PART1-WEEK3-REVIEW

实验出来的小技巧：

`shellcheck`指令 — 需要提前下载在vagrant里

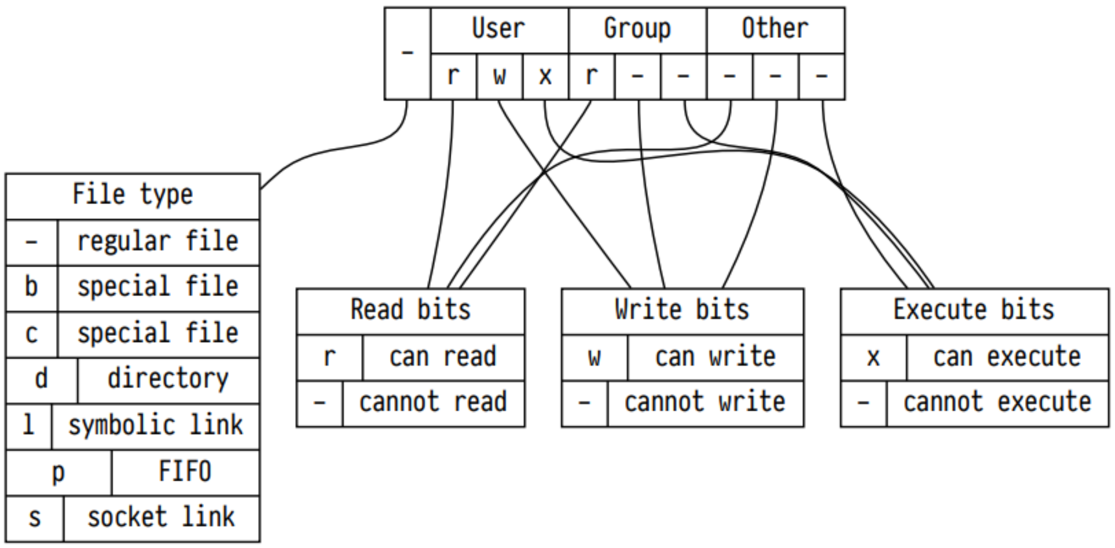
strace —需要下载在vagrant里

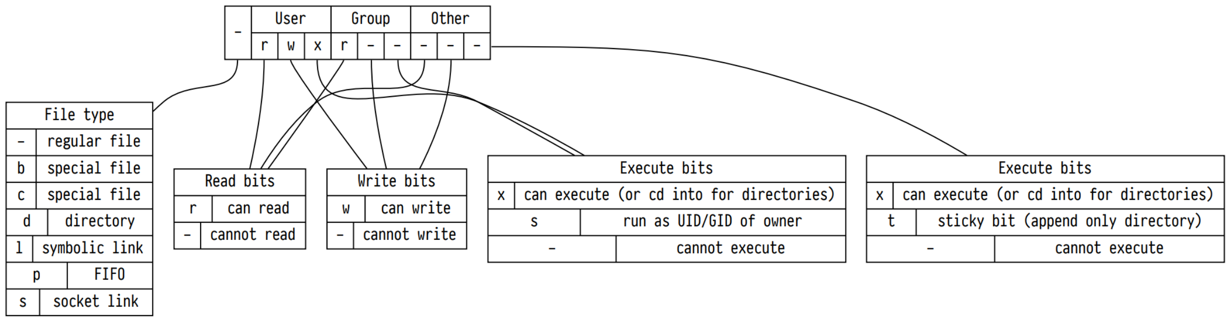
如何消除列： control + v 进入visual block模式 — 向上/下的方向键进入列选中 — 按d键列删除 - shell checking的时候

Slides- Permissions

root - super user / system administrator / UID 0

Permission Picture- UNIX DAC





t → Sticky bit t is mostly for **log** directories and **temporary** directories.

s → 把这个文件run的时候的UID/GID 设置成owner的

日志(log) → can append but not delete them.

