

# 课程概览与 shell

## 动机

作为计算机科学家，我们都知道计算机最擅长帮助我们完成重复性的工作。但是我们却常常忘记这一点也适用于我们使用计算机的方式，而不仅仅是利用计算机程序去帮我们求解问题。在从事与计算机相关的工作时，我们有很多触手可及的工具可以帮助我们更高效的解决问题。但是我们中的大多数人实际上只利用了这些工具中的很少一部分，我们常常只是死记硬背一些如咒语般的命令，或是当我们卡住的时候，盲目地从网上复制粘贴一些命令。

本课程意在帮你解决这一问题。

我们希望教会您如何挖掘现有工具的潜力，并向您介绍一些新的工具。也许我们还可以促使您想要去探索（甚至是去开发）更多的工具。我们认为这是大多数计算机科学相关课程中缺少的重要一环。

## 课程结构

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](#)。尽管这些讲座之间基本上是各自独立的，但随着课程的进行，我们会假定您已经掌握了之前的内容。每个讲座都有在线笔记供查阅，但是课上的很多内容并不会包含在笔记中。因此我们也会把课程录制下来发布到互联网上供大家观看学习。

我们希望能在这 11 个一小时讲座中涵盖大部分必须的内容，因此课程的信息密度是相当大的。为了能帮助您以自己的节奏来掌握讲座内容，每次课程都包含一组练习来帮助您掌握本节课的重点。课后我们会安排答疑的时间来回答您的问题。如果您参加的是在线课程，可以发送邮件到 [missing-semester@mit.edu](mailto:missing-semester@mit.edu) 来联系我们。

由于时长的限制，我们不可能达到那些专门课程一样的细致程度，我们会适时地将您介绍一些优秀的资源，帮助您深入的理解相关的工具或主题。但是如果您还有一些特别关注的话题，也请联系我们。

## 主题 1: The Shell

### shell 是什么？

如今的计算机有着多种多样的交互接口让我们可以进行指令的输入，从炫酷的图像用户界面 (GUI)，语音输入甚至是 AR/VR 都已经无处不在。这些交互接口可以覆盖 80% 的使用场景，但是它们也从根本上限制了您的操作方式——你不能点击一个不存在的按钮或者是用语音输入一个还没有被录入的指令。为了充分利用计算机的能力，我们不得不回到最根本的方式，使用文字接口：Shell

几乎所有您能够接触到的平台都支持某种形式的 shell，有些甚至还提供了多种 shell 供您选择。虽然它们之间有些细节上的差异，但是其核心功能都是一样的：它允许你执行程序，输入并获取

某种半结构化的输出。

本节课我们会使用 Bourne Again SHell, 简称 “bash” 。这是被最广泛使用的一种 shell, 它的语法和其他的 shell 都是类似的。打开shell 提示符 (您输入指令的地方), 您首先需要打开 终端。您的设备通常都已经内置了终端, 或者您也可以安装一个, 非常简单。

## 使用 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 (“ current working directory” ) 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 (稍后会介绍)。在这个提示符中, 您可以输入 命令, 命令最终会被 shell 解析。最简单的命令是执行一个程序:

```
missing:~$ date  
Fri 10 Jan 2020 11:49:31 AM EST  
missing:~$
```

这里, 我们执行了 date 这个程序, 不出意料地, 它打印出了当前的日期和时间。然后, shell 等待我们输入其他命令。我们可以在执行命令的同时向程序传递 参数:

```
missing:~$ echo hello  
hello
```

上例中, 我们让 shell 执行 echo , 同时指定参数 hello 。echo 程序将该参数打印出来。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 ) 。

但是, shell 是如何知道去哪里寻找 date 或 echo 的呢? 其实, 类似于 Python 或 Ruby, shell 是一个编程环境, 所以它具备变量、条件、循环和函数 (下一课进行讲解)。当你在 shell 中执行命令时, 您实际上是在执行一段 shell 可以解释执行的简短代码。如果你要求 shell 执行某个指令, 但是该指令并不是 shell 所了解的编程关键字, 那么它会去咨询 环境变量 \$PATH , 它会列出当 shell 接到某条指令时, 进行程序搜索的路径:

```
missing:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
missing:~$ which echo
/bin/echo
missing:~$ /bin/echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

当我们执行 echo 命令时，shell 了解到需要执行 echo 这个程序，随后它便会在 \$PATH 中搜索由 : 所分割的一系列目录，基于名字搜索该程序。当找到该程序时便执行（假定该文件是可执行程序，后续课程将详细讲解）。确定某个程序名代表的是哪个具体的程序，可以使用 which 程序。我们也可以绕过 \$PATH，通过直接指定需要执行的程序的路径来执行该程序

## 在shell中导航

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 \。路径 / 代表的是系统的根目录，所有的文件夹都包括在这个路径之下，在 Windows 上每个盘都有一个根目录（例如：C:\）。我们假设您在学习本课程时使用的是 Linux 文件系统。如果某个路径以 / 开头，那么它是一个 绝对路径，其他的都是 相对路径。相对路径是指相对于当前工作目录的路径，当前工作目录可以使用 pwd 命令来获取。此外，切换目录需要使用 cd 命令。在路径中，. 表示的是当前目录，而 .. 表示上级目录：

```
missing:~$ pwd
/home/missing
missing:~$ cd /home
missing:/home$ pwd
/home
missing:/home$ cd ..
missing:/$ pwd
/
missing:/$ cd ./home
missing:/home$ pwd
/home
missing:/home$ cd missing
missing:~$ pwd
/home/missing
missing:~$ ../../bin/echo hello
hello
```

注意，shell 会实时显示当前的路径信息。您可以通过配置 shell 提示符来显示各种有用的信息，这一内容我们会在后面的课程中进行讨论。

一般来说，当我们运行一个程序时，如果我们没有指定路径，则该程序会在当前目录下执行。例如，我们常常会搜索文件，并在需要时创建文件。

为了查看指定目录下包含哪些文件，我们使用 ls 命令：

```
missing:~$ ls
missing:~$ cd ..
missing:/home$ ls
missing
missing:/home$ cd ..
missing:/$ ls
bin
boot
dev
etc
home
...
...
```

除非我们利用第一个参数指定目录，否则 `ls` 会打印当前目录下的文件。大多数的命令接受标记和选项（带有值的标记），它们以 `-` 开头，并可以改变程序的行为。通常，在执行程序时使用 `-h` 或 `--help` 标记可以打印帮助信息，以便了解有哪些可用的标记或选项。例如，`ls --help` 的输出如下：

<code>-l</code>	use a long listing format
-----------------	---------------------------

<code>missing:~\$ ls -l /home</code>	<code>drwxr-xr-x 1 missing users 4096 Jun 15 2019 missing</code>
--------------------------------------	--

这个参数可以更加详细地列出目录下文件或文件夹的信息。首先，本行第一个字符 `d` 表示 `missing` 是一个目录。然后接下来的九个字符，每三个字符构成一组。`(rwx)`。它们分别代表了文件所有者 (`missing`)，用户组 (`users`) 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。从上面的信息来看，只有文件所有者可以修改 (`w`)，`missing` 文件夹（例如，添加或删除文件夹中的文件）。为了进入某个文件夹，用户需要具备该文件夹以及其父文件夹的“搜索”权限（以“可执行”：`x`）权限表示。为了列出它的包含的内容，用户必须对该文件夹具备读权限 (`r`)。对于文件来说，权限的意义也是类似的。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。

在这个阶段，还有几个趁手的命令是您需要掌握的，例如 `mv`（用于重命名或移动文件）、`cp`（拷贝文件）以及 `mkdir`（新建文件夹）。

如果您想要知道关于程序参数、输入输出的信息，亦或是想要了解它们的工作方式，请试试 `man` 这个程序。它会接受一个程序名作为参数，然后将它的文档（用户手册）展现给您。注意，使用 `q` 可以退出该程序。

<code>missing:~\$ man ls</code>
---------------------------------

## 在程序间创建连接

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，一个程序的输入输出流都是您的终端。也就是，您的键盘作为输入，显示器作为输出。但是，我们也可以重定向这些流！

最简单的重定向是 `< file` 和 `> file`。这两个命令可以将程序的输入输出流分别重定向到文件：

```
missing:~$ echo hello > hello.txt
missing:~$ cat hello.txt
hello
missing:~$ cat < hello.txt
hello
missing:~$ cat < hello.txt > hello2.txt
missing:~$ cat hello2.txt
hello
```

您还可以使用 `>>` 来向一个文件追加内容。使用管道（*pipes*），我们能够更好的利用文件重定向。`|` 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：

```
missing:~$ ls -l / | tail -n1
drwxr-xr-x 1 root root 4096 Jun 20 2019 var
missing:~$ curl --head --silent google.com | grep --ignore-case content-l
219
```

我们会在数据清理一章中更加详细的探讨如何更好的利用管道。

## 一个功能全面又强大的工具

对于大多数的类 Unix 系统，有一类用户是非常特殊的，那就是：根用户（root user）。您应该已经注意到了，在上面的输出结果中，根用户几乎不受任何限制，他可以创建、读取、更新和删除系统中的任何文件。通常在我们并不会以根用户的身份直接登录系统，因为这样可能会因为某些错误的操作而破坏系统。取而代之的是我们会在需要的时候使用 `sudo` 命令。顾名思义，它的作用是让您可以在 `su`（super user 或 root 的简写）的身份执行一些操作。当您遇到拒绝访问（`permission denied`）的错误时，通常是因为此时您必须是根用户才能操作。然而，请再次确认您是真的要执行此操作。

有一件事情是您必须作为根用户才能做的，那就是向 `sysfs` 文件写入内容。系统被挂载在 `/sys` 下，`sysfs` 文件则暴露了一些内核（kernel）参数。因此，您不需要借助任何专用的工具，就可以轻松地在运行期间配置系统内核。**注意 Windows 和 macOS 没有这个文件**

例如，您笔记本电脑的屏幕亮度写在 `brightness` 文件中，它位于

```
/sys/class/backlight
```

通过将数值写入该文件，我们可以改变屏幕的亮度。现在，蹦到您脑袋里的第一个想法可能是：

```
$ sudo find -L /sys/class/backlight -maxdepth 2 -name '*brightness*'
/sys/class/backlight/thinkpad_screen/brightness
$ cd /sys/class/backlight/thinkpad_screen
$ sudo echo 3 > brightness
An error occurred while redirecting file 'brightness'
open: Permission denied
```

出乎意料的是，我们还是得到了一个错误信息。毕竟，我们已经使用了 `sudo` 命令！关于 shell，有件事我们必须要知道。`|`、`>` 和 `<` 是通过 shell 执行的，而不是被各个程序单独执行。`echo` 等程序并不知道 `|` 的存在，它们只知道从自己的输入输出流中进行读写。对于上面这种情况，`shell`（权限为您的当前用户）在设置 `sudo echo` 前尝试打开 `brightness` 文件并写入，但是系统拒绝了 `shell` 的操作因为此时 `shell` 不是根用户。

明白这一点后，我们可以这样操作：

```
$ echo 3 | sudo tee brightness
```

因为打开 `/sys` 文件的是 `tee` 这个程序，并且该程序以 `root` 权限在运行，因此操作可以进行。这样您就可以在 `/sys` 中愉快地玩耍了，例如修改系统中各种LED的状态（路径可能会有所不同）：

```
$ echo 1 | sudo tee /sys/class/leds/input6::scrolllock/brightness
```

## 接下来.....

学到这里，您掌握的 shell 知识已经可以完成一些基础的任务了。您应该已经可以查找感兴趣的文件并使用大多数程序的基本功能了。在下一场讲座中，我们会探讨如何利用 shell 及其他工具执行并自动化更复杂的任务。

## 课后练习

习题解答 本课程中的每节课都包含一系列练习题。有些题目是有明确目的的，另外一些则是开放题，例如“尝试使用 X 和 Y”，我们强烈建议您一定要动手实践，用于尝试这些内容。此外，我们没有为这些练习题提供答案。如果有任何困难，您可以发送邮件给我们并描述你已经做出的尝试，我们会设法帮您解答。

1. 本课程需要使用类 Unix shell，例如 Bash 或 ZSH。如果您在 Linux 或者 MacOS 上面完成本课程的练习，则不需要做任何特殊的操作。如果您使用的是 Windows，则您不应该使用 cmd 或是 Powershell；您可以使用[Windows Subsystem for Linux](#)或者是 Linux 虚拟机。使用

echo \$SHELL 命令可以查看您的 shell 是否满足要求。如果打印结果为 /bin/bash 或 /usr/bin/zsh 则是可以的。

2. 在 /tmp 下新建一个名为 missing 的文件夹。
3. 用 man 查看程序 touch 的使用手册。
4. 用 touch 在 missing 文件夹中新建一个叫 semester 的文件。
5. 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
```

第一行可能有点棘手， # 在Bash中表示注释，而 ! 即使被双引号 ( " ) 包裹也具有特殊的含义。单引号 ( ' ) 则不一样，此处利用这一点解决输入问题。更多信息请参考 [Bash quoting 手册](#)

6. 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。
7. 查看 chmod 的手册(例如，使用 man chmod 命令)
8. 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](#)
9. 使用 | 和 > ，将 semester 文件输出的最后更改日期信息，写入主目录下的 last-modified.txt 的文件中
10. 写一段命令来从 /sys 中获取笔记本的电量信息，或者台式机 CPU 的温度。注意：macOS 并没有 sysfs，所以 Mac 用户可以跳过这一题。

# Shell 工具和脚本

在这节课中，我们将会展示 bash 作为脚本语言的一些基础操作，以及几种最常用的 shell 工具。

## Shell 脚本

到目前为止，我们已经学习来如何在 shell 中执行命令，并使用管道将命令组合使用。但是，很多情况下我们需要执行一系列的操作并使用条件或循环这样的控制流。

shell 脚本是一种更加复杂度的工具。

大多数shell都有自己的一套脚本语言，包括变量、控制流和自己的语法。shell脚本与其他脚本语言不同之处在于，shell 脚本针对 shell 所从事的相关工作进行来优化。因此，创建命令流程（pipelines）、将结果保存到文件、从标准输入中读取输入，这些都是 shell 脚本中的原生操作，这让它比通用的脚本语言更易用。本节中，我们会专注于 bash 脚本，因为它最流行，应用更为广泛。

在bash中为变量赋值的语法是 `foo=bar`，访问变量中存储的数值，其语法为 `$foo`。需要注意的是，`foo = bar`（使用空格隔开）是不能正确工作的，因为解释器会调用程序 `foo` 并将 `=` 和 `bar` 作为参数。**总的来说，在shell脚本中使用空格会起到分割参数的作用，有时候可能会造成混淆，请务必多加检查。**

Bash中的字符串通过' 和 " 分隔符来定义，但是它们的含义并不相同。以' 定义的字符串为原义字符串，其中的变量不会被转义，而 " 定义的字符串会将变量值进行替换。

```
foo=bar
echo "$foo"
# 打印 bar
echo '$foo'
# 打印 $foo
```

和其他大多数的编程语言一样，bash也支持 `if`, `case`, `while` 和 `for` 这些控制流关键字。同样地，bash 也支持函数，它可以接受参数并基于参数进行操作。下面这个函数是一个例子，它会创建一个文件夹并使用 `cd` 进入该文件夹。

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

这里 `$1` 是脚本的第一个参数。与其他脚本语言不同的是，bash使用了很多特殊的变量来表示参数、错误代码和相关变量。下面是列举来其中一些变量，更完整的列表可以参考 [这里](#)。

- `$0` - 脚本名

- \$1 到 \$9 - 脚本的参数。 \$1 是第一个参数，依此类推。
- \$@ - 所有参数
- \$# - 参数个数
- \$? - 前一个命令的返回值
- \$\$ - 当前脚本的进程识别码
- !! - 完整的上一条命令，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。

命令通常使用 STDOUT 来返回输出值，使用 STDERR 来返回错误及错误码，便于脚本以更加友好的方式报告错误。返回码或退出状态是脚本/命令之间交流执行状态的方式。返回值0表示正常执行，其他所有非0的返回值都表示有错误发生。

退出码可以搭配 && (与操作符) 和 || (或操作符) 使用，来进行条件判断，决定是否执行其他程序。它们都属于短路运算符 (short-circuiting) 同一行的多个命令可以用 ; 分隔。程序 true 的返回码永远是 0， false 的返回码永远是 1。让我们看几个例子

```
false || echo "Oops, fail"
# Oops, fail

true || echo "Will not be printed"
#
# Things went well
# Things went well

false && echo "Will not be printed"
#
# This will always run"
# This will always run
```

另一个常见的模式是以变量的形式获取一个命令的输出，这可以通过 命令替换 (*command substitution*) 实现。

当您通过 \$( CMD ) 这样的方式来执行 CMD 这个命令时，它的输出结果会替换掉 \$( CMD ) 。例如，如果执行 for file in \$(ls) ， shell首先将调用 ls ，然后遍历得到的这些返回值。还有一个冷门的类似特性是 进程替换 (*process substitution*) ， <( CMD ) 会执行 CMD 并将结果输出到一个临时文件中，并将 <( CMD ) 替换成临时文件名。这在我们希望返回值通过文件而不是STDIN传递时很有用。例如， diff <(ls foo) <(ls bar) 会显示文件夹 foo 和 bar 中文件的区别。

说了很多，现在该看例子了，下面这个例子展示了一部分上面提到的特性。这段脚本会遍历我们提供的参数，使用 grep 搜索字符串 foobar ，如果没有找到，则将其作为注释追加到文件中。

```
#!/bin/bash

echo "Starting program at $(date)" # date会被替换成日期和时间

echo "Running program $0 with $# arguments with pid $$"

for file in "$@"; do
    grep foobar "$file" > /dev/null 2> /dev/null
    # 如果模式没有找到，则grep退出状态为 1
    # 我们将标准输出流和标准错误流重定向到Null，因为我们并不关心这些信息
    if [[ $? -ne 0 ]]; then
        echo "File $file does not have any foobar, adding one"
        echo "# foobar" >> "$file"
    fi
done
```

在条件语句中，我们比较 `$?` 是否等于0。Bash实现了许多类似的比较操作，您可以查看 [test 手册](#)。在bash中进行比较时，尽量使用双方括号 `[[ ]]` 而不是单方括号 `[ ]`，这样会降低犯错的几率，尽管这样并不能兼容 sh。更详细的说明参见[这里](#)。

当执行脚本时，我们经常需要提供形式类似的参数。bash使我们可以轻松的实现这一操作，它可以基于文件扩展名展开表达式。这一技术被称为shell的 通配 (globbing)

- 通配符 - 当你想要利用通配符进行匹配时，你可以分别使用 `?` 和 `*` 来匹配一个或任意个字符。例如，对于文件 `foo`, `foo1`, `foo2`, `foo10` 和 `bar`, `rm foo?` 这条命令会删除 `foo1` 和 `foo2`，而 `rm foo*` 则会删除除了 `bar` 之外的所有文件。
- 花括号 `{}` - 当你有一系列的指令，其中包含一段公共子串时，可以用花括号来自动展开这些命令。这在批量移动或转换文件时非常方便。

```

convert image.{png,jpg}
# 会展开为
convert image.png image.jpg

cp /path/to/project/{foo,bar,baz}.sh /newpath
# 会展开为
cp /path/to/project/foo.sh /path/to/project/bar.sh /path/to/project/baz.sh

# 也可以结合通配使用
mv *{.py,.sh} folder
# 会移动所有 *.py 和 *.sh 文件

mkdir foo bar

# 下面命令会创建 foo/a, foo/b, ... foo/h, bar/a, bar/b, ... bar/h 这些文件
touch {foo,bar}/{a..h}
touch foo/x bar/y
# 比较文件夹 foo 和 bar 中包含文件的不同
diff <(ls foo) <(ls bar)
# 输出
# < x
# ---
# > y

```

编写 bash 脚本有时候会很别扭和反直觉。例如 [shellcheck](#) 这样的工具可以帮助你定位 sh/bash 脚本中的错误。

注意，脚本并不一定只有用 bash 写才能在终端里调用。比如说，这是一段 Python 脚本，作用是将输入的参数倒序输出：

```

#!/usr/local/bin/python
import sys
for arg in reversed(sys.argv[1:]):
    print(arg)

