

🐱 MIT The Missing Semester of Your CS Education

1. shell

1. 基础知识

```
Last login: Thu Jan  7 19:58:46 on ttys000  
~  21:14:10  
$
```

看起来像这样的就是一个shell \$表示你的身份不是root权限

我们可以通过echo来输出

```
~  21:16:32  
[$ echo hello  
hello
```

shell基于空格分割命令进行解析 将后面的单词作为程序的参数，如果传递中的参数包括空格 可以使用单引号双引号 或者是转义符号

那么 echo 的位置在哪儿

其实shell也是一个编程环境 所以变量 条件 循环和函数他也都有（类似 import）

通过环境变量\$PATH（大写）进行相关的搜索

```
~  21:21:23  
$ echo $PATH  
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Applications/VMware Fusion.app/Contents/Public:/Library/Apple/usr/bin
```

2.shell中的一些重要命令

Pwd 显示当前目录

cd (change dir来切换命令)

路径中的当前目录是. 上级目录是..

利用ls 查看目录下的文件 利用cat文件进行查看或者vim进行修改

ls -l 可以查看权限

```
$ ls -l
total 8
drwxr-xr-x  7 miewang  staff  224  1  7 21:24 My-winter-vacation
-rw-r--r--  1 miewang  staff   15  1  7 19:27 test.txt
```

d表示dir这个是一个目录

然后接下来的字符每三个为一组

r(read 读权限)

w(write 写权限)

x(执行权限)

分别代表文件所有者 staff组以及其他人的权限

-表示没有的权限

也要使用 mv cp mkdir等功能

可以使用 man来查看

<>可以用这样的流来写入文件

>> 来向一个文件追加内容

```
~/Documents/2021 on [?]master! 21:25:05
$ echo 123 >test.txt
```

```
~/Documents/2021 on [?]master! 21:32:44
```

```
$ cat test.txt
```

```
123
```

| 操作符允许我们将一个程序的输出和另外一个程序的输入连接起来:

```
$ ls -l /|tail -n1
drwxr-xr-x@ 2 root  wheel     64  1  1  2020 xarts
```

3.课后作业

1. 在 `/tmp` 下新建一个名为 `missing` 的文件夹。
2. 用 `man` 查看程序 `touch` 的使用手册。
3. 用 `touch` 在 `missing` 文件夹中新建一个叫 `semester` 的文件。
4. 将以下内容一行一行地写入 `semester`

文件:

```
#!/bin/sh
curl --head --silent https://missing.csail.mit.edu
```

5.尝试执行这个文件。例如，将该脚本的路径（`./semester`）输入到您的shell中并回车。如果程序无法执行，请使用 `ls` 命令来获取信息并理解其不能执行的原因。

6.查看 `chmod` 的手册(例如，使用 `man chmod` 命令)

7.使用 `chmod` 命令改变权限，使 `./semester` 能够成功执行，不要使用 `sh semester` 来执行该程序。您的 shell 是如何知晓这个文件需要使用 `sh` 来解析呢？更多信息请参考：[shebang](#)

8.使用 `|` 和 `>`，将 `semester` 文件输出的最后更改日期信息，写入家目录下的 `last-modified.txt` 的文件中

4.课后作业解答

1.

```
/ ⌚ 22:12:56
$ cd /tmp/
/tmp ⌚ 22:12:59
$ mkdir missing

/tmp ⌚ 22:13:10
$ ls
com.apple.launchd.ZCyT5CuwJz  mysql.sock.lock
com.google.Keystone           mysqlx.sock
fsevents-d-uuid                mysqlx.sock.lock
missing                         powerlog
mysql.sock
```

2.

```

touch(1)           BSD General Commands Manual          touch(1)

NAME
    touch -- change file access and modification times

SYNOPSIS
    touch [-A [-] [[hh]mm]SS] [-acfhm] [-r file] [-t [[CC]YY]MMDDhhmm[.SS]]
            file ...

DESCRIPTION
    The touch utility sets the modification and access times of files. If
    any file does not exist, it is created with default permissions.

    By default, touch changes both modification and access times. The -a and
    -m flags may be used to select the access time or the modification time
    individually. Selecting both is equivalent to the default. By default,
    the timestamps are set to the current time. The -t flag explicitly spec-
    ifies a different time, and the -r flag specifies to set the times those
    of the specified file. The -A flag adjusts the values by a specified
    amount.

    The following options are available:
```

3.

```

/tmp/missing 22:17:36
$ touch -f semester

/tmp/missing 22:17:48
$ ls
semester
```

4.

```

/tmp/missing 22:24:02
$ cat semester
#!/bin/sh

/tmp/missing 22:24:05
$ echo curl --head --silent https://missing.csail.mit.edu>>semester

/tmp/missing 22:24:26
$ cat semester
#!/bin/sh
curl --head --silent https://missing.csail.mit.edu
```

5.

```
/tmp/missing 22:24:29
$ ./semester
zsh: permission denied: ./semester

/tmp/missing 22:25:43
$ ls -l semester
-rw-r--r-- 1 miewang wheel 61 1 7 22:24 semester
```

没有执行权限

6.man chmod

7.

```
$ chmod 777 semester

/tmp/missing 22:27:44
$ ./semester
HTTP/1.1 200 Connection established

HTTP/2 200
content-type: text/html; charset=utf-8
server: GitHub.com
x-origin-cache: HIT
last-modified: Thu, 31 Dec 2020 22:23:08 GMT
access-control-allow-origin: *
etag: "5fee4f4c-1e9f"
expires: Thu, 07 Jan 2021 14:37:47 GMT
cache-control: max-age=600
x-proxy-cache: HIT
x-github-request-id: 4A9C:68A6:10154D:187727:5FF71A6A
accept-ranges: bytes
date: Thu, 07 Jan 2021 14:27:55 GMT
via: 1.1 varnish
age: 0
```

8.

```
/tmp/missing 22:31:25
$ ./semester | grep -i "last-modified" > ~/last-modified.txt
```

2.Shell tools and scripting

```
~ 9:04:41
[$ foo=bar

~ 9:16:39
[$ echo $foo
bar

~ 9:16:45
[$ foo = bar
zsh: command not found: foo
```

如果有空格出现则不行

可以用双引号解析字符串

```
~ 9:16:53
[$ echo "hello $foo"
hello bar

~ 9:19:52
[$ echo 'hello $foo'
hello $foo
```

我们尝试一下写一个.sh脚本

vim mcd.sh

```
mc当地( ) {  
    mkdir -p "$1"  
    cd "$1"  
}
```

这里面的\$1就相当于其他编程中的参数args

- \$0 - 脚本名
- \$1 到 \$9 - 脚本到参数。 \$1 是第一个参数，依此类推。
- \$@ - 所有参数
- \$# - 参数个数
- \$? - 前一个命令到返回值
- \$\$ - 当前脚本到进程识别码
- !! - 完整到上一条命令，包括参数。常见应用：当你因为权限不足执行命令失败时，可以使用 sudo !! 再尝试一次。
- \${_} - 上一条命令的最后一个参数。如果你正在使用的是交互式shell，你可以通过按下 Esc 之后键入 . 来获取这个值。

定义这个函数的作用就是建立一个目录 然后我们切换到这个目录

当我们运行这个脚本之后

```
~/Documents/2021/My-winter-vacation on ?master! 9:21:10
$ vim mcd.sh

~/Documents/2021/My-winter-vacation on ?master! 9:25:22
$ source mcd.sh

~/Documents/2021/My-winter-vacation on ?master! 9:25:28
$ mcd _mtools_drives:3: command not found: mtoolstest
_mtools_drives:3: command not found: mtoolstest
_mtools_drives:3: command not found: mtoolstest
t mcd t _mtools_drives:3: command not found: mtoolstest
_mtools_drives:3: command not found: mtoolstest
_mtools_drives:3: command not found: mtoolstest
e mcd te _mtools_drives:3: command not found: mtoolstest
_mtools_drives:3: command not found: mtoolstest
_mtools_drives:3: command not found: mtoolstest
s mcd tes _mtools_drives:3: command not found: mtoolstest
_mtools_drives:3: command not found: mtoolstest
_mtools_drives:3: command not found: mtoolstest
t mcd test

~/Documents/2021/My-winter-vacation/test on ?master! 9:25:33
```

source命令的作用就是读取脚本中的bash命令 将里面配置的文件全加在环境中

```
~/Documents/2021/My-winter-vacation on ?master! 9:29:00
$ mkdir test

~/Documents/2021/My-winter-vacation on ?master! 9:29:25
$ cd $_
```

当我们grep并没有搜到相关字符串的时候 会将错误代码置为1

```
~/Documents/2021/My-winter-vacation on ?master! 9:31:03
$ grep foobar mcd.sh

~/Documents/2021/My-winter-vacation on ?master! 9:31:13
$ echo $?
1
```

运算符

```
$ false||echo "Oops fail"
Oops fail
```

短路运算

```
~/Documents/2021/My-winter-vacation on master! 9:33
$ true || echo "Oh I love you"

~/Documents/2021/My-winter-vacation on master! 9:34
[$ true && echo 1
1
```

也可以变量赋值成如下

```
~/Documents/2021/My-winter-vacation on master! 9:34:26
$ foo=$(pwd)

~/Documents/2021/My-winter-vacation on master! 9:35:17
$ echo $foo/
/Users/miewang/Documents/2021/My-winter-vacation/
```

看一个demo bash

```
1#!/bin/bash
2
3echo "Starting program at $(date)" # Date will be substituted
4
5echo "Running program $0 with $# arguments with pid $$"
6
7for file in "$@"; do
8    grep foobar "$file" > /dev/null 2> /dev/null
9    # When pattern is not found, grep has exit status 1
10   # We redirect STDOUT and STDERR to a null register
11   # about them
12   if [[ "$?" -ne 0 ]]; then
13       echo "File $file does not have any foobar, adding one"
14       echo "# foobar" >> "$file"
15   fi
16done
```

主要关注点是这个 /dev/null 2> /dev/null

看了官网的解释

我们将标准输出流和标准错误流重定向到Null,

如果没找到字符串 grep就退出好了 不关心接下来怎么样

这个ne是不等于的意思 是linux shell的命令

```
tools >>> touch foo foo1 foo2 foo10          x 130 master
tools >>> cp project{1,2}                      x 130 master
tools >>> touch project1/src/test/test1.py project1/src/test/test2.py p
/project1/src/test/test3.py project2/src/test/test1.py project2/src/test/test2.p
ject2/src/test/test3.py
tools >>> touch project1/src/test/test1.py project1/src/test/test2.py p
/project1/src/test/test3.py project2/src/test/test1.py project2/src/test/test2.p
ject2/src/test/test3.py
tools >>> mkdir foo bar
tools >>> touch {foo,bar}/{a..j}■
```

用花括号可以匹配多个东西创建

shellcheck来debug shell脚本

some useful tools

man /convert/tar/find...

可以通过history来查看自己的键入记录

```
3 touch root,z,s,4
4 convert
5 ls
6 shellcheck mcd.sh
7 shellcheck
8 ls -l
9 ./mcd.sh
10 tar
11 tar -h
12 find ".flag*"
13 find "*/flag.php"
14 find / -name="fla*"
15 rg "import requests" -t py
16 ripgrep
17 man rg
18 rg
19 ripgrep
20 ack
21 grep
22 history
23 history 1 | grep ctf
```

最后附上一个我们在ctf awd常用的一个 备份文件

用tar打包然后下载下来

```
[$ tar -cvf 1.tar ~miiewang/Desktop/
```

tar -xvf 1.tar 来解压

课后练习

1. 阅读 `man ls`，然后使用 `ls` 命令进行如下操作：

- 所有文件（包括隐藏文件）
- 文件打印以人类可以理解的格式输出（例如，使用454M而不是454279954）
- 文件以最近访问顺序排序
- 以彩色文本显示输出结果

典型输出如下：

```
-rw-r--r-- 1 user group 1.1M Jan 14 09:53 baz
drwxr-xr-x 5 user group 160 Jan 14 09:53 .
-rw-r--r-- 1 user group 514 Jan 14 06:42 bar
-rw-r--r-- 1 user group 106M Jan 13 12:12 foo
drwx-----+ 47 user group 1.5K Jan 12 18:08 ..
```

2. 编写两个bash函数 `marco` 和 `polo` 执行下面的操作。每当你执行 `marco` 时，当前的工作目录应当以某种形式保存，当执行 `polo` 时，无论现在处在什么目录下，都应当 `cd` 回到当时执行 `marco` 的目录。为了方便debug，你可以把代码写在单独的文件 `marco.sh` 中，并通过 `source marco.sh` 命令，（重新）加载函数。
3. 假设您有一个命令，它很少出错。因此为了在出错时能够对其进行调试，需要花费大量的时间重现错误并捕获输出。编写一段bash脚本，运行如下的脚本直到它出错，将它的标准输出和标准错误流记录到文件，并在最后输出所有内容。加分项：报告脚本在失败前共运行了多少次。

```
#!/usr/bin/env bash

n=$(( RANDOM % 100 ))

if [[ n -eq 42 ]]; then
    echo "Something went wrong"
    >&2 echo "The error was using magic numbers"
    exit 1
fi

echo "Everything went according to plan"
```

4. 本节课我们讲解了 `find` 命令的 `-exec` 参数非常强大，它可以对我们查找的文件进行操作。但是，如果我们要对所有文件进行操作呢？例如创建一个zip压缩文件？我们已经知道，命令行可以从参数或标准输入接受输入。在用管道连接命令时，我们将标准输出和标准输入连接起来，但是有些命令，例如 `tar` 则需要从参数接受输入。这里我们可以使用 `xargs` 命令，它可以使用标准输入中的内容作为参数。例如 `ls | xargs rm` 会删除当前目录中的所有文件。
您的任务是编写一个命令，它可以递归地查找文件夹中所有的HTML文件，并将它们压缩成zip文件。注意，即使文件名中包含空格，您的命令也应该能够正确执行（提示：查看 `xargs` 的参数 `-d`）
5. (进阶) 编写一个命令或脚本递归的查找文件夹中最近使用的文件。更通用的做法，你可以按照最近的使用时间列出文件吗？

my solution

1.

```
$ ls -ahuGlf
total 104
drwxr-xr-x 16 miewang staff 512B 1 8 12:12 .
drwxr-xr-x  6 miewang staff 192B 1 7 19:28 ../
-rw-r--r--@  1 miewang staff 6.0K 1 7 20:49 .DS_Store
drwxr-xr-x 13 miewang staff 416B 1 7 20:42 .git/
-rw-r--r--  1 miewang staff 12K 1 8 12:12 .marco.sh.swp
-rw-r--r--@  1 miewang staff 441B 1 8 11:10 README.md
-rw-r--r--  1 miewang staff 5B 1 8 11:10 demo.txt
-rw-r--r--  1 miewang staff 0B 1 8 10:36 foo
-rw-r--r--  1 miewang staff 0B 1 8 10:36 foo2
-rw-r--r--  1 miewang staff 0B 1 8 10:36 foo3
-rw-r--r--  1 miewang staff 0B 1 8 10:36 foo4
-rw-r--r--  1 miewang staff 126B 1 8 12:09 marco.sh
-rw-r--r--  1 miewang staff 35B 1 8 11:10 mcd.sh
-rw-r--r--  1 miewang staff 49B 1 8 11:55 pwd.txt
drwxr-xr-x  2 miewang staff 64B 1 8 11:10 test/
-rw-r--r--@  1 miewang staff 10K 1 8 12:12 🐱MIT The Missing Semester of Your CS Education.md
```