`setuid/setgid` bits are used for privilege separation(权限分离) - 程序存在

`passwd` program changes the password, 指令是 `ls -l $(command -v passwd)`

setuid: `su` switch to user (by default root) with their password-要密码，默认root

`sudo` switch to user if the sysadmin says you're **allowed** to with your password

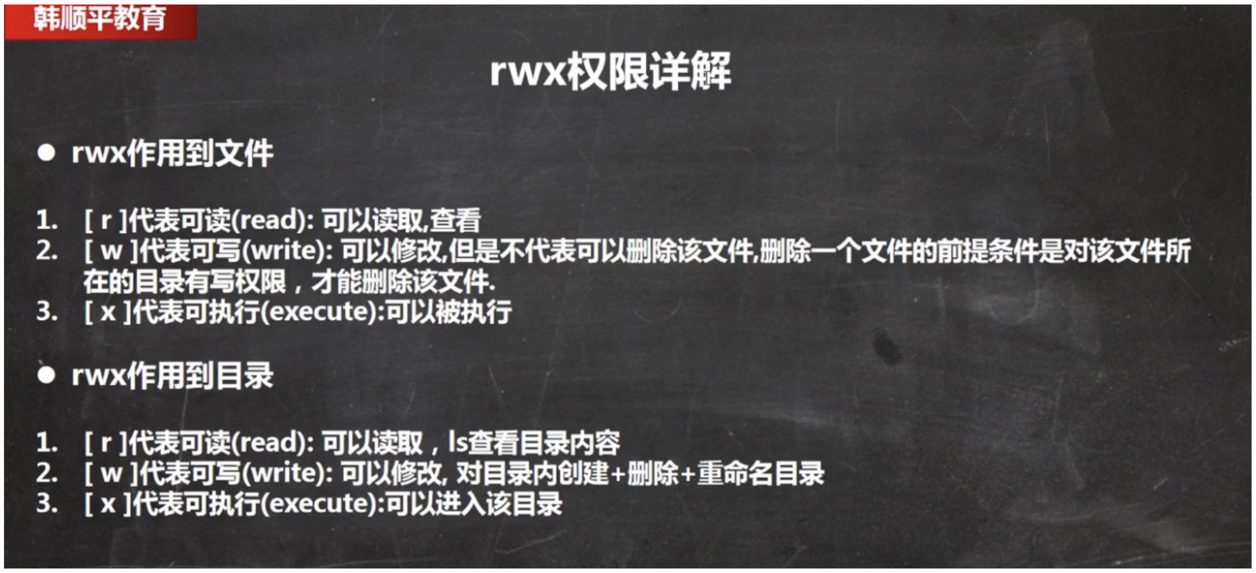
`doas` modern rewrite of sudo with less bugs and Spiderman references重写sudo减少错误

`msn su; man sudo; man doas` → see difference

`chmod` to change permission; `chown` to change file owners

```
# Change who owns a file:
ls -l exam
-rw-r--r-- 1 joseph joseph 0 Jan 12 11:49 exam
chown joseph:staff exam
# Alternatively...
chown :staff exam
ls -l exam
-rw-r--r-- 1 joseph staff 0 Jan 12 11:49 exam
#(See man 1 chown)
# Change a file permissions: +是得到, -是删除
chmod go-wx exam
```

```
ls -l exam
-rw-r--r-- 1 joseph staff 0 Jan 12 11:49 exam
```



韩顺平教育

rwx权限详解

● rwx作用到文件

1. [r]代表可读(read): 可以读取,查看
2. [w]代表可写(write): 可以修改,但是不代表可以删除该文件,删除一个文件的前提条件是对该文件所在的目录有写权限，才能删除该文件.
3. [x]代表可执行(execute):可以被执行

● rwx作用到目录

1. [r]代表可读(read): 可以读取，ls查看目录内容
2. [w]代表可写(write): 可以修改, 对目录内创建+删除+重命名目录
3. [x]代表可执行(execute):可以进入该目录

Slides- ShellScripting

Anything has to do more than once → write a script for it.

