Shellscripting and Buildtools

Joseph Hallett

February 5, 2024

Whats all this about?

I've written a *lot* of code over the years:

Assembly, C and Java as an engineer

Commonlisp for my own projects

Haskell to build compilers

PostScript to draw really efficient diagrams

LaTeX to publish books

...several a dozen other things too

Which language have I written the most code in?

Which language do I use to solve most tasks?

Which language do I like the least?

Shellscripting!

Normally we type commands for the terminal on a commandline...

- But we can automate them and stick them into scripts

Anything you have to do more than once...

Write a script for it!

- Saves a tonne of time
- Often easier than writing a full program

For example...

${GREP} -Pi "^${1}$" /usr/share/dict/words：使用之前设定的 grep 命令（标准或GNU版本）来在文件 /usr/share/dict/words 中搜索一个指定的单词。这个文件通常包含系统的英文单词列表。
-Pi：-P 选项启用 Perl 正则表达式，-i 选项使搜索不区分大小写。
"^${1}$"：这是一个正则表达式，用来匹配完整的行。^ 表示行的开始，${1} 是脚本的第一个参数（用户提供的单词），$ 表示行的结束。整个表达式的意思是精确匹配整行为第一个参数提供的单词。
/usr/share/dict/words：这是被搜索的文件路径，通常包含一个标准的单词列表。

```
#! /bin/sh
# Solve
GREP=grep
if [ $(uname) = "OpenBSD" ]; then
   # Use GNU Grep on OpenBSD
   GREP=ggrep
fi

${GREP} -Pi "^${1}$" /usr/share/dict/words
```

```
#! /usr/bin/env bash
# Solve Puzzmo's wordbind

DICT=/usr/share/dict/words
REGEX="$(sed -e "s/[ \t]//g" <<<"${*}" \
       | sed -e "s/\(.\)/\1*/g")"
<"${DICT}" awk <<EOS
/^${REGEX}\$/ {
    printf(\"%d\t%s\n\", length(\$0), \$0)
}
EOS \
| sort -n
```

```
~/.local/bin/knotwords 'st[^aeo]pid'
```

- stupid

```
~/.local/bin/wordbind superlite striae
```

- 7 sullies
- 7 suppers
- 7 suppler
- 7 surlier
- 8 peerless
- 8 supplest
- 8 supplier
- 8 supplies
- 8 suppress
- 8 surliest

使用 sed 命令将输入的参数中的空格和制表符都去除，然后使用另一个 sed 命令将每个字符替换为一个字符和 *，生成了一个正则表达式 REGEX。

使用 awk 命令从字典文件中读取内容，并针对每一行进行匹配。

Or for example...

首先检查传递给脚本的第一个、第二个和第三个参数是否分别等于 "should"、"also" 和 "run"。如果是，则执行以下操作：
移除已经检查过的前三个参数。
使用 gum confirm "Run 'doas $*'" 提示用户是否运行命令，如果用户确认，则执行 doas $* 命令。
如果第一个、第二个和第三个参数分别等于 "should"、"also" 和 "remove"，则执行以下操作：
提示用户确认是否删除第四个参数指定的文件或目录。
如果用户确认，则执行 doas rm -fr "${4}" 命令来删除该文件或目录。
如果以上条件都不满足，则输出警告信息，提醒用户更仔细地阅读粘贴的命令。

```
#! /usr/bin/env bash
if [ $1 = "should" -a $2 = "also" -a $3 = "run" ]; then
  shift 3
  gum confirm "Run 'doas $*'?" && doas $*
elif [ $1 = "should" -a $2 = "also" -a $3 = "remove" ]; then
  gum confirm "Delete '$4'?" && doas rm -fr "${4}"
else
  2>&1 printf "[WARNING] You should read the commands you"
  2>&1 printf "paste more carefully\n"
fi
```

Sometimes when I upgrade my computer it tells me to delete some files or run some commands:

`You should also run rcctl restart pf`

Copying and pasting the precise text is a pain...

- Can I just copy the whole line and run that?

(Of course I can... should I though?)