```

内核知道去用 python 解释器而不是 shell 命令来运行这段脚本，是因为脚本的开头第一行的 [shebang](#)。

在 shebang 行中使用 [env](#) 命令是一种好的实践，它会利用环境变量中的程序来解析该脚本，这样就提高来您的脚本的可移植性。env 会利用我们第一节讲座中介绍过的 PATH 环境变量来进行定位。例如，使用了 env 的 shebang 看上去时这样的 `#!/usr/bin/env python`。

shell 函数和脚本有如下一些不同点：

- 函数只能与shell使用相同的语言，脚本可以使用任意语言。因此在脚本中包含 shebang 是很重要的。
- 函数仅在定义时被加载，脚本会在每次被执行时加载。这让函数的加载比脚本略快一些，但每次修改函数定义，都要重新加载一次。
- 函数会在当前的shell环境中执行，脚本会在单独的进程中执行。因此，函数可以对环境变量进行更改，比如改变当前工作目录，脚本则不行。脚本需要使用 [export](#) 将环境变量导出，并将值传递给环境变量。
- 与其他程序语言一样，函数可以提高代码模块性、代码复用性并创建清晰性的结构。shell脚本中往往也会包含它们自己的函数定义。

## Shell 工具

### 查看命令如何使用

看到这里，您可能会有疑问，我们应该如何为特定的命令找到合适的标记呢？例如 `ls -l`, `mv -i` 和 `mkdir -p`。更普遍的是，给您一个命令行，您应该怎样了解如何使用这个命令行并找出它的不同的选项呢？一般来说，您可能会先去网上搜索答案，但是，UNIX 可比 StackOverflow 出现的早，因此我们的系统里其实早就包含了可以获取相关信息的方法。

在上一节中我们介绍过，最常用的方法是为对应的命令行添加 `-h` 或 `--help` 标记。另外一个更详细的方法则是使用 `man` 命令。[man](#) 命令是手册 (manual) 的缩写，它提供了命令的用户手册。

例如，`man rm` 会输出命令 `rm` 的说明，同时还有其标记列表，包括之前我们介绍过的 `-i`。事实上，目前我们给出的所有命令的说明链接，都是网页版的Linux命令手册。即使是您安装的第三方命令，前提是开发者编写了手册并将其包含在了安装包中。在交互式的、基于字符处理的终端窗口中，一般也可以通过 `:help` 命令或键入 `?` 来获取帮助。

有时候手册内容太过详实，让我们难以在其中查找哪些最常用的标记和语法。[TLDR pages](#) 是一个很不错的替代品，它提供了一些案例，可以帮助您快速找到正确的选项。

例如，自己就常常在tldr上搜索 [tar](#) 和 [ffmpeg](#) 的用法。

### 查找文件

程序员们面对的最常见的重复任务就是查找文件或目录。所有的类UNIX系统都包含一个名为 [find](#) 的工具，它是 shell 上用于查找文件的绝佳工具。`find` 命令会递归地搜索符合条件的文件，例如：

```
# 查找所有名称为src的文件夹
find . -name src -type d
# 查找所有文件夹路径中包含test的python文件
find . -path '*/test/*.py' -type f
# 查找前一天修改的所有文件
find . -mtime -1
# 查找所有大小在500k至10M的tar.gz文件
find . -size +500k -size -10M -name '*.tar.gz'
```

除了列出所寻找的文件之外，find 还能对所有查找到的文件进行操作。这能极大地简化一些单调的任务。

```
# 删除全部扩展名为.tmp 的文件
find . -name '*.\tmp' -exec rm {} \;
# 查找全部的 PNG 文件并将其转换为 JPG
find . -name '*.\png' -exec convert {} {}.\jpg \;
```

尽管 find 用途广泛，它的语法却比较难以记忆。例如，为了查找满足模式 PATTERN 的文件，您需要执行 `find -name '*PATTERN*'` (如果您希望模式匹配时是不区分大小写，可以使用 `-iname` 选项)

您当然可以使用 alias 设置别名来简化上述操作，但 shell 的哲学之一便是寻找（更好用的）替代方案。记住，shell 最好的特性就是您只是在调用程序，因此您只要找到合适的替代程序即可（甚至自己编写）。

例如，`fd` 就是一个更简单、更快速、更友好的程序，它可以用来作为 find 的替代品。它有很多不错的默认设置，例如输出着色、默认支持正则匹配、支持unicode并且我认为它的语法更符合直觉。以模式 PATTERN 搜索的语法是 `fd PATTERN`。

大多数人都认为 find 和 fd 已经很好用了，但是有的人可能想知道，我们是不是可以有更高效的方法，例如不要每次都搜索文件而是通过编译索引或建立数据库的方式来实现更加快速地搜索。

这就要靠 [locate](#) 了。 locate 使用一个由 [updatedb](#) 负责更新的数据库，在大多数系统中 updatedb 都会通过 [cron](#) 每日更新。这便需要我们在速度和时效性之间作出权衡。而且，find 和类似的工具可以通过别的属性比如文件大小、修改时间或是权限来查找文件，locate 则只能通过文件名。[这里](#)有一个更详细的对比。

## 查找代码

查找文件是很有用的技能，但是很多时候您的目标其实是查看文件的内容。一个最常见的场景是您希望查找具有某种模式的全部文件，并找它们的位置。

为了实现这一点，很多类UNIX的系统都提供了 [grep](#) 命令，它是用于对输入文本进行匹配的通用工具。它是一个非常重要的shell工具，我们会在后续的数据清理课程中深入的探讨它。

grep 有很多选项，这也使它成为一个非常全能的工具。其中我经常使用的有 -C：获取查找结果的上下文（Context）； -v 将对结果进行反选（Invert），也就是输出不匹配的结果。举例来说， grep -C 5 会输出匹配结果前后五行。当需要搜索大量文件的时候，使用 -R 会递归地进入子目录并搜索所有的文本文件。

但是，我们有很多办法可以对 grep -R 进行改进，例如使其忽略 .git 文件夹，使用多CPU等等。

因此也出现了很多它的替代品，包括 [ack](#), [ag](#) 和 [rg](#)。它们都特别好用，但是功能也都差不多，我比较常用的是 ripgrep (rg)，因为它速度快，而且用法非常符合直觉。例子如下：

```
# 查找所有使用了 requests 库的文件
rg -t py 'import requests'

# 查找所有没有写 shebang 的文件（包含隐藏文件）
rg -u --files-without-match "^#!"

# 查找所有的 foo 字符串，并打印其之后的 5 行
rg foo -A 5

# 打印匹配的统计信息（匹配的行和文件的数量）
rg --stats PATTERN
```

与 find / fd 一样，重要的是你要知道有些问题使用合适的工具就会迎刃而解，而具体选择哪个工具则不是那么重要。

## 查找 shell 命令

目前为止，我们已经学习了如何查找文件和代码，但随着你使用shell的时间越来越久，您可能想要找到之前输入过的某条命令。首先，按向上的方向键会显示你使用过的上一条命令，继续按上键则会遍历整个历史记录。

`history` 命令允许您以程序员的方式来访问shell中输入的历史命令。这个命令会在标准输出中打印shell中的里面命令。如果我们要搜索历史记录，则可以利用管道将输出结果传递给 grep 进行模式搜索。`history | grep find` 会打印包含find子串的命令。

对于大多数的shell来说，您可以使用 `Ctrl+R` 对命令历史记录进行回溯搜索。敲 `Ctrl+R` 后您可以输入子串来进行匹配，查找历史命令行。

反复按下就会在所有搜索结果中循环。在 [zsh](#) 中，使用方向键上或下也可以完成这项工作。

`Ctrl+R` 可以配合 [fzf](#) 使用。`fzf` 是一个通用对模糊查找工具，它可以和很多命令一起使用。这里我们可以对历史命令进行模糊查找并将结果以赏心悦目的格式输出。

另外一个和历史命令相关的技巧我喜欢称之为**基于历史的自动补全**。这一特性最初是由 [fish](#) shell 创建的，它可以根据您最近使用过的开头相同的命令，动态地对当前对shell命令进行补全。这一功能在 [zsh](#) 中也可以使用，它可以极大的提高用户体验。

你可以修改 shell history 的行为，例如，如果在命令的开头加上一个空格，它就不会被加进shell记录中。当你输入包含密码或是其他敏感信息的命令时会用到这一特性。为此你需要

在 `.bashrc` 中添加 `HISTCONTROL=ignorespace` 或者向 `.zshrc` 添加 `setopt HIST_IGNORE_SPACE`。如果你不小心忘了在前面加空格，可以通过编辑 `bash_history` 或 `.zhistory` 来手动地从历史记录中移除那一项。

## 文件夹导航

之前对所有操作我们都默认一个前提，即您已经位于想要执行命令的目录下，但是如何才能高效地在目录间随意切换呢？有很多简便的方法可以做到，比如设置alias，使用 [ln -s](#) 创建符号连接等。而开发者们已经想到了很多更为精妙的解决方案。

由于本课程的目的是尽可能对你的日常习惯进行优化。因此，我们可以使用 [fasd](#) 和 [autojump](#) 这两个工具来查找最常用或最近使用的文件和目录。

`Fasd` 基于 [\*frecency\*](#) 对文件和文件排序，也就是说它会同时针对频率 (*frequency*) 和时效 (*recency*) 进行排序。默认情况下，`fasd` 使用命令 `z` 帮助我们快速切换到最常访问的目录。例如，如果您经常访问 `/home/user/files/cool_project` 目录，那么可以直接使用 `z cool` 跳转到该目录。对于 `autojump`，则使用 `j cool` 代替即可。

还有一些更复杂的工具可以用来概览目录结构，例如 [tree](#), [broot](#) 或更加完整的文件管理器，例如 [nnn](#) 或 [ranger](#)。

## 课后练习

### 习题解答

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. 本节课我们讲解的 `find` 命令中的 `-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`，译注：MacOS 上的 `xargs` 没有 `-d`，[查看这个issue](#)）

如果您使用的是 MacOS，请注意默认的 BSD `find` 与 [GNU coreutils](#) 中的是不一样的。你可以为 `find` 添加 `-print0` 选项，并为 `xargs` 添加 `-0` 选项。作为 Mac 用户，您需要注意 mac 系统自带的命令行工具和 GNU 中对应的工具是有区别的；如果你想使用 GNU 版本的工具，也可以使用 [brew 来安装](#)。

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？

# 编辑器 (Vim)

写作和写代码其实是两项非常不同的活动。当我们编程的时候，会经常在文件间进行切换、阅读、浏览和修改代码，而不是连续编写一大段的文字。因此代码编辑器和文本编辑器是很不同的两种工具（例如微软的 Word 与 Visual Studio Code）。

作为程序员，我们大部分时间都花在代码编辑上，所以花点时间掌握某个适合自己的编辑器是非常值得的。通常学习使用一个新的编辑器包含以下步骤：

- 阅读教程（比如这节课以及我们为您提供的资源）
- 坚持使用它来完成你所有的编辑工作（即使一开始这会让你的工作效率降低）
- 随时查阅：如果某个操作看起来像是有更方便的实现方法，一般情况下真的会有

如果您能够遵循上述步骤，并且坚持使用新的编辑器完成您所有的文本编辑任务，那么学习一个复杂的代码编辑器的过程一般是这样的：头两个小时，您会学习到编辑器的基本操作，例如打开和编辑文件、保存与退出、浏览缓冲区。当学习时间累计达到20个小时之后，您使用新编辑器的效率应该已经和使用老编辑器一样快。在此之后，其益处开始显现：有了足够的知识和肌肉记忆后，使用新编辑器将大大节省你的时间。而现代文本编辑器都是些复杂且强大的工具，永远有新东西可学：学的越多，效率越高。

## 该学哪个编辑器？

程序员们对自己正在使用的文本编辑器通常有着 非常强的执念。

现在最流行的编辑器是什么？[Stack Overflow 的调查](#)（这个调查可能并不如我们想象的那样客观，因为 Stack Overflow 的用户并不能代表所有程序员）显示，[Visual Studio Code](#) 是目前最流行的代码编辑器。而 [Vim](#) 则是最流行的基于命令行的编辑器。

## Vim

这门课的所有教员都使用 Vim 作为编辑器。Vim 有着悠久历史；它始于 1976 年的 Vi 编辑器，到现在还在 不断开发中。Vim 有很多聪明的设计思想，所以很多其他工具也支持 Vim 模式（比如，140 万人安装了 [Vim emulation for VS code](#)）。即使你最后使用 其他编辑器，Vim 也值得学习。

由于不可能在 50 分钟内教授 Vim 的所有功能，我们会专注于解释 Vim 的设计哲学，教你基础知识，并展示一部分高级功能，然后给你掌握这个工具所需要的资源。

## 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 是一个多模式编辑器：它对于插入文字和操纵文字有不同的模式。Vim 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。

这样的设计哲学使得 Vim 成为了一个能跟上你思维速度的编辑器。

## 编辑模式

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**: 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
- **可视化模式** (一般, 行, 块) : 选中文本块
- **命令模式**: 用于执行命令

在不同的操作模式下, 键盘敲击的含义也不同。比如, `x` 在插入模式会插入字母 `x`, 但是在正常模式会删除当前光标所在的字母, 在可视模式下则会删除选中文块。

在默认设置下, Vim 会在左下角显示当前的模式。Vim 启动时的默认模式是正常模式。通常你会把大部分时间花在正常模式和插入模式。

你可以按下 `<ESC>` (退出键) 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。

因为你会在使用 Vim 时大量使用 `<ESC>` 键, 所以可以考虑把大小写锁定键重定义成 `<ESC>` 键 ([MacOS 教程](#))。

## 基本操作

### 插入文本

在正常模式, 键入 `i` 进入插入模式。现在 Vim 跟很多其他的编辑器一样, 直到你键入 `<ESC>` 返回正常模式。你只需要掌握这一点和上面介绍的所有基础知识就可以使用 Vim 来编辑文件了 (虽然如果你一直停留在插入模式内不一定高效)。

### 缓存, 标签页, 窗口

Vim 会维护一系列打开的文件, 称为 “缓存”。一个 Vim 会话包含一系列标签页, 每个标签页包含一系列窗口 (分隔面板)。每个窗口显示一个缓存。跟网页浏览器等其他你熟悉的程序不一样的是, 缓存和窗口不是一一对应的关系; 窗口只是视角。一个缓存可以在多个窗口打开, 甚至在同一个标签页内的多个窗口打开。这个功能其实很好用, 比如在查看同一个文件的不同部分的时候。

Vim 默认打开一个标签页, 这个标签也包含一个窗口。

## 命令行

在正常模式下键入 `:` 进入命令行模式。在键入 `:` 后，你的光标会立即跳到屏幕下方的命令行。这个模式有很多功能，包括打开，保存，关闭文件，以及 [退出 Vim](#)。

- `: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 {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

## Vim 的接口其实是一种编程语言

Vim 最重要的设计思想是 Vim 的界面本身是一个程序语言。键入操作（以及他们的助记名）本身就是命令，这些命令可以组合使用。这使得移动和编辑更加高效，特别是一旦形成肌肉记忆。

## 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 里面移动也被称为“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: `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)
- 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: `v`
- 可视化行: `V`
- 可视化块: `Ctrl+v`

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，dθ 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串，包括周围的单引号

## 演示

这里是一个有问题的 [fizz buzz](#) 实现：

```

def fizz_buzz(limit):
    for i in range(limit):
        if i % 3 == 0:
            print('fizz')
        if i % 5 == 0:
            print('buzz')
        if i % 3 and i % 5:
            print(i)

def main():
    fizz_buzz(10)

```

我们会修复以下问题：

- 主函数没有被调用
- 从 0 而不是 1 开始
- 在 15 的整数倍的时候在不用行打印 “fizz” 和 “buzz”
- 在 5 的整数倍的时候打印 “fizz”
- 采用硬编码的参数 10 而不是从命令控制行读取参数
  
- 主函数没有被调用
  - G 文件尾
  - o 向下打开一个新行
  - 输入 “if name ...”
- 从 0 而不是 1 开始
  - 搜索 /range
  - ww 向前移动两个词
  - i 插入文字, “1, ”
  - ea 在 limit 后插入, “+1”
- 在新的一行 “fizzbuzz”
  - jj\$ i 插入文字到行尾
  - 加入 “, end=' ' ”
  - jj. 重复第二个打印
  - jjo 在 if 打开一行
  - 加入 “else: print()”
- fizz fizz
  - ci' 变到 fizz
- 命令控制行参数
  - gg0 向上打开
  - “import sys”
  - /10
  - ci( to “int(sys.argv[1])”

展示详情请观看课程视频。比较上面用 Vim 的操作和你可能使用其他程序的操作。值得一提的是 Vim 需要很少的键盘操作，允许你编辑的速度跟上你思维的速度。

# 自定义 Vim

Vim 由一个位于 `~/.vimrc` 的文本配置文件（包含 Vim 脚本命令）。你可能会启用很多基本设置。

我们提供一个文档详细的基本设置，你可以用它当作你的初始设置。我们推荐使用这个设置因为它修复了一些 Vim 默认设置奇怪行为。[在这儿 下载我们的设置，然后将它保存成 `~/.vimrc`。](#)

Vim 能够被重度自定义，花时间探索自定义选项是值得的。你可以参考其他人的在 GitHub 上共享的设置文件，比如，你的授课人的 Vim 设置 ([Anish, Jon](#) (uses [neovim](#)), [Jose](#))。有很多好的博客文章也聊到了这个话题。尽量不要复制粘贴别人的整个设置文件，而是阅读和理解它，然后采用对你有用的部分。

## 扩展 Vim

Vim 有很多扩展插件。跟很多互联网上已经过时的建议相反，你不需要在 Vim 使用一个插件管理器（从 Vim 8.0 开始）。你可以使用内置的插件管理系统。只需要创建一个 `~/.vim/pack/vendor/start/` 的文件夹，然后把插件放到这里（比如通过 `git clone`）。

以下是一些我们最爱的插件：

- [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

我们尽量避免在这里提供一份冗长的插件列表。你可以查看讲师们的开源的配置文件 ([Anish, Jon, Jose](#)) 来看看我们使用的其他插件。浏览 [Vim Awesome](#) 来了解一些很棒的插件。这个话题也有很多博客文章：搜索“best Vim plugins”。

## 其他程序的 Vim 模式

很多工具提供了 Vim 模式。这些 Vim 模式的质量参差不齐；取决于具体工具，有的提供了很多酷炫的 Vim 功能，但是大多数对基本功能支持的很好。

## Shell

如果你是一个 Bash 用户，用 `set -o vi`。如果你用 Zsh: `bindkey -v`。Fish 用 `fish_vi_key_bindings`。另外，不管利用什么 shell，你可以 `export EDITOR=vim`。这是一个用来决定当一个程序需要启动编辑时启动哪个的环境变量。例如，`git` 会使用这个编辑器来编辑 `commit` 信息。

## Readline

很多程序使用 [GNU Readline](#) 库来作为它们的命令控制行界面。Readline 也支持基本的 Vim 模式，可以通过在 `~/.inputrc` 添加如下行开启：

```
set editing-mode vi
```

比如，在这个设置下，Python REPL 会支持 Vim 快捷键。

## 其他

甚至有 Vim 的网页浏览快捷键 [browsers](#)，受欢迎的有用于 Google Chrome 的 [Vimium](#) 和用于 Firefox 的 [Tridactyl](#)。你甚至可以在 [Jupyter notebooks](#) 中用 Vim 快捷键。[这个列表](#) 中列举了支持类 vim 键位绑定的软件。

## Vim 进阶

这里我们提供了一些展示这个编辑器能力的例子。我们无法把所有的这样的事情都教给你，但是你可以在使用中学习。一个好的对策是：当你在使用你的编辑器的时候感觉“一定有更好的方法来做这个”，那么很可能真的有：上网搜寻一下。