Shekkscripting is about automating all those tedious(乏味) little jobs:

1. Byzantine syntax (based on shell commands) 2. Awful for debugging

shellcheck网址

1.Commandline tool available 2.Run it on everything you ever write 3.shellcheck is great

写脚本：#! 开头，脚本路径+参数(arguments)

For portable POSIX shellscripts `#! /bin/sh/` : 可移植
For less portable BASH scripts
`#! /usr/bin/env bash` 不可移植的bash脚本

```
chmod +x my-script.sh # 给执行权限
./my-script.sh # 运行脚本 OR
sh my-script.sh # 不想让他executable可以用这个
```

各个分区：beginning: `/bin` was reserved for just system programs

`/usr/bin` for admin installed programs

`/usr/local/bin` for locally installed programs

`/opt/bin` for optional installed programs

`/opt/local/bin` for optional locally installed programs

`~/.local/bin` for a users programs

因为太乱了，So, sticking everything in `/bin` , others stuck them in `/usr/bin` but symlinked them to `/bin` -用链接练到bin里面

env - What env does is look through the PATH and tries to find the program specified and runs it（查找path并尝试找到指定的程序并且运行它**) -** set and print environment

Path - environment variable called PATH that tells the system where all the programs are.

Alter it: `export PATH="${PATH}:/extra/directory/to/search”`

Basic Syntax: `A; B` run A then run B

`A | B` run A and feed its output as the input to B(pipes)

`A && B` run A and if successful run B

`A || B` run A and if not successful run B

How to know **successful? — return 1 byte exit value**

This gets stored into the **variable ${?}** after every command runs.

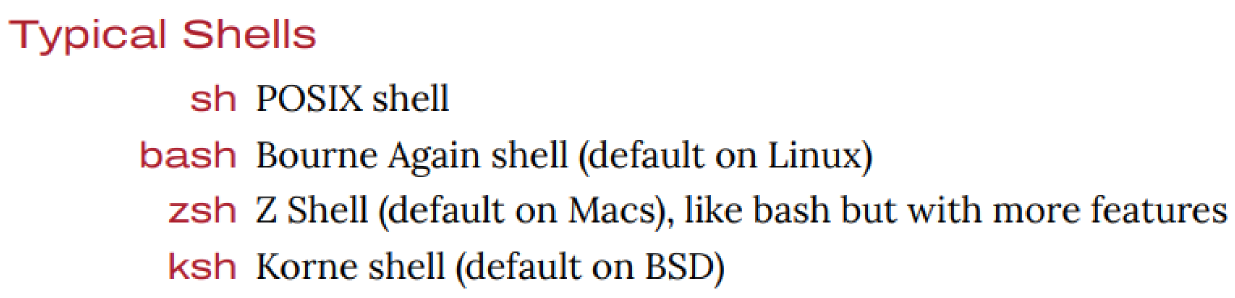
0 indicates **success** (usually) ; **>0** indicates **failure** (usually)

summary: Include a `#!` ;Always use **env**; `$?` contains the exit code

Why is this the case? `[ $? -eq 0 ]` # works, but `[$? -eq 0]` # doesn't work

在shell脚本中，方括号`[ ]`周围的空格很重要。方括号实际上是一个**命令**，括号内的表达式是该命令的**参数**。因此，当你写`[ $? -eq 0 ]`时，实际上是运行了一个名为`[`的命令，

带有参数`$?`、`-eq`和`0`。当去掉空格，写成`[$? -eq 0]`时，它不起作用，因为shell将其解释为一个**单一的标记**，没有任何命令，因此无法识别该怎么处理它。因此，在shell脚本中，一定要记得在方括号`[ ]`周围包含空格。



Typical Shells

sh	POSIX shell
bash	Bourne Again shell (default on Linux)
zsh	Z Shell (default on Macs), like bash but with more features
ksh	Korne shell (default on BSD)

Variables： create a variable: `GREETING="Hello World!”` (不能有空格); use a variable: `echo "${GREETING}”` want variable exist in the programs→ start as: `export GREETING` ;get rid of a variable: `unset GREETING` | No penalty for using one undefined.

Standard variables: `${0}` Name of the script; `${1}, ${2}, ${3}…` Arguments passed to your script; `${#}` The number of arguments passed to your script; `${@}` and `${*}` All the arguments.