-a 是 if 语句中的一个条件测试操作符，用于表示 "and"，即逻辑与。在这个脚本中，[$1 = "should" -a $2 = "also" -a $3 = "run"] 表示当 $1 等于 "should" 且 $2 等于 "also" 且 $3 等于 "run" 时条件成立。

2>&1 是一种重定向操作符，它将标准错误（文件描述符2）重定向到标准输出（文件描述符1）。在这个脚本中，2>&1 将标准错误流（stderr）与标准输出流（stdout）合并，这样产生的输出就会同时包含标准输出和标准错误的内容。

Or for a further example...

```
#! /usr/bin/env bash
# Fix kitty
/usr/local/opt/bin/fix-kitty

# Update sources
cd /usr/src && cvs -q up -Pd -A
cd /usr/ports && cvs -q up -Pd -A
cd /usr/xenocara && cvs -q up -Pd -A
```

After I upgrade my computer I need to run a couple of standard commands.

- I can never remember them
- Batch them up!

So whats this really about?

Shellscripting is about automating all those tedious little jobs

- Byzantine syntax (based on shell commands)
- Awful for debugging
- Requires magical knowledge
- Probably the most useful thing you'll ever learn

Luckily we have help

Shell scripting is somewhat magical, and there are *lots* of gotchas...

https://www.shellcheck.net

Wonderful tool to spot unportable/dangerous things in shell scripts

- Commandline tool available
- Run it on *everything* you ever write
- `shellcheck` is great

```
shellcheck `command -v knotwords`
In /home/joseph/.local/bin/knotwords line 2:
GREP=grep
^--^ SC2209 (warning): Use var=$(command) to assign output (or quote to assign string).


In /home/joseph/.local/bin/knotwords line 3:
if [ $(uname) = "OpenBSD" ]; then
    ^------^ SC2046 (warning): Quote this to prevent word splitting.

For more information:
  https://www.shellcheck.net/wiki/SC2046 -- Quote this to prevent word splitt...
  https://www.shellcheck.net/wiki/SC2209 -- Use var=$(command) to assign outp...
```

So how do you write one?

Start the file with the *shebang* `#!` then the path to the interpreter of the script plus any arguments:

For portable POSIX shellscripts `#! /bin/sh/`

For less portable BASH scripts `#! /usr/bin/env bash`

Then

- `chmod +x my-script.sh`
- `./my-script.sh`

chmod +x 和 chmod a+x 一样

The rest of the file will be run by the interpretter you specified

- or `sh my-script.sh` if you don't want to/can't mark it executable.

(Hey this is also why Python scripts start `#! /usr/bin/env python3`)

#!/bin/sh：这行指定了使用 /bin/sh 作为脚本的解释器。/bin/sh 是 POSIX 标准所要求的基本 shell 解释器，因此这种写法适用于所有支持 POSIX 标准的系统，可以保证脚本在大多数 Unix-like 系统上运行。

#!/usr/bin/env bash：这行通过 env 命令来查找系统中可用的 bash 解释器的路径。这种写法更具可移植性，因为不同系统中 bash 可能安装在不同的路径下。这种写法通常用于脚本需要使用 bash 特性的情况下，但并不要求遵循 POSIX 标准的环境。

Why env?

Hang on, you might be saying, I know that bash is always in `/bin/bash`... can I just put that as my interpretter path?

Yes, but...

In the beginning `/bin` was reserved for just *system* programs

- and `/usr/bin` for admin installed programs
- and `/usr/local/bin` for locally installed programs
- and `/opt/bin` for optional installed programs
- and `/opt/local/bin` for optional locally installed programs
- and `~/.local/bin` for a *users* programs
- ...oh and sometimes they're even mounted on different disks!

This is *kinda* madness.

- So *most* Linux systems said look we'll just stick everything in `/usr/bin` and stop using multiple partitions
- But some said no it should be `/bin`, one said `/Applications/`, and others stuck them in `/usr/bin` but symlinked them to `/bin`
- And on some systems users grew fed up of the outdated system `bash` and compiled their own and installed it in `~/.local/bin` ...
- ...and ever tried using Python `venv`?

env

```
ENV(1)                    General Commands Manual                    ENV(1)

NAME
     env – set and print environment

SYNOPSIS
     env [-i] [name=value ...] [utility [argument ...]]

DESCRIPTION
     env executes utility after modifying the environment as specified on the
     command line.  The option name=value specifies an environment variable,
     name, with a value of value.
```

What env does is look through the PATH and tries to find the program specified and runs it.

...Path?