### 搜索和替换

:s (替换) 命令 ([文档](#))。

- %s/foo/bar/g
  - 在整个文件中将 foo 全局替换成 bar
- %s/\[.\*\](\(.\*)\)/\1/g
  - 将有命名的 Markdown 链接替换成简单 URLs

### 多窗口

- 用 :sp / :vsp 来分割窗口
- 同一个缓存可以在多个窗口中显示。

### 宏

- q{字符} 来开始在寄存器 {字符} 中录制宏
- q 停止录制
- @{字符} 重放宏
- 宏的执行遇错误会停止
- {计数}@{字符} 执行一个宏{计数}次
- 宏可以递归
  - 首先用 q{字符}q 清除宏
  - 录制该宏，用 @{字符} 来递归调用该宏（在录制完成之前不会有任何操作）
- 例子：将 xml 转成 json ([file](#))
  - 一个有 “name” / “email” 键对象的数组
  - 用一个 Python 程序？
  - 用 sed / 正则表达式
    - g/people/d
    - %s/<person>/{/g

- `%s/<name>\(.*)<\/name>/{"name": "\1", /g`
- ...
- Vim 命令 / 宏
  - Gdd, ggdd 删除第一行和最后一行
  - 格式化最后一个元素的宏 (寄存器 e)
    - 跳转到有 `<name>` 的行
    - `qe^r" f>s": "<ESC>f<C"<ESC>q`
  - 格式化一个的宏
    - 跳转到有 `<person>` 的行
    - `qpS{<ESC>j@eA,<ESC>j@ejS},<ESC>q`
  - 格式化一个标签然后转到另外一个的宏
    - 跳转到有 `<person>` 的行
    - `qq@pjq`
  - 执行宏到文件尾
    - `999@qq`
  - 手动移除最后的 , 然后加上 [ 和 ] 分隔符

## 扩展资料

- vimtutor 是一个 Vim 安装时自带的教程
- [Vim Adventures](#) 是一个学习使用 Vim 的游戏
- [Vim Tips Wiki](#)
- [Vim Advent Calendar](#) 有很多 Vim 小技巧
- [Vim Golf](#) 是用 Vim 的用户界面作为程序语言的 [code golf](#)
- [Vi/Vim Stack Exchange](#)
- [Vim Screencasts](#)
- [Practical Vim](#) (书籍)

## 课后练习

### 习题解答

1. 完成 vimtutor。备注：它在一个 [80x24](#) (80 列, 24 行) 终端窗口看起来效果最好。
2. 下载我们提供的 [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. 阅读这个插件的 [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 完成所有的文件编辑。每当不够高效的时候，或者你感觉“一定有一个更好的方式”时，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，可以来我们的答疑时间或者给我们发邮件。
6. 在其他工具中设置 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。

# 数据整理

您是否曾经有过这样的需求，将某种格式存储的数据转换成另外一种格式？肯定有过，对吧！这也正是我们这节课所要讲授的主要内容。具体来讲，我们需要不断地对数据进行处理，直到得到我们想要的最终结果。

在之前的课程中，其实我们已经接触到了一些数据整理的基本技术。可以这么说，每当您使用管道运算符的时候，其实就是在进行某种形式的数据整理。

例如这样一条命令 `journalctl | grep -i intel`，它会找到所有包含intel(不区分大小写)的系统日志。您可能并不认为这是数据整理，但是它确实将某种形式的数据（全部系统日志）转换成了另外一种形式的数据（仅包含intel的日志）。大多数情况下，数据整理需要您能够明确哪些工具可以被用来达成特定数据整理的目的，并且明白如何组合使用这些工具。

让我们从头讲起。既然是学习数据整理，那有两样东西自然是必不可少的：用来整理的数据以及相关的应用场景。日志处理通常是一个比较典型的使用场景，因为我们经常需要在日志中查找某些信息，这种情况下通读日志是不现实的。现在，让我们研究一下系统日志，看看哪些用户曾经尝试过登录我们的服务器：

```
ssh myserver journalctl
```

内容太多了。现在让我们把涉及 sshd 的信息过滤出来：

```
ssh myserver journalctl | grep sshd
```

注意，这里我们使用管道将一个远程服务器上的文件传递给本机的 grep 程序！ssh 太牛了，下一节课我们会讲授命令行环境，届时我们会详细讨论 ssh 的相关内容。此时我们打印出的内容，仍然比我们需要的要多得多，读起来也非常费劲。我们来改进一下：

```
ssh myserver 'journalctl | grep sshd | grep "Disconnected from"' | less
```

多出来的引号是什么作用呢？这么说吧，我们的日志是一个非常大的文件，把这么大的文件流直接传输到我们本地的电脑上再进行过滤是对流量的一种浪费。因此我们采取另外一种方式，我们先在远端机器上过滤文本内容，然后再将结果传输到本机。less 为我们创建来一个文件分页器，使我们可以通过翻页的方式浏览较长的文本。为了进一步节省流量，我们甚至可以将当前过滤出的日志保存到文件中，这样后续就不需要再次通过网络访问该文件了：

```
$ ssh myserver 'journalctl | grep sshd | grep "Disconnected from"' > ssh.log  
$ less ssh.log
```

过滤结果中仍然包含不少没用的数据。我们有很多办法可以删除这些无用的数据，但是让我们先研究一下 sed 这个非常强大的工具。

sed 是一个基于文本编辑器 ed 构建的“流编辑器”。在 sed 中，您基本上是利用一些简短的命令来修改文件，而不是直接操作文件的内容（尽管您也可以选择这样做）。相关的命令行非常多，但是最常用的是 s，即替换命令，例如我们可以这样写：

```
ssh myserver journalctl
| grep sshd
| grep "Disconnected from"
| sed 's/.*Disconnected from //'
```

上面这段命令中，我们使用了一段简单的正则表达式。正则表达式是一种非常强大的工具，可以让我们基于某种模式来对字符串进行匹配。s 命令的语法如下：`s/REGEX/SUBSTITUTION/`，其中 REGEX 部分是我们需要使用的正则表达式，而 SUBSTITUTION 是用于替换匹配结果的文本。

## 正则表达式

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：

- . 除换行符之外的“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

sed 的正则表达式有些时候是比较奇怪的，它需要你在这些模式前添加 \ 才能使其具有特殊含义。或者，您也可以添加 -E 选项来支持这些匹配。

回过头我们再看 `/.*Disconnected from /`，我们会发现这个正则表达式可以匹配任何以若干任意字符开头，并接着包含“Disconnected from”的字符串。这也正式我们所希望的。但是请注意，正则表达式并不容易写对。如果有人将“Disconnected from”作为自己的用户名会怎样呢？



```
Jan 17 03:13:00 thesquareplanet.com sshd[2631]: Disconnected from invalid
```

正则表达式会如何匹配？`*` 和 `+` 在默认情况下是贪婪模式，也就是说，它们会尽可能多的匹配文本。因此对上述字符串的匹配结果如下：

```
46.97.239.16 port 55920 [preauth]
```

这可不是我们想要的结果。对于某些正则表达式的实现来说，您可以给 `*` 或 `+` 增加一个`?` 后缀使其变成非贪婪模式，但是很可惜 `sed` 并不支持该后缀。不过，我们可以切换到 `perl` 的命令行模式，该模式支持编写这样的正则表达式：

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

让我们回到 `sed` 命令并使用它完成后续的任务，毕竟对于这一类任务，`sed` 是最常见的工具。`sed` 还可以非常方便的做一些事情，例如打印匹配后的内容，一次调用中进行多次替换搜索等。但是这些内容我们并不会在此进行介绍。`sed` 本身是一个非常全能的工具，但是在具体功能上往往能找到更好的工具作为替代品。

好的，我们还需要去掉用户名后面的后缀，应该如何操作呢？

想要匹配用户名后面的文本，尤其是当这里的用户名可以包含空格时，这个问题变得非常棘手！这里我们需要做的是匹配一整行：

```
| sed -E 's/.+Disconnected from (invalid |authenticating )?user .+ [^ ]+'
```

让我们借助正则表达式在线调试工具[regex debugger](#) 来理解这段表达式。OK，开始的部分和以前是一样的，随后，我们匹配两种类型的“user”（在日志中基于两种前缀区分）。再然后我们匹配属于用户名的所有字符。接着，再匹配任意一个单词（`[^ ]+` 会匹配任意非空且不包含空格的序列）。紧接着后面匹配单“port”和它后面的一串数字，以及可能存在的后缀`[preauth]`，最后再匹配行尾。

注意，这样做的话，即使用户名是“Disconnected from”，对匹配结果也不会有任何影响，您知道这是为什么吗？

问题还没有完全解决，日志的内容全部被替换了空字符串，整个日志的内容因此都被删除了。我们实际上希望能够将用户名保留下来。对此，我们可以使用“捕获组 (capture groups)”来完成。被圆括号内的正则表达式匹配到的文本，都会被存入一系列以编号区分的捕获组中。捕获组的内容可以在替换字符串时使用（有些正则表达式的引擎甚至支持替换表达式本身），例如`\1`、`\2`、`\3` 等等，因此可以使用如下命令：

```
| sed -E 's/.+Disconnected from (invalid |authenticating )?user (.*) [^ ]+'
```

想必您已经意识到了，为了完成某种匹配，我们最终可能会写出非常复杂的正则表达式。例如，这里有一篇关于如何匹配电子邮箱地址的文章[e-mail address](#)，匹配电子邮箱可一点也不简单。

网络上还有很多关于如何匹配电子邮箱地址的讨论。人们还为其编写了测试用例及测试矩阵。您甚至可以编写一个用于判断一个数是否为质数的正则表达式。

正则表达式是出了名的难以写对，但是它仍然会是您强大的常备工具之一。

## 回到数据整理

OK，现在我们有如下表达式：

```
ssh myserver journalctl
| grep sshd
| grep "Disconnected from"
| sed -E 's/.*Disconnected from (invalid |authenticating )?user (.*) [^'
```

sed 还可以做很多各种各样有趣的事情，例如文本注入：(使用 i 命令)，打印特定的行 (使用 p 命令)，基于索引选择特定行等等。详情请见 man sed！

现在，我们已经得到了一个包含用户名的列表，列表中的用户都曾经尝试过登录我们的系统。但这还不够，让我们过滤出那些最常出现的用户：

```
ssh myserver journalctl
| grep sshd
| grep "Disconnected from"
| sed -E 's/.*Disconnected from (invalid |authenticating )?user (.*) [^'
| sort | uniq -c
```

sort 会对其输入数据进行排序。uniq -c 会把连续出现的行折叠为一行并使用出现次数作为前缀。我们希望按照出现次数排序，过滤出最常出现的用户名：

```
ssh myserver journalctl
| grep sshd
| grep "Disconnected from"
| sed -E 's/.*Disconnected from (invalid |authenticating )?user (.*) [^'
| sort | uniq -c
| sort -nk1,1 | tail -n10
```

sort -n 会按照数字顺序对输入进行排序 (默认情况下是按照字典序排序) -k1,1 则表示“仅基于以空格分割的第一列进行排序”。,n 部分表示“仅排序到第 n 个部分”，默认情况是到行尾。就本例来说，针对整个行进行排序也没有任何问题，我们这里主要是为了学习这一用法！

如果我们希望得到登录次数最少的用户，我们可以使用 head 来代替 tail。或者使用 sort -r 来进行倒序排序。

相当不错。但我们只想获取用户名，而且不要一行一个地显示。

```
ssh myserver journalctl
| grep sshd
| grep "Disconnected from"
| sed -E 's/.*Disconnected from (invalid |authenticating )?user (.*) [^'
| sort | uniq -c
| sort -nk1,1 | tail -n10
| awk '{print $2}' | paste -sd,
```

如果您使用的是 MacOS：注意这个命令并不能配合 MacOS 系统默认的 BSD `paste` 使用。参考 [课程概览与 shell](#) 的习题内容获取更多相关信息。

我们可以利用 `paste` 命令来合并行 (`-s`)，并指定一个分隔符进行分割 (`-d`)，那 `awk` 的作用又是什么呢？

## awk – 另外一种编辑器

`awk` 其实是一种编程语言，只不过它碰巧非常善于处理文本。关于 `awk` 可以介绍的内容太多了，限于篇幅，这里我们仅介绍一些基础知识。

首先，`{print $2}` 的作用是什么？`awk` 程序接受一个模式串（可选），以及一个代码块，指定当模式匹配时应该做何种操作。默认当模式串即匹配所有行（上面命令中当用法）。在代码块中，`$0` 表示整行的内容，`$1` 到 `$n` 为一行中的 `n` 个区域，区域的分割基于 `awk` 的域分隔符（默认是空格，可以通过 `-F` 来修改）。在这个例子中，我们的代码意思是：对于每一行文本，打印其第二个部分，也就是用户名。

让我们康康，还有什么炫酷的操作可以做。让我们统计一下所有以 `c` 开头，以 `e` 结尾，并且仅尝试过一次登录的用户。

```
| awk '$1 == 1 && $2 ~ /^c[^ ]*e$/ { print $2 }' | wc -l
```

让我们好好分析一下。首先，注意这次我们为 `awk` 指定了一个匹配模式串（也就是 `{...}` 前面的那部分内容）。该匹配要求文本的第一部分需要等于1（这部分刚好是 `uniq -c` 得到的计数值），然后其第二部分必须满足给定的一个正则表达式。代码块中的内容则表示打印用户名。然后我们使用 `wc -l` 统计输出结果的行数。

不过，既然 `awk` 是一种编程语言，那么则可以这样：

```
BEGIN { rows = 0 }
$1 == 1 && $2 ~ /^c[^ ]*e$/ { rows += $1 }
END { print rows }
```

BEGIN 也是一种模式，它会匹配输入的开头（END 则匹配结尾）。然后，对每一行第一个部分进行累加，最后将结果输出。事实上，我们完全可以抛弃 grep 和 sed，因为 awk 就可以解决所有问题。至于怎么做，就留给读者们做课后练习吧。

## 分析数据

想做数学计算也是可以的！例如这样，您可以将每行的数字加起来：

```
| paste -sd+ | bc -l
```

下面这种更加复杂的表达式也可以：

```
echo "2*($(data | paste -sd+))" | bc -l
```

您可以通过多种方式获取统计数据。如果已经安装了R语言，[st](#) 是个不错的选择：

```
ssh myserver journalctl
| grep sshd
| grep "Disconnected from"
| sed -E 's/.*Disconnected from (invalid |authenticating )?user (.*) [^'
| sort | uniq -c
| awk '{print $1}' | R --slave -e 'x <- scan(file="stdin", quiet=TRUE);'
```

R 也是一种编程语言，它非常适合被用来进行数据分析和[绘制图表](#)。这里我们不会讲的特别详细，您只需要知道 summary 可以打印某个向量的统计结果。我们将输入的一系列数据存放在一个向量后，利用R语言就可以得到我们想要的统计数据。

如果您希望绘制一些简单的图表，gnuplot 可以帮助到您：

```
ssh myserver journalctl
| grep sshd
| grep "Disconnected from"
| sed -E 's/.*Disconnected from (invalid |authenticating )?user (.*) [^'
| sort | uniq -c
| sort -nk1,1 | tail -n10
| gnuplot -p -e 'set boxwidth 0.5; plot "-" using 1:xtic(2) with boxes'
```

## 利用数据整理来确定参数

有时候您要利用数据整理技术从一长串列表里找出你所需要安装或移除的东西。我们之前讨论的相关技术配合 xargs 即可实现：

```
rustup toolchain list | grep nightly | grep -vE "nightly-x86" | sed 's/-x/-'
```

## 整理二进制数据

虽然到目前为止我们的讨论都是基于文本数据，但对于二进制文件其实同样有用。例如我们可以用 ffmpeg 从相机中捕获一张图片，将其转换成灰度图后通过SSH将压缩后的文件发送到远端服务器，并在那里解压、存档并显示。

```
ffmpeg -loglevel panic -i /dev/video0 -frames 1 -f image2 -
| convert - -colorspace gray -
| gzip
| ssh mymachine 'gzip -d | tee copy.jpg | env DISPLAY=:0 feh -'
```

## 课后练习

### 习题解答

- 学习一下这篇简短的 [交互式正则表达式教程](#)。
- 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

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，过滤掉每次都不相同的部分，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 (因为这些内容是三次启动日志中的重复部分) 。
6. 在网上找一个类似 [这个](#) 或者 [这个](#) 的数据集。或者从[这里](#)找一些。使用 `curl` 获取数据集并提取其中两列数据，如果您想要获取的是HTML数据，那么 [`pup`](#) 可能会更有帮助。对于JSON类型的数据，可以试试 [`jq`](#)。请使用一条指令来找出其中一列的最大值和最小值，用另外一条指令计算两列之间差的总和。

# 命令行环境

当您使用 shell 进行工作时，可以使用一些方法改善您的工作流，本节课我们就来讨论这些方法。

我们已经使用 shell 一段时间了，但是到目前为止我们的关注点主要集中在使用不同的命令上面。现在，我们将会学习如何同时执行多个不同的进程并追踪它们的状态、如何停止或暂停某个进程以及如何使进程在后台运行。

我们还将学习一些能够改善您的 shell 及其他工具的工作流的方法，这主要是通过定义别名或基于配置文件对其进行配置来实现的。这些方法都可以帮您节省大量的时间。例如，仅需要执行一些简单的命令，我们就可以在所有的主机上使用相同的配置。我们还会学习如何使用 SSH 操作远端机器。

## 任务控制

某些情况下我们需要中断正在执行的任务，比如当一个命令需要执行很长时间才能完成时（假设我们在使用 `find` 搜索一个非常大的目录结构）。大多数情况下，我们可以使用 `Ctrl-C` 来停止命令的执行。但是它的工作原理是什么呢？为什么有的时候会无法结束进程？

### 结束进程

您的 shell 会使用 UNIX 提供的信号机制执行进程间通信。当一个进程接收到信号时，它会停止执行、处理该信号并基于信号传递的信息来改变其执行。就这一点而言，信号是一种软件中断。

在上面的例子中，当我们输入 `Ctrl-C` 时，shell 会发送一个 `SIGINT` 信号到进程。

下面这个 Python 程序向您展示了捕获信号 `SIGINT` 并忽略它的基本操作，它并不会让程序停止。为了停止这个程序，我们需要使用 `SIGQUIT` 信号，通过输入 `Ctrl-\` 可以发送该信号。

```
#!/usr/bin/env python
import signal, time

def handler(signum, time):
    print("\nI got a SIGINT, but I am not stopping")

signal.signal(signal.SIGINT, handler)
i = 0
while True:
    time.sleep(.1)
    print("\r{}".format(i), end="")
    i += 1
```

如果我们向这个程序发送两次 `SIGINT`，然后再发送一次 `SIGQUIT`，程序会有什么反应？注意<sup>^</sup> 是我们在终端输入 `Ctrl` 时的表示形式：

```
$ python sigint.py
24^C
I got a SIGINT, but I am not stopping
26^C
I got a SIGINT, but I am not stopping
30^[[1] 39913 quit      python sigint.pyf
```

尽管 SIGINT 和 SIGQUIT 都常常用来发出和终止程序相关的请求。 SIGTERM 则是一个更加通用的、也更加优雅地退出信号。为了发出这个信号我们需要使用 [kill](#) 命令, 它的语法是:  
`kill -TERM <PID>`。

## 暂停和后台执行进程

信号可以让进程做其他的事情, 而不仅仅是终止它们。例如, SIGSTOP 会让进程暂停。在终端中, 键入 `Ctrl-Z` 会让 shell 发送 SIGTSTP 信号, SIGTSTP 是 Terminal Stop 的缩写 (即 terminal 版本的 SIGSTOP)。

我们可以使用 [fg](#) 或 [bg](#) 命令恢复暂停的工作。它们分别表示在前台继续或在后台继续。

[jobs](#) 命令会列出当前终端会话中尚未完成的全部任务。您可以使用 pid 引用这些任务 (也可以用 [pgrep](#) 找出 pid)。更加符合直觉的操作是您可以使用百分号 + 任务编号 ([jobs](#) 会打印任务编号) 来选取该任务。如果要选择最近的一个任务, 可以使用 `$!` 这一特殊参数。