2.

```
~/Documents/2021/My-winter-vacation on [?]master! ⌚ 11:55:06
$ cat pwd.txt
/Users/miewang/Documents/2021/My-winter-vacation
```

```
~/Documents/2021/My-winter-vacation on [?]master! ⌚ 11:55:12
```

```
$ cat marco.sh
```

```
#!/bin/sh
marco(){
```

```
    echo $(pwd) > pwd.txt
}
```

```
#!/bin/sh
```

```
marco(){
```

```
    echo $(pwd) > pwd.txt
```

```
}
```

```
polo(){
```

```
    cat /Users/miewang/Documents/2021/My-winter-vacation/pwd.txt | cd\
```

但我这个polo死活也没出来，感觉自己没写错

为什么要这样。。。我可以直接用个变量保存(ps 第一次忘了变量赋值左右不能有空格)

```
#!/usr/local/bin/zsh

var=$(pwd)
|
polo(){
    cd $var
}

~
```

3.

```
1 #!/bin/sh
2 count=0
3 while true; do
4     ./test.sh &> tmp
5     if [[ "$?" -ne 0 ]]; then
6         break
7     fi
8     count=$((count+1))
9 done
10 echo "total run times: $count"
11 cat tmp
```

./demo.sh: line 5: [[0: command not found
./demo.sh: line 5: [[1: command not found
./demo.sh: line 5: [[0: command not found
^Z
[8] + 29421 suspended ./demo.sh

~/Documents/2021/My-winter-vacation on [?]master! 13:45
\$ vim demo.sh

~/Documents/2021/My-winter-vacation on [?]master! 13:45
\$./demo.sh
total run times: 148
Something went wrong
The error was using magic numbers

写这个脚本真的是写了好久

首先注意的点就是 while 的语法

Do 和;之间要有空格

而shell中变量赋值不可以有空格

&>是标准和错误输出

if的[]和代码要严格分

4.

```
$ find . -name "*.html" -print0 | xargs -0 tar -cvf 1.tar  
a ./1123.html  
a ./1123/53.html  
a ./1123/465.html  
a ./1123/1223.html  
a ./1123/123.html  
a ./7789.html  
a ./12.html  
a ./2.html  
a ./456.html  
a ./2 3.html
```

find -print0和xargs -0原理及用法

平常我们经常把find和xargs搭配使用，例如：

```
find . -name "*.txt" | xargs rm
```

但是这个命令如果遇到文件名里有空格或者换行符，就会出错。因为xargs识别字符段的标识是空格或者换行符，所以如果一个文件名里有空格或者换行符，xargs就会把它识别成两个字符串，自然就出错了。

这时候就需要-print0和-0了。

find -print0表示在find的每一个结果之后加一个NULL字符，而不是默认加一个换行符。find的默认在每一个结果后加一个'\n'，所以输出结果是一行一行的。当使用了-print0之后，就变成一行了

```
[Yihuas-MacBook-Pro:Documents yliu$ find . -name "*.txt"  
./result.txt  
./test.txt  
[Yihuas-MacBook-Pro:Documents yliu$ find . -name "*.txt" -print0  
./result.txt./test.txtYihuas-MacBook-Pro:Documents yliu$
```

然后xargs -0表示xargs用NULL来作为分隔符。这样前后搭配就不会出现空格和换行符的错误了。选择NULL做分隔符，是因为一般编程语言把NULL作为字符串结束的标志，所以文件名不可能以NULL结尾，这样确保万无一失。

所以比较我们推荐的比较保险的命令是

```
find . -name "*.txt" -print0 | xargs -0 rm
```

5. ls -R | head -n 2

3.Vim

1.vim的哲学

在编程的时候，你会把大量时间花在阅读/编辑而不是在写代码上。所以，Vim是一个多模态编辑器：它对于插入文字和操纵文字有不同的模式。Vim既是可编程的（可以使用Vimscript或者像Python一样的其他程序语言），Vim的接口本身也是一个程序语言：键入操作（以及其助记名）是命令，这些命令也是可组合的。Vim避免了使用鼠标，因为那样太慢了；Vim甚至避免用上下左右键因为那样需要太多的手指移动。

这样的设计哲学的结果是一个能跟上你思维速度的编辑器。

2. 编辑模式

- 正常模式：在文件中四处移动光标进行修改
- 插入模式：插入文本
- 替换模式：替换文本
- 可视化（一般，行，块）模式：选中文本块
- 命令模式：用于执行命令

不同的操作模式，键盘敲击的含义不同。当 `x` 在插入模式会插入 `x`，但是在正常模式会删除当前光标的字体，可是模式会删除选中文块。

你可以按下 `<esc>`（逃脱键）从任何其他模式返回正常模式。在正常模式，键入 `i` 进入插入模式，`R` 进入替换模式，`v` 进入可视（一般）模式，`V` 进入可视（行）模式，`<c-v>`（`Ctrl-V`，有时也写作 `^V`）进入可视（块）模式，`:` 进入命令模式。

3. 基本操作

`:q` 退出

`:w` 保存

`:wq` 保存退出

`:e` 打开要编辑的文件

`:ls` 显示打开的缓存

`:help {标题}` 打开帮助文档

`:help :w` 打开 `w` 命令的帮助文档

`:help w` 打开 `w` 移动的帮助文档

移动

多数时候你会在正常模式下，使用移动命令在缓存中导航。在 Vim 里面移动也被成为“名词”，因为它们指向文字块。

- 基本移动：`h j k l`（左，下，上，右）
- 词：`w`（下一个词），`b`（词初），`e`（词尾）
- 行：`o`（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）
- 翻页：`Ctrl-u`（上翻），`Ctrl-d`（下翻）
- 文件：`gg`（文件头），`G`（文件尾）
- 行数：`:{行数}<CR>` 或者 `{行数}G`（`{行数}` 为行数）
- 杂项：`%`（找到配对，比如括号或者 `/* */` 之类的注释对）
- 查找：

f{字符}

,

t{字符}

,

F{字符}

,

T{字符}

- 查找/到 向前/向后 在本行的{字符}
- , / ; 用于导航匹配
- 搜索: /{正则表达式}, n / N 用于导航匹配

选择

可视化模式:

- 可视化
- 可视化行
- 可视化块

可以用移动命令来选中。

编辑

所有你需要用鼠标做的事，你现在都可以用键盘：采用编辑命令和移动命令的组合来完成。这就是 Vim 的界面开始看起来像一个程序语言的时候。Vim 的编辑命令也被称为“动词”，因为动词可以施动于名词。

i

进入插入模式

- 但是对于操纵/编辑文本，不单想用退格键完成
- o / O 在之上/之下插入行

d{移动命令}

删除 {移动命令}

- 例如, `dw` 删除词, `d$` 删除到行尾, `d0` 删除到行头。

c{移动命令}

改变 {移动命令}

- 例如, `cw` 改变词
- 比如 `d{移动命令}` 再 `i`
- `x` 删除字符 (等同于 `dl`)
- `s` 替换字符 (等同于 `xi`)
- 可视化模式 + 操作
 - 选中文字, `d` 删除 或者 `c` 改变
- `u` 撤销, `<C-r>` 重做
- `y` 复制 / “yank” (其他一些命令比如 `d` 也会复制)
- `p` 粘贴
- 更多值得学习的: 比如 `~` 改变字符的大小写

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` 向前移动三个词
- `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。修饰语有 `i`, 表示“内部”或者“在内”, 和 `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窗, 包括周围的单引号

课后练习

1. 完成 `vimtutor`。备注: 它在一个 `80x24` (80列, 24行) 终端窗口看起来最好。
2. 下载我们的 [基本 vimrc](#), 然后把它保存到 `~/.vimrc`。通读这个注释详细的文件 (用 Vim!) , 然后观察 Vim 在这个新的设置下看起来和使用起来有哪些细微的区别。
3. 安装和配置一个插件: `ctrlp.vim`
 1. 用 `mkdir -p ~/.vim/pack/vendor/start` 创建插件文件夹
 2. 下载这个插件: `cd ~/.vim/pack/vendor/start; git clone https://github.com/ctrlpvim/ctrlp.vim`

3. 读这个插件的 [文档](#)。尝试用 CtrlP 来在一个工程文件夹里定位一个文件， 打开 Vim, 然后用 Vim 命令控制行开始 `:CtrlP`.
4. 自定义 CtrlP：添加 [configuration](#) 到你的 `~/.vimrc` 来用按 Ctrl-P 打开 CtrlP
4. 练习使用 Vim, 在你自己的机器上重做 [演示](#)。
5. 下个月用 Vim 做你 [所有](#) 文件编辑。每当不够高效的时候, 或者你感觉“一定有一个更好的方式”，尝试求助搜索引擎，很有可能有一个更好的方式。如果你遇到难题，来我们的答疑时间或者给我们发邮件。
6. 在你的其他工具中设置 Vim 快捷键（见上面的操作指南）。
7. 进一步自定义你的 `~/.vimrc` 和安装更多插件。
8. (高阶) 用 Vim 宏将 XML 转换到 JSON ([例子文件](#))。尝试着先完全自己做，但是在你卡住的时候可以查看上面 [宏](#) 章节。

My solution

因为目前需求度不高，只掌握了常见的几种模式的切换以及初步使用，当经常使用时再来补这些练习题

4.Data Wrangling

1.Concept

当我们在使用管道命令 | 的时候，实际上我们就在进行某种形式的数据整理，本质就是从一种形式转换为另一种形式

```
$ log show | grep ssd
2021-01-04 09:16:40.753307+0800 0x328      Error      0x0
    7 corebrightnessssd: (CoreDisplay) [com.apple.CoreDisplay:default]
<decode: missing data>
2021-01-04 09:16:40.822205+0800 0x328      Error      0x0
    7 corebrightnessssd: (CoreDisplay) [com.apple.CoreDisplay:default]
<decode: missing data>
2021-01-04 09:16:40.873066+0800 0x328      Error      0x0
    7 corebrightnessssd: (CoreDisplay) [com.apple.CoreDisplay:default]
<decode: missing data>
2021-01-04 09:16:40.923999+0800 0x328      Error      0x0
    7 corebrightnessssd: (CoreDisplay) [com.apple.CoreDisplay:default]
<decode: missing data>
2021-01-04 09:16:40.974597+0800 0x328      Error      0x0
    7 corebrightnessssd: (CoreDisplay) [com.apple.CoreDisplay:default]
<decode: missing data>
```

在linux上使用journalctl

可以看到我们把系统日志 通过grep查找出来

2.基本操作

可以看到即使我们通过grep来稍微过滤了一下，数据还是很冗杂

我们首先可以使用 less 打过ctf的都知道 如果cat 被ban了，可以考虑使用less的好处是能够翻页浏览文本

```
$ log show | grep ssh | grep "apple" > ssh.log
```

```
2021-01-04 09:00:42.073066+0000 0x328      Error      0x0          122    7  c
ghtnssd: (com.apple.CoreDisplay:default) [ERROR] - <decode: missing data>
2021-01-04 09:01:42.073066+0000 0x328      Error      0x0          122    7  c
ghtnssd: (com.apple.CoreDisplay:default) [ERROR] - <decode: missing data>
2021-01-04 09:02:42.073066+0000 0x328      Error      0x0          122    7  c
ghtnssd: (com.apple.CoreDisplay:default) [ERROR] - <decode: missing data>
2021-01-04 09:03:42.073066+0000 0x328      Error      0x0          122    7  c
ghtnssd: (com.apple.CoreDisplay:default) [ERROR] - <decode: missing data>
2021-01-04 09:04:42.073066+0000 0x328      Error      0x0          122    7  c
ghtnssd: (com.apple.CoreDisplay:default) [ERROR] - <decode: missing data>
2021-01-04 09:05:42.073066+0000 0x328      Error      0x0          122    7  c
ghtnssd: (com.apple.CoreDisplay:default) [ERROR] - <decode: missing data>
2021-01-04 09:06:42.073066+0000 0x328      Error      0x0          122    7  c
ghtnssd: (com.apple.CoreDisplay:default) [ERROR] - <decode: missing data>
2021-01-04 09:07:42.073066+0000 0x328      Error      0x0          122    7  c
ghtnssd: (com.apple.CoreDisplay:default) [ERROR] - <decode: missing data>
2021-01-04 09:08:42.073066+0000 0x328      Error      0x0          122    7  c
ghtnssd: (com.apple.CoreDisplay:default) [ERROR] - <decode: missing data>
2021-01-04 09:09:42.073066+0000 0x328      Error      0x0          122    7  c
ghtnssd: (com.apple.CoreDisplay:default) [ERROR] - <decode: missing data>
2021-01-04 09:10:42.073066+0000 0x328      Error      0x0          122    7  c
ghtnssd: (com.apple.CoreDisplay:default) [ERROR] - <decode: missing data>
2021-01-04 09:11:42.073066+0000 0x328      Error      0x0          122    7  c
ghtnssd: (com.apple.CoreDisplay:default) [ERROR] - <decode: missing data>
2021-01-04 09:12:42.073066+0000 0x328      Error      0x0          122    7  c
ghtnssd: (com.apple.CoreDisplay:default) [ERROR] - <decode: missing data>
2021-01-04 09:13:42.073066+0000 0x328      Error      0x0          122    7  c
ghtnssd: (com.apple.CoreDisplay:default) [ERROR] - <decode: missing data>
2021-01-04 09:14:42.073066+0000 0x328      Error      0x0          122    7  c
ghtnssd: (com.apple.CoreDisplay:default) [ERROR] - <decode: missing data>
```

然后学习一个命令 sed

```
less ssh.log | grep "apple" | sed 's/apple/xiaomi/'
```

语法s/regex/substitution 前一个是查找 后一个替换

希望库克不会告我🙏 (233

```
xiaomi.passd.listener.resumed.
xiaomi.passkit:Connections] Adding con
xiaomi.cards.all-access
xiaomi.passes.add-silently
xiaomi.payment.all-access
xiaomi.application-identifier
xiaomi.passkit:Connections] PDPassLibr
xiaomi.passd.listener.resumed.
```

我们已经有很多的编程经验

正则表达式非常常见也非常有用，值得您花些时间去理解它。让我们从这一句正则表达式开始学习：

`/.*Disconnected from /`。正则表达式通常以（尽管并不总是）`/`开始和结束。大多数的 ASCII 字符都表示它们本来的含义，但是有一些字符确实具有表示匹配行为的“特殊”含义。不同字符所表示的含义，根据正则表达式的实现方式不同，也会有所变化，这一点确实令人沮丧。常见的模式有：

- `.` 除空格之外的“任意单个字符”
- `*` 匹配前面字符零次或多次
- `+` 匹配前面字符一次或多次
- `[abc]` 匹配 `a`, `b` 和 `c` 中的任意一个
- `(RX1|RX2)` 任何能够匹配 `RX1` 或 `RX2` 的结果
- `^` 行首
- `$` 行尾

这里要简单说下贪婪匹配和非贪婪匹配

*和+一般都是贪婪模式，尽可能多的匹配 加上? 可以变成非贪婪模式

但是sed不支持？所以我们可以使用perl

```
perl -pe 's/.*?Disconnected from //'
```

我们可以尝试一下几个例子

```
~/Documents/2021/My-winter-vacation on ?master! 9:33:57
$ echo 'aba' | sed 's/[ab]//'
ba

~/Documents/2021/My-winter-vacation on ?master! 9:34:03
$ echo 'aba' | sed 's/[a]//'
ba

~/Documents/2021/My-winter-vacation on ?master! 9:35:44
$ echo 'aba' | sed 's/[b]//'
aa

~/Documents/2021/My-winter-vacation on ?master! 9:37:48
$ echo 'abcababc' | sed -E 's/(ab|bc)*//g'
cc
```