<< (Here String)允许你提供一个输入区块（多行文本），直到遇到一个预定义的结束标记。

<<< (Here Document)允许你提供单行字符串作为输入。

There is an environment variable called `PATH` that tells the system where all the programs are:

- Colon separated list of paths

If you want to alter it you can add a line like to your shell's config

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

Your shells config is *possibly* in `~/.profile` but it often varies... check the `man` page for your `${SHELL}`
Also some shells have different syntax (e.g. `fish`)...

```
$ tr ':' $'\n' <<< $PATH
/home/joseph/.local/share/python/bin
/bin
/usr/bin
/sbin
/usr/sbin
/usr/X11R6/bin
/usr/local/bin
/usr/local/sbin
/home/joseph/.local/bin
/usr/local/opt/bin
/usr/games
/usr/local/games
/usr/local/jdk-17/bin
/home/joseph/.local/share/go/bin
```

Basic Syntax

Shell scripts are written by chaining commands together

- `A ; B` run A then run B
- `A | B` run A and feed its output as the input to B
- `A && B` run A and if successful run B
- `A || B` run A and if not successful run B

How does it know if its successful?

Programs *return* a 1 byte exit value (e.g. C main ends with `return 0;`)

- This gets stored into the variable `${?}` after every command runs.
- `0` indicates success (usually)
- `>0` indicates failure (usually)

This can then be used with commands like `test`:

```
do_long_running_command
test $? -eq 0 && printf "Command succeeded\n"
```

Or the slightly shorter:

```
do_long_running_command
[ $? -eq 0 ] && printf "Command succeeded\n"
```

`test $? -eq 0`：

`$?`：这是一个特殊的变量，用于存储上一个执行的命令的退出状态码。在 Unix 和 Linux 系统中，命令成功执行通常返回 0，而失败则返回非零值（具体值取决于失败的类型）。

`test`：这是一个用于检查条件并返回真（0）或假（非0）的命令。在这个上下文中，它用于测试 `$?`（即上一个命令的退出状态码）是否等于0。

`-eq`：这是一个比较操作符，意为“等于（equal）”。

`0`：这里指的是期望的退出状态码，表示成功。

Bonus puzzle

在Bash脚本中，[是一个命令，也称为test。因此，当你写[$? -eq 0]时，你实际上是在调用test命令，其参数为$?、-eq和0。该命令检查最后一个命令的退出状态（$?）是否等于0。

然而，当你写[$? -eq 0]时，Bash将[和]视为普通字符，而不是shell的特殊字符。在这种情况下，Bash尝试执行一个名为[$? -eq 0]的命令，这个命令是不存在的，导致出现错误。

Why is this the case?

```
[ $? -eq 0 ] # works
[$? -eq 0] # doesn't work
```

Variables

All programs have variables... Shell languages are no different:

To create a variable:

```
GREETING="Hello World!"
```

(No spaces around the =)

To use a variable

```
echo "${GREETING}"
```

If you want your variable to exist in the programs you start as an *environment variable*:

```
export GREETING
```

To get rid of a variable

```
unset GREETING
```

Well...

Variables in shell languages *tend* to act more like macro variables.

- There's no penalty for using one thats not defined.

```
NAME='Joe'
unset NAME
echo "Hello, '${NAME}'"
```

```
Hello, ''
```

If this bothers you:

```
set -o nounset
echo "${NAME:? variable 1 passed to program}"
```

(There are a *bunch* of these shell parameter expansion tricks beyond `:?` which can do search and replace, string length and various magic...)

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

If statements and *for* loops, with *globbing*, are available:

```
# Or [ -x myscript.sh ];
# Or [[ -x myscript.sh ]]; if using Bash
if test -x myscript.sh; then
   ./myscript.sh
fi

for file in *.py; do
   python "${file}"
done
```

Other loops

Well...okay you only have `for` really... but you can do other things with it:

```
for n in 1 2 3 4 5; do
   echo -n "${n} "
done
```

1 2 3 4 5

```
seq 5
```

1 2 3 4 5

```
for n in $(seq 5); do
   echo -n "${n} "
done
```

1 2 3 4 5

```
seq -s, 5
```

1,2,3,4,5

```
# IFS = In Field Separator
IFS=','
for n in $(seq -s, 5); do
   echo -n "${n} "
done
```

1 2 3 4 5

Case statements too!

```
# Remove everything upto the last / from ${SHELL}
case "${SHELL##*/}" in
   bash) echo "I'm using bash!" ;;
   zsh) echo "Ooh fancy a zsh user!" ;;
   fish) echo "Something's fishy!" ;;
   *)   echo "Ooh something else!" ;;
esac
```

```
case "${SHELL##*/}" in
```

