同
git diff <reference> <reference> <filename>两个指定快照间文件的不同

git branch -vv会列出在本地仓库的所有分支,-vv是详细信息。
git branch <name> 创建新分支。
git checkout -b <name> 创建并切换到新分支

git merge <name> 将此分支合并到当前分支
git mergetool 启用专门工具处理合并冲突

一般来说冲突文件中会包含如下内容:
<<<<< HEAD
...当前分支的
==========
...合并分支的
>>>>> branch name

git remote 可以列出当前仓库所知道的所有远程仓库。
git remote add <name> <url> 添加一个远程仓库。
git push <remote> <local branch>:<remote branch>
通过本地分支创建远程仓库上的分支。
git branch --set-upsteram-to=<remote>/<remote branch> 将本地分支与远程分支相对应。
这会简化git pull/push,不用输入大量参数。
git fetch 获取远端更改，从远端获取对象和引用，不会更改任何本地历史记录。
git pull= git fetch+git merge

git config 自定义，自己修改~/.gitconfig亦可。
git clone --depth=1 不会clone历史提交记录
git add -p 交互式的暂存
git blame 显示每一行是谁修改提交的
git show 显示提交信息
git stash 暂时移除更改 git stash pop是相对应的逆过程
git bisect (binary search history?)
.gitignore 使用gitignore文件指定无需上传的文件类型 eg:*.DS_Store

git commit --amend 修改一个commit的内容和信息
git reset HEAD <file>:反暂存文件
git checkout -- <file>:舍弃修改
```

7 Debugging and Profiling

7.1 Debugging

相较于使用printf来进行调试，使用日志logging进行调试的优势在于:

- 您可以将日志写入文件、socket 或者甚至是发送到远端服务器而不仅仅是标准输出。
- 日志可以支持严重等级（例如 INFO, DEBUG, WARN, ERROR等)，这使您可以根据需要过滤日志。
- 对于新发现的问题，很可能您的日志中已经包含了可以帮助您定位问题的足够的信息。
- 可以在终端中使用彩色文本以提供日志的可读性。

大多数的程序都会将日志保存在您的系统中的某个地方。对于UNIX系统来说，程序的日志通常存放在 /var/log。

大多数Linux 系统都会使用 systemd， systemd 会将日志以某种特殊格式存放于/ var/ log/ journal ，您可以使用 journalctl 命令显示这些消息。

```
eg: journalctl --since ‘‘1m ago’’ | grep Hello
```

很多时候应当使用调试器对程序进行调试，这样调试时展示的信息更多。

你可以使用增强的调试器，例如pdb的增强版ipdb。其提供了高亮，tab补全等等功能。

即使您需要调试的程序是一个二进制的黑盒程序， 仍然有一些工具可以帮助到您。 当您的程序需要执行一些只有操作系统内核才能完成的操作(系统调用)时。有一些命令可以帮助您追踪您的程序执行的系统调用。在 Linux 中可以使用 strace来完成此类操作。

有些问题是不需要执行代码就能发现的。 例如仔细观察一段代码， 您就能发现某个循环变量覆盖了某个已经存在的变量或函数名或是有个变量在被读取之前并没有被定义。

这种情况下静态分析工具就可以帮我们找到问题。

静态分析工具可以参考这个列表:analysis-tools-dev/static-analysis:

对于 linters 则可以参考这个列表：caramelomartins/awesome-linters

7.2 Profiling

调试代码类似，大多数情况下我们只需要打印两处代码之间的时间计算时间差即可发现问题。

不过在时间差之中，我们需要区分真实时间、用户(User)时间和系统(Sys)时间。

- 真实时间 - 从程序开始到结束流失掉的真实时间， 包括其他进程的执行时间以及阻塞消耗的时间（例如等待 I/O或网络）
- User - CPU 执行用户代码所花费的时间
- Sys - CPU 执行系统内核代码所花费的时间

但是使用Printf来进行分析是难以维护的，所以我们要使用Profiler。

CPU 性能分析工具有两种：追踪分析器（*tracing*）及采样分析器（*sampling*）。

追踪分析器会记录程序的每一次函数调用，会延长被分析程序的运行时间。

而采样分析器则只会周期性的监测（通常为每毫秒）程序并记录程序堆栈以得到函数调用的结果。

更加符合直觉的显示分析信息的方式是包括每行代码的执行时间，这也是行分析器 *(line profiler)*的作用。其会显示出程序中每一行所花费的时间，相比显示的是每次函数调用的时间的profiler，其可以更直观的理解数据。

像 C 或者 C++ 这样的语言，程序在使用完内存后不去释放它会导致内存泄漏。此时我们就需要类似Valgrind这样的工具来检查内存泄漏问题。

不同与strace, perf 命令将 CPU 的区别进行了抽象，它不会报告时间和内存的消耗，而是报告与您的程序相关的系统事件。

例如， perf 可以报告不佳的缓存局部性（poor cache locality）、大量的页错误（page faults）或活锁（livelocks）。

- perf list - 列出可以被 pref 追踪的事件
- perf stat COMMAND ARG1 ARG2 - 收集与某个进程或指令相关的事件
- perf record COMMAND ARG1 ARG2 - 记录命令执行的采样信息并将统计数据储存在 perf.data 中
- perf report - 格式化并打印 perf.data 中的数据

您可以使用类似Flame Graphs之类的工具来实现profile的可视化。

有很多工具可以被用来显示不同的系统资源，例如 CPU 占用、内存使用、网络、磁盘使用等。

- *通用监控* - 最流行的工具要数htop 。 htop 可以显示当前运行进程 的多种统计信息。还可以留意一下Glances ，它的实现类似但是用户界面更好。
- *I/O 操作* - iotop命令可以显示实时 I/O 占用信息而且可以非常方便地检查某个进程是否正在执行大量的磁盘读写操作
- *磁盘使用* - df 命令可以显示每个分区的信息，而 du 命令则可以显示当前目录下每个文件的磁盘使用情况（ disk usage）。 -h 选项可以使命令以对人类（human）更加友好的格式显示数据。ncdu是一个交互性更好的 du ，它可以让您在不同目录下导航、删除文件和文件夹
- *内存使用* - free命令可以显示系统当前空闲的内存。内存也可以使用 htop 这样的工具来显示
- *打开文件* - lsof命令可以列出被进程打开的文件信息。 当我们需要查看某个文件是被哪个进程打开的时候，这个命令非常有用
- *网络连接和配置* - ss命令能帮助我们监控网络包的收发情况以及网络接口的显示信息。 ss 常见的 一个使用场景是找到端口被进程占用的信息。如果要显示路由、网络设备和接口信息，您可以使用 ip 命令。**netstat 和 ifconfig 这两个命令已经被前面那些工具所代替了**
- *网络使用* -raboof/nethogs和iftop是非常好的用于对网络占用进行监控的交互式命令行工具

如果您希望测试一下这些工具，您可以使用 stress 命令来为系统人为地增加负载。

有时候您只需要对黑盒程序进行基准测试(例如find和fd之间的比较)并依此对软件选择进行评估。类似hyperfine这样的命令行可以帮您快速进行基准测试。

8 MetaProgramming

我们本章要讲的东西，更多是关于**流程** ，而不是写代码或者去更高效的工作。本节课我们会学习构建系统、代码测试以及依赖管理。

必须要指出的是，元编程也有用于Metaprogramming - Wikipedia即操作程序的程序之含义，这和我们本章所介绍的概念是完全不同的。

8.1 Build systems

对于大多数系统来说，不论其是否包含代码，都会包含一个“构建过程”。有时需要执行一系列操作。有很多工具可以帮助我们完成这些操作。这些工具通常被称为 “构建系统”。

如何选择工具完全取决于您当前要完成的任务以及项目的规模。 从本质上讲，这些工具都是非常类似的。您需要定义*依赖*、*目标和规则*。

您必须告诉构建系统您具体的构建*目标*，系统的任务则是找到构建这些目标所需要的*依赖*，并根据*规则*构建所需的中间产物，直到最终目标被构建出来。

make 是最常用的构建系统之一。

