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` （print working directory） 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：

```
  -l                         use a long listing format
```

```
missing:~$ ls -l /home
drwxr-xr-x 1 missing  users  4096 Jun 15  2019 missing
```

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。

`cd -` 可以实现在两个dir反复横跳，表示跳到上一个路径处

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）。

`rm` 默认不是递归的，需要 `rm -r` 才能删除文件夹，`rmdir` 是更安全的做法，只能删除空文件夹

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。

```
missing:~$ man ls
```

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
//以hello.txt作为输入流，hello2.txt作为输出流
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：他们本身并不耦合，只是我们改变了输入输出流

```
missing:~$ ls -l / | tail -n1
//列举根目录，tail表示输出输入的倒数n后面那个数字的行数的东西
drwxr-xr-x 1 root  root  4096 Jun 20  2019 var
missing:~$ curl --head --silent google.com | grep --ignore-case content-length | cut --deli
219
```

`curl` 根据URL抓取界面

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以 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
//tee 的作用是输入东西，并输出给
```

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
```

也可以直接 `sudo su` 进入root

`xdg-open xxx` 在macos或者linux上会以合适的程序打开

HW

https://missing-semester-cn.github.io/missing-notes-and-solutions/2020/solutions/course-shell-solution/

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
```

1. 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
2. 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。
3. 查看 `chmod` 的手册(例如，使用 `man chmod` 命令)
4. 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
5. 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
6. 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。

```
cd tmp
mkdir missing
man touch
touch semester
echo #!/bin/sh > semester
echo curl --head --silent https://missing.csail.mit.edu >> semester
cat semester

./semester
man chmod
chmod 777 semester
./semester
./semester | grep last-modified > ~/last-modified.txt
```

- touch：新建文件或者修改时间戳

- 注意流的append >>
- chmod修改权限
- grep寻找

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。

```
1 foo=bar
2 echo "$foo"
3 # 打印 bar
4 echo '$foo'
5 # 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。

```
1 mcd () {
2     mkdir -p "$1"
3     cd "$1"
4 }
```

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
  - Error code
    - 0表示成功
    - 1表示失败
- `$$` - 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，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, failtrue || echo "Will not be printed"
#true && echo "Things went well"
# Things went well false && echo "Will not be printed"
#false ; echo "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/bashecho
"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
    # 循环变量，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
    # ne是not equal
    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 /newpath

# 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
# 先创建2个文件夹，然后在各自目录下面创建那些文件
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。

```
shellcheck mcd.sh
```

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看上去是这样的 `#!/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。

`locate xxx`

`updatedb`

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。

```
grep -R foobar .
```

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`。

升级版

https://zhuanlan.zhihu.com/p/438124776

- Homebrew
- iTerm2
- Oh my zsh
- 内置 Shell 命令增强
  - bat代替cat
  - exa代替ls
  - fd代替find
  - rg代替grep
- asdf - 使用单个工具对多个运行时进行版本管理
- fzf模糊查找
  - fzf 最常用的功能
    - Ctrl + C：查找以往使用过的命令历史
    - Ctrl + T：在当前路径及子路径下查找文件
    - Ctrl + R：查找历史命令行指令
- tldr - Too Long; Didn't Read
- Nnn
  - 直接输入nnn即可
- Ranger
- The Fuck - 纠正上一条命令的拼写错误
- tig
- 快速跳转

- Fasd
  - Z 快速切换到最常访问的目录
- Autojump
- z.lua
  - Z

HW

答案：https://missing-semester-cn.github.io/2020/shell-tools/

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 ..
```

```
ls -a
ls -la
ls -ltr
ls --color=auto
```

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。

```
vim marco.sh

#!/bin/bash
marco()
{
    echo "$(pwd)" > $HOME/marco_history.log
    echo "save pwd $(pwd)"
}
polo()
{
    cd "$(cat "$HOME/marco_history.log")"
}
```

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"
```

```
count=1

 while true
 do
     ./buggy.sh 2> out.log
     if [[ $? -ne 0 ]]; then
         echo "failed after $count times"cat out.log
         break
     fi((count++))done
```

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，它可以使用标准输入中的内容作为参数。 例如 `ls | xargs rm` 会删除当前目录中的所有文件。

您的任务是编写一个命令，它可以递归地查找文件夹中所有的HTML文件，并将它们压缩成zip文件。注意，即使文件名中包含空格，您的命令也应该能够正确执行（提示：查看 `xargs` 的参数 `-d` ，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。

```
//mac

mkdir html
cd html
touch {1..10}.html
mkdir htmlhtml
cd htmlhtml
touch xxx.html
cd ..
cd ..
tree
find html -name "*.html" -print0 | xargs -0 tar vcf html.zip

```

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？

```
find . -type f -mmin -60 -print0 | xargs -0 ls -lt | head -10
```