3.课后习题

1. 统计words文件 (`/usr/share/dict/words`) 中包含至少三个`a`且不以`'s`结尾的单词个数。这些单词中，出现频率前三的末尾两个字母是什么？`sed`的`y`命令，或者`tr`程序也许可以帮你解决大小写的问题。共存在多少种词尾两字母组合？还有一个很有挑战性的问题：哪个组合从未出现过？

- 进行原地替换听上去很有诱惑力，例如：`sed s/REGEX/SUBSTITUTION/ input.txt > input.txt`。但是这并不是一个明智的做法，为什么呢？还是说只有 `sed` 是这样的？查看 `man sed` 来完成这个问题
- 找出您最近十次开机的开机时间平均数、中位数和最长时间。在Linux上需要用到

```
journalctl
```

，而在 macOS 上使用

```
log show
```

。找到每次起到开始和结束时的时间戳。在Linux上类似这样操作：

```
Logs begin at ...
```

和

```
systemd[577]: Startup finished in ...
```

在 macOS 上, [查找](#):

```
==== system boot:
```

和

```
Previous shutdown cause: 5
```

- 查看之前三次重启启动信息中不同的部分 (参见 `journalctl` 的 `-b` 选项)。将这一任务分为几个步骤，首先获取之前三次启动的启动日志，也许获取启动日志的命令就有合适的选项可以帮助您提取前三次启动的日志，亦或者您可以使用 `sed '0,/STRING/d'` 来删除 `STRING` 匹配到的字符串前面的全部内容。然后，过滤掉每次都不同的部分，例如时间戳。下一步，重复记录输入行并对其计数(可以使用 `uniq`)。最后，删除所有出现过3次的内容 (因为这些内容上三次启动日志中的重复部分)。

4.My Solution

1.

```
$ cat /usr/share/dict/words | grep -E "(.*a){3}" | grep -Ev "\'s\$" | wc -l  
7047
```

```
/usr/share/dict/words | grep -E "(.*a){3}" | grep -Ev "\'s\$" | sed -E 's/.*/(..)$/\1/' | sort |uniq -c | wc -l  
150
```

```
$ cat /usr/share/dict/words | grep -E "(.*a){3}" | grep -Ev "\'s\$" | sed -E 's/.*/(..)$/\1/' | sort |uniq -c | sort -rn | head -n3  
1031 al  
752 ia  
682 an
```

2. sed -i 's /find/replace/g' filename

3.没拿出mac的解 附上linux的吧

4. Find your average, median, and max system boot time over the last ten boots

Answer: using R to calculate the statistics:

```
journalctl  
| grep 'systemd\[1\]: Startup finished in'  
| tail -n10  
| sed -E "s/.* \\\(userspace\\\) = (.*)s\\.\$/\\1/"  
| R --slave -e 'x <- scan(file="stdin", quiet=TRUE); summary(x)'
```

5. Look for boot messages that are not shared between your past three reboots.

Answer:

```
{journalctl -b & journalctl -b -1 & journalctl -b -2}  
| sed -E "s/^.*vostro //"  
| uniq -c  
| sort -nk1,1  
| grep "\s[1-2] .*\]::.*$"
```

4.未做

5.Command-line environment

1.job control

```
~/Documents/2021/My-winter-vacation on master! 9:18:40  
$ sleep 20  
^C  
  
~/Documents/2021/My-winter-vacation on master! 9:18:46  
$
```

Ctrl +c 可以中断当前进程

当你使用ctrl +c的时候 会发送signal t 信号来打断

但我们也可以在执行脚本的时候处理这个异常 不进行退出

```
~/cmd >>> python sigint.py
33^C
I got a SIGINT, but I am not stopping
149
```

然后我们使用ctrl +\ 发送SIGQUIT 不是同一个信号来中断掉

ctrl +z 暂停

通过jobs看我暂停的

```
~/Documents/2021/My-winter-vacation on ?master! 6
$ jobs
[1] - suspended sleep 20
[2] + suspended man signal
```

然后用bg 运行

```
~/Documents/2021/My-winter-vacation on ?master! 9:26:35
$ bg %2
[2] - 2747 continued man signal
[2] + 2747 suspended (tty output) man signal
```

当我们用kill -KILL的时候 则可以杀死进程

```
~/Documents/2021/My-winter-vacation on ?master! 9:27:33
$ jobs
[1] - suspended sleep 20
[2] + suspended (tty output) man signal

~/Documents/2021/My-winter-vacation on ?master! 9:27:39
$ kill -KILL %1
[1] - 2685 killed sleep 20

~/Documents/2021/My-winter-vacation on ?master! 9:28:54
$ kill -KILL %2
[2] + 2747 killed man signal

~/Documents/2021/My-winter-vacation on ?master! 9:28:57
$ jobs
```

2.Terminal multiplexer

-Sessions

-windows

-panes

用几个tmux 进行分离

The screenshot shows a tmux session with the following structure:

- (1) -> 0: zsh* (1 panes) "miewangDeMac"
- (2) - 1: 2 windows
- (3) -> 0: zsh- (1 panes) "miewangDeMac"
- (4) -> 1: zsh* (1 panes) "miewangDeMac" (highlighted)
- (5) - 2: 1 windows
- (6) -> 0: zsh* (1 panes) "miewangDeMac"
- (7) - 3: 2 windows (attached)
- (8) -> 0: zsh* (1 panes) "miewangDeMac"
- (9) -> 1: zsh- (1 panes) "miewangDeMac"

Below the list, there is a terminal prompt:

```
1 (sort: index)
$
```

At the bottom left, it says **~ 10:10:49**. On the right, there is a small square icon containing the number **0**.

tmux的上手很快

创建窗口：

Ctrl+b c

查看窗口列表

Ctrl+b w

切换到指定窗口，只需要先按下Ctrl-b，然后再按下想切换的窗口所对应的数字。

Ctrl+b 0

切换到下一个窗口

Ctrl+b n

切换到上一个窗口

Ctrl+b p

在相邻的两个窗口里切换

Ctrl+b l

重命名窗口

Ctrl+b ,

在多个窗口里搜索关键字

Ctrl+b f

删除窗口

Ctrl+b &



```
tmux new -s foo # 新建名称为 foo 的会话  
tmux ls # 列出所有 tmux 会话  
tmux a # 恢复至上一次的会话  
tmux a -t foo # 恢复名称为 foo 的会话，会话默认名称为数字  
tmux kill-session -t foo # 删除名称为 foo 的会话  
tmux kill-server # 删除所有的会话
```



浏览各个窗口

3.alias

起别名

```
[~] $ alias ll="ls -lah"

~ 10:21:45
[~] ll
total 992
drwxr-xr-x+ 83 miewang staff 2.6K 1 12 10:21 .
drwxr-xr-x 5 root admin 160B 1 1 2020 ..
drwx----- 4 miewang staff 128B 2 16 2020 .BurpSuite
-r----- 1 miewang staff 9B 12 15 22:56 .CFUserTextE
-rw-r--r--@ 1 miewang staff 36K 1 10 21:33 .DS_Store
drwxr-xr-x 3 miewang staff 96B 10 11 2019 .Shadowsocks
drwxr-xr-x 8 miewang staff 256B 2 29 2020 .Shadowsocks
drwx----- 5 miewang staff 160B 1 10 21:12 .Trash
drwxr-xr-x 2 miewang staff 64B 8 3 20:32 .android
-rw----- 1 miewang staff 12K 1 7 09:04 bash_history
```

我觉得比较方便的就是git的时候

```
[~] ~/alias gu="git push -u origin master"
```

注意等号两边不能有空格

你使用bash的时候 当你关闭终端的时候，别名会消失

所以要 修改配置文件

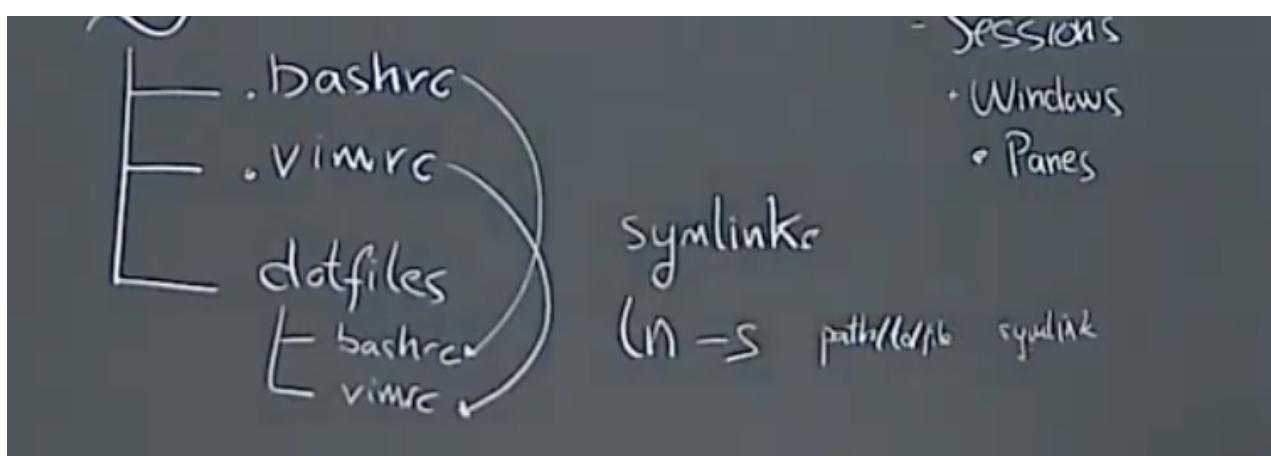
而我用的是zsh 无需source

4. 配置shell

因为我已经配置好了我的shell,所以这部分我就没什么记录

比较重要的点就是.文件,

在linux 或者mac下 以.开头rc结尾的配置文件



在这里简单介绍了软链接

5. Remote machines

主要介绍的就是ssh

国内可以考虑阿里云或者腾讯云的学生计划

9块或者10块一个月的主机

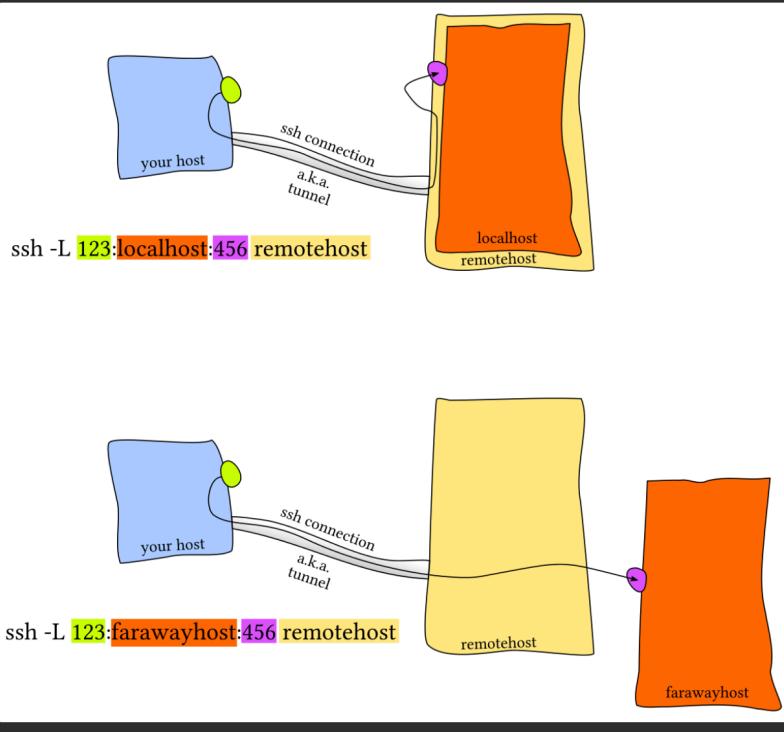
安全人员很多都需要一个远程服务器来反弹shell。或者是搭建自己的博客，开一个docker之类的。这里面个人推荐finalshell 或者xshell 都是比较好的图形化界面的管理软件

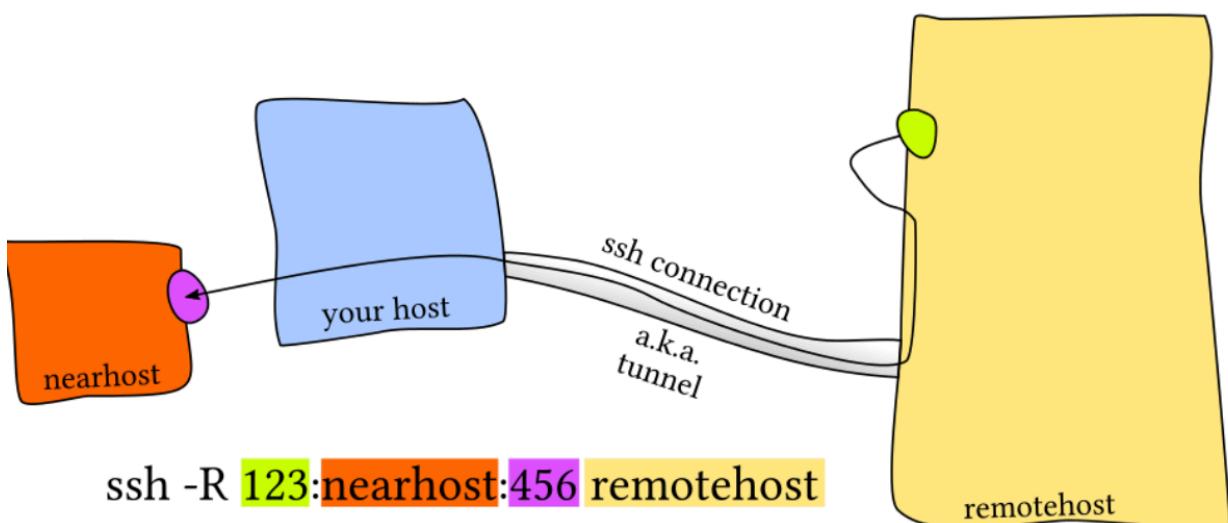
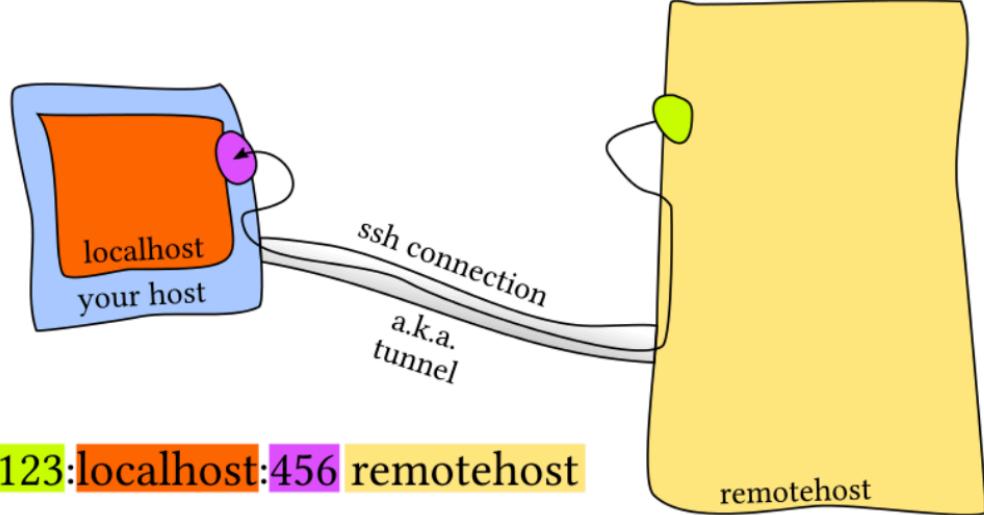
这里面一个蛮有意思的问题 如果处理远程的tmux 可以换一些键位绑定