```
paper.pdf: paper.tex plot-data.png
           pdflatex paper.tex
plot-%.png: %.dat plot.py
           ./plot.py -i $*.dat -o $@
```

paper.pdf即为目标(target)，paper.tex,plot-data.png即为依赖(dependency)。**:**左边是目标，右边是依赖项。

规则中的 % 是一种模式匹配。例如，如果目标是 plot-foo.png ，make会去寻找foo.dat作为依赖。

make会以工作量最小为目的来构建，故而依赖没有变动的目标一般是不会在重复构建的过程中重新构建的。

8.2 Dependency management

就项目来说，它的依赖可能本身也是其他的项目，也许会依赖某些程序(例如 python)、 系统包(例如 openssl)或相关编程语言的库(例如 matplotlib)。 现在，大多数的依赖可以通过某些软件仓库来获取(例如pypi)。

大多数被其他项目所依赖的项目都会在每次发布新版本时创建一个版本号。

版本号有很多用途，其中最重要的作用是保证软件能够运行。

不同项目所用的版本号其具体含义并不完全相同， 但是一个相对比较常用的标准是语义版本号sematic versioning， 这种版本号具有不同的语义， 它的格式是这样的： 主版本号. 次版本号. 补丁号(eg:8.3.2)。相关规则有：

- 如果新的版本没有改变 API，请将补丁号递增
- 如果您添加了 API 并且该改动是向后兼容的，请将次版本号递增
- 如果您修改了 API 但是它并不向后兼容，请将主版本号递增

这么做有很多好处。现在如果我们的项目是基于项目构建的， 那么只要最新版本的主版本号只要没变就是安全的，次版本号不低于之前我们使用的版本即可。

使用依赖管理系统的时候，您可能会遇到锁文件（lock files）这一概念。

锁文件列出了您当前每个依赖所对应的具体版本号。通常需要执行升级程序才能更新依赖的版本。这么做的原因有很多， 例如避免不必要的重新编译、 创建可复现的软件版本或禁止自动升级到最新版本（可能会包含 bug）。

还有一种极端的依赖锁定叫做 vendoring，它会把您的依赖中的所有代码直接拷贝到您的项目中，这样您就能够完全掌控代码的任何修改，同时您也可以将自己的修改添加进去，不过这也意味着如果该依赖的维护者更新了某些代码，您也必须要自己去拉取这些更新。

8.3 Continuous integration systems

持续集成，或者叫做CI，它指的是那些“当您的代码变动时，自动运行的东西”。

它们的工作原理都是类似的：您需要在代码仓库中添加一个文件，描述当前仓库发生任何修改时，应该如何应对。目前为止，最常见的是：如果有人提交代码，执行测试套件或是代码的风格检查。

多数的大型软件都有测试套件。

- 测试套件：所有测试的统称。
- 单元测试：一种“微型测试”，用于对某个封装的特性进行测试。例如测试微信机器人的api查询功能。
- 一种“宏观测试”，针对系统的某一大部分进行，测试其不同的特性或组件是否能**协同**工作。例如测试微信机器人整体对于客户消息的处理。
- 回归测试：一种实现特定模式的测试，测试以前出过错的内容。
- 模拟（Mocking）：使用一个假的实现来替换模块， 函数或类型， 屏蔽那些和测试不相关的内容。例如，您可能会模拟网络连接或模拟网络请求。

9 Security and Cryptography