Control flow: dd-得自己试着打一下，看不懂思密达

Basename and Dirname: `$(basename "${shell}")`

Pipelines: How many processes is Firefox using? `ps -A | grep -i firefox`

`awk` command - 竖向剪切 留下特定列`ps -A | grep -i firefox | awk '{print $1, $5}’`

drop the last line: `ps -A | grep -i firefox | awk '{print $1, $5}' | ghead -n -1`

各种各样的piping都是啥：

The `|` pipe copies standard output to standard input...
The
`>` pipe copies standard output to a named file... (e.g. ps -A >processes.txt, see also the
tee command)
The
`>>` pipe appends standard output to a named file...
The
`<` pipe reads a file into standard input... (e.g. grep firefox <processes.txt)

The
`<<<` pipe takes a string and places it on standard input
You can even copy and merge streams if you know their file descriptors (e.g. appending

`2>&1` to a command will run it with standard error merged into standard output)

Slides- Build Tools

in order to automate → **Make**

BSD Make(developing BSD) → old fashioned, POSIX;

GNU Make(others using it(More fashion)) → featureful, default on Linux

make arguments(in Makefile, then it will run that commands in Makefile)

当Makefile Changing, makefile is smart to rebuild **every** rule line containing **changed** program in file.就算只make了其中一个also changing。

make what to do when then run: Phony targets(often includes all; clean; install)

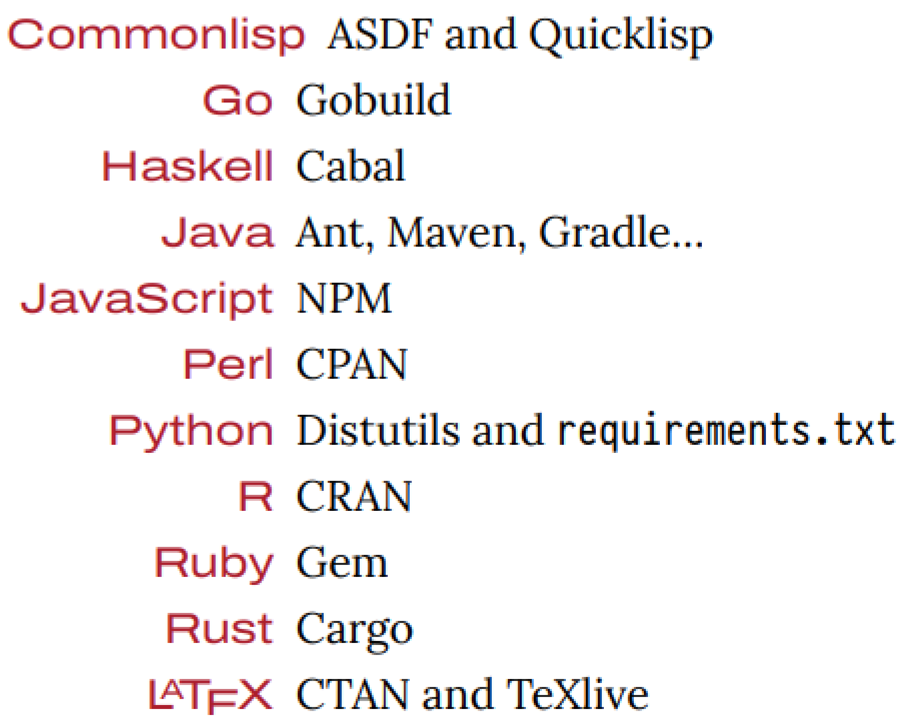
`all` typically first rule in a file (or marked .default): depends on everything you'd like to build /\ `.PHONY: all clean` makefile里的一行,用作标记phony,not find file but 视为目标

additional: 同时，`all` 规则应该依赖于构建项目中的所有其他目标，以确保在运行 `make all` 命令时，能够构建整个项目中的所有内容。

Pattern rules: better generalize!

```
.PHONY: all clean #虚拟目标（不会寻找同名all或clean文件）
figures=$(patsubst .dot,.pdf,$(wildcard *.dot)) #内置参数后缀dot-
all: hello coursework.zip ${figures}
clean:
git clean -dfx
hello: hello.c library.o extra-library.o
%.zip: %
zip -r $@ $< # $@ :要生成的的目标文件名 and $<：规则的第一个依赖项：源
%.pdf: %.dot
dot -Tpdf $< -O $@ # %-> 任意符号
```

Modern build tooling : language + library management tooling→ needs specify dependencies + tell compiler how to rebuild the project