比如ctrl+b -> ctrl +a

端口转发

本地端口转发





常见的情景是使用本地端口转发，即远端设备上的服务监听一个端口，而您希望在本地设备上的一个端口建立连接并转发到远程端口上。例如，我们在远端服务器上运行 Jupyter notebook 并监听 8888 端口。然后，建立从本地端口 9999 的转发，使用 `ssh -L 9999:localhost:8888 foobar@remote_server`。这样只需要访问本地的 `localhost:9999` 即可。

一个比较有意思的命令

本机使用过最多的10条命令

```
history | awk '{\$1="";print substr(\$0,2)}' | sort | uniq -c | sort -n | tail -n 10
```

6.Summary

课后练习偏实操，大家可以试一遍，搜索引擎可以解决100%的问题

5.Version Control(git)

版本控制的优点 基于多分支并行开发

进行版本控制的方法很多。Git 拥有一个经过精心设计的模型，这使其能够支持版本控制所需的所有特性，例如维护历史记录、支持分支和促进协作。

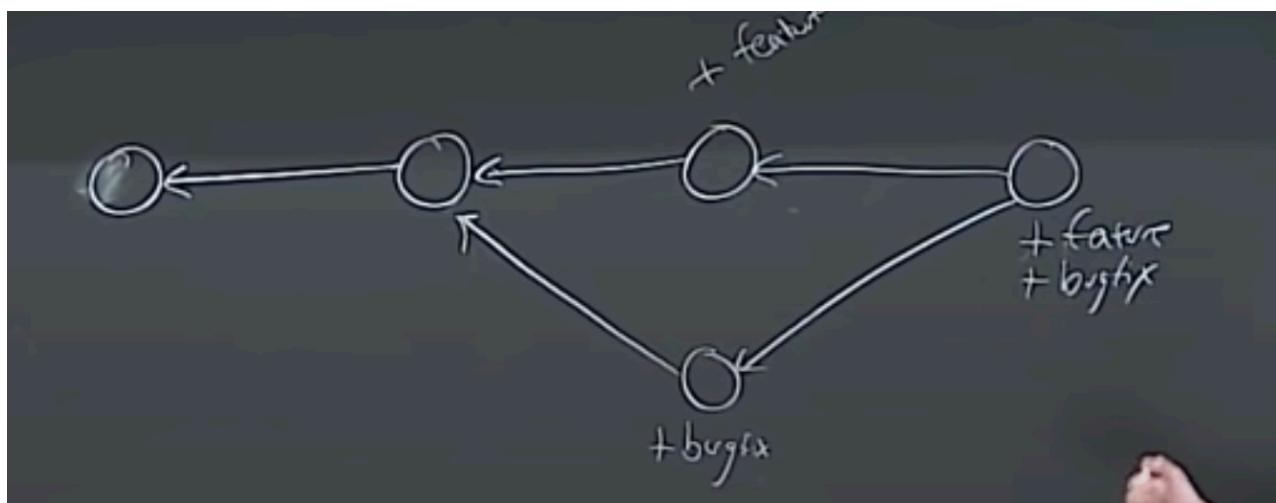
快照

Git 将顶级目录中的文件和文件夹作为集合，并通过一系列快照来管理其历史记录。在Git的术语里，文件被称作Blob对象（数据对象），也就是一组数据。目录则被称之为“树”，它将名字与Blob对象或树对象进行映射（使得目录中可以包含其他目录）。快照则是被追踪的最顶层的树。例如，一个树看起来可能是这样的：

```
<root> (tree)
|
+- foo (tree)
|   |
|   + bar.txt (blob, contents = "hello world")
|
+- baz.txt (blob, contents = "git is wonderful")
```

这个顶层的树包含了两个元素，一个名为“foo”的树(它本身包含了一个blob对象“bar.txt”），以及一个对blob对象“baz.txt”。

关联快照与分支



git中这种快照被称为提交，当我们想要同时开发两个不同的特性，他们直接是相互独立的，我们首先要用不同的分支来表示，等开发完成后，将分支合并为一个新的提交，这个提交就含有以前的特性

But 合并时可能会出现冲突或者各种未知的错误，所以解决原因还得看自己

git的数据模型

这里面用伪代码表示

数据模型及其伪代码表示

以伪代码的形式来学习 Git 的数据模型，可能更加清晰：

```
// 文件就是一组数据
type blob = array<byte>

// 一个包含文件和目录的目录
type tree = map<string, tree | file>

// 每个提交都包含一个父辈，元数据和顶层树
type commit = struct {
    parent: array<commit>
    author: string
    message: string
    snapshot: tree
}
```

这是一种简洁的历史模型。

对象和内存寻址

Git 中的对象可以是 blob、树或提交：

```
type object = blob | tree | commit
```

Git 在储存数据时，所有的对象都会基于它们的[SHA-1 hash](#)进行寻址。

```
objects = map<string, object>

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

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

Blobs、树和提交都一样，它们都是对象。当它们引用其他对象时，它们并没有真正的在硬盘上保存这些对象，而是仅仅保存了它们的哈希值作为引用。

例如，[above](#)例子中的树（可以通过`git cat-file -p 698281bc680d1995c5f4caaf3359721a5a58d48d`来进行可视化），看上去是这样的：

```
100644 blob 4448adbf7ecd394f42ae135bbeed9676e894af85      baz.txt
040000 tree c68d233a33c5c06e0340e4c224f0afca87c8ce87      foo
```

树本身会包含一些指向其他内容的指针，例如`baz.txt` (blob) 和`foo` (树)。如果我们用`git cat-file -p 4448adbf7ecd394f42ae135bbeed9676e894af85`，即通过哈希值查看`baz.txt`的内容，会得到以下信息：

```
git is wonderful
```

引用

现在，所有的快照都可以通过它们的 SHA-1 哈希值来标记了。但这也太不方便了，谁也记不住一串 40 位的十六进制字符。

针对这一问题，Git 的解决方法是给这些哈希值赋予人类可读的名字，也就是引用（references）。引用是指向提交的指针。与对象不同的是，它是可变的（引用可以被更新，指向新的提交）。例如，`master` 引用通常会指向主分支的最新一次提交。

```
references = map<string, string>

def update_reference(name, id):
    references[name] = id

def read_reference(name):
    return references[name]

def load_reference(name_or_id):
    if name_or_id in references:
        return load(references[name_or_id])
    else:
        return load(name_or_id)
```

这样，Git 就可以使用诸如“`master`”这样人类刻度的名称来表示历史记录中某个特定的提交，而不需要在使用一长串十六进制字符了。

有一个细节需要我们注意，通常情况下，我们会想要知道“我们当前所在位置”，并将其标记下来。这样当我们创建新的快照的时候，我们就可以知道它的相对位置（如何设置它的“父辈”）。在 Git 中，我们当前的位置有一个特殊的索引，它就是“`HEAD`”。

git命令行是引用的操作

一个比较有意思的命令

git log --all --graph --decorate 可以把commit记录弄成图表的形式

```
* commit 50e348cf9bf772f073682761397400fd5076a5e6 (HEAD)
| Author: T4rn <73047111+T4rnRookie@users.noreply.github.com>
| Date:   Sun Jan 10 12:42:06 2021 +0800
|
|   Update README.md
|
* commit ae8d3019df8e6e7abbfa351fd570891a8d5c6fd1
| Author: T4rn <73047111+T4rnRookie@users.noreply.github.com>
| Date:   Sun Jan 10 12:37:51 2021 +0800
|
|   Delete 2020xiangyun.md
|
* commit 5c7815e813b1bdee816f7aeedd3e56b89ea4f989
| Author: T4rnRookie <root@hackerpoet.com>
| Date:   Sun Jan 10 12:33:31 2021 +0800
|
|   CTFwriteups
|
* commit 00b8f6a670e55fcdbb710239c811070169d2b0dd
| \ Merge: f9dab34 5918e8b
| | Author: T4rnRookie <root@hackerpoet.com>
| | Date:   Sun Jan 10 12:25:20 2021 +0800
```

像这样

还可以用git branch对分支进行操作

比如课程的操作 先用vv查看上游分支的名字

然后新建分支

```
Lecture 6: Version Control (git) (2020)
.../demo > git branch cat
.../demo > git log --all --graph --decorate
* commit fee073115567a5d9eda3e8bb21a0f37c25938e9e (HEAD -> master, cat)
| Author: Anish Athalye <me@anishathalye.com>
| Date:   Thu Jan 23 13:15:59 2020 -0500
|
|   Add animal.py
```

利用git checkout来切换分支

git checkout -b 新建分支并切换

用两个分支并行开发 最后合并

git merge 来进行合并

但是就像我之前说的 merge的时候可能会出现合并冲突

```
.../demo > git merge dog
Auto-merging animal.py
CONFLICT (content): Merge conflict in animal.py
Automatic merge failed; fix conflicts and then commit the result.
```

解决合并冲突的工具 git mergetool

.git/文件夹

```
$ ls .git/
COMMIT_EDITMSG ORIG_HEAD      hooks        logs
FETCH_HEAD       config        index        objects
HEAD            description   info         refs
```

利用git blame 加文件查看修改记录

```
a29c01dc (T4rnRookie 2021-01-07 20:51:00 +0800 2) ## 1. 😊My python note
a29c01dc (T4rnRookie 2021-01-07 20:51:00 +0800 3)
a29c01dc (T4rnRookie 2021-01-07 20:51:00 +0800 4) ## 2. 😊Nankai University information security note
a29c01dc (T4rnRookie 2021-01-07 20:51:00 +0800 5)
a29c01dc (T4rnRookie 2021-01-07 20:51:00 +0800 6) ## 3. 😊Standford Computer Internet
a29c01dc (T4rnRookie 2021-01-07 20:51:00 +0800 7)
a29c01dc (T4rnRookie 2021-01-07 20:51:00 +0800 8) ## 4. 😊MIT The Missing Semester of Your CS Education
a29c01dc (T4rnRookie 2021-01-07 20:51:00 +0800 9)
a29c01dc (T4rnRookie 2021-01-07 20:51:00 +0800 10) ## 5. 😊CTF writeups
a29c01dc (T4rnRookie 2021-01-07 20:51:00 +0800 11)
a29c01dc (T4rnRookie 2021-01-07 20:51:00 +0800 12) ## 6. 😊Some Attack lab
a29c01dc (T4rnRookie 2021-01-07 20:51:00 +0800 13)
50e348cf (T4rn 2021-01-10 12:42:06 +0800 14) SO, this winter vacation I want to learn something new. and I will upload some notes and something fun that I think.
a29c01dc (T4rnRookie 2021-01-07 20:51:00 +0800 15)
50e348cf (T4rn 2021-01-10 12:42:06 +0800 16) Hope you can benifite from them
```

也可以通过git show来查看两个版本的修改记录

```
diff --git a/README.md b/README.md
index 59c8608..02307c5 100644
--- a/README.md
+++ b/README.md
@@ -11,8 +11,8 @@
## 6. 😊Some Attack lab

-SO, this winter vacation I want to learn something new. so I will upload some notes and something fun that I think.
+SO, this winter vacation I want to learn something new. and I will upload some notes and something fun that I think.

-Hope you can also benifite from them
+Hope you can benifite from them

-Tips:Don't forget to wear mask ❤️😊
\ No newline at end of file
+Tips:Don't forget to wear mask ❤️😊
(END)
```

上传代码比如c 的时候，当你编译的过程的一些文件比如.elf可能会跟着上传上去

如果你不想这样的话可以配置.gitignore文件

The screenshot shows a terminal window with a dark background. On the left, there is a vertical list of file names starting with a yellow '1' followed by '.DS_Store'. The rest of the list consists of numerous 'z' characters, indicating they are hidden files. At the bottom of the terminal window, there is a status bar with several pieces of information: '1' in a yellow box, '.gitignore | +' in a grey box, and 'unix | utf-' on the right.

1.部分solution

1.将版本历史可视化并进行探索

```
git show
```

2.是谁最后修改来 README.md文件? (提示: 使用 git log 命令并添加合适的参数)

```
git log -1 --all README.md
```

3最后一次修改_config.yml 文件中 collections: 行时的提交信息是什么? (提示: 使用git blame 和 git show)

```
git blame _config.yml | grep collections: | awk '{print $1}' | xargs git show | head -n6
```

4.与其他的命令行工具一样, Git 也提供了一个名为 ~/.gitconfig 配置文件 (或 dotfile)。请在 ~/.gitconfig 中创建一个别名, 使您在运行 git graph 时, 您可以得到 git log --all --graph --decorate --oneline的输出结果;

就是用alias

```
[alias]
# git log history with all
gra = log --all --graph --decorate
# git log history with oneline
gro = log --all --graph --decorate --oneline
```

5.克隆 [本课程网站的仓库](#), 找找有没有错别字或其他可以改进的地方, 在 GitHub 上发起拉取请求 (Pull Request) ;

```
/Documents/2021/My winter vacation on [master] 12.02.22
$ git clone https://github.com/missing-semester/missing-semester missing
Cloning into 'missing'...
```

上网查了下 pull request的意义

先fork一份 修改自己的 然后发起pull request 原项目者可以merge你们的代码

The screenshot shows a GitHub repository page for 'piaoliangkb / missing-semester-2020'. At the top, there's a navigation bar with links for Code, Issues, Pull requests, Actions, Projects, Wiki, Security, and Insights. Below the navigation, a section titled 'Comparing changes' asks to choose two branches for comparison or to start a new pull request. It also mentions the option to compare across forks. A dropdown menu for the base repository is set to 'base repository: piaoliangkb/missing-semest...'. The dropdown for the head repository is set to 'head repository: T4rnRookie/missing-semeste...'. The dropdown for the compare branch is set to 'compare: master'. At the bottom, a message states 'There isn't anything to compare.' with a small icon.

可以在github上面实现

6.Debugging and profiling

编程只能使代码按照你的想法运行，但是他的耗时或者一些你想法的错误是我们需要额外注意的

最简单方便的debug就是在合适的地方加输出

另外一个方法是使用日志

日志的优点

- 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。