还有一件事情需要掌握, 那就是命令中的 `&` 后缀可以让命令在直接在后台运行, 这使得您可以直接在 shell 中继续做其他操作, 不过它此时还是会使用 shell 的标准输出, 这一点有时会比较恼人 (这种情况可以使用 shell 重定向处理)。

让已经在运行的进程转到后台运行, 您可以键入 `Ctrl-Z`, 然后紧接着再输入 `bg`。注意, 后台的进程仍然是您的终端进程的子进程, 一旦您关闭终端 (会发送另外一个信号 SIGHUP), 这些后台的进程也会终止。为了防止这种情况发生, 您可以使用 [nohup](#) (一个用来忽略 SIGHUP 的封装) 来运行程序。针对已经运行的程序, 可以使用 `disown`。除此之外, 您可以使用终端多路复用器来实现, 下一章节我们会进行详细地探讨。

下面这个简单的会话中展示来了些概念的应用。

```
$ sleep 1000
^Z
[1] + 18653 suspended sleep 1000

$ nohup sleep 2000 &
[2] 18745
 appending output to nohup.out

$ jobs
[1] + suspended sleep 1000
[2] - running nohup sleep 2000

$ bg %1
[1] - 18653 continued sleep 1000

$ jobs
[1] - running sleep 1000
[2] + running nohup sleep 2000

$ kill -STOP %1
[1] + 18653 suspended (signal) sleep 1000

$ jobs
[1] + suspended (signal) sleep 1000
[2] - running nohup sleep 2000

$ kill -SIGHUP %1
[1] + 18653 hangup sleep 1000

$ jobs
[2] + running nohup sleep 2000

$ kill -SIGHUP %2

$ jobs
[2] + running nohup sleep 2000

$ kill %2
[2] + 18745 terminated nohup sleep 2000

$ jobs
```

SIGKILL 是一个特殊的信号，它不能被进程捕获并且它会马上结束该进程。不过这样做会有一些副作用，例如留下孤儿进程。

您可以在 [这里](#) 或输入 `man signal` 或使用 `kill -l` 来获取更多关于信号的信息。

# 终端多路复用

当您在使用命令行时，您通常会希望同时执行多个任务。举例来说，您可以想要同时运行您的编辑器，并在终端的另外一侧执行程序。尽管再打开一个新的终端窗口也能达到目的，使用终端多路复用器则是一种更好的办法。

像 [tmux](#) 这类的终端多路复用器可以允许我们基于面板和标签分割出多个终端窗口，这样您便可以同时与多个 shell 会话进行交互。

不仅如此，终端多路复用使我们可以分离当前终端会话并在将来重新连接。

这让您操作远端设备时的工作流大大改善，避免了 nohup 和其他类似技巧的使用。

现在最流行的终端多路器是 [tmux](#)。[tmux](#) 是一个高度可定制的工具，您可以使用相关快捷键创建多个标签页并在它们间导航。

[tmux](#) 的快捷键需要我们掌握，它们都是类似 <C-b> x 这样的组合，即需要先按下 **Ctrl+b**，松开后再按下 x。[tmux](#) 中对象的继承结构如下：

- **会话** - 每个会话都是一个独立的工作区，其中包含一个或多个窗口
  - [tmux](#) 开始一个新的会话
  - [tmux new -s NAME](#) 以指定名称开始一个新的会话
  - [tmux ls](#) 列出当前所有会话
  - 在 [tmux](#) 中输入 <C-b> d，将当前会话分离
  - [tmux a](#) 重新连接最后一个会话。您也可以通过 -t 来指定具体的会话
- **窗口** - 相当于编辑器或是浏览器中的标签页，从视觉上将一个会话分割为多个部分
  - <C-b> c 创建一个新的窗口，使用 <C-d> 关闭
  - <C-b> N 跳转到第 N 个窗口，注意每个窗口都是有编号的
  - <C-b> p 切换到前一个窗口
  - <C-b> n 切换到下一个窗口
  - <C-b> , 重命名当前窗口
  - <C-b> w 列出当前所有窗口
- **面板** - 像 vim 中的分屏一样，面板使我们可以在一个屏幕里显示多个 shell
  - <C-b> " 水平分割
  - <C-b> % 垂直分割
  - <C-b> <方向> 切换到指定方向的面板，<方向> 指的是键盘上的方向键
  - <C-b> z 切换当前面板的缩放
  - <C-b> [ 开始往回卷动屏幕。您可以按下空格键来开始选择，回车键复制选中的部分
  - <C-b> <空格> 在不同的面板排布间切换

扩展阅读：[这里](#) 是一份 [tmux](#) 快速入门教程，[而这一篇](#) 文章则更加详细，它包含了 screen 命令。您也许想要掌握 [screen](#) 命令，因为在大多数 UNIX 系统中都默认安装有该程序。

# 别名

输入一长串包含许多选项的命令会非常麻烦。因此，大多数 shell 都支持设置别名。shell 的别名相当于一个长命令的缩写，shell 会自动将其替换成原本的命令。例如，bash 中的别名语法如下：

```
alias alias_name="command_to_alias arg1 arg2"
```

注意， = 两边是没有空格的，因为 `alias` 是一个 shell 命令，它只接受一个参数。

别名有许多很方便的特性：

```
# 创建常用命令的缩写
alias ll="ls -lh"

# 能够少输入很多
alias gs="git status"
alias gc="git commit"
alias v="vim"

# 手误打错命令也没关系
alias sl=ls

# 重新定义一些命令行的默认行为
alias mv="mv -i"          # -i prompts before overwrite
alias mkdir="mkdir -p"      # -p make parent dirs as needed
alias df="df -h"           # -h prints human readable format

# 别名可以组合使用
alias la="ls -A"
alias lla="la -l"

# 在忽略某个别名
\ls
# 或者禁用别名
unalias la

# 获取别名的定义
alias ll
# 会打印 ll='ls -lh'
```

值得注意的是，在默认情况下 shell 并不会保存别名。为了让别名持续生效，您需要将配置放进 shell 的启动文件里，像是 `.bashrc` 或 `.zshrc`，下一节我们就会讲到。

# 配置文件 (Dotfiles)

很多程序的配置都是通过纯文本格式的被称作点文件的配置文件来完成的（之所以称为点文件，是因为它们的文件名以`.`开头，例如`~/.vimrc`。也正因为此，它们默认是隐藏文件，`ls`并不会显示它们）。

shell 的配置也是通过这类文件完成的。在启动时，您的 shell 程序会读取很多文件以加载其配置项。根据 shell 本身的不同，您从登录开始还是以交互的方式完成这一过程可能会有很大的不同。关于这一话题，[这里](#)有非常好的资源

对于 bash 来说，在大多数系统下，您可以通过编辑`.bashrc`或`.bash_profile`来进行配置。在文件中您可以添加需要在启动时执行的命令，例如上文我们讲到过的别名，或者是您的环境变量。

实际上，很多程序都要求您在 shell 的配置文件中包含一行类似`export PATH="$PATH:/path/to/program/bin"`的命令，这样才能确保这些程序能够被 shell 找到。

还有一些其他的工具也可以通过点文件进行配置：

- bash -`~/.bashrc`,`~/.bash_profile`
- git -`~/.gitconfig`
- vim -`~/.vimrc` 和`~/.vim` 目录
- ssh -`~/.ssh/config`
- tmux -`~/.tmux.conf`

我们应该如何管理这些配置文件呢，它们应该在它们的文件夹下，并使用版本控制系统进行管理，然后通过脚本将其**符号链接**到需要的地方。这么做有如下好处：

- **安装简单**: 如果您登录了一台新的设备，在这台设备上应用您的配置只需要几分钟的时间；
- **可以执行**: 您的工具在任何地方都以相同的配置工作
- **同步**: 在一处更新配置文件，可以同步到其他所有地方
- **变更追踪**: 您可能要在整个程序员生涯中持续维护这些配置文件，而对于长期项目而言，版本历史是非常重要的

配置文件中需要放些什么？您可以通过在线文档和[帮助手册](#)了解所使用工具的设置项。另一个方法是在网上搜索有关特定程序的文章，作者们在文章中会分享他们的配置。还有一种方法就是直接浏览其他人的配置文件：您可以在[这里](#)找到无数的[dotfiles 仓库](#)——其中最受欢迎的那些可以在[这里](#)找到（我们建议您不要直接复制别人的配置）。[这里](#)也有一些非常有用的资源。

本课程的老师们也在 GitHub 上开源了他们的配置文件：[Anish](#), [Jon](#), [Jose](#).

## 可移植性

配置文件的一个常见的痛点是它可能并不能在多种设备上生效。例如，如果您在不同设备上使用的操作系统或者 shell 是不同的，则配置文件是无法生效的。或者，有时您仅希望特定的配置只在某些设备上生效。

有一些技巧可以轻松达成这些目的。如果配置文件 if 语句，则您可以借助它针对不同的设备编写不同的配置。例如，您的 shell 可以这样做：

```
if [[ "$uname" == "Linux" ]]; then {do_something}; fi

# 使用和 shell 相关的配置时先检查当前 shell 类型
if [[ "$SHELL" == "zsh" ]]; then {do_something}; fi

# 您也可以针对特定的设备进行配置
if [[ "$hostname" == "myServer" ]]; then {do_something}; fi
```

如果配置文件支持 include 功能，您也可以多加利用。例如：`~/.gitconfig` 可以这样编写：

```
[include]
path = ~/.gitconfig_local
```

然后我们可以在日常使用的设备上创建配置文件 `~/.gitconfig_local` 来包含与该设备相关的特定配置。您甚至应该创建一个单独的代码仓库来管理这些与设备相关的配置。

如果您希望在不同的程序之间共享某些配置，该方法也适用。例如，如果您想要在 bash 和 zsh 中同时启用一些别名，您可以把它们写在 `.aliases` 里，然后在这两个 shell 里应用：

```
# Test if ~/.aliases exists and source it
if [ -f ~/.aliases ]; then
    source ~/.aliases
fi
```

## 远端设备

对于程序员来说，在他们的日常工作中使用远程服务器已经非常普遍了。如果您需要使用远程服务器来部署后端软件或您需要一些计算能力强大的服务器，您就会用到安全 shell (SSH)。和其他工具一样，SSH 也是可以高度定制的，也值得我们花时间学习它。

通过如下命令，您可以使用 ssh 连接到其他服务器：

```
ssh foo@bar.mit.edu
```

这里我们尝试以用户名 `foo` 登录服务器 `bar.mit.edu`。服务器可以通过 URL 指定（例如 `bar.mit.edu`），也可以使用 IP 指定（例如 `foobar@192.168.1.42`）。后面我们会介绍如何修改 ssh 配置文件使我们可以用类似 `ssh bar` 这样的命令来登录服务器。

## 执行命令

`ssh` 的一个经常被忽视的特性是它可以直接远程执行命令。`ssh foobar@server ls` 可以直接在用 `foobar` 的命令下执行 `ls` 命令。想要配合管道来使用也可以，`ssh foobar@server`

`ls | grep PATTERN` 会在本地查询远端 `ls` 的输出而 `ls | ssh foobar@server grep PATTERN` 会在远端对本地 `ls` 输出的结果进行查询。

## SSH 密钥

基于密钥的验证机制使用了密码学中的公钥，我们只需要向服务器证明客户端持有对应的私钥，而不需要公开其私钥。这样您就可以避免每次登录都输入密码的麻烦了秘密就可以登录。不过，私钥(通常是 `~/.ssh/id_rsa` 或者 `~/.ssh/id_ed25519`) 等效于您的密码，所以一定要好好保存它。

### 密钥生成

使用 `ssh-keygen` 命令可以生成一对密钥：

```
ssh-keygen -o -a 100 -t ed25519 -f ~/.ssh/id_ed25519
```

您可以为密钥设置密码，防止有人持有您的私钥并使用它访问您的服务器。您可以使用 `ssh-agent` 或 `gpg-agent`，这样就不需要每次都输入该密码了。

如果您曾经配置过使用 SSH 密钥推送到 GitHub，那么可能您已经完成了[这里](#) 介绍的这些步骤，并且已经有了一个可用的密钥对。要检查您是否持有密码并验证它，您可以运行 `ssh-keygen -y -f /path/to/key`。

### 基于密钥的认证机制

`ssh` 会查询 `.ssh/authorized_keys` 来确认那些用户可以被允许登录。您可以通过下面的命令将一个公钥拷贝到这里：

```
cat .ssh/id_ed25519 | ssh foobar@remote 'cat >> ~/.ssh/authorized_keys'
```

如果支持 `ssh-copy-id` 的话，可以使用下面这种更简单的解决方案：

```
ssh-copy-id -i .ssh/id_ed25519.pub foobar@remote
```

## 通过 SSH 复制文件

使用 `ssh` 复制文件有很多方法：

- `ssh+tee`，最简单的方法是执行 `ssh` 命令，然后通过这样的方法利用标准输入实现 `cat localfile | ssh remote_server tee serverfile`。回忆一下，`tee` 命令会将标准输出写入到一个文件；
- `scp`：当需要拷贝大量的文件或目录时，使用 `scp` 命令则更加方便，因为它可以方便的遍历相关路径。语法如下：`scp path/to/local_file remote_host:path/to/remote_file`；
- `rsync` 对 `scp` 进行了改进，它可以检测本地和远端的文件以防止重复拷贝。它还可以提供一些诸如符号连接、权限管理等精心打磨的功能。甚至还可以基于 `--partial` 标记实现断点续

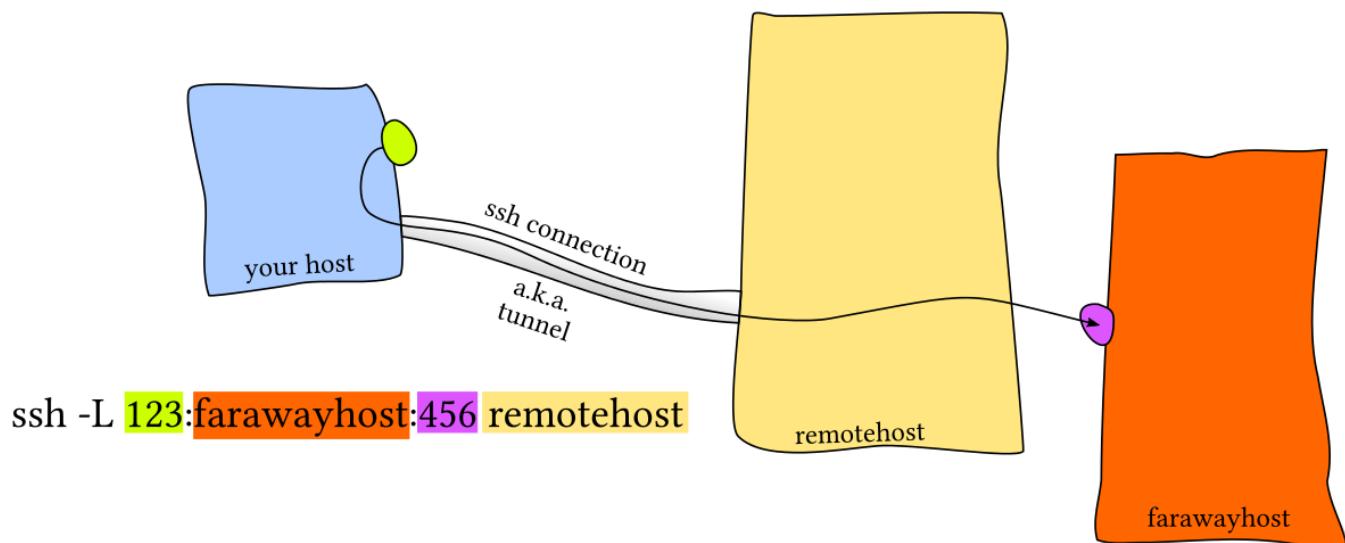
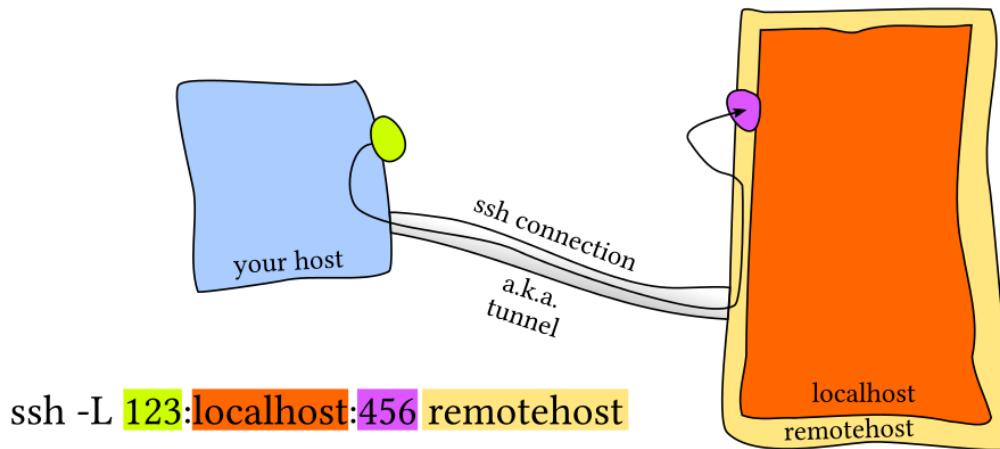
传。rsync 的语法和 scp 类似；

## 端口转发

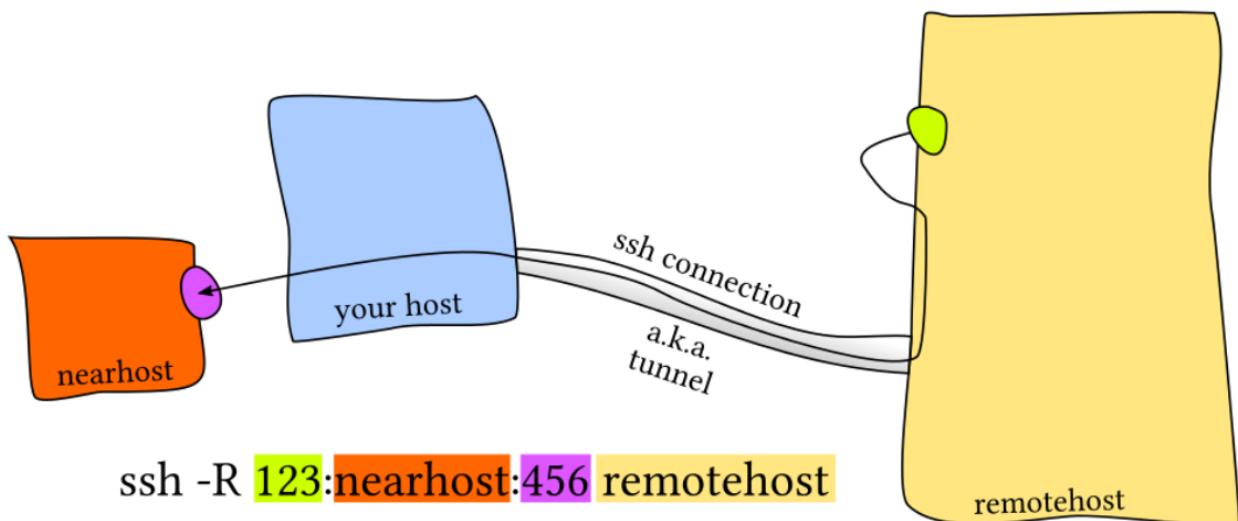
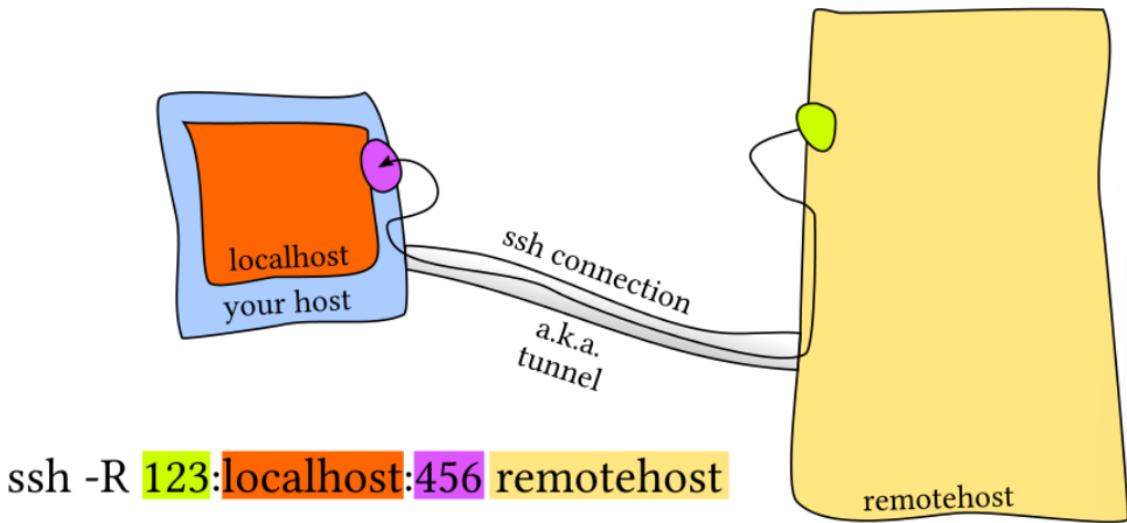
很多情况下我们都会遇到软件需要监听特定设备的端口。如果是在您的本机，可以使用 `localhost:PORT` 或 `127.0.0.1:PORT`。但是如果需要监听远程服务器的端口该如何操作呢？这种情况下远端的端口并不会直接通过网络暴露给您。