Commonlisp	ASDF and Quicklisp
Go	Gobuild
Haskell	Cabal
Java	Ant, Maven, Gradle...
JavaScript	NPM
Perl	CPAN
Python	Distutils and `requirements.txt`
R	CRAN
Ruby	Gem
Rust	Cargo
LATEX	CTAN and TeXlive

example: pom.xml → 重要

some useful commands: `mvn test` run the test suite

`mvn install` install the JAR into your local JAR packages

`mvn clean` delete everything

EXERCISES

Create a user: `sudo adduser NAME`

Change the user: `sudo USERNAME`

SETUID/GID: `s` 在文件权限里

complier helper exercise（生成代码但实现不了在输入./b hello.c的情况下compile并且run）

```
#!/bin/sh

# Function to compile a C file

compile(){
  src_file="$1"
  if [ ! -f "$src_file" ]; then
    src_file="${src_file%.c}.c" # Append .c if not provided
    if [ ! -f "$src_file" ]; then
      echo "Error: Source file '$1' not found." >&2
      return 1
    fi
  fi

  if gcc -Wall -std=c99 -g "$src_file" -o "${src_file%.c}"; ther
    echo "Compilation successful."
    return 0
  else
    echo "Error: Compilation failed." >&2
    return 1
  fi
}

# Function to run a compiled program
run() {
  program="$1"
  if [ "${program: -2}" = ".c" ]; then
    compile "$program"
    if [ $? -eq 0 ]; then
      ./"${program%.c}"
    else
      echo "Error: Compilation failed for '$program'." >&2
      return 1
    fi
  elif [ -f "$program" ]; then
```

```
    if [ ! -x "$program" ]; then
      echo "Error: Program '$program' not found or not executabl
      return 1
    fi
    ./"$program"
  else
    echo "Error: Program '$program' not found." >&2
    return 1
  fi
}

#Main script logic

if [ $# -eq 0 ]; then
  echo "Usage: $0 [compile|run|build] [filename]" >&2
  exit 1
fi

case "$1" in
 compile)
    shift
    if [ $# -eq 0 ]; then
      echo "Error: Missing filename." >&2
      exit 1
    fi
    compile "$1"
    ;;
  run)
    shift
    if [ $# -eq 0 ]; then
      echo "Error: Missing program name" >&2
      exit 1
    fi
    run "${1%.c}"
    ;;
  build)
```

```
    shift
    if [ $# -eq 0 ]; then
      echo "Error: Missing filename." >&2
      exit 1
    fi
    if compile "$1"; then
      run "${2%.c}"
    fi
    ;;
  *)
    echo "Usage: $1 [compile|run|build] [filename]" >&2
    exit 1
    ;;
esac

exit 0
```

Strict Model: example: `-Werror` in c → all warnings as errors

top of the shell scripts: `set -euo pipefail`

`set` is a shell **internal** command that sets shell flags which controls how commands are run. — shell 内部命令，用于设置控制命令运行方式的 shell 标志。

`set -e` : on the top. command success →return 0; any fail command → stop running

similar to command `|| exit $?` on the end of every command

`set -u` - referencing an undefined variable is an error (原本引用非空variable不会报错)

`set -o pipefail` - changing how pipes works. normal: very last command in pipe.

`pipefail` option: any command in the pipeline fails, return that command's exit code.

默认：command1|command2|command3 , 1,3成功，2失败，管道返回3(看起来成功)

pipefail启动：相同command，1，3成功，2失败，管道立即失败，返回2的退出状态（任意一个失败就退出程序并且返回失败command状态作为整个管道的退出状态。

notes about `set -u`: 如果您编写类似 `rm -rf $FOLDER/` 和 `$FOLDER` 未设置的内容，那么您不会意外地删除整个系统！当然，大多数实现将拒绝在没有该选项的情况下 `rm` 删除.