```
$ ls /var/log/
AccessoryVersionInfo.txt  fsck_apfs.log          wifi.log
Bluetooth                  fsck_apfs_error.log    wifi.log.0.bz2
CoreDuet                   fsck_hfs.log          wifi.log.1.bz2
DiagnosticMessages          install.log           wifi.log.10.bz2
apache2                     mDNSResponder        wifi.log.2.bz2
asl                         monthly.out          wifi.log.3.bz2
com.apple.wifiveLOCITY     powermanagement      wifi.log.4.bz2
com.apple.xpc.launchd      ppp                  wifi.log.5.bz2
cups                        shutdown_monitor.log   wifi.log.6.bz2
daily.out                  system.log           wifi.log.7.bz2
displaypolicy               system.log.0.gz       wifi.log.8.bz2
displaypolicyd.stdout.log  uucp                 wifi.log.9.bz2
dm                           vnetlib
```

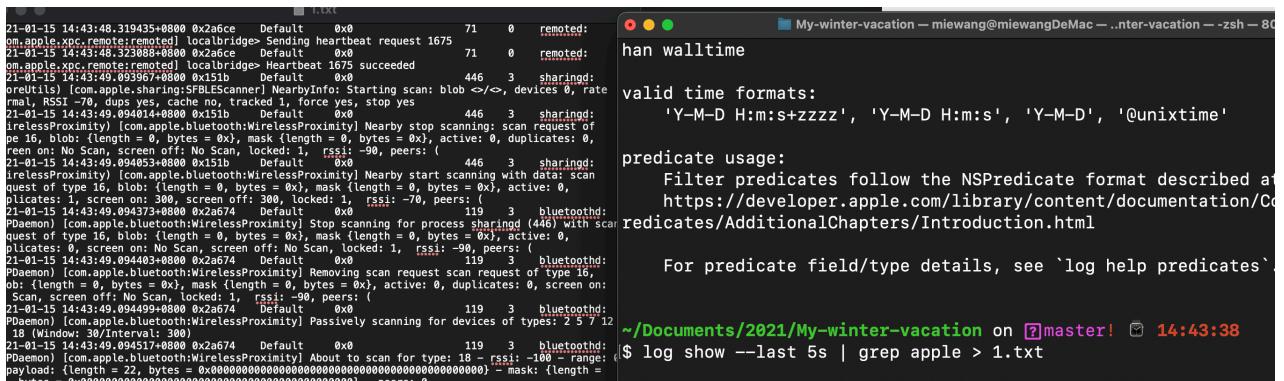
Unix下的日志文件 一般在/var/log下面

mac的一般在/var/log/system.log

mac可以用log show来检查日志内容

而linux可以用journalctl

和前面学的内容来结合



```
21-01-15 14:43:48.319435+0800 0x2a6ce Default 0x0 71 0 remoteds:
on.apple.xpc.remote:remoted localbridge> Sending heartbeat request 1675
21-01-15 14:43:48.323088+0800 0x2a6ce Default 0x0 71 0 remoteds:
on.apple.xpc.remote:remoted localbridge> Heartbeat 1675 succeeded
21-01-15 14:43:49.093967+0800 0x151b Default 0x0 446 3 sharingd:
oreutis:[com.apple.sharing:SFBLEScanner] Nearby start scan: blob <->, devices 0, rate
RSSI: 0, force scan off, stop yes
21-01-15 14:43:49.094014+0800 0x151b Default 0x0 446 3 sharingd:
iorelessproximity:[com.apple.bluetooth:WirelessProximity] Nearby stop scanning: scan request of
pe 16, blob: {length = 0, bytes = 0x}, mask {length = 0, bytes = 0x}, active: 0, duplicates: 0,
reen on: No Scan, screen off: No Scan, locked: 1, rssl: -90, peers: (
21-01-15 14:43:49.094053+0800 0x151b Default 0x0 446 3 sharingd:
iorelessproximity:[com.apple.bluetooth:WirelessProximity] Nearby start scanning with data: scan
quest of type 16, blob: {length = 0, bytes = 0x}, mask {length = 0, bytes = 0x}, active: 0,
duplicates: 0, screen on: No Scan, screen off: No Scan, locked: 1, rssl: -70, peers: (
21-01-15 14:43:49.094073+0800 0x2a674 Default 0x0 119 3 bluetoothhd:
PDaemon:[com.apple.bluetooth:WirelessProximity] Stop scanning for process sharingd (446) with scan
quest of type 16, blob: {length = 0, bytes = 0x}, mask {length = 0, bytes = 0x}, active: 0,
duplicates: 0, screen on: No Scan, screen off: No Scan, locked: 1, rssl: -90, peers: (
21-01-15 14:43:49.094093+0800 0x2a674 Default 0x0 119 3 bluetoothhd:
PDaemon:[com.apple.bluetooth:WirelessProximity] Removing scan request scan request of type 16,
blob: {length = 0, bytes = 0x}, mask {length = 0, bytes = 0x}, active: 0, duplicates: 0, screen on:
Scan quest off: No Scan, locked: 1, rssl: -100, peers: (
21-01-15 14:43:49.094109+0800 0x2a674 Default 0x0 119 3 bluetoothhd:
PDaemon:[com.apple.bluetooth:WirelessProximity] Passively scanning for devices of types: 2 5 7 12
18 (Window: 30/Interval: 300)
21-01-15 14:43:49.094517+0800 0x2a674 Default 0x0 119 3 bluetoothhd:
PDaemon:[com.apple.bluetooth:WirelessProximity] About to scan for type: 18 - rssl: -100 - range: (
payload: {length = 22, bytes = 0x0000000000000000000000000000000000000000000000000000000000000000} - mask: {length =
bytes = 0x0000000000000000000000000000000000000000000000000000000000000001} - peers: 0
~/Documents/2021/My-winter-vacation on master! 14:43:38
```

当然我们也支持编程语言或者是terminal 来写进系统日志

利用Logger

```
~/Documents/2021/My-winter-vacation on master! 14:49:04
$ logger "hello"

~/Documents/2021/My-winter-vacation on master! 14:49:39
$ log show --last 1m | grep "hello"
2021-01-15 14:49:13.509768+0800 0x2a96c      Default      0x0
    0    QQ: (libboringssl.dylib) [com.apple.network:boringssl] boringss
_info_handler(1836) [C2970.1:4][0x7fe9464f3ef0] Client handshake state:
nt read_server_hello
2021-01-15 14:49:13.690145+0800 0x2b667      Default      0x0
    0    QQ: (libboringssl.dylib) [com.apple.network:boringssl] boringss
_info_handler(1836) [C2970.1:4][0x7fe9464f3ef0] Client handshake state:
nt read_server_hello_done
```

利用现成的debugger

比如python中可以使用ipdb

pwn师傅经常使用的gdb调试工具

静态分析

有些时候 你不需要执行代码，就能用眼睛debugger 这时候静态分析就会帮助我们

比如

```
import time

def foo():
    return 42

for foo in range(5):
    print(foo)
bar = 1
bar *= 0.2
time.sleep(60)
print(baz)
```

当我们执行的时候 会崩溃掉，当我们使用python的静态分析工具

Pyflakes

```
~/Documents/2021/My-winter-vacation on master! 14:49:04
KeyboardInterrupt
~/p/m/deb >>> pyflakes
lint.py:6: redefinition of unused 'foo' from line 3
lint.py:11: undefined name 'baz'
```



还可以利用mypy进行类型检查



```
~/p/m/deb >>> mypy lint.py
lint.py:6: error: Incompatible types in assignment (expression has type "float", variable has type "Callable[[], Any]")
lint.py:9: error: Incompatible types in assignment (expression has type "float", variable has type "int")
lint.py:11: error: Name 'baz' is not defined
```

在shell那节我们也说过shellcheck

来检查shell脚本

debugger在web开发的时候 比如审查元素

或者是wireshark抓包

Profiling

time

```
import time, random
n = random.randint(1, 10) * 100

# 获取当前时间
start = time.time()

# 执行一些操作
print("Sleeping for {} ms".format(n))
time.sleep(n/1000)

# 比较当前时间和起始时间
print(time.time() - start)

# Output
# Sleeping for 500 ms
# 0.5713930130004883
```

一般来说我们计算一个程序运行的时间，只需要开始和结束减掉就可以

所以 用户时间 + 系统时间代表了消耗的实际cpu

- 真实时间 - 从程序开始到结束流失掉到真实时间，包括其他进程到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。

比如我们执行一个 curl

```
$ time curl https://www.baidu.com &> /dev/null
curl https://www.baidu.com &> /dev/null  0.01s user 0.00s sys
                                         tem 8% cpu 0.219 total
```

性能分析工具 (profilers)

一般性能分析工具指的都是cpu，大多数编译器都有基于命令行的分析器

python中我们可以使用cProfile模块分析函数调用的时间

```
import sys, re

def grep(pattern, file):
    with open(file, 'r') as f:
        print(file)
        for i, line in enumerate(f.readlines()):
            pattern = re.compile(pattern)
            match = pattern.search(line)
            if match is not None:
                print("{}: {}".format(i, line), end="")

if __name__ == '__main__':
    times = int(sys.argv[1])
    pattern = sys.argv[2]
    for i in range(times):
        for file in sys.argv[3:]:
            grep(pattern, file)
```

运行

```
$ python -m cProfile -s tottime grep.py 1000 '^import|\.def)[^,]*$' *.py
[omitted program output]

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      8000    0.266    0.000    0.292    0.000 {built-in method io.open}
      8000    0.153    0.000    0.894    0.000 grep.py:5(grep)
     17000    0.101    0.000    0.101    0.000 {built-in method builtins.print}
      8000    0.100    0.000    0.129    0.000 {method 'readlines' of
'_io._IOBase' objects}
    93000    0.097    0.000    0.111    0.000 re.py:286(_compile)
```

```

93000    0.069    0.000    0.069    0.000 {method 'search' of
'_sre.SRE_Pattern' objects}
93000    0.030    0.000    0.141    0.000 re.py:231(compile)
17000    0.019    0.000    0.029    0.000 codecs.py:318(decode)
1      0.017    0.017    0.911    0.911 grep.py:3(<module>)

```

但他每次执行显示的是每次调用函数的时间。所以更加符合只觉得应该是包括每行代码的执行时间
demo

```

#!/usr/bin/env python
import requests
from bs4 import BeautifulSoup

# 这个装饰器会告诉分析器
# 我们想要分析这个函数
@profile
def get_urls():
    response = requests.get('https://missing.csail.mit.edu')
    s = BeautifulSoup(response.content, 'lxml')
    urls = []
    for url in s.find_all('a'):
        urls.append(url['href'])

if __name__ == '__main__':
    get_urls()

```

如果用cProfile数据太冗杂 我们可以使用 line_profiler

```

$ kernprof -l -v a.py
Wrote profile results to urls.py.lprof
Timer unit: 1e-06 s

Total time: 0.636188 s
File: a.py
Function: get_urls at line 5

Line #  Hits           Time  Per Hit   % Time  Line Contents
=====
5                               @profile
6                               def get_urls():
7       1    613909.0  613909.0     96.5      response = requ
8       1    21559.0   21559.0      3.4      s = BeautifulSoup
9       1        2.0      2.0      0.0      urls = []
10      25      685.0    27.4      0.1      for url in s.fi
11      24      33.0     1.4      0.0      urls.append

```

事件分析

在我们使用 `strace` 调试代码的时候，您可能会希望忽略一些特殊的代码并希望在分析时将其当作黑盒处理。`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）。下面是关于常见命令的简介：

- `perf list` - 列出可以被 perf 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。

可视化在 Python 中您可以使用 `pycallgraph` 来生成这些图片。

显示调用图和控制流图等等

一些资源监控工具

Htop

demo

端口不知道被什么占用的时候(我们模拟开一个服务器)

```

> python -m http.server 4444

```

```
jjgo@lion ~ >>> sudo lsof | grep ":4444 .LISTEN"
python 25031 jjgo 4u IPv4 26630769 0t0
TCP *:4444 (LISTEN)
```

专用工具

有时候，您只需要对黑盒程序进行基准测试，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。例如，我们在 shell 工具和脚本那一节课中我们推荐使用 `fd` 来代替 `find`。我们这里可以用 `hyperfine` 来比较一下它们。

例如，下面的例子中，我们可以看到 `fd` 比 `find` 要快20倍。

```
$ hyperfine --warmup 3 'fd -e jpg' 'find . -iname "*.jpg"'
Benchmark #1: fd -e jpg
Time (mean ± σ):      51.4 ms ± 2.9 ms      [User: 121.0 ms, System: 160.5
ms]
Range (min ... max): 44.2 ms ... 60.1 ms    56 runs

Benchmark #2: find . -iname "*.jpg"
Time (mean ± σ):      1.126 s ± 0.101 s      [User: 141.1 ms, System: 956.1
ms]
Range (min ... max): 0.975 s ... 1.287 s    10 runs

Summary
'fd -e jpg' ran
21.89 ± 2.33 times faster than 'find . -iname "*.jpg"'
```

Summary

课后答案也是对自己需要的简单动手就可以8

8.Metaprogramming

对于解释出自官方项目

我们这里说的“元编程（metaprogramming）”是什么意思呢？好吧，对于本文要介绍的这些内容，这是我们能够想到的最能概括它们的词。因为我们今天要讲的东西，更多是关于 流程，而不是写代码或更高效的工作。本节课我们会学习构建系统、代码测试以及依赖管理。在您还是学生的时候，这些东西看上去似乎对您来说没那么重要，不过当您开始实习或走进社会的时候，您将会接触到大型的代码库，本节课讲授的这些东西也会变得随处可见。必须要指出的是，“元编程”也有[用于操作程序的程序](#)之含义，这和我们今天讲座所介绍的概念是完全不同的。

构建系统

`make` 是最常用的构建系统之一，您会发现它通常被安装到了几乎所有基于UNIX的系统中。`make` 并不完美，但是对于中小型项目来说，它已经足够好了。当您执行 `make` 时，它会去参考当前目录下名为 `Makefile` 的文件。所有构建目标、相关依赖和规则都需要在该文件中定义，它看上去是这样的：

```
paper.pdf: paper.tex plot-data.png
    pdflatex paper.tex

plot-%.png: %.dat plot.py
    ./plot.py -i $*.dat -o $@
```

这个文件中的指令，即如何使用右侧文件构建左侧文件的规则。或者，换句话说，冒号左侧的是构建目标，冒号右侧的是构建它所需的依赖。缩进的部分是从依赖构建目标时需要用到的一段程序。在 `make` 中，第一条指令还指明了构建的目的，如果您使用不带参数的 `make`，这便是我们最终的构建结果。或者，您可以使用这样的命令来构建其他目标：`make plot-data.png`。

规则中的 `%` 是一种模式，它会匹配其左右两侧相同的字符串。例如，如果目标是 `plot-foo.png`，`make` 会去寻找 `foo.dat` 和 `plot.py` 作为依赖。现在，让我们看看如果在一个空的源码目录中执行 `make` 会发生什么？

```
$ make
make: *** No rule to make target 'paper.tex', needed by 'paper.pdf'. Stop.
```

`make` 会告诉我们，为了构建出 `paper.pdf`，它需要 `paper.tex`，但是并没有一条规则能够告诉它如何构建该文件。让我们构建它吧！

```
$ touch paper.tex
$ make
make: *** No rule to make target 'plot-data.png', needed by 'paper.pdf'. Stop.
```

哟，有意思，我们是有构建 `plot-data.png` 的规则的，但是这是一条模式规则。因为源文件 `foo.dat` 并不存在，因此 `make` 就会告诉您它不能构建 `plot-data.png`，让我们创建这些文件：

```
$ cat paper.tex
\documentclass{article}
\usepackage{graphicx}
\begin{document}
\includegraphics[scale=0.65]{plot-data.png}
\end{document}
$ cat plot.py
#!/usr/bin/env python
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import argparse
```

```
parser = argparse.ArgumentParser()
parser.add_argument('-i', type=argparse.FileType('r'))
parser.add_argument('-o')
args = parser.parse_args()

data = np.loadtxt(args.i)
plt.plot(data[:, 0], data[:, 1])
plt.savefig(args.o)
$ cat data.dat
1 1
2 2
3 3
4 4
5 8
```

当我们执行 `make` 时会发生什么？

```
$ make
./plot.py -i data.dat -o plot-data.png
pdflatex paper.tex
... lots of output ...
```

看！PDF！

如果再次执行 `make` 会怎样？