此时就需要进行 端口转发。端口转发有两种，一种是本地端口转发和远程端口转发（参见下图，该图片引用自这篇[StackOverflow 文章](#)）中的图片。

### 本地端口转发



### 远程端口转发



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。

## SSH 配置

我们已经介绍了很多参数。为它们创建一个别名是个好想法，我们可以这样做：

```
alias my_server="ssh -i ~/.id_ed25519 --port 2222 -L 9999:localhost:8888
```

不过，更好的方法是使用 `~/.ssh/config`。

```
Host vm
  User foobar
  HostName 172.16.174.141
  Port 2222
  IdentityFile ~/.ssh/id_ed25519
  LocalForward 9999 localhost:8888

# 在配置文件中也可以使用通配符
Host *.mit.edu
  User foobaz
```

这么做好处是，使用 `~/.ssh/config` 文件来创建别名，类似 `scp`、`rsync` 和 `mosh` 的这些命令都可以读取这个配置并将设置转换为对应的命令行选项。

注意，`~/.ssh/config` 文件也可以被当作配置文件，而且一般情况下也是可以被导入其他配置文件的。不过，如果您将其公开到互联网上，那么其他人都将会看到您的服务器地址、用户名、开放端口等等。这些信息可能会帮助到那些企图攻击您系统的黑客，所以请务必三思。

服务器侧的配置通常放在 `/etc/ssh/sshd_config`。您可以在这里配置免密认证、修改 ssh 端口、开启 X11 转发等等。您也可以为每个用户单独指定配置。

## 杂项

连接远程服务器的一个常见痛点是遇到由关机、休眠或网络环境变化导致的掉线。如果连接的延迟很高也很让人讨厌。[Mosh](#)（即 mobile shell）对 ssh 进行了改进，它允许连接漫游、间歇连接及智能本地回显。

有时将一个远端文件夹挂载到本地会比较方便，[sshfs](#) 可以将远端服务器上的一个文件夹挂载到本地，然后您就可以使用本地的编辑器了。

## Shell & 框架

在 shell 工具和脚本那节课中我们已经介绍了 `bash shell`，因为它是目前最通用的 shell，大多数的系统都将其作为默认 shell。但是，它并不是唯一的选项。

例如，`zsh shell` 是 `bash` 的超集并提供了一些方便的功能：

- 智能替换, \*\*
- 行内替换/通配符扩展
- 拼写纠错
- 更好的 tab 补全和选择
- 路径展开 (`cd /u/lo/b` 会被展开为 `/usr/local/bin`)

**框架** 也可以改进您的 shell。比较流行的通用框架包括[prezto](#) 或 [oh-my-zsh](#)。还有一些更精简的框架，它们往往专注于某一个特定功能，例如[zsh 语法高亮](#) 或 [zsh 历史子串查询](#)。像 [fish](#) 这样

的 shell 包含了很多用户友好的功能，其中一些特性包括：

- 向右对齐
- 命令语法高亮
- 历史子串查询
- 基于手册页面的选项补全
- 更智能的自动补全
- 提示符主题

需要注意的是，使用这些框架可能会降低您 shell 的性能，尤其是如果这些框架的代码没有优化或者代码过多。您随时可以测试其性能或禁用某些不常用的功能来实现速度与功能的平衡。

## 终端模拟器

和自定义 shell 一样，花点时间选择适合您的 **终端模拟器** 并进行设置是很有必要的。有许多终端模拟器可供您选择（这里有一些关于它们之间**比较**的信息）

您会花上很多时间在使用终端上，因此研究一下终端的设置是很有必要的，您可以从下面这些方面来配置您的终端：

- 字体选择
- 彩色主题
- 快捷键
- 标签页/面板支持
- 回退配置
- 性能（像 [Alacritty](#) 或者 [kitty](#) 这种比较新的终端，它们支持GPU加速）。

## 课后练习

### 习题解答

### 任务控制

1. 我们可以使用类似 `ps aux | grep` 这样的命令来获取任务的 pid，然后您可以基于pid 来结束这些进程。但我们其实有更好的方法来做这件事。在终端中执行 `sleep 10000` 这个任务。然后用 `Ctrl-Z` 将其切换到后台并使用 `bg` 来继续允许它。现在，使用 [pgrep](#) 来查找 pid 并使用 [pkill](#) 结束进程而不需要手动输入pid。（提示：：使用 `-af` 标记）。
2. 如果您希望某个进程结束后再开始另外一个进程，应该如何实现呢？在这个练习中，我们使用 `sleep 60 &` 作为先执行的程序。一种方法是使用 [wait](#) 命令。尝试启动这个休眠命令，然后待其结束后再执行 `ls` 命令。

但是，如果我们在不同的 bash 会话中进行操作，则上述方法就不起作用了。因为 `wait` 只能对子进程起作用。之前我们没有提过的一个特性是，`kill` 命令成功退出时其状态码为 0，其他状态则是非0。`kill -0` 则不会发送信号，但是会在进程不存在时返回一个不为0的状态

码。请编写一个 bash 函数 `pidwait`，它接受一个 pid 作为输入参数，然后一直等待直到该进程结束。您需要使用 `sleep` 来避免浪费 CPU 性能。

## 终端多路复用

1. 请完成这个 tmux 教程 参考这些步骤来学习如何自定义 tmux。

## 别名

1. 创建一个 `dc` 别名，它的功能是当我们错误的将 `cd` 输入为 `dc` 时也能正确执行。
2. 执行 `history | awk '{\$1=""};print substr(\$0,2)' | sort | uniq -c | sort -n | tail -n 10` 来获取您最常用的十条命令，尝试为它们创建别名。注意：这个命令只在 Bash 中生效，如果您使用 ZSH，使用 `history 1` 替换 `history`。

## 配置文件

让我们帮助您进一步学习配置文件：

1. 为您的配置文件新建一个文件夹，并设置好版本控制
2. 在其中添加至少一个配置文件，比如说您的 shell，在其中包含一些自定义设置（可以从设置 `$PS1` 开始）。
3. 建立一种在新设备进行快速安装配置的方法（无需手动操作）。最简单的方法是写一个 shell 脚本对每个文件使用 `ln -s`，也可以使用[专用工具](#)
4. 在新的虚拟机上测试该安装脚本。
5. 将您现有的所有配置文件移动到项目仓库里。
6. 将项目发布到GitHub。

## 远端设备

进行下面的练习需要您先安装一个 Linux 虚拟机（如果已经安装过则可以直接使用），如果您对虚拟机尚不熟悉，可以参考[这篇教程](#)来进行安装。

1. 前往 `~/.ssh/` 并查看是否已经存在 SSH 密钥对。如果不存在，请使用 `ssh-keygen -o -a 100 -t ed25519` 来创建一个。建议为密钥设置密码然后使用 `ssh-agent`，更多信息可以参考[这里](#)；
2. 在 `.ssh/config` 加入下面内容：

```
Host vm
  User username_goes_here
  HostName ip_goes_here
  IdentityFile ~/.ssh/id_ed25519
  LocalForward 9999 localhost:8888
```

1. 使用 `ssh-copy-id vm` 将您的 ssh 密钥拷贝到服务器。
2. 使用 `python -m http.server 8888` 在您的虚拟机中启动一个 Web 服务器并通过本机的 `http://localhost:9999` 访问虚拟机上的 Web 服务器

3. 使用 `sudo vim /etc/ssh/sshd_config` 编辑 SSH 服务器配置，通过修改 `PasswordAuthentication` 的值来禁用密码验证。通过修改 `PermitRootLogin` 的值来禁用 root 登录。然后使用 `sudo service sshd restart` 重启 ssh 服务器，然后重新尝试。
4. (附加题) 在虚拟机中安装 [mosh](#) 并启动连接。然后断开服务器/虚拟机的网络适配器。mosh 可以恢复连接吗？
5. (附加题) 查看 ssh 的 -N 和 -f 选项的作用，找出在后台进行端口转发的命令是什么？

# 版本控制(Git)

版本控制系统 (VCSs) 是一类用于追踪源代码（或其他文件、文件夹）改动的工具。顾名思义，这些工具可以帮助我们管理代码的修改历史；不仅如此，它还可以让协作编码变得更方便。VCS 通过一系列的快照将某个文件夹及其内容保存了起来，每个快照都包含了文件或文件夹的完整状态。同时它还维护了快照创建者的信息以及每个快照的相关信息等等。

为什么说版本控制系统非常有用？即使您只是一个人进行编程工作，它也可以帮您创建项目的快照，记录每个改动的目的、基于多分支并行开发等等。和别人协作开发时，它更是一个无价之宝，您可以看到别人对代码进行的修改，同时解决由于并行开发引起的冲突。

现代的版本控制系统可以帮助您轻松地（甚至自动地）回答以下问题：

- 当前模块是谁编写的？
- 这个文件的这一行是什么时候被编辑的？是谁作出的修改？修改原因是什么呢？
- 最近的1000个版本中，何时/为什么导致了单元测试失败？

尽管版本控制系统有很多，其事实上的标准则是 **Git**。而这篇 [XKCD 漫画](#) 则反映出了人们对 Git 的评价：

因为 Git 接口的抽象泄漏 (leaky abstraction) 问题，通过自顶向下的方式（从命令行接口开始）学习 Git 可能会让人感到非常困惑。很多时候您只能死记硬背一些命令行，然后像使用魔法一样使用它们，一旦出现问题，就只能像上面那幅漫画里说的那样去处理了。

尽管 Git 的接口有些丑陋，但是它的底层设计和思想却是非常优雅的。丑陋的接口只能靠死记硬背，而优雅的底层设计则非常容易被人理解。因此，我们将通过一种自底向上的方式向您介绍 Git。我们会从数据模型开始，最后再学习它的接口。一旦您搞懂了 Git 的数据模型，再学习其接口并理解这些接口是如何操作数据模型的就非常容易了。

## Git 的数据模型

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")

```

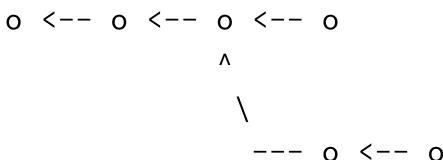
这个顶层的树包含了两个元素，一个名为“foo”的树（它本身包含了一个blob对象“bar.txt”），以及一个blob对象“baz.txt”。

## 历史记录建模：关联快照

版本控制系统和快照有什么关系呢？线性历史记录是一种最简单的模型，它包含了一组按照时间顺序线性排列的快照。不过出于种种原因，Git并没有采用这样的模型。

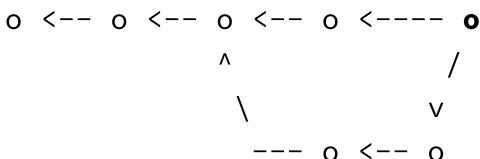
在 Git 中，历史记录是一个由快照组成的有向无环图。有向无环图，听上去似乎是什么高大上的数学名词。不过不要怕，您只需要知道这代表 Git 中的每个快照都有一系列的“父辈”，也就是其之前的一系列快照。注意，快照具有多个“父辈”而非一个，因为某个快照可能由多个父辈而来。例如，经过合并后的两条分支。

在 Git 中，这些快照被称为“提交”。通过可视化的方式来表示这些历史提交记录时，看起来差不多是这样的：



上面是一个 ASCII 码构成的简图，其中的 o 表示一次提交（快照）。

箭头指向了当前提交的父辈（这是一种“在...之前”，而不是“在...之后”的关系）。在第三次提交之后，历史记录分岔成了两条独立的分支。这可能因为此时需要同时开发两个不同的特性，它们之间是相互独立的。开发完成后，这些分支可能会被合并并创建一个新的提交，这个新的提交会同时包含这些特性。新的提交会创建一个新的历史记录，看上去像这样（最新的合并提交用粗体标记）：



Git 中的提交是不可改变的。但这并不代表错误不能被修改，只不过这种“修改”实际上是创建了一个全新的提交记录。而引用（参见下文）则被更新为指向这些新的提交。

## 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 | blob>

// 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 哈希](#) 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，[上面](#)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`” 这样人类可读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`”。

## 仓库

最后，我们可以粗略地给出 Git 仓库的定义了：对象 和 引用。

在硬盘上，Git 仅存储对象和引用：因为其数据模型仅包含这些东西。所有的 `git` 命令都对应着对提交树的操作，例如增加对象，增加或删除引用。

当您输入某个指令时，请思考一下这条命令是如何对底层的图数据结构进行操作的。另一方面，如果您希望修改提交树，例如“丢弃未提交的修改和将‘master’引用指向提交 5d83f9e 时，有什么命令可以完成该操作（针对这个具体问题，您可以使用 `git checkout master; git reset --hard 5d83f9e`）

## 暂存区

Git 中还包括一个和数据模型完全不相关的概念，但它确是创建提交的接口的一部分。

就上面介绍的快照系统来说，您也许会期望它的实现里包括一个“创建快照”的命令，该命令能够基于当前工作目录的当前状态创建一个全新的快照。有些版本控制系统确实是这样工作的，但 Git 不是。我们希望简洁的快照，而且每次从当前状态创建快照可能效果并不理想。例如，考虑如下场景，您开发了两个独立的特性，然后您希望创建两个独立的提交，其中第一个提交仅包含第一个特性，而第二个提交仅包含第二个特性。或者，假设您在调试代码时添加了很多打印语句，然后您仅仅希望提交和修复 bug 相关的代码而丢弃所有的打印语句。

Git 处理这些场景的方法是使用一种叫做“暂存区（staging area）”的机制，它允许您指定下次快照中要包括那些改动。

## Git 的命令行接口

为了避免重复信息，我们将不会详细解释以下命令行。强烈推荐您阅读 [Pro Git 中文版](#) 或以观看本讲座的视频来学习。

### 基础

- `git help <command>`: 获取 git 命令的帮助信息
- `git init`: 创建一个新的 git 仓库，其数据会存放在一个名为 `.git` 的目录下
- `git status`: 显示当前的仓库状态
- `git add <filename>`: 添加文件到暂存区
- `git commit`: 创建一个新的提交
  - 如何编写 [良好的提交信息！](#)
  - 为何要 [编写良好的提交信息](#)
- `git log`: 显示历史日志
- `git log --all --graph --decorate`: 可视化历史记录（有向无环图）
- `git diff <filename>`: 显示与暂存区文件的差异
- `git diff <revision> <filename>`: 显示某个文件两个版本之间的差异
- `git checkout <revision>`: 更新 HEAD 和目前的分支

### 分支和合并

- `git branch`: 显示分支
- `git branch <name>`: 创建分支
- `git checkout -b <name>`: 创建分支并切换到该分支
  - 相当于 `git branch <name>; git checkout <name>`

- git merge <revision>: 合并到当前分支
- git mergetool: 使用工具来处理合并冲突
- git rebase: 将一系列补丁变基 (rebase) 为新的基线

## 远端操作

- git remote: 列出远端
- git remote add <name> <url>: 添加一个远端
- git push <remote> <local branch>:<remote branch>: 将对象传送至远端并更新远端引用
- git branch --set-upstream-to=<remote>/<remote branch>: 创建本地和远端分支的关联关系
- git fetch: 从远端获取对象/索引
- git pull: 相当于 git fetch; git merge
- git clone: 从远端下载仓库

## 撤销

- git commit --amend: 编辑提交的内容或信息
- git reset HEAD <file>: 恢复暂存的文件
- git checkout -- <file>: 丢弃修改
- git restore: git 2.32 版本后取代 git reset 进行许多撤销操作

# Git 高级操作

- git config: Git 是一个 [高度可定制的](#) 工具
- git clone --depth=1: 浅克隆 (shallow clone), 不包括完整的版本历史信息
- git add -p: 交互式暂存
- git rebase -i: 交互式变基
- git blame: 查看最后修改某行的人
- git stash: 暂时移除工作目录下的修改内容
- git bisect: 通过二分查找搜索历史记录
- .gitignore: [指定](#) 故意不追踪的文件

## 杂项

- **图形用户界面:** Git 的 [图形用户界面客户端](#) 有很多, 但是我们自己并不使用这些图形用户界面的客户端, 我们选择使用命令行接口
- **Shell 集成:** 将 Git 状态集成到您的 shell 中会非常方便。([zsh](#), [bash](#))。[Oh My Zsh](#) 这样的框架中一般以及集成了这一功能
- **编辑器集成:** 和上面一条类似, 将 Git 集成到编辑器中好处多多。[fugitive.vim](#) 是 Vim 中集成 Git 的常用插件
- **工作流:** 我们已经讲解了数据模型与一些基础命令, 但还没讨论到进行大型项目时的一些惯例 ([有很多不同的处理方法](#))

- **GitHub:** Git 并不等同于 GitHub。在 GitHub 中您需要使用一个被称作[拉取请求 \(pull request\)](#) 的方法来向其他项目贡献代码
- **其他 Git 提供商:** GitHub 并不是唯一的。还有像 [GitLab](#) 和 [BitBucket](#) 这样的平台。

## 资源

- [Pro Git](#)，**强烈推荐！** 学习前五章的内容可以教会您流畅使用 Git 的绝大多数技巧，因为您已经理解了 Git 的数据模型。后面的章节提供了很多有趣的高级主题。 ([Pro Git 中文版](#)) ;
- [Oh Shit, Git!?!?](#)，简短的介绍了如何从 Git 错误中恢复；
- [Git for Computer Scientists](#)，简短的介绍了 Git 的数据模型，与本文相比包含较少量的伪代码以及大量的精美图片；
- [Git from the Bottom Up](#)详细的介绍了 Git 的实现细节，而不仅仅局限于数据模型。好奇的同学可以看看；
- [How to explain git in simple words](#);
- [Learn Git Branching](#) 通过基于浏览器的游戏来学习 Git ；

## 课后练习

### 习题解答

1. 如果您之前从来没有用过 Git，推荐您阅读 [Pro Git](#) 的前几章，或者完成像 [Learn Git Branching](#)这样的教程。重点关注 Git 命令和数据模型相关内容；
2. Fork [本课程网站的仓库](#)
  1. 将版本历史可视化并进行探索
  2. 是谁最后修改了 README.md 文件？(提示：使用 git log 命令并添加合适的参数)
  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 )
3. 使用 Git 时的一个常见错误是提交本不应该由 Git 管理的大文件，或是将含有敏感信息的文件提交给 Git 。尝试向仓库中添加一个文件并添加提交信息，然后将其从历史中删除 ([这篇文章也许会有帮助](#))；
4. 从 GitHub 上克隆某个仓库，修改一些文件。当您使用 git stash 会发生什么？当您执行 git log --all --oneline 时会显示什么？通过 git stash pop 命令来撤销 git stash 操作，什么时候会用到这一技巧？
5. 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；
6. 您可以通过执行 git config --global core.excludesfile ~/.gitignore\_global 在 ~/.gitignore\_global 中创建全局忽略规则。配置您的全局 gitignore 文件来自动忽略系统或编辑器的临时文件，例如 .DS\_Store ；
7. 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) ；

# 调试及性能分析

代码不能完全按照您的想法运行，它只能完全按照您的写法运行，这是编程界的一条金科玉律。

让您的写法符合您的想法是非常困难的。在这节课中，我们会传授给您一些非常有用的技术，帮您处理代码中的 bug 和程序性能问题。

## 调试代码

### 打印调试法与日志

“最有效的 debug 工具就是细致的分析，配合恰当位置的打印语句” — Brian Kernighan, *Unix 新手入门*。

调试代码的第一种方法往往是在您发现问题的地方添加一些打印语句，然后不断重复此过程直到您获取了足够的信息并找到问题的根本原因。

另外一个方法是使用日志，而不是临时添加打印语句。日志较普通的打印语句有如下的一些优势：

- 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。

[这里](#) 是一个包含日志的例程序：