Build tools：C

指令：`tar -zxvf FILENAME` ：解压缩一个以 gzip 压缩的 tar 文件（.tar.gz 或 .tgz 格式）

- `z`: 表示使用 gzip 解压缩，`tar` 会调用 `gzip` 进行解压缩。
- `x`: 表示解压缩操作。
- `v`: 表示详细模式（verbose），将显示解压缩过程中的详细信息。
- `f FILENAME`: 表示指定要解压缩的文件为 FILENAME

Briefly, the shell commands `./configure; make; make install` should configure, build, and install this package.

make: `sudo apt install make`
configureation variables(everything passed with a
-D)

- 一些是功能开关
- 特定操作系统和编译器的选项 e.g. `-DHAVE_STRING_H` 定义 `string.h` 是否存在在系统中

```
#if HAVE_READLINXE 可以转换#ifdef命令到command命令，例如：
include <readline/readline.h>
include <readline/history.h>
#endif
# 会被转换为：
gcc [lots of options] -g -O2 -o sqlite3
# 一行打不下，接上
sqlite3-shell.o sqlite3-sqlite3.o -lreadline -lcurses
# 命令可以构建一个可以run的sqlite3（.q quit出这个语句）
```

- 如果不能找到.h文件且连接不到library file咋办？→ apt-file `apt-file search <name of file>` → 找出哪个包提供缺少的文件

`libffi:so` : can let me know what **package** might have **provided** it?

Build tools： Python

安装pip：用pip进行安装 ： `pip3 install --user moduleName`

避免: `sudo pip install` 用↑ 这种方法将软件包安装到主目录`~/.local` 中file

↑ 会安装到`/usr` 需要root权限的文件夹

`scipy` → `apt search scipy` 搜索已有版本

venv(虚拟环境): Create a **virtual python install** that is owned by a user

- `pip freeze | tee requirements.txt` : **list** all the packages your using and what **version** they are and **save them in a file called requirment.txt**

在这之后，代码`pip install -r requirements.txt`就可以再次安装.Easy知道所有正确依赖

Build tools： JAVA maven — java包管理和构建工具

- 编译器`javac`将源文件（`.java`）变成`.class`文件；
- 该`jar.jar`工具将class文件打包成文件；
- 该`java`命令运行类文件或 jar 文件

有多个包时，设置`JAVA_HOME`和`PATH`变量指向安装。例子：

二进制文件(binaries folder)：`PATH` ；解压jdk的文件夹：`JAVA_HOME`

```
export JAVA_HOME='/usr/lib/jvm/java-17-openjdk'
export PATH="${PATH}:${JAVA_HOME}/bin"
```

Running maven:

`mvn archetype:generate` : *generate an artifact from an archetype.*→ maven-speak会创建一个有maven文件的新文件夹.(no found error: 路径不对)

一个项目会被输入三元组：groupId, artifactId, version。Maven 创建了一个以你的 artifactId 命名的文件夹，但如果你愿意，你可以移动和重命名它，只要你从文件夹内运行它，maven 就不会介意。使用 cd project 或您调用的任何名称进入文件夹。

POSIX shell → `find .` shouldd show everything [windows: `start .`

```
.
./src
./src/main
./src/main/java
./src/main/java/org
./src/main/java/org/example
./src/main/java/org/example/App.java
./src/test
./src/test/java
./src/test/java/org
./src/test/java/org/example
./src/test/java/org/example/AppTest.java
./pom.xml
```

In a `standard` maven folder structure. Your java sources live under `src/main/java`, and the default package name is `org.example` or whatever you put as your groupId so the main file is currently `src/main/java/org/example/App.java`

POM file:

- **artifact's identifier**(group id, artifact id, version)
- **build properties:** 确定要编译的java版本 1.8
- **dependencies section：** 添加要使用库的位置;声明`<scope>test</scope>`它仅用于测试，而不是项目本身。声明项目真正依赖不能有这行。
- `<plugins>` **section:** maven 用于编译和构建项目的插件, in order to lock 特定版本

Add here is the `exec-maven-plugin` as follows, so that you can actually run your project:
第二行：

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>exec-maven-plugin</artifactId>
    <version>3.0.0</version>
    <configuration>
        <mainClass>org.example.App</mainClass>   # important li
    # set full name(with path components) of class with the mai
    </configuration>
</plugin>
```

Compile, run and develop:

`mvn compile`：编译该项目 `mvn clean`：删除所有编译文件

`mvn exec:java`：设置插件之后，通过它运行编译后的项目

flow: 进行编辑，然后运行`mvn compile test exec:java`重新编译，运行测试，然后运行程序

`mvn test`运行测试`src/test/java`；`mvn package`在文件夹中创建项目的 jar 文件`target/`

Build tools: Spring

Web applications listen to a port (normally TCP port 80 for HTTP, 443 for HTTPS in production; 8000 or 8080 while in development).