```
$ make
make: 'paper.pdf' is up to date.
```

什么事情都没做！为什么？好吧，因为它什么都不需要做。`make`回去检查之前的构建是因其依赖改变而需要被更新。让我们试试修改 `paper.tex` 在重新执行 `make`：

```
$ vim paper.tex
$ make
pdflatex paper.tex
...
```

注意 `make` 并没有重新构建 `plot.py`，因为没必要；`plot-data.png` 的所有依赖都没有发生改变。

依赖管理

就您的项目来说，它的依赖可能本身也是其他的项目。您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，这些仓库会在一个地方托管大量的依赖，我们则可以通过一套非常简单的机制来安装依赖。例如 Ubuntu 系统下面有Ubuntu软件包仓库，您可以通过 `apt` 这个工具来访问，RubyGems 则包含了 Ruby 的相关库，PyPi 包含了 Python 库，Arch Linux 用户贡献的库则可以在 Arch User Repository 中

找到。

由于每个仓库、每种工具的运行机制都不太一样，因此我们并不会在本节课深入讲解具体的细节。我们会介绍一些通用的术语，例如版本控制。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。通常看上去像 8.1.3 或 64.1.20192004。版本号一般是数字构成的，但也并不绝对。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。试想一下，假如我的库要发布一个新版本，在这个版本里面我重命名了某个函数。如果有人在我的库升级版本后，仍希望基于它构建新的软件，那么很可能构建会失败，因为它希望调用的函数已经不复存在了。有了版本控制就可以很好的解决这个问题，我们可以指定当前项目需要基于某个版本，甚至某个范围内的版本，或是某些项目来构建。这么做的话，即使某个被依赖的库发生了变化，依赖它的软件可以基于其之前的版本进行构建。

这样还并不理想！如果我们发布了一项和安全相关的升级，它并没有影响到任何公开接口（API），但是出于安全的考虑，依赖它的项目都应该立即升级，那应该怎么做呢？这也是版本号包含多个部分的原因。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](#)，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。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您的项目构建的，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。换句话说，如果我依赖的版本是 1.3.7，那么使用 1.3.8、1.6.1，甚至是 1.3.0 都是可以的。如果版本号是 2.2.4 就不一定能用了，因为它的主版本号增加了。我们可以将 Python 的版本号作为语义版本号的一个实例。您应该知道，Python 2 和 Python 3 的代码是不兼容的，这也是为什么 Python 的主版本号改变的原因。类似的，使用 Python 3.5 编写的代码在 3.7 上可以运行，但是在 3.4 上可能会不行。

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，您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，不过这也意味着如何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。

持续集成系统

随着您接触到的项目规模越来越大，您会发现修改代码之后还有很多额外的工作要做。您可能需要上传一份新版本的文档、上传编译后的文件到某处、发布代码到 pypi，执行测试套等等。或许您希望每次有人提交代码到 GitHub 的时候，他们的代码风格被检查过并执行过某些基准测试？如果您有这方面的需求，那么请花些时间了解一下持续集成。

持续集成，或者叫做 CI 是一种雨伞术语（umbrella term），它指的是那些“当您的代码变动时，自动运行的东西”，市场上有很多提供各式各样 CI 工具的公司，这些工具大部分都是免费或开源的。比较大的有 Travis CI、Azure Pipelines 和 GitHub Actions。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，最常见的规则是：如果有人提交代码，执行测试套。当这个事件被触发时，CI 提供方会启动一个（或多个）虚拟机，执行您制定的规则，并且通常会记录下相关的执行结果。您可以进行某些设置，这样当测试套失败时您能够收到通知或者当测试全部通过时，您的仓库主页会显示一个徽标。

本课程的网站基于 GitHub Pages 构建，这就是一个很好的例子。Pages 在每次 master 有代码更新时，会执行 Jekyll 博客软件，然后使您的站点可以通过某个 GitHub 域名来访问。对于我们来说这些事情太琐碎了，我现在我们只需要在本地进行修改，然后使用 git 提交代码，发布到远端。CI 会自动帮我们处理后续的事情。

测试简介

多数的大型软件都有“测试套”。您可能已经对测试的相关概念有所了解，但是我们认为有些测试方法和测试术语还是应该再次提醒一下：

- 测试套：所有测试的统称
- 单元测试：一个“微型测试”，用于对某个封装的特性进行测试
- 集成测试：一个“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。
- 回归测试：用于保证之前引起问题的 bug 不会再次出现
- 模拟（Mocking）：使用一个假的实现来替换函数、模块或类型，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“模拟硬盘”

课后练习（无解）

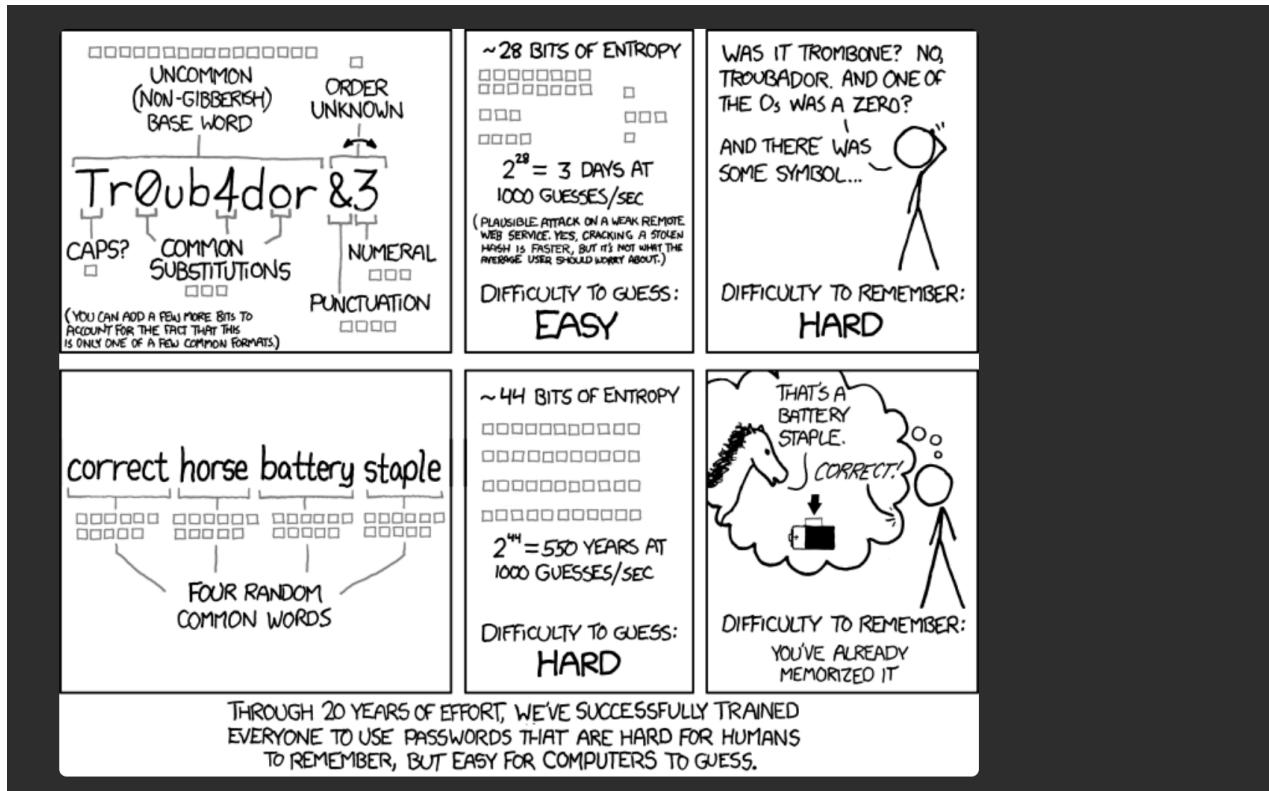
1. 大多数的 makefiles 都提供了一个名为 `clean` 的构建目标，这并不是说我们会生成一个名为 `clean` 的文件，而是我们可以使用它清理文件，让 make 重新构建。您可以理解为它的作用是“撤销”所有构建步骤。在上面的 makefile 中为 `paper.pdf` 实现一个 `clean` 目标。您需要构建 [phony](#)。您也许会发现 [git ls-files](#) 子命令很有用。其他一些有用的 make 构建目标可以在[这里](#)找到；
2. 指定版本要求的方法很多，让我们学习一下 [Rust的构建系统](#)的依赖管理。大多数的包管理仓库都支持类似的语法。对于每种语法(尖号、波浪号、通配符、比较、乘积)，构建一种场景使其具有实际意义；
3. Git 可以作为一个简单的 CI 系统来使用，在任何 git 仓库中的 `.git/hooks` 目录中，您可以找到一些文件（当前处于未激活状态），它们的作用和脚本一样，当某些事件发生时便可以自动执行。请编写一个 [pre-commit](#) 钩子，当执行 `make` 命令失败后，它会执行 `make paper.pdf` 并拒绝您的提交。这样做可以避免产生包含不可构建版本的提交信息；
4. 基于 [GitHub Pages](#) 创建任意一个可以自动发布的页面。添加一个[GitHub Action](#) 到该仓库，对仓库中的所有 shell 文件执行 [shellcheck \(方法之一\)](#)；
5. [构建属于您的 GitHub action](#)，对仓库中所有的 `.md` 文件执行 [proselint](#) 或 [write-good](#)，在您的仓库中开启这一功能，提交一个包含错误的文件看看该功能是否生效。

9.Security and cryptography

这节课对我这种安全学习者要好好学习

1. 熵

度量了不确定性并可以用来决定密码的强度。



正如上面的 [XKCD 漫画](#) 所描述的，“correcthorsebatterystaple”这个密码比“Tr0ub4dor&3”更安全——可是熵是如何量化安全性的呢？

熵的单位是 **比特**。对于一个均匀分布的随机离散变量，熵等于 $\log_2(\text{所有可能的个数, 即n})$ 。扔一次硬币的熵是1比特。掷一次（六面）骰子的熵大约为2.58比特。

一般我们认为攻击者了解密码的模型（最小长度，最大长度，可能包含的字符种类等），但是不了解某个密码是如何随机选择的——比如[掷骰子](#)。

使用多少比特的熵取决于应用的威胁模型。上面的XKCD漫画告诉我们，大约40比特的熵足以对抗在线穷举攻击（受限于网络速度和应用认证机制）。而对于离线穷举攻击（主要受限于计算速度），一般需要更强的密码（比如80比特或更多）。

2. hash function

课程中介绍了sha1

```
$ echo hello | sha1sum  
f572d396fae9206628714fb2ce00f72e94f2258f
```

长度为40的hex

这种加密的一些特性

- 确定性：对于不变的输入永远有相同的输出。

- 不可逆性：对于 `hash(m) = h`，难以通过已知的输出 `h` 来计算出原始输入 `m`。
- 目标碰撞抵抗性/弱无碰撞：对于一个给定输入 `m_1`，难以找到 `m_2 != m_1` 且 `hash(m_1) = hash(m_2)`。
- 碰撞抵抗性/强无碰撞：难以找到一组满足 `hash(m_1) = hash(m_2)` 的输入 `m_1, m_2`（该性质严格强于目标碰撞抵抗性）。

所以你听到的很多破解md5 破解sha1 很多可能只是彩虹表而已

而git 也是使用sha1来进行传输

```
commit 00b8f6a670e55fcdbb710239c811070169d2b0dd
Merge: f9dab34 5918e8b
Author: T4rnRookie <root@hackerpoet.com>
Date:   Sun Jan 10 12:25:20 2021 +0800
```

我们还可以用sha1来加密文件名

```
$ sha1sum README.md
a2eb39d2ac5afb29c525ee230bf41901fdde0459  README.md
```

比较有意思的点就是这个，因为我们可能很多时候都要下载文件，但是由于很多事国外的，所以我们可能会考虑一些镜像站来下载软件的副本，我们为了以防后门可以与官方发布的hash值进行比对

Hash函数的承诺机制

假设我希望承诺一个值，但之后再透露它——比如在没有一个可信的、双方可见的硬币的情况下在我的脑海中公平的“扔一次硬币”。我可以选择一个值 `r = random()`，并和你分享它的哈希值 `h = sha256(r)`。这时你可以开始猜硬币的正反：我们一致同意偶数 `r` 代表正面，奇数 `r` 代表反面。你猜完了以后，我告诉你值 `r` 的内容，得出胜负。同时你可以使用 `sha256(r)` 来检查我分享的哈希值 `h` 以确认我没有作弊。

###

密钥生成函数的应用

- 从密码生成可以在其他加密算法中使用的密钥，比如对称加密算法（见下）。
- 存储登录凭证时不可直接存储明文密码。

正确的方法是针对每个用户随机生成一个盐 `salt = random()`，并存储盐，以及密钥生成函数对连接了盐的明文密码生成的哈希值 `KDF(password + salt)`。

在验证登录请求时，使用输入的密码连接存储的盐重新计算哈希值 `KDF(input + salt)`，并与存储的哈希值对比。

对称加密

说到加密，可能你会首先想到隐藏明文信息。对称加密使用以下几个方法来实现这个功能：

```
keygen() -> key (这是一个随机方法)
```

```
encrypt(plaintext: array<byte>, key) -> array<byte> (输出密文)  
decrypt(ciphertext: array<byte>, key) -> array<byte> (输出明文)
```

加密方法 `encrypt()` 输出的密文 `ciphertext` 很难在不知道 `key` 的情况下得出明文 `plaintext`。解密方法 `decrypt()` 有明显的正确性。因为功能要求给定密文及其密钥，解密方法必须输出明文：`decrypt(encrypt(m, k), k) = m`。

[AES](#) 是现在常用的一种对称加密系统。

对称加密的应用

- 加密不信任的云服务上存储的文件。对称加密和密钥生成函数配合起来，就可以使用密码加密文件：将密码输入密钥生成函数生成密钥 `key = KDF(passphrase)`，然后存储 `encrypt(file, key)`。



```
> openssl aes-256-cbc -salt -in README.md -out README.enc.md  
enter aes-256-cbc encryption password:  
Verifying - enter aes-256-cbc encryption password:  
> cat README.enc.md  
Salted__uo5-|,S  
  
(Gj|Tgl5BE]Q霍 |LJ@K_  
N@0sH^6E|*ay5,{B#[tKN$/=,-Gc  
i@Id  
  
> openssl aes-256-cbc -d -in README.enc.md -out README.dec.md  
enter aes-256-cbc decryption password:  
> cmp README.md README.dec.md  
> echo $? 所以关于对称密钥密码学的任何问题  
( - ) ( mast  
( - ) ( mast
```

是相同的文件

只不过是加密版本

非对称加密

非对称加密的“非对称”代表在其环境中，使用两个具有不同功能的密钥：一个是私钥(private key)，不向外公布；另一个是公钥(public key)，公布公钥不像公布对称加密的共享密钥那样可能影响加密体系的安全性。

非对称加密使用以下几个方法来实现加密/解密(encrypt/decrypt)，以及签名/验证(sign/verify)：

```
keygen() -> (public key, private key) (这是一个随机方法)

encrypt(plaintext: array<byte>, public key) -> array<byte> (输出密文)
decrypt(ciphertext: array<byte>, private key) -> array<byte> (输出明文)

sign(message: array<byte>, private key) -> array<byte> (生成签名)
verify(message: array<byte>, signature: array<byte>, public key) -> bool (验证
签名是否是由和这个公钥相关的私钥生成的)
```

非对称的加密/解密方法和对称的加密/解密方法有类似的特征。

信息在非对称加密中使用 公钥 加密，且输出的密文很难在不知道 私钥 的情况下得出明文。

解密方法 `decrypt()` 有明显的正确性。给定密文及私钥，解密方法一定会输出明文：