```
$ python logger.py
# Raw output as with just prints
$ python logger.py log
# Log formatted output
$ python logger.py log ERROR
# Print only ERROR levels and above
$ python logger.py color
# Color formatted output
```

有很多技巧可以使日志的可读性变得更好，我最喜欢的一个技巧是对其进行着色。到目前为止，您应该已经知道，以彩色文本显示终端信息时可读性更好。但是应该如何设置呢？

ls 和 grep 这样的程序会使用 [ANSI escape codes](#)，它是一系列的特殊字符，可以使您的 shell 改变输出结果的颜色。例如，执行 echo -e "\e[38;2;255;0;0mThis is red\e[0m" 会打印红色的字符串：This is red。只要您的终端支持[真彩色](#)。如果您的终端不支持真彩色（例如 MacOS 的 Terminal.app），您可以使用支持更加广泛的 16 色，例如：“\e[31;1mThis is red\e[0m”。

下面这个脚本向您展示了如何在终端中打印多种颜色（只要您的终端支持真彩色）

```
#!/usr/bin/env bash
for R in $(seq 0 20 255); do
    for G in $(seq 0 20 255); do
        for B in $(seq 0 20 255); do
            printf "\e[38;2;${R};${G};${B}m■\e[0m";
        done
    done
done
```

## 第三方日志系统

如果您正在构建大型软件系统，您很可能会使用到一些依赖，有些依赖会作为程序单独运行。如 Web 服务器、数据库或消息代理都是此类常见的第三方依赖。

和这些系统交互的时候，阅读它们的日志是非常必要的，因为仅靠客户端侧的错误信息可能并不足以定位问题。

幸运的是，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`。例如，[NGINX](#) web 服务器就将其日志存放于 `/var/log/nginx`。

目前，系统开始使用 **system log**，您所有的日志都会保存在这里。大多数（但不是全部的）Linux 系统都会使用 `systemd`，这是一个**系统守护进程**，它会控制您系统中的很多东西，例如哪些服务应该启动并运行。`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。

类似地，在 macOS 系统中是 `/var/log/system.log`，但是有更多的工具会使用系统日志，它的内容可以使用 [`log show`](#) 显示。

对于大多数的 UNIX 系统，您也可以使用 [`dmesg`](#) 命令来读取内核的日志。

如果您希望将日志加入到系统日志中，您可以使用 [`logger`](#) 这个 shell 程序。下面这个例子显示了如何使用 `logger` 并且如何找到能够将其存入系统日志的条目。

不仅如此，大多数的编程语言都支持向系统日志中写日志。

```
logger "Hello Logs"
# On macOS
log show --last 1m | grep Hello
# On Linux
journalctl --since "1m ago" | grep Hello
```

正如我们在数据整理那节课上看到的那样，日志的内容可以非常的多，我们需要对其进行处理和过滤才能得到我们想要的信息。

如果您发现您需要对 `journalctl` 和 `log show` 的结果进行大量的过滤，那么此时可以考虑使用它们自带的选项对其结果先过滤一遍再输出。还有一些像 [lnav](#) 这样的工具，它为日志文件提供了更好的展现和浏览方式。

## 调试器

当通过打印已经不能满足您的调试需求时，您应该使用调试器。

调试器是一种可以允许我们和正在执行的程序进行交互的程序，它可以做到：

- 当到达某一行时将程序暂停；
- 一次一条指令地逐步执行程序；
- 程序崩溃后查看变量的值；
- 满足特定条件时暂停程序；
- 其他高级功能。

很多编程语言都有自己的调试器。Python 的调试器是 [pdb](#) .

下面对 pdb 支持的命令进行简单的介绍：

- **l(ist)** - 显示当前行附近的11行或继续执行之前的显示；
- **s(tep)** - 执行当前行，并在第一个可能的地方停止；
- **n(ext)** - 继续执行直到当前函数的下一条语句或者 `return` 语句；
- **b(reak)** - 设置断点（基于传入的参数）；
- **print** - 在当前上下文对表达式求值并打印结果。还有一个命令是 **pp**，它使用 [pprint](#) 打印；
- **return** - 继续执行直到当前函数返回；
- **q(uit)** - 退出调试器。

让我们使用 pdb 来修复下面的 Python 代码（参考讲座视频）

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(n):
            if arr[j] > arr[j+1]:
                arr[j] = arr[j+1]
                arr[j+1] = arr[j]
    return arr

print(bubble_sort([4, 2, 1, 8, 7, 6]))
```

注意，因为 Python 是一种解释型语言，所以我们可以通过 pdb shell 执行命令。[ipdb](#) 是一种增强型的 pdb，它使用 [IPython](#) 作为 REPL 并开启了 tab 补全、语法高亮、更好的回溯和更好的内省，同时还保留了 pdb 模块相同的接口。

对于更底层的编程语言，您可能需要了解一下 [gdb](#)（以及它的改进版 [pwndbg](#)）和 [lldb](#)。

它们都对类 C 语言的调试进行了优化，它允许您探索任意进程及其机器状态：寄存器、堆栈、程序计数器等。

## 专门工具

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时，它需要使用 [系统调用](#)。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](#)，在 macOS 和 BSD 中可以使用 [dtrace](#)。dtrace 用起来可能有些别扭，因为它使用的是它自有的 D 语言，但是我们可以使用一个叫做 [dtruss](#) 的封装使其具有和 strace (更多信息参考 [这里](#))类似的接口

下面的例子展现来如何使用 strace 或 dtruss 来显示 ls 执行时，对 [stat](#) 系统调用进行追踪对结果。若需要深入了解 strace，[这篇文章](#) 值得一读。

```
# On Linux
sudo strace -e lstat ls -l > /dev/null
4
# On macOS
sudo dtruss -t lstat64_extended ls -l > /dev/null
```

有些情况下，我们需要查看网络数据包才能定位问题。像 [tcpdump](#) 和 [Wireshark](#) 这样的网络数据包分析工具可以帮助您获取网络数据包的内容并基于不同的条件进行过滤。

对于 web 开发，Chrome/Firefox 的开发者工具非常方便，功能也很强大：

- 源码 - 查看任意站点的 HTML/CSS/JS 源码；
- 实时地修改 HTML, CSS, JS 代码 - 修改网站的内容、样式和行为用于测试（从这一点您也能看出来，网页截图是不可靠的）；
- Javascript shell - 在 JS REPL 中执行命令；
- 网络 - 分析请求的时间线；
- 存储 - 查看 Cookies 和本地应用存储。

## 静态分析

有些问题是您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。静态分析会将程序的源码作为输入然后基于编码规则对其进行分析并对代码的正确性进行推理。

下面这段 Python 代码中存在几个问题。首先，我们的循环变量 foo 覆盖了之前定义的函数 foo。最后一行，我们还把 bar 错写成了 baz，因此当程序完成 sleep (一分钟)后，执行到这一行的时候便会崩溃。

```

import time

def foo():
    return 42

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

```

静态分析工具可以发现此类的问题。当我们使用 [pyflakes](#) 分析代码的时候，我们会得到与这两处 bug 相关的错误信息。[mypy](#) 则是另外一个工具，它可以对代码进行类型检查。这里，mypy 会经过我们 bar 起初是一个 int，然后变成了 float。这些问题都可以在不运行代码的情况下被发现。

```

$ pyflakes foobar.py
foobar.py:6: redefinition of unused 'foo' from line 3
foobar.py:11: undefined name 'baz'

$ mypy foobar.py
foobar.py:6: error: Incompatible types in assignment (expression has type
foobar.py:9: error: Incompatible types in assignment (expression has type
foobar.py:11: error: Name 'baz' is not defined
Found 3 errors in 1 file (checked 1 source file)

```

在 shell 工具那一节课的时候，我们介绍了 [shellcheck](#)，这是一个类似的工具，但它是应用于 shell 脚本的。

大多数的编辑器和 IDE 都支持在编辑界面显示这些工具的分析结果、高亮有警告和错误的位置。这个过程通常称为 **code linting**。风格检查或安全检查的结果同样也可以进行相应的显示。

在 vim 中，有 [ale](#) 或 [syntastic](#) 可以帮助您做同样的事情。在 Python 中，[pylint](#) 和 [pep8](#) 是两种用于进行风格检查的工具，而 [bandit](#) 工具则用于检查安全相关的问题。

对于其他语言的开发者来说，静态分析工具可以参考这个列表：[Awesome Static Analysis](#)（您也许会对 *Writing* 一节感兴趣）。对于 linters 则可以参考这个列表：[Awesome Linters](#)。

对于风格检查和代码格式化，还有以下一些工具可以作为补充：用于 Python 的 [black](#)、用于 Go 语言的 [gofmt](#)、用于 Rust 的 [rustfmt](#) 或是用于 JavaScript, HTML 和 CSS 的 [prettier](#)。这些工具可以自动格式化您的代码，这样代码风格就可以与常见的风格保持一致。尽管您可能并不想对代码进行风格控制，标准的代码风格有助于方便别人阅读您的代码，也可以方便您阅读它的代码。

# 性能分析

即使您的代码能够像您期望的一样运行，但是如果它消耗了您全部的 CPU 和内存，那么它显然也不是个好程序。算法课上我们通常会介绍大O标记法，但却没教给我们如何找到程序中的热点。鉴于 [过早的优化是万恶之源](#)，您需要学习性能分析和监控工具，它们会帮助您找到程序中最耗时、最耗资源的部分，这样您就可以有针对性的进行性能优化。

## 计时

和调试代码类似，大多数情况下我们只需要打印两处代码之间的时间即可发现问题。下面这个例子中，我们使用了 Python 的 [time](#) 模块。

```
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
```

不过，执行时间 (wall clock time) 也可能会误导您，因为您的电脑可能也在同时运行其他进程，也可能在此期间发生了等待。对于工具来说，需要区分真实时间、用户时间和系统时间。通常来说，用户时间+系统时间代表了您的进程所消耗的实际 CPU (更详细的解释可以参照[这篇文章](#))。

- 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。

例如，试着执行一个用于发起 HTTP 请求的命令并在其前面添加 [time](#) 前缀。网络不好的情况下您可能会看到下面的输出结果。请求花费了 2s 才完成，但是进程仅花费了 15ms 的 CPU 用户时间和 12ms 的 CPU 内核时间。

```
$ time curl https://missing.csail.mit.edu &> /dev/null
real    0m2.561s
user    0m0.015s
sys     0m0.012s
```

## 性能分析工具 (profilers)

### CPU

大多数情况下，当人们提及性能分析工具的时候，通常指的是 CPU 性能分析工具。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，而采样分析器则只会周期性的监测（通常为每毫秒）您的程序并记录程序堆栈。它们使用这些记录来生成统计信息，显示程序在哪些事情上花费了最多的时间。如果您希望了解更多相关信息，可以参考[这篇](#)介绍性的文章。

大多数的编程语言都有一些基于命令行的分析器，我们可以使用它们来分析代码。它们通常可以集成在 IDE 中，但是本节课我们会专注于这些命令行工具本身。

在 Python 中，我们使用 `cProfile` 模块来分析每次函数调用所消耗的时间。在下面的例子中，我们实现了一个基础的 `grep` 命令：

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

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)
```

我们可以使用下面的命令来对这段代码进行分析。通过它的输出我们可以知道，IO 消耗了大量的时间，编译正则表达式也比较耗费时间。因为正则表达式只需要编译一次，我们可以将其移动到 `for` 循环外面来改进性能。

```
$ 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.TextIOWrapper' 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>)
```

[omitted lines]



关于 Python 的 `cProfile` 分析器（以及其他一些类似的分析器），需要注意的是它显示的是每次函数调用的时间。看上去可能快到反直觉，尤其是如果您在代码里面使用了第三方的函数库，因为内部函数调用也会被看作函数调用。

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，这也是行分析器的工作。例如，下面这段 Python 代码会向本课程的网站发起一个请求，然后解析响应返回的页面中的全部 URL：

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

# 这个装饰器会告诉行分析器
# 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()
```

如果我们使用 Python 的 `cProfile` 分析器，我们会得到超过2500行的输出结果，即使对其进行排序，我仍然搞不懂时间到底都花在哪了。如果我们使用 [line profiler](#)，它会基于行来显

示时间：

```
$ 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          1      613909.0  613909.0     96.5    @profile
6          1      21559.0   21559.0      3.4    def get_urls():
7          1      21559.0   21559.0      3.4        response = requests.get(
8          1          2.0       2.0      0.0        s = BeautifulSoup(respon
9          1          2.0       2.0      0.0        urls = []
10         25       685.0     27.4      0.1        for url in s.find_all('a
11         24       33.0      1.4      0.0        urls.append(url['href'])


```

## 内存

像 C 或者 C++ 这样的语言，内存泄漏会导致您的程序在使用完内存后不去释放它。为了应对内存类的 Bug，我们可以使用类似 [Valgrind](#) 这样的工具来检查内存泄漏问题。

对于 Python 这类具有垃圾回收机制的语言，内存分析器也是很有用的，因为对于某个对象来说，只要有指针还指向它，那它就不会被回收。

下面这个例子及其输出，展示了 [memory-profiler](#) 是如何工作的（注意装饰器和 `line-profiler` 类似）。

```
@profile
def my_func():
    a = [1] * (10 ** 6)
    b = [2] * (2 * 10 ** 7)
    del b
    return a

if __name__ == '__main__':
    my_func()
```

```
$ python -m memory_profiler example.py
Line #      Mem usage  Increment  Line Contents
=====
3                  @profile
4      5.97 MB      0.00 MB  def my_func():
5     13.61 MB      7.64 MB      a = [1] * (10 ** 6)
6    166.20 MB    152.59 MB      b = [2] * (2 * 10 ** 7)
7     13.61 MB  -152.59 MB      del b
8     13.61 MB      0.00 MB  return a
```

## 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。

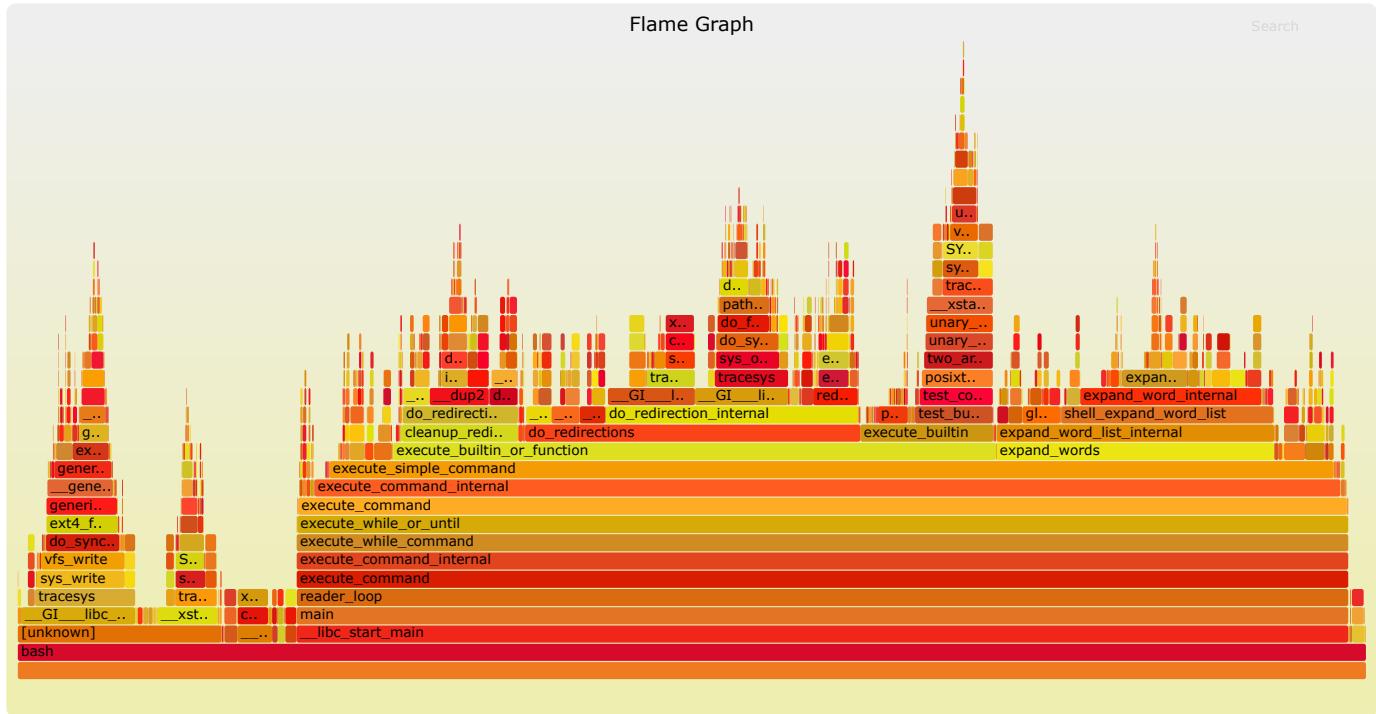
例如，`perf` 可以报告不佳的缓存局部性（poor cache locality）、大量的页错误（page faults）或活锁（livelocks）。下面是关于常见命令的简介：

- `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。

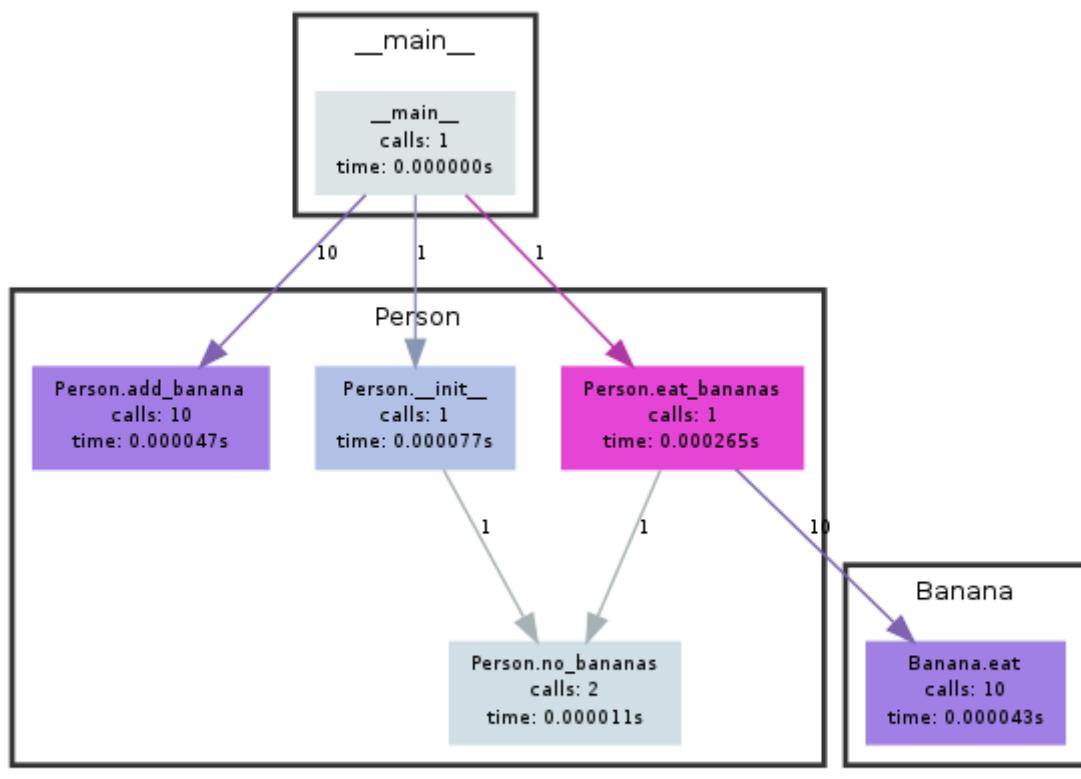
## 可视化

使用分析器来分析真实的程序时，由于软件的复杂性，其输出结果中将包含大量的信息。人类是一种视觉动物，非常不善于阅读大量的文字。因此很多工具都提供了可视化分析器输出结果的功能。