```
decrypt(encrypt(m, public key), private key) = m。
```

对称加密和非对称加密可以类比为机械锁。对称加密就好比一个防盗门：只要是有钥匙的人都可以开门或者锁门。非对称加密好比一个可以拿下来的挂锁。你可以把打开状态的挂锁（公钥）给任何一个人并保留唯一的钥匙（私钥）。这样他们将给你的信息装进盒子里并用这个挂锁锁上以后，只有你可以用保留的钥匙开锁。

签名/验证方法具有和书面签名类似的特征。

在不知道 私钥 的情况下，不管需要签名的信息为何，很难计算出一个可以使 `verify(message, signature, public key)` 返回为真的签名。

对于使用私钥签名的信息，验证方法验证和私钥相对应的公钥时一定返回为真：`verify(message, sign(message, private key), public key) = true`。

非对称加密面对的主要挑战是，如何分发公钥并对应现实世界中存在的人或组织。

Signal的信任模型是，信任用户第一次使用时给出的身份(trust on first use)，同时支持用户线下(out-of-band)、面对面交换公钥 (Signal里的safety number)。

PGP使用的是[信任网络](#)。简单来说，如果我想加入一个信任网络，则必须让已经在信任网络中的成员对我进行线下验证，比如对比证件。验证无误后，信任网络的成员使用私钥对我的公钥进行签名。这样我就成为了信任网络的一部分。只要我使用签名过的公钥所对应的私钥就可以证明“我是我”。

Keybase主要使用[社交网络证明 \(social proof\)](#)，和一些别的精巧设计。

每个信任模型有它们各自的优点：我们（讲师）更倾向于 Keybase 使用的模型。

案例分析

密码管理器

每个人都应该尝试使用密码管理器，比如[KeePassXC](#)。

密码管理器会帮助你对每个网站生成随机且复杂（表现为高熵）的密码，并使用你指定的主密码配合密钥生成函数来对称加密它们。

你只需要记住一个复杂的主密码，密码管理器就可以生成很多复杂度高且不会重复使用的密码。密码管理器通过这种方式降低密码被猜出的可能，并减少网站信息泄露后对其他网站密码的威胁。

两步验证（双因子验证）

[两步验证\(2FA\)](#)要求用户同时使用密码（“你知道的信息”）和一个身份验证器（“你拥有的物品”，比如[YubiKey](#)）来消除密码泄露或者[钓鱼攻击](#)的威胁。

全盘加密

对笔记本电脑的硬盘进行全盘加密是防止因设备丢失而信息泄露的简单且有效方法。Linux的[cryptsetup + LUKS](#), Windows的[BitLocker](#), 或者macOS的[FileVault](#)都使用一个由密码保护的对称密钥来加密盘上的所有信息。

聊天加密

[Signal](#)和[Keybase](#)使用非对称加密对用户提供端到端(End-to-end)安全性。

获取联系人的公钥非常关键。为了保证安全性，应使用线下方式验证Signal或者Keybase的用户公钥，或者信任Keybase用户提供的社交网络证明。

SSH

我们在[之前的一堂课](#)讨论了SSH和SSH密钥的使用。那么我们今天从密码学的角度来分析一下它们。

当你运行 `ssh-keygen` 命令，它会生成一个非对称密钥对：公钥和私钥 (`public_key`, `private_key`)。生成过程中使用的随机数由系统提供的熵决定。这些熵可以来源于硬件事件 (hardware events)等。公钥最终会被分发，它可以直接明文存储。但是为了防止泄露，私钥必须加密存储。`ssh-keygen` 命令会提示用户输入一个密码，并将它输入密钥生成函数产生一个密钥。最终，`ssh-keygen` 使用对称加密算法和这个密钥加密私钥。

在实际运用中，当服务器已知用户的公钥（存储在 `.ssh/authorized_keys` 文件中，一般在用户HOME目录下），尝试连接的客户端可以使用非对称签名来证明用户的身份——这便是[挑战应答方式](#)。简单来说，服务器选择一个随机数字发送给客户端。客户端使用用户私钥对这个数字信息签名后返回服务器。服务器随后使用 `.ssh/authorized_keys` 文件中存储的用户公钥来验证返回的信息是否由所对应的私钥所签名。这种验证方式可以有效证明试图登录的用户持有所需的私钥。

非对称加密面对的主要挑战是，如何分发公钥并对应现实世界中存在的人或组织。

Signal的信任模型是，信任用户第一次使用时给出的身份(trust on first use)，同时支持用户线下(out-of-band)、面对面交换公钥 (Signal里的safety number) 。

PGP使用的是[信任网络](#)。简单来说，如果我想加入一个信任网络，则必须让已经在信任网络中的成员对我进行线下验证，比如对比证件。验证无误后，信任网络的成员使用私钥对我的公钥进行签名。这样我就成为了信任网络的一部分。只要我使用签名过的公钥所对应的私钥就可以证明“我是我”。

Keybase主要使用[社交网络证明 \(social proof\)](#)，和一些别的精巧设计。

每个信任模型有它们各自的优点：我们（讲师）更倾向于 Keybase 使用的模型。

案例分析

密码管理器

每个人都应该尝试使用密码管理器，比如[KeePassXC](#)。

密码管理器会帮助你对每个网站生成随机且复杂（表现为高熵）的密码，并使用你指定的主密码配合密钥生成函数来对称加密它们。

你只需要记住一个复杂的主密码，密码管理器就可以生成很多复杂度高且不会重复使用的密码。密码管理器通过这种方式降低密码被猜出的可能，并减少网站信息泄露后对其他网站密码的威胁。

两步验证（双因子验证）

[两步验证](#)(2FA)要求用户同时使用密码（“你知道的信息”）和一个身份验证器（“你拥有的物品”，比如[YubiKey](#)）来消除密码泄露或者[钓鱼攻击](#)的威胁。

全盘加密

对笔记本电脑的硬盘进行全盘加密是防止因设备丢失而信息泄露的简单且有效方法。Linux的[cryptsetup + LUKS](#)，Windows的[BitLocker](#)，或者macOS的[FileVault](#)都使用一个由密码保护的对称密钥来加密盘上的所有信息。

聊天加密

[Signal](#)和[Keybase](#)使用非对称加密对用户提供端到端(End-to-end)安全性。

获取联系人的公钥非常关键。为了保证安全性，应使用线下方式验证Signal或者Keybase的用户公钥，或者信任Keybase用户提供的社交网络证明。

SSH

我们在[之前的一堂课](#)讨论了SSH和SSH密钥的使用。那么我们今天从密码学的角度来分析一下它们。

当你运行`ssh-keygen`命令，它会生成一个非对称密钥对：公钥和私钥(`public_key`, `private_key`)。生成过程中使用的随机数由系统提供的熵决定。这些熵可以来源于硬件事件(hardware events)等。公钥最终会被分发，它可以直接明文存储。但是为了防止泄露，私钥必须加密存储。`ssh-keygen`命令会提示用户输入一个密码，并将它输入密钥生成函数产生一个密钥。最终，`ssh-keygen`使用对称加密算法和这个密钥加密私钥。

在实际运用中，当服务器已知用户的公钥（存储在`.ssh/authorized_keys`文件中，一般在用户HOME目录下），尝试连接的客户端可以使用非对称签名来证明用户的身份——这便是[挑战应答方式](#)。简单来说，服务器选择一个随机数字发送给客户端。客户端使用用户私钥对这个数字信息签名后返回服务器。服务器随后使用`.ssh/authorized_keys`文件中存储的用户公钥来验证返回的信息是否由所对应的私钥所签名。这种验证方式可以有效证明试图登录的用户持有所需的私钥。

非对称加密面对的主要挑战是，如何分发公钥并对应现实世界中存在的人或组织。

Signal的信任模型是，信任用户第一次使用时给出的身份(trust on first use)，同时支持用户线下(out-of-band)、面对面交换公钥（Signal里的safety number）。

PGP使用的是[信任网络](#)。简单来说，如果我想加入一个信任网络，则必须让已经在信任网络中的成员对我进行线下验证，比如对比证件。验证无误后，信任网络的成员使用私钥对我的公钥进行签名。这样我就成为了信任网络的一部分。只要我使用签名过的公钥所对应的私钥就可以证明“我是我”。

Keybase主要使用[社交网络证明 \(social proof\)](#)，和一些别的精巧设计。

每个信任模型有它们各自的优点：我们（讲师）更倾向于 Keybase 使用的模型。

案例分析

密码管理器

每个人都应该尝试使用密码管理器，比如[KeePassXC](#)。

密码管理器会帮助你对每个网站生成随机且复杂（表现为高熵）的密码，并使用你指定的主密码配合密钥生成函数来对称加密它们。

你只需要记住一个复杂的主密码，密码管理器就可以生成很多复杂度高且不会重复使用的密码。密码管理器通过这种方式降低密码被猜出的可能，并减少网站信息泄露后对其他网站密码的威胁。

两步验证（双因子验证）

[两步验证](#)(2FA)要求用户同时使用密码（“你知道的信息”）和一个身份验证器（“你拥有的物品”，比如[YubiKey](#)）来消除密码泄露或者[钓鱼攻击](#)的威胁。

全盘加密

对笔记本电脑的硬盘进行全盘加密是防止因设备丢失而信息泄露的简单且有效方法。Linux的[cryptsetup + LUKS](#)，Windows的[BitLocker](#)，或者macOS的[FileVault](#)都使用一个由密码保护的对称密钥来加密盘上的所有信息。

聊天加密

[Signal](#)和[Keybase](#)使用非对称加密对用户提供端到端(End-to-end)安全性。

获取联系人的公钥非常关键。为了保证安全性，应使用线下方式验证Signal或者Keybase的用户公钥，或者信任Keybase用户提供的社交网络证明。

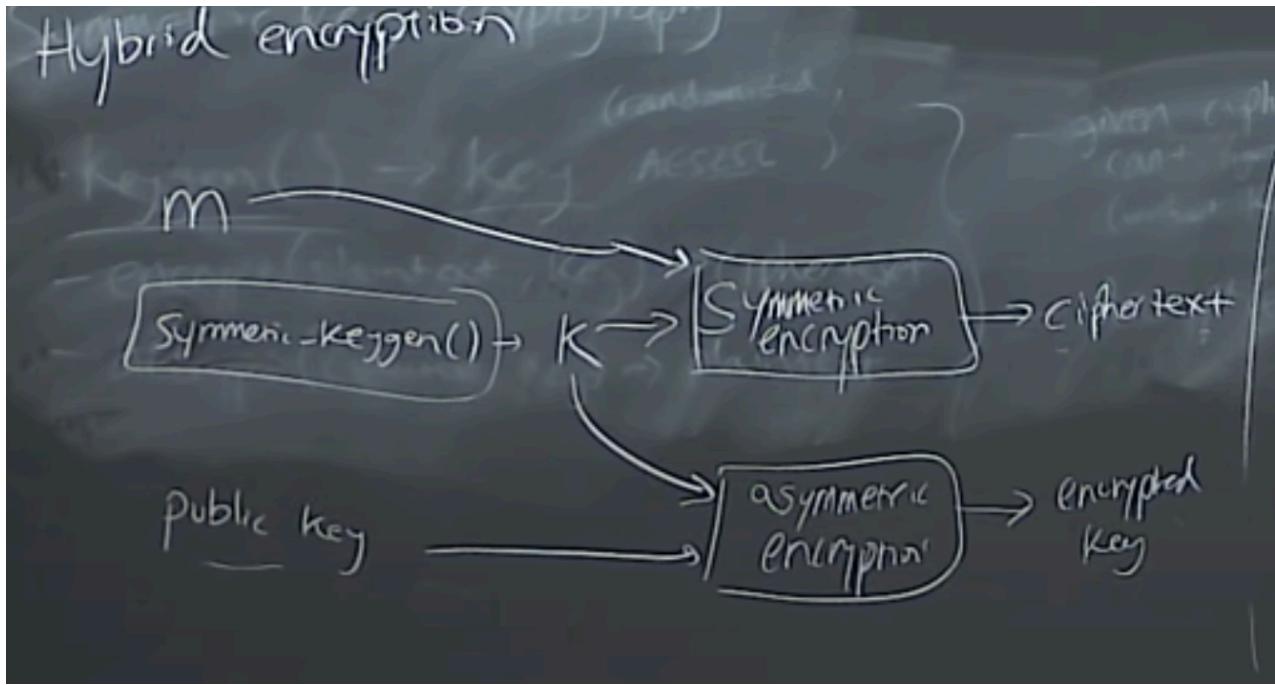
SSH

我们在[之前的一堂课](#)讨论了SSH和SSH密钥的使用。那么我们今天从密码学的角度来分析一下它们。

当你运行 `ssh-keygen` 命令，它会生成一个非对称密钥对：公钥和私钥 (`public_key`, `private_key`)。生成过程中使用的随机数由系统提供的熵决定。这些熵可以来源于硬件事件 (hardware events)等。公钥最终会被分发，它可以直接明文存储。但是为了防止泄露，私钥必须加密存储。`ssh-keygen` 命令会提示用户输入一个密码，并将它输入密钥生成函数产生一个密钥。最终，`ssh-keygen` 使用对称加密算法和这个密钥加密私钥。

在实际运用中，当服务器已知用户的公钥（存储在 `.ssh/authorized_keys` 文件中，一般在用户HOME目录下），尝试连接的客户端可以使用非对称签名来证明用户的身份——这便是[挑战应答方式](#)。简单来说，服务器选择一个随机数字发送给客户端。客户端使用用户私钥对这个数字信息签名后返回服务器。服务器随后使用 `.ssh/authorized_keys` 文件中存储的用户公钥来验证返回的信息是否由所对应的私钥所签名。这种验证方式可以有效证明试图登录的用户持有所需的私钥。

对称非对称结合



课后练习

参考主站

10.potpourri

这里面很多我都知道了。所以唯一一个想要以后了解的

<https://www.hammerspoon.org/>

就是这个，到时候再尝试自己写lua

11.question&answer

学习操作系统相关内容的推荐，比如进程，虚拟内存，中断，内存管理等

首先，不清楚你是不是真的需要了解这些更底层的话题。当你开始编写更加底层的代码，比如实现或修改内核的时候，这些内容是很重要的。除了其他课程中简要介绍过的进程和信号量之外，大部分话题都不相关。

学习资源：

- [MIT's 6.828 class](#) - 研究生阶段的操作系统课程（课程资料是公开的）。
- 现代操作系统 第四版 (*Modern Operating Systems 4th ed*) - 作者是Andrew S. Tanenbaum 这本书对上述很多概念都有很好的描述。
- FreeBSD的设计与实现 (*The Design and Implementation of the FreeBSD Operating System*) - 关于FreeBSD OS 不错的资源(注意，FreeBSD OS 不是 Linux)。
- 其他的指南例如 [用 Rust 写操作系统](#) 这里用不同的语言逐步实现了内核，主要用于教学的目的。

你会优先学习的工具有那些？

值得优先学习的内容：

- 多去使用键盘，少使用鼠标。这一目标可以通过多加利用快捷键，更换界面等来实现。
- 学好编辑器。作为程序员你大部分时间都是在编辑文件，因此值得学好这些技能。
- 学习怎样去自动化或简化工作流程中的重复任务。因为这会节省大量的时间。
- 学习像 Git 之类的版本控制工具并且知道如何与 GitHub 结合，以便在现代的软件项目中协同工作。

使用 Python VS Bash脚本 VS 其他语言？

通常来说，Bash 脚本对于简短的一次性脚本有效，比如当你想要运行一系列的命令的时候。但是Bash 脚本有一些比较奇怪的地方，这使得大型程序或脚本难以用 Bash 实现：

- Bash 可以获取简单的用例，但是很难获得全部可能的输入。例如，脚本参数中的空格会导致Bash 脚本出错。
- Bash 对于代码重用并不友好。因此，重用你先前已经写好的代码很困难。通常 Bash 中没有软件库的概念。
- Bash 依赖于一些像 `$?` 或 `$@` 的特殊字符指代特殊的值。其他的语言却会显式地引用，比如 `exitCode` 或 `sys.argv`。

因此，对于大型或者更加复杂的脚本我们推荐使用更加成熟的脚本语言例如 Python 和 Ruby。你可以找到很多用这些语言编写的，用来解决常见问题的在线库。如果你发现某种语言实现了你所需要的特定功能库，最好的方式就是直接去使用那种语言。

source script.sh 和 `./script.sh` 有什么区别？`