对于采样分析器来说，常见的显示 CPU 分析数据的形式是 [火焰图](#)，火焰图会在 Y 轴显示函数调用关系，并在 X 轴显示其耗时的比例。火焰图同时还是可交互的，您可以深入程序的某一具体部分，并查看其栈追踪（您可以尝试点击下面的图片）。



调用图和控制流图可以显示子程序之间的关系，它将函数作为节点并把函数调用作为边。将它们和分析器的信息（例如调用次数、耗时等）放在一起使用时，调用图会变得非常有用，它可以帮助我们分析程序的流程。在 Python 中您可以使用 [pycallgraph](#) 来生成这些图片。



## 资源监控

有时候，分析程序性能的第一步是搞清楚它所消耗的资源。程序变慢通常是因为它所需要的资源不够了。例如，没有足够的内存或者网络连接变慢的时候。

有很多很多的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](#)了，它是 [top](#)的改进版。htop 可以显示当前运行进程的多种统计信息。htop 有很多选项和快捷键，常见的有：<F6> 进程排序、t 显示树状结构和 h 打开或折叠线程。还可以留意一下 [glances](#)，它的实现类似但是用户界面更好。如果需要合并测量全部的进程，[dstat](#) 是也是一个非常好用的工具，它可以实时地计算不同子系统资源的度量数据，例如 I/O、网络、CPU 利用率、上下文切换等等；
- **I/O 操作** - [iostop](#) 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](#) 可以显示每个分区的信息，而 [du](#) 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](#) 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](#) 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](#) 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。注意，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** - [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。

## 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: 16
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: 95
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"'
```

和 debug 一样，浏览器也包含了很多不错的性能分析工具，可以用来分析页面加载，让我们可以搞清楚时间都消耗在什么地方（加载、渲染、脚本等等）。更多关于 [Firefox](#) 和 [Chrome](#) 的信息可以点击链接。

## 课后练习

### 习题解答

#### 调试

- 使用 Linux 上的 `journalctl` 或 macOS 上的 `log show` 命令来获取最近一天中超级用户的登录信息及其所执行的指令。如果找不到相关信息，您可以执行一些无害的命令，例如 `sudo ls` 然后再次查看。
- 学习 [这份](#) `pdb` 实践教程并熟悉相关的命令。更深入的信息您可以参考[这份](#)教程。
- 安装 [shellcheck](#) 并尝试对下面的脚本进行检查。这段代码有什么问题吗？请修复相关问题。在您的编辑器中安装一个linter插件，这样它就可以自动地显示相关警告信息。

```
#!/bin/sh
## Example: a typical script with several problems
for f in $(ls *.m3u)
do
    grep -qi hq.*mp3 $f \
        && echo -e 'Playlist $f contains a HQ file in mp3 format'
done
```

- (进阶题) 请阅读 [可逆调试](#) 并尝试创建一个可以工作的例子（使用 [rr](#) 或 [RevPDB](#)）。

## 性能分析

- 这里有一些排序算法的实现。请使用 [cProfile](#) 和 [line\\_profiler](#) 来比较插入排序和快速排序的性能。两种算法的瓶颈分别在哪里？然后使用 `memory_profiler` 来检查内存消耗，为什么插入排序更好一些？然后再看看原地排序版本的快排。附加题：使用 `perf` 来查看不同算法的循环次数及缓存命中及丢失情况。
- 这里有一些用于计算斐波那契数列 Python 代码，它为计算每个数字都定义了一个函数：

```

#!/usr/bin/env python

def fib0(): return 0

def fib1(): return 1

s = """def fib{}(): return fib{}() + fib{}()"""

if __name__ == '__main__':
    for n in range(2, 10):
        exec(s.format(n, n-1, n-2))
        # from functools import lru_cache
        # for n in range(10):
        #     exec("fib{} = lru_cache(1)(fib{})".format(n, n))
    print(eval("fib9()"))

```

将代码拷贝到文件中使其变为一个可执行的程序。首先安装 [pycallgraph](#) 和 [graphviz](#) (如果您能够执行 dot，则说明已经安装了 GraphViz)。并使用 pycallgraph graphviz -- ./fib.py 来执行代码并查看 pycallgraph.png 这个文件。fib0 被调用了多少次？我们可以通过记忆法来对其进行优化。将注释掉的部分放开，然后重新生成图片。这回每个 fibN 函数被调用了多少次？

3. 我们经常会遇到的情况是某个我们希望去监听的端口已经被其他进程占用了。让我们通过进程的PID查找相应的进程。首先执行 python -m http.server 4444 启动一个最简单的 web 服务器来监听 4444 端口。在另外一个终端中，执行 lsof | grep LISTEN 打印出所有监听端口的进程及相应的端口。找到对应的 PID 然后使用 kill <PID> 停止该进程。
4. 限制进程资源也是一个非常有用的技术。执行 stress -c 3 并使用 htop 对 CPU 消耗进行可视化。现在，执行 taskset --cpu-list 0,2 stress -c 3 并可视化。stress 占用了3个CPU吗？为什么没有？阅读[man taskset](#)来寻找答案。附加题：使用[cgroups](#)来实现相同的操作，限制 stress -m 的内存使用。
5. (进阶题) curl ipinfo.io 命令或执行 HTTP 请求并获取关于您 IP 的信息。打开 [Wireshark](#) 并抓取 curl 发起的请求和收到的回复报文。（提示：可以使用 http 进行过滤，只显示 HTTP 报文）

# 元编程

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

如果您使用 LaTeX 来编写论文，您需要执行哪些命令才能编译出您想要的论文呢？执行基准测试、绘制图表然后将其插入论文的命令又有哪些？或者，如何编译本课程提供的代码并执行测试呢？

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，您需要执行一系列操作。通常，这一过程包含了很多步骤，很多分支。执行一些命令来生成图表，然后执行另外的一些命令生成结果，然后再执行其他的命令来生成最终的论文。有很多事情需要我们完成，您并不是第一个因此感到苦恼的人，幸运的是，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”，而且这些工具还不少。如何选择工具完全取决于您当前手上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。理想的情况下，如果目标的依赖没有发生改动，并且我们可以从之前的构建中复用这些依赖，那么与其相关的构建规则并不会被执行。

`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'. Std
```

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'.
```

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 data.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。

## 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，执行测试套件等等。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，涵盖了一组术语的术语)，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，执行测试套件。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，这样当测试套件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，我们现在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“测试套件”。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。
- 集成测试：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。
- 回归测试：一种实现特定模式的测试，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

### 习题解答

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。您需要将构建目标设置为 [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，它会在提交前执行 `make paper.pdf` 并在出现构建失败的情况下拒绝您的提交。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。

# 安全和密码学

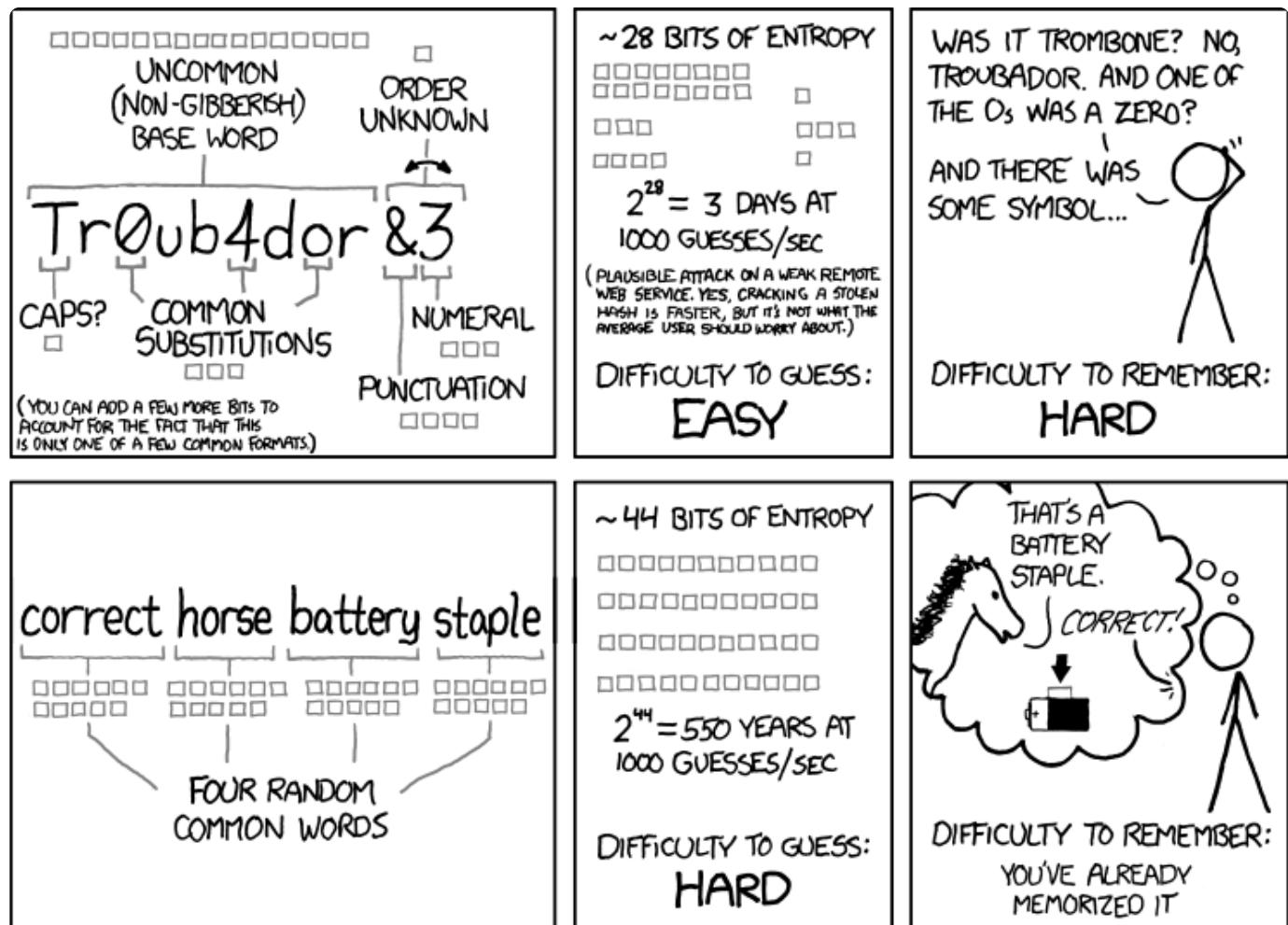
去年的这节课我们从计算机 用户的角度探讨了增强隐私保护和安全的方法。今年我们将关注比如散列函数、密钥生成函数、对称/非对称密码体系这些安全和密码学的概念是如何应用于前几节课所学到的工具 (Git和SSH) 中的。

本课程不能作为计算机系统安全 (6.858) 或者 密码学 (6.857) 以及 6.875 的替代。如果你不是密码学的专家,请不要试图创造或者修改加密算法。从事和计算机系统安全相关的工作同理。

这节课将对一些基本的概念进行简单 (但实用) 的说明。虽然这些说明不足以让你学会如何 设计安全系统或者加密协议,但我们希望你可以对现在使用的程序和协议有一个大概了解。

## 熵

**熵**(Entropy) 度量了不确定性并可以用来决定密码的强度。



THROUGH 20 YEARS OF EFFORT, WE'VE SUCCESSFULLY TRAINED EVERYONE TO USE PASSWORDS THAT ARE HARD FOR HUMANS TO REMEMBER, BUT EASY FOR COMPUTERS TO GUESS.

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  $\log_2$ (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）。

## 散列函数

[密码散列函数](#) (Cryptographic hash function) 可以将任意大小的数据映射为一个固定大小的输出。除此之外，还有一些其他特性。一个散列函数的大概规范如下：

```
hash(value: array<byte>) -> vector<byte, N> (N对于该函数固定)
```

[SHA-1](#)是Git中使用的一种散列函数，它可以将任意大小的输入映射为一个160比特（可被40位十六进制数表示）的输出。下面我们用 `sha1sum` 命令来测试SHA1对几个字符串的输出：

```
$ printf 'hello' | sha1sum
aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d
$ printf 'hello' | sha1sum
aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d
$ printf 'Hello' | sha1sum
f7ff9e8b7bb2e09b70935a5d785e0cc5d9d0abf0
```

抽象地讲，散列函数可以被认为是一个不可逆，且看上去随机（但具确定性）的函数（这就是[散列函数的理想模型](#)）。一个散列函数拥有以下特性：

- 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  $m_2 \neq m_1$  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）。

注：虽然SHA-1还可以用于特定用途，但它已经不再被认为是一个强密码散列函数。你可参照[密码散列函数的生命周期](#)这个表格了解一些散列函数是何时被发现弱点及破解的。请注意，针对应用推荐特定的散列函数超出了本课程内容的范畴。如果选择散列函数对于你的工作非常重要，请先系统学习信息安全及密码学。

## 密码散列函数的应用

- Git中的内容寻址存储(Content addressed storage)：[散列函数](#)是一个宽泛的概念（存在非密码学的散列函数），那么Git为什么要特意使用密码散列函数？

- 文件的信息摘要(Message digest): 像Linux ISO这样的软件可以从非官方的（有时不太可信的）镜像站下载，所以需要设法确认下载的软件和官方一致。官方网站一般会在（指向镜像站的）下载链接旁边备注安装文件的哈希值。用户从镜像站下载安装文件后可以对照公布的哈希值来确定安装文件没有被篡改。
- 承诺机制(Commitment scheme): 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  $r = \text{random}()$ ，并和你分享它的哈希值  $h = \text{sha256}(r)$ 。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  $\text{sha256}(r)$  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

**密钥生成函数** (Key Derivation Functions) 作为密码散列函数的相关概念，被应用于包括生成固定长度，可以使用在其他密码算法中的密钥等方面。为了对抗穷举法攻击，密钥生成函数通常较慢。

### 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  $salt = \text{random}()$ ，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)。

## 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
```



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.`

## 非对称加密的应用

- PGP电子邮件加密：用户可以将所使用的公钥在线发布，比如：PGP密钥服务器或 Keybase。  
任何人都可以向他们发送加密的电子邮件。
- 聊天加密：像 Signal 和 Keybase 使用非对称密钥来建立私密聊天。
- 软件签名：Git 支持用户对提交(commit)和标签(tag)进行GPG签名。任何人都可以使用软件开发者公布的签名公钥验证下载的已签名软件。

## 密钥分发

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、pass 和 1Password。

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。

## 资源

- [去年的讲稿](#): 更注重于计算机用户可以如何增强隐私保护和安全
- [Cryptographic Right Answers](#): 解答了在一些应用环境下“应该使用什么加密？”的问题

## 课后练习

### 习题解答

#### 1. 熵

1. 假设一个密码是由四个小写的单词拼接组成，每个单词都是从一个含有10万单词的字典中随机选择，且每个单词选中的概率相同。一个符合这样构造的例子是 `correcthorsebatterystaple`。这个密码有多少比特的熵？
  2. 假设另一个密码是用八个随机的大小写字母或数字组成。一个符合这样构造的例子是 `rg8Ql34g`。这个密码又有多少比特的熵？
  3. 哪一个密码更强？
  4. 假设一个攻击者每秒可以尝试1万个密码，这个攻击者需要多久可以分别破解上述两个密码？
2. **密码散列函数** 从[Debian镜像站](#)下载一个光盘映像（比如这个来自阿根廷镜像站的[映像](#)）。使用 `sha256sum` 命令对比下载映像的哈希值和官方Debian站公布的哈希值。如果你下载了上面的映像，官方公布的哈希值可以参考[这个文件](#)。
  3. **对称加密** 使用[OpenSSL](#)的AES模式加密一个文件: `openssl aes-256-cbc -salt -in {源文件名} -out {加密文件名}`。使用 `cat` 或者 `hexdump` 对比源文件和加密的文件，再用 `openssl aes-256-cbc -d -in {加密文件名} -out {解密文件名}` 命令解密刚刚加密的文件。最后使用 `cmp` 命令确认源文件和解密后的文件内容相同。
  4. **非对称加密**
    1. 在你自己的电脑上使用更安全的[ED25519算法](#)生成一组[SSH 密钥对](#)。为了确保私钥不使用时的安全，一定使用密码加密你的私钥。

2. [配置GPG](#)。
3. 给Anish发送一封加密的电子邮件 ([Anish的公钥](#)) 。
4. 使用 `git commit -S` 命令签名一个Git提交，并使用 `git show --show-signature` 命令验证这个提交的签名。或者，使用 `git tag -s` 命令签名一个Git标签，并使用 `git tag -v` 命令验证标签的签名。

# 大杂烩

## 目录

- [修改键位映射](#)
- [守护进程](#)
- [FUSE](#)
- [备份](#)
- [API \(应用程序接口\)](#)
- [常见命令行标志参数及模式](#)
- [窗口管理器](#)
- [VPN](#)
- [Markdown](#)
- [Hammerspoon \(macOS桌面自动化\)](#)
  - 资源
- [开机引导以及 Live USB](#)
- [Docker, Vagrant, VMs, Cloud, OpenStack](#)
- [交互式记事本编程](#)
- [GitHub](#)

## 修改键位映射

作为一名程序员，键盘是你的主要输入工具。它像计算机里的其他部件一样是可配置的，而且值得你在这上面花时间。

一个很常见的配置是修改键位映射。通常这个功能由在计算机上运行的软件实现。当某一个按键被按下，软件截获键盘发出的按键事件（keypress event）并使用另外一个事件取代。比如：

- 将 Caps Lock 映射为 Ctrl 或者 Escape：Caps Lock 使用了键盘上一个非常方便的位置而它的功能却很少被用到，所以我们（讲师）非常推荐这个修改；
- 将 PrtSc 映射为播放/暂停：大部分操作系统支持播放/暂停键；
- 交换 Ctrl 和 Meta 键（Windows 的徽标键或者 Mac 的 Command 键）。

你也可以将键位映射为任意常用的指令。软件监听到特定的按键组合后会运行设定的脚本。

- 打开一个新的终端或者浏览器窗口；
- 输出特定的字符串，比如：一个超长邮件地址或者 MIT ID；
- 使计算机或者显示器进入睡眠模式。

甚至更复杂的修改也可以通过软件实现：

- 映射按键顺序，比如：按 Shift 键五下切换大小写锁定；
- 区别映射单点和长按，比如：单点 Caps Lock 映射为 Escape，而长按 Caps Lock 映射为 Ctrl；

- 对不同的键盘或软件保存专用的映射配置。

下面是一些修改键位映射的软件：

- macOS - [karabiner-elements](#), [skhd](#) 或者 [BetterTouchTool](#)
- Linux - [xmodmap](#) 或者 [Autokey](#)
- Windows - 控制面板, [AutoHotkey](#) 或者 [SharpKeys](#)
- QMK - 如果你的键盘支持定制固件, [QMK](#) 可以直接在键盘的硬件上修改键位映射。保留在键盘里的映射免除了在别的机器上的重复配置。

## 守护进程

即便守护进程 (daemon) 这个词看上去有些陌生，你应该已经大约明白它的概念。大部分计算机都有一系列在后台保持运行，不需要用户手动运行或者交互的进程。这些进程就是守护进程。以守护进程运行的程序名一般以 d 结尾，比如 SSH 服务端 sshd，用来监听传入的 SSH 连接请求并对用户进行鉴权。

Linux 中的 systemd (the system daemon) 是最常用的配置和运行守护进程的方法。运行 systemctl status 命令可以看到正在运行的所有守护进程。这里面有很多可能你没有见过，但是掌管了系统的核心部分的进程：管理网络、DNS解析、显示系统的图形界面等等。用户使用 systemctl 命令和 systemd 交互来 enable (启用)、 disable (禁用)、 start (启动)、 stop (停止)、 restart (重启)、或者 status (检查) 配置好的守护进程及系统服务。

systemd 提供了一个很方便的界面用于配置和启用新的守护进程或系统服务。下面的配置文件使用了守护进程来运行一个简单的 Python 程序。文件的内容非常直接所以我们不对它详细阐述。systemd 配置文件的详细指南可参见 [freedesktop.org](#)。

```

# /etc/systemd/system/myapp.service
[Unit]
# 配置文件描述
Description=My Custom App
# 在网络服务启动后启动该进程
After=network.target

[Service]
# 运行该进程的用户
User=foo
# 运行该进程的用户组
Group=foo
# 运行该进程的根目录
WorkingDirectory=/home/foo/projects/mydaemon
# 开始该进程的命令
ExecStart=/usr/bin/local/python3.7 app.py
# 在出现错误时重启该进程
Restart=on-failure

[Install]
# 相当于Windows的开机启动。即使GUI没有启动，该进程也会加载并运行
WantedBy=multi-user.target
# 如果该进程仅需要在GUI活动时运行，这里应写作：
# WantedBy=graphical.target
# graphical.target在multi-user.target的基础上运行和GUI相关的服务

```

如果你只是想定期运行一些程序，可以直接使用 [cron](#)。它是一个系统内置的，用来执行定期任务的守护进程。

## FUSE

现在的软件系统一般由很多模块化的组件构建而成。你使用的操作系统可以通过一系列共同的方式使用不同的文件系统上的相似功能。比如当你使用 `touch` 命令创建文件的时候，`touch` 使用系统调用（system call）向内核发出请求。内核再根据文件系统，调用特有的方法来创建文件。这里的问题是，UNIX 文件系统在传统上是以内核模块的形式实现，导致只有内核可以进行文件系统相关的调用。

[FUSE](#)（用户空间文件系统）允许运行在用户空间上的程序实现文件系统调用，并将这些调用与内核接口联系起来。在实践中，这意味着用户可以在文件系统调用中实现任意功能。

FUSE 可以用于实现如：一个将所有文件系统操作都使用 SSH 转发到远程主机，由远程主机处理后返回结果到本地计算机的虚拟文件系统。这个文件系统里的文件虽然存储在远程主机，对于本地计算机上的软件而言和存储在本地别无二致。`sshfs` 就是一个实现了这种功能的 FUSE 文件系统。

一些有趣的 FUSE 文件系统包括：

- [sshfs](#): 使用 SSH 连接在本地打开远程主机上的文件
- [rclone](#): 将 Dropbox、Google Drive、Amazon S3、或者 Google Cloud Storage 一类的云存储服务挂载为本地文件系统
- [gocryptfs](#): 覆盖在加密文件上的文件系统。文件以加密形式保存在磁盘里，但该文件系统挂载后用户可以直接从挂载点访问文件的明文
- [kbfs](#): 分布式端到端加密文件系统。在这个文件系统里有私密 (private)，共享 (shared)，以及公开 (public) 三种类型的文件夹
- [borgbackup](#): 方便用户浏览删除重复数据后的压缩加密备份

## 备份

任何没有备份的数据都可能在一个瞬间永远消失。复制数据很简单，但是可靠地备份数据很难。下面列举了一些关于备份的基础知识，以及一些常见做法容易掉进的陷阱。

首先，复制存储在同一个磁盘上的数据不是备份，因为这个磁盘是一个单点故障 (single point of failure)。这个磁盘一旦出现问题，所有的数据都可能丢失。放在家里的外置磁盘因为火灾、抢劫等原因可能会和源数据一起丢失，所以是一个弱备份。推荐的做法是将数据备份到不同的地点存储。

同步方案也不是备份。即使方便如 Dropbox 或者 Google Drive，当数据在本地被抹除或者损坏，同步方案可能会把这些“更改”同步到云端。同理，像 RAID 这样的磁盘镜像方案也不是备份。它不能防止文件被意外删除、损坏、或者被勒索软件加密。

有效备份方案的几个核心特性是：版本控制，删除重复数据，以及安全性。对备份的数据实施版本控制保证了用户可以从任何记录过的历史版本中恢复数据。在备份中检测并删除重复数据，使其仅备份增量变化可以减少存储开销。在安全性方面，作为用户，你应该考虑别人需要有什么信息或者工具才可以访问或者完全删除你的数据及备份。最后一点，不要盲目信任备份方案。用户应该经常检查备份是否可以用来恢复数据。

备份不局限于备份在本地计算机上的文件。云端应用的重大发展使得我们很多的数据只存储在云端。当我们无法登录这些应用，在云端存储的网络邮件，社交网络上的照片，流媒体音乐播放列表，以及在线文档等等都会随之丢失。用户应该有这些数据的离线备份，而且已经有项目可以帮助下载并存储它们。

如果想要了解更多具体内容，请参考本课程2019年关于备份的[课堂笔记](#)。

## API (应用程序接口)

关于如何使用计算机有效率地完成 本地 任务，我们这堂课已经介绍了很多方法。这些方法在互联网上其实也适用。大多数线上服务提供的 API (应用程序接口) 让你可以通过编程方式来访问这些服务的数据。比如，美国国家气象局就提供了一个可以从 shell 中获取天气预报的 API。

这些 API 大多具有类似的格式。它们的结构化 URL 通常使用 `api.service.com` 作为根路径，用户可以访问不同的子路径来访问需要调用的操作，以及添加查询参数使 API 返回符合查询参数条件的结果。

以美国天气数据为例，为了获得某个地点的天气数据，你可以发送一个 GET 请求（比如使用 curl）到 <https://api.weather.gov/points/42.3604,-71.094>。返回中会包括一系列用于获取特定信息（比如小时预报、气象观察站信息等）的 URL。通常这些返回都是 JSON 格式，你可以使用 jq 等工具来选取需要的部分。

有些需要认证的 API 通常要求用户在请求中加入某种私密令牌（secret token）来完成认证。请阅读你想访问的 API 所提供的文档来确定它请求的认证方式，但是其实大多数 API 都会使用 OAuth。OAuth 通过向用户提供一系列仅可用于该 API 特定功能的私密令牌进行校验。因为使用了有效 OAuth 令牌的请求在 API 看来就是用户本人发出的请求，所以请一定保管好这些私密令牌。否则其他人就可以冒用你的身份进行任何你可以在这个 API 上进行的操作。

[IFTTT](#) 这个网站可以将很多 API 整合在一起，让某 API 发生的特定事件触发在其他 API 上执行的任务。IFTTT 的全称 If This Then That 足以说明它的用法，比如在检测到用户的新推文后，自动发布在其他平台。但是你可以对它支持的 API 进行任意整合，所以试着来设置一下任何你需要的功能吧！

## 常见命令行标志参数及模式

命令行工具的用法千差万别，阅读 man 页面可以帮助你理解每种工具的用法。即便如此，下面我们将介绍一下命令行工具一些常见的共同功能。

- 大部分工具支持 --help 或者类似的标志参数（flag）来显示它们的简略用法。
- 会造成不可撤回操作的工具一般会提供“空运行”（dry run）标志参数，这样用户可以确认工具真实运行时会进行的操作。这些工具通常也会有“交互式”（interactive）标志参数，在执行每个不可撤回的操作前提示用户确认。
- --version 或者 -V 标志参数可以让工具显示它的版本信息（对于提交软件问题报告非常重要）。
- 基本所有的工具支持使用 --verbose 或者 -v 标志参数来输出详细的运行信息。多次使用这个标志参数，比如 -vvv，可以让工具输出更详细的信息（经常用于调试）。同样，很多工具支持 --quiet 标志参数来抑制除错误提示之外的其他输出。
- 大多数工具中，使用 - 代替输入或者输出文件名意味着工具将从标准输入（standard input）获取所需内容，或者向标准输出（standard output）输出结果。
- 会造成破坏性结果的工具一般默认进行非递归的操作，但是支持使用“递归”（recursive）标志函数（通常是 -r）。
- 有的时候你可能需要向工具传入一个看上去像标志参数的普通参数，比如：
  - 使用 rm 删除一个叫 -r 的文件；
  - 在通过一个程序运行另一个程序的时候（ssh machine foo），向内层的程序（foo）传递一个标志参数。

这时候你可以使用特殊参数 -- 让某个程序停止处理 -- 后面出现的标志参数以及选项（以 - 开头的内容）：

- rm -- -r 会让 rm 将 -r 当作文件名；
- ssh machine --for-ssh -- foo --for-foo 的 -- 会让 ssh 知道 --for-foo 不是 ssh 的标志参数。

## 窗口管理器

大部分人适应了 Windows、macOS、以及 Ubuntu 默认的“拖拽”式窗口管理器。这些窗口管理器的窗口一般就堆在屏幕上，你可以拖拽改变窗口的位置、缩放窗口、以及让窗口堆叠在一起。这种堆叠式 (floating/stacking) 管理器只是窗口管理器中的一种。特别在 Linux 中，有很多其他的管理器。

平铺式 (tiling) 管理器就是一个常见的替代。顾名思义，平铺式管理器会把不同的窗口像贴瓷砖一样平铺在一起而不和其他窗口重叠。这和 [tmux](#) 管理终端窗口的方式类似。平铺式管理器按照写好的布局显示打开的窗口。如果只打开一个窗口，它会填满整个屏幕。新开一个窗口的时候，原来的窗口会缩小到比如三分之二或者三分之一的大小来腾出空间。打开更多的窗口会让已有的窗口进一步调整。

就像 tmux 那样，平铺式管理器可以让你在完全不使用鼠标的情况下使用键盘切换、缩放、以及移动窗口。它们值得一试！

## VPN

VPN 现在非常火，但我们不清楚这是不是因为[一些好的理由](#)。你应该了解 VPN 能提供的功能和它的限制。使用了 VPN 的你对于互联网而言，**最好的情况下**也就是换了一个网络供应商 (ISP)。所有你发出的流量看上去来源于 VPN 供应商的网络而不是你的“真实”地址，而你实际接入的网络只能看到加密的流量。

虽然这听上去非常诱人，但是你应该知道使用 VPN 只是把原本对网络供应商的信任放在了 VPN 供应商那里——网络供应商 **能看到的**，VPN 供应商 **也都能看到**。如果相比网络供应商你更信任 VPN 供应商，那当然很好。反之，则连接VPN的价值不明确。机场的不加密公共热点确实不可以信任，但是在家庭网络环境里，这个差异就没有那么明显。

你也应该了解现在大部分包含用户敏感信息的流量已经被 HTTPS 或者 TLS 加密。这种情况下你所处的网络环境是否“安全”不太重要：供应商只能看到你和哪些服务器在交谈，却不能看到你们交谈的内容。

这一切的大前提是“最好的情况”。曾经发生过 VPN 提供商错误使用弱加密或者直接禁用加密的先例。另外，有些恶意的或者带有投机心态的供应商会记录和你有关的所有流量，并很可能会将这些信息卖给第三方。找错一家 VPN 经常比一开始就不用 VPN 更危险。

MIT 向有访问校内资源需求的成员开放自己运营的 [VPN](#)。如果你也想自己配置一个 VPN，可以了解一下 [WireGuard](#) 以及 [Algo](#)。

## Markdown

你在职业生涯中大概率会编写各种各样的文档。在很多情况下这些文档需要使用标记来增加可读性，比如：插入粗体或者斜体内容，增加页眉、超链接、以及代码片段。

在不使用 Word 或者 LaTeX 等复杂工具的情况下，你可以考虑使用 [Markdown](#) 这个轻量化的标记语言 (markup language)。你可能已经见过 Markdown 或者它的一个变种。很多环境都支持并使用 Markdown 的一些子功能。

Markdown 致力于将人们编写纯文本时的一些习惯标准化。比如：

- 用 \* 包围的文字表示强调（斜体），或者用 \*\* 表示特别强调（粗体）；
- 以 # 开头的行是标题，# 的数量表示标题的级别，比如：## 二级标题；
- 以 - 开头代表一个无序列表的元素。一个数字加 . (比如 1.) 代表一个有序列表元素；
- 反引号 ` (backtick) 包围的文字会以代码字体显示。如果要显示一段代码，可以在每一行前加四个空格缩进，或者使用三个反引号包围整个代码片段：

就像这样

- 如果要添加超链接，将需要显示的文字用方括号包围，并在后面紧接着用圆括号包围链接：[显示文字] (指向的链接)。

Markdown 不仅容易上手，而且应用非常广泛。实际上本课程的课堂笔记和其他资料都是使用 Markdown 编写的。点击[这个链接](#)可以看到本页面的原始 Markdown 内容。

## Hammerspoon (macOS 桌面自动化)

[Hammerspoon](#) 是面向 macOS 的一个桌面自动化框架。它允许用户编写和操作系统功能挂钩的 Lua 脚本，从而与键盘、鼠标、窗口、文件系统等交互。

下面是 Hammerspoon 的一些示例应用：

- 绑定移动窗口到的特定位置的快捷键
- 创建可以自动将窗口整理成特定布局的菜单栏按钮
- 在你到实验室以后，通过检测所连接的 WiFi 网络自动静音扬声器
- 在你不小心拿了朋友的充电器时弹出警告

从用户的角度，Hammerspoon 可以运行任意 Lua 代码，绑定菜单栏按钮、按键、或者事件。Hammerspoon 提供了一个全面的用于和系统交互的库，因此它能没有限制地实现任何功能。你可以从头编写自己的 Hammerspoon 配置，也可以结合别人公布的配置来满足自己的需求。

## 资源

- [Getting Started with Hammerspoon](#): Hammerspoon 官方教程
- [Sample configurations](#): Hammerspoon 官方示例配置
- [Anish's Hammerspoon config](#): Anish 的 Hammerspoon 配置

## 开机引导以及 Live USB

在你的计算机启动时，[BIOS](#) 或者 [UEFI](#) 会在加载操作系统之前对硬件系统进行初始化，这被称为引导 (booting)。你可以通过按下计算机提示的键位组合来配置引导，比如 Press F9 to configure BIOS. Press F12 to enter boot menu。在 BIOS 菜单中你可以对硬件相关的设置进行更改，也可以在引导菜单中选择从硬盘以外的其他设备加载操作系统——比如 Live USB。

[Live USB](#) 是包含了完整操作系统的闪存盘。Live USB 的用途非常广泛，包括：

- 作为安装操作系统的启动盘；
- 在不将操作系统安装到硬盘的情况下，直接运行 Live USB 上的操作系统；
- 对硬盘上的相同操作系统进行修复；
- 恢复硬盘上的数据。

Live USB 通过在闪存盘上 写入 操作系统的镜像制作，而写入不是单纯的往闪存盘上复制 .iso 文件。你可以使用 [UNetbootin](#)、[Rufus](#) 等 Live USB 写入工具制作。

## Docker, Vagrant, VMs, Cloud, OpenStack

[虚拟机](#) (Virtual Machine) 以及容器化 (containerization) 等工具可以帮助你模拟一个包括操作系统的完整计算机系统。虚拟机可以用于创建独立的测试或者开发环境，以及用作安全测试的沙盒。

[Vagrant](#) 是一个构建和配置虚拟开发环境的工具。它支持用户在配置文件中写入比如操作系统、系统服务、需要安装的软件包等描述，然后使用 `vagrant up` 命令在各种环境 (VirtualBox, KVM, Hyper-V等) 中启动一个虚拟机。[Docker](#) 是一个使用容器化概念的类似工具。

租用云端虚拟机可以享受以下资源的即时访问：

- 便宜、常开、且有公共IP地址的虚拟机用来托管网站等服务
- 有大量 CPU、磁盘、内存、以及 GPU 资源的虚拟机
- 超出用户可以使用的物理主机数量的虚拟机
  - 相比物理主机的固定开支，虚拟机的开支一般按运行的时间计算。所以如果用户只需要在短时间内使用大量算力，租用1000台虚拟机运行几分钟明显更加划算。

受欢迎的 VPS 服务商有 [Amazon AWS](#), [Google Cloud](#)、[Microsoft Azure](#)以及 [DigitalOcean](#)。

MIT CSAIL 的成员可以使用 [CSAIL OpenStack instance](#) 申请免费的虚拟机用于研究。

## 交互式记事本编程

[交互式记事本](#)可以帮助开发者进行与运行结果交互等探索性的编程。现在最受欢迎的交互式记事本环境大概是 [Jupyter](#)。它的名字来源于所支持的三种核心语言：Julia、Python、R。[Wolfram Mathematica](#) 是另外一个常用于科学计算的优秀环境。

## GitHub

[GitHub](#) 是最受欢迎的开源软件开发平台之一。我们课程中提到的很多工具，从 [vim](#) 到 [Hammerspoon](#)，都托管在 Github 上。向你每天使用的开源工具作出贡献其实很简单，下面是两种贡献者们经常使用的方法：

- 创建一个[议题 \(issue\)](#)。议题可以用来反映软件运行的问题或者请求新的功能。创建议题并不需要创建者阅读或者编写代码，所以它是一个轻量化的贡献方式。高质量的问题报告对于开发者十分重要。在现有的议题发表评论也可以对项目的开发作出贡献。

– 使用[拉取请求 \(pull request\)](#) 提交代码更改。由于涉及到阅读和编写代码，提交拉取请求总的来说比创建议题更加深入。拉取请求是请求别人把你自己的代码拉取（且合并）到他们的仓库里。很多开源项目仅允许认证的管理者管理项目代码，所以一般需要[复刻 \(fork\)](#) 这些项目的上游仓库（upstream repository），在你的 Github 账号下创建一个内容完全相同但是由你控制的复刻仓库。这样你就可以在这个复刻仓库自由创建新的分支并推送修复问题或者实现新功能的代码。完成修改以后再回到开源项目的 Github 页面[创建一个拉取请求](#)。

提交请求后，项目管理者会和你交流拉取请求里的代码并给出反馈。如果没有问题，你的代码会和上游仓库中的代码合并。很多大的开源项目会提供贡献指南，容易上手的议题，甚至专门的指导项目来帮助参与者熟悉这些项目。

# 提问&回答

最后一节课，我们回答学生提出的问题：

- [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](#)
- [你会优先学习的工具有那些？](#)
- [使用 Python VS Bash脚本 VS 其他语言？](#)
- [source script.sh 和 ./script.sh 有什么区别？](#)
- [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？](#)
- [我应该用 apt-get install 还是 pip install 去下载软件包呢？](#)
- [用于提高代码性能，简单好用的性能分析工具有哪些？](#)
- [你使用那些浏览器插件？](#)
- [有哪些有用的数据整理工具？](#)
- [Docker和虚拟机有什么区别？](#)
- [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）？](#)
- [使用 Vim 编辑器 VS Emacs 编辑器？](#)
- [机器学习应用的提示或技巧？](#)
- [还有更多的 Vim 小窍门吗？](#)
- [2FA是什么，为什么我需要使用它？](#)
- [对于不同的 Web 浏览器有什么评价？](#)

## 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 对于简单的使用情形没什么问题，但是很难对于所有可能的输入都正确。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 - 系统用户的主目录
- /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 )通常重启时删除
- /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，[\\_bpfttrace](#) 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程中，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，而 Safari 则基于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。