这两种情况 `script.sh` 都会在bash会话中被读取和执行，不同点在于哪个会话执行这个命令。对于 `source` 命令来说，命令是在当前的bash会话中执行的，因此当 `source` 执行完毕，对当前环境的任何更改（例如更改目录或是定义函数）都会留存在当前会话中。单独运行 `./script.sh` 时，当前的bash 会话将启动新的bash会话（实例），并在新实例中运行命令 `script.sh`。因此，如果 `script.sh` 更改目录，新的bash会话（实例）会更改目录，但是一旦退出并将控制权返回给父bash会话，父会话仍然留在先前的位置（不会有目录的更改）。同样，如果 `script.sh` 定义了要在终端中访问的函数，需要用 `source` 命令在当前bash会话中定义这个函数。否则，如果你运行 `./script.sh`，只有新的bash会话（进程）才能执行定义的函数，而当前的shell不能。

各种软件包和工具存储在哪里？引用过程是怎样的? `/bin` 或 `/lib` 是什么？

根据你在命令行中运行的程序，这些包和工具会全部在 `PATH` 环境变量所列出的目录中查找到，你可以使用 `which` 命令(或是 `type` 命令)来检查你的shell在哪里发现了特定的程序。一般来说，特定种类的文件存储有一定的规范，[文件系统，层次结构标准 \(Filesystem, Hierarchy Standard\)](#) 可以查到我们讨论内容的详细列表。

- `/bin` - 基本命令二进制文件
- `/sbin` - 基本的系统二进制文件，通常是root运行的

- `/dev` - 设备文件，通常是硬件设备接口文件
- `/etc` - 主机特定的系统配置文件
- `/home` - 系统用户的家目录
- `/lib` - 系统软件通用库
- `/opt` - 可选的应用软件
- `/sys` - 包含系统的信息和配置([第一堂课](#)介绍的)
- `/tmp` - 临时文件(`/var/tmp`)通常在重启之间删除

`/usr/`

- 只读的用户数据

- `/usr/bin` - 非必须的命令二进制文件
- `/usr/sbin` - 非必须的系统二进制文件，通常是由root运行的
- `/usr/local/bin` - 用户编译程序的二进制文件
- `/var` - 变量文件 像日志或缓存

我应该用 `apt-get install` 还是 `pip install` 去下载软件包呢？

这个问题没有普遍的答案。这与使用系统程序包管理器还是特定语言的程序包管理器来安装软件这一更笼统的问题相关。需要考虑的几件事：

- 常见的软件包都可以通过这两种方法获得，但是小众的软件包或较新的软件包可能不在系统程序包管理器中。在这种情况下，使用特定语言的程序包管理器是更好的选择。
- 同样，特定语言的程序包管理器相比系统程序包管理器有更多的最新版本的程序包。
- 当使用系统软件包管理器时，将在系统范围内安装库。如果出于开发目的需要不同版本的库，则系统软件包管理器可能不能满足你的需要。对于这种情况，大多数编程语言都提供了隔离或虚拟环境，因此你可以用特定语言的程序包管理器安装不同版本的库而不会发生冲突。对于 Python，可以使用 `virtualenv`，对于 Ruby，使用 `RVM`。
- 根据操作系统和硬件架构，其中一些软件包可能会附带二进制文件或者软件包需要被编译。例如，在树莓派（Raspberry Pi）之类的ARM架构计算机中，在软件附带二进制文件和软件包需要被编译的情况下，使用系统包管理器比特定语言包管理器更好。这在很大程度上取决于你的特定设置。你应该仅使用一种解决方案，而不同时使用两种方法，因为这可能会导致难以解决的冲突。我们的建议是尽可能使用特定语言的程序包管理器，并使用隔离的环境（例如 Python 的 `virtualenv`）以避免影响全局环境。

用于提高代码性能，简单好用的性能分析工具有哪些？

性能分析方面相当有用和简单工具是[print timing](#)。你只需手动计算代码不同部分之间花费的时间。通过重复执行此操作，你可以有效地对代码进行二分法搜索，并找到花费时间最长的代码段。

对于更高级的工具，Valgrind的[Callgrind](#)可让你运行程序并计算所有的时间花费以及所有调用堆栈（即哪个函数调用了另一个函数）。然后，它会生成带注释的代码版本，其中包含每行花费的时间。但是，它会使程序运行速度降低一个数量级，并且不支持线程。其他的，[perf](#)工具和其他特定语言的采样性能分析器可以非常快速地输出有用的数据。[Flamegraphs](#)是对采样分析器结果的可视化工具。你还可以使用针对特定编程语言或任务的工具。例如，对于Web开发而言，Chrome和Firefox内置的开发工具具有出色的性能分析器。

有时，代码中最慢的部分是系统等待磁盘读取或网络数据包之类的事件。在这些情况下，需要检查根据硬件性能估算的理论速度是否不偏离实际数值，也有专门的工具来分析系统调用中的等待时间，包括用于用户程序内核跟踪的[eBPF](#)。如果需要低级的性能分析，[bpftrace](#)值得一试。

你使用那些浏览器插件？

我们钟爱的插件主要与安全性与可用性有关：

- [uBlock Origin](#) - 是一个[用途广泛 \(wide-spectrum\)](#) 的拦截器，它不仅可以拦截广告，还可以拦截第三方的页面，也可以拦截内部脚本和其他种类资源的加载。如果你打算花更多的时间去配置，前往[中等模式 \(medium mode\)](#) 或者 [强力模式 \(hard mode\)](#)。在你调整好设置之前一些网站会停止工作，但是这些配置会显著提高你的网络安全水平。另外，[简易模式 \(easy mode\)](#) 作为默认模式已经相当不错了，可以拦截大部分的广告和跟踪，你也可以自定义规则来拦截网站对象。
- [Stylus](#) - 是Stylish的分支（不要使用Stylish，它会[窃取浏览记录](#)），这个插件可让你将自定义CSS样式加载到网站。使用Stylus，你可以轻松地自定义和修改网站的外观。可以删除侧边框，更改背景颜色，更改文字大小或字体样式。这可以使你经常访问的网站更具可读性。此外，Stylus可以找到其他用户编写并发布在[userstyles.org](#)中的样式。大多数常用的网站都有一个或几个深色主题样式。
- 全页屏幕捕获 - 内置于Firefox和[Chrome 扩展程序](#)中。这些插件提供完整的网站截图，通常比打印要好用。
- [多账户容器](#) - 该插件使你可以将Cookie分为“容器”，从而允许你以不同的身份浏览web网页并且/或确保网站无法在它们之间共享信息。
- 密码集成管理器 - 大多数密码管理器都有浏览器插件，这些插件帮你将登录凭据输入网站的过程不仅方便，而且更加安全。与简单复制粘贴用户名和密码相比，这些插件将首先检查网站域是否与列出的条目相匹配，以防止冒充网站的网络钓鱼窃取登录凭据。

有哪些有用的数据整理工具？

在数据整理那一节课中，我们没有时间讨论一些数据整理工具，包括分别用于JSON和HTML数据的专用解析器，[jq](#) 和 [pup](#)。Perl语言是另一个更高级的可以用于数据整理管道的工具。另一个技巧是使用[column -t](#)命令，可以将空格文本（不一定对齐）转换为对齐的文本。

一般来说，vim和Python是两个不常规的数据整理工具。对于某些复杂的多行转换，vim宏是非常有用的工具。你可以记录一系列操作，并根据需要重复执行多次，例如，在编辑的[讲义](#)（去年[视频](#)）中，有一个示例是使用vim宏将XML格式的文件转换为JSON。

对于通常以CSV格式显示的表格数据，Python [pandas](#)库是一个很棒的工具。不仅因为它能让复杂操作的定义（如分组依据、联接或过滤器）变得非常容易，而且还便于根据不同属性绘制数据。它还支持导出多种表格格式，包括XLS、HTML或LaTeX。另外，R语言（一种有争议的[不好的](#)语言）具有很多功能，可以计算数据的统计数字，这在管道的最后一步中非常有用。[ggplot2](#)是R中很棒的绘图库。

Docker和虚拟机有什么区别?

Docker 基于容器这个更为概括的概念。关于容器和虚拟机之间最大的不同是，虚拟机会执行整个的 OS 栈，包括内核（即使这个内核和主机内核相同）。与虚拟机不同，容器避免运行其他内核实例，而是与主机分享内核。在Linux环境中，有LXC机制来实现，并且这能使一系列分离的主机像是在使用自己的硬件启动程序，而实际上是共享主机的硬件和内核。因此容器的开销小于完整的虚拟机。

另一方面，容器的隔离性较弱而且只有在主机运行相同的内核时才能正常工作。例如，如果你在macOS 上运行 Docker，Docker 需要启动 Linux虚拟机去获取初始的 Linux内核，这样的开销仍然很大。最后，Docker 是容器的特定实现，它是为软件部署而定制的。基于这些，它有一些奇怪之处：例如，默认情况下，Docker 容器在重启之间不会有以任何形式的存储。

不同操作系统的优缺点是什么，我们如何选择（比如选择最适用于我们需求的Linux发行版）？

关于Linux发行版，尽管有相当多的版本，但大部分发行版在大多数使用情况下的表现是相同的。可以使用任何发行版去学习 Linux 与 UNIX 的特性和其内部工作原理。发行版之间的根本区别是发行版如何处理软件包更新。某些版本，例如 Arch Linux 采用滚动更新策略，用了最前沿的软件包（bleeding-edge），但软件可能并不稳定。另外一些发行版（如Debian，CentOS 或 Ubuntu LTS）其更新策略要保守得多，因此更新的内容会更稳定，但会牺牲一些新功能。我们建议你使用 Debian 或 Ubuntu 来获得简单稳定的台式机和服务器体验。

Mac OS 是介于 Windows 和 Linux 之间的一个操作系统，它有很漂亮的界面。但是，Mac OS 是基于 BSD 而不是 Linux，因此系统的某些部分和命令是不同的。另一种值得体验的是 FreeBSD。虽然某些程序不能在 FreeBSD 上运行，但与 Linux 相比，BSD 生态系统的碎片化程度要低得多，并且说明文档更加友好。除了开发Windows应用程序或需要使用某些Windows系统更好支持的功能（例如对游戏的驱动程序支持）外，我们不建议使用 Windows。

对于双系统，我们认为最有效的是 macOS 的 bootcamp，长期来看，任何其他组合都可能会出现问题，尤其是当你结合了其他功能比如磁盘加密。

使用 Vim 编辑器 VS Emacs 编辑器？

我们三个都使用 vim 作为我们的主要编辑器。但是 Emacs 也是一个不错的选择，你可以两者都尝试，看看那个更适合你。Emacs 不使用 vim 的模式编辑，但是这些功能可以通过 Emacs 插件像[Evil](#) 或 [Doom Emacs](#) 来实现。Emacs的优点是可以用Lisp语言进行扩展（Lisp比vim默认的脚本语言vimscript要更好用）。

机器学习应用的提示或技巧？

课程的一些经验可以直接用于机器学习程序。就像许多科学学科一样，在机器学习中，你需要进行一系列实验，并检查哪些数据有效，哪些无效。你可以使用 Shell 轻松快速地搜索这些实验结果，并且以合理的方式汇总。这意味着需要在限定时间内或使用特定数据集的情况下，检查所有实验结果。通过使用 JSON文件记录实验的所有相关参数，使用我们在本课程中介绍的工具，这件事情可以变得极其简单。最后，如果你不使用集群提交你的 GPU 作业，那你应该研究如何使该过程自动化，因为这是一项非常耗时的任务，会消耗你的精力。

还有更多的 Vim 小窍门吗？

更多的窍门：

- 插件 - 花时间去探索插件。有很多不错的插件修复了vim的缺陷或者增加了能够与现有vim工作流结合的新功能。关于这部分内容，资源是[VimAwesome](#) 和其他程序员的dotfiles。
- 标记 - 在vim里你可以使用 `m<x>` 为字母 `x` 做标记，之后你可以通过 `'<x>` 回到标记位置。这可以让你快速定位到文件内或文件间的特定位置。
- 导航 - `Ctrl+O` 和 `Ctrl+I` 命令可以使你在最近访问位置前后移动。
- 撤销树 - vim 有不错的更改跟踪机制，不同于其他的编辑器，vim存储变更树，因此即使你撤销后做了一些修改，你仍然可以通过撤销树的导航回到初始状态。一些插件比如 [gundo.vim](#) 和 [undotree](#) 通过图形化来展示撤销树。
- 时间撤销 - `:earlier` 和 `:later` 命令使得你可以用时间而非某一时刻的更改来定位文件。
- 持续撤销 - 是一个默认未被开启的vim的内置功能，它在vim启动之间保存撤销历史，需要配置在 `.vimrc` 目录下的 `undofile` 和 `undodir`，vim会保存每个文件的修改历史。
- 热键（Leader Key） - 热键是一个用于用户自定义配置命令的特殊按键。这种模式通常是按下后释放这个按键（通常是空格键）并与其他的按键组合去实现一个特殊的命令。插件也会用这些按键增加它们的功能，例如，插件UndoTree使用 `<Leader> u` 去打开撤销树。
- 高级文本对象 - 文本对象比如搜索也可以用vim命令构成。例如，`d/<pattern>` 会删除下一处匹配 `pattern` 的字符串，`cgn` 可以用于更改上次搜索的关键字。

2FA是什么，为什么我需要使用它？

双因子验证（Two Factor Authentication 2FA）在密码之上为帐户增加了一层额外的保护。为了登录，你不仅需要知道密码，还必须以某种方式“证明”可以访问某些硬件设备。最简单的情形是可以通过接收手机的 SMS 来实现（尽管 SMS 2FA 存在[已知问题](#)）。我们推荐使用[YubiKey](#)之类的[U2F](#)方案。

对于不同的 Web 浏览器有什么评价？

2020的浏览器现状是，大部分的浏览器都与 Chrome 类似，因为它们都使用同样的引擎(Blink)。Microsoft Edge 同样基于 Blink，至于 Safari 基于 WebKit(与Blink类似的引擎)，这些浏览器仅仅是更糟糕的 Chrome 版本。不管是在性能还是可用性上，Chrome 都是一款很不错的浏览器。如果你想要替代品，我们推荐 Firefox。Firefox 与 Chrome 的在各方面不相上下，并且在隐私方面更加出色。有一款目前还没有完成的叫 Flow 的浏览器，它实现了全新的渲染引擎，有望比现有引擎速度更快。