这一行开始一个 case 语句，用于根据 ${SHELL##*/} 的值来执行不同的代码块。${SHELL} 是一个环境变量，包含当前用户登录 shell 的完整路径，例如 /bin/bash。
${SHELL##*/} 是一种参数扩展，用来从 ${SHELL} 中删除最后一个 / 之前的所有内容，因此只保留 shell 名称（例如，如果 ${SHELL} 是 /bin/bash，则 ${SHELL##*/} 结果为 bash）。

Basename and Dirname

In the previous example I used the "${VAR##*/}" trick to remove everything up to the last /...
Which gives you the name of the file neatly...
...but I have to look this up everytime I use it.
Instead we can use $(basename "${shell}") to get the same info.

```
echo "${SHELL}"
echo "${SHELL##*/}"
echo "$(basename "${SHELL}")"
echo "$(dirname "${SHELL}")"
```

basename 命令用于从完整的路径中提取文件名（或目录名），在这里它用于提取 shell 的名称。
"$(...)" 是命令替换的语法，它执行括号内的命令，并将输出结果替换到当前位置。
如果 ${SHELL} 是 /bin/bash，basename "${SHELL}" 的结果也是 bash。这一行命令通过调用外部命令 basename 来获取并打印 shell 的名称

You can even use it to remove *file extensions*:

```
for f in *.jpg; do
  convert "${f}" "$(basename "${f}" .jpg).png"
done
```

dirname 命令用于从完整的路径中提取目录路径，忽略最后的文件名（或目录名）部分。
如果 ${SHELL} 是 /bin/bash，dirname "${SHELL}" 的结果是 /bin。这一行命令用于获取并打印当前 shell 所在的目录路径。

convert 命令是 ImageMagick 工具套件的一部分，用于转换图像格式、大小调整、裁剪、应用各种效果等。在这里，它用于将 JPG 文件转换为 PNG 文件。
"${f}" 是当前循环迭代中被匹配到的 .jpg 文件名。
$(basename "${f}" .jpg) 使用 basename 命令从文件名 f 中去除路径（如果有）和 .jpg 扩展名，只留下基本的文件名。然后，通过添加 .png 扩展名，将输出文件的格式指定为 PNG。

Pipelines

As part of shell scripting, its often useful to build commands out of chains of other commands. For example I can use `ps` to list all the processes on my computer and `grep` to search.

- How many processes is *Firefox* using?

```
ps -A | grep -i firefox
```

```
44179   ??   SpU   0:29.59   firefox
60731   ??   Ip    0:00.08   /usr/local/lib/firefox/firefox   -contentproc   -appDir
57651   ??   IpU   0:00.30   /usr/local/lib/firefox/firefox   -contentproc   {e3aaae0
78402   ??   SpU   0:08.66   /usr/local/lib/firefox/firefox   -contentproc   {1ddabe3
53121   ??   SpU   0:01.79   /usr/local/lib/firefox/firefox   -contentproc   {5f676d2
79118   ??   IpU   0:00.21   /usr/local/lib/firefox/firefox   -contentproc   {40690c1
38067   ??   IpU   0:00.20   /usr/local/lib/firefox/firefox   -contentproc   {6be551d
33456   ??   IpU   0:00.20   /usr/local/lib/firefox/firefox   -contentproc   {8c295ac
82061   ??   R/3   0:00.00   grep                             -i             firefox
```

Too much info!

Lets use the awk command to cut it to just the first and fourth columns!

```
ps -A | grep -i firefox | awk '{print $1, $4}'
```

```
                                44179   firefox
                                60731   /usr/local/lib/firefox/firefox
                                57651   /usr/local/lib/firefox/firefox
                                78402   /usr/local/lib/firefox/firefox
                                53121   /usr/local/lib/firefox/firefox
                                79118   /usr/local/lib/firefox/firefox
                                38067   /usr/local/lib/firefox/firefox
                                33456   /usr/local/lib/firefox/firefox
                                24087   grep
```

ps -A
ps 命令用于显示当前系统的进程状态。
-A 选项告诉 ps 显示所有的进程，不仅仅是与当前终端相关的进程。

| awk '{print $1, $4}'
再次使用管道操作符将 grep 命令的输出传递给 awk 命令。
awk 是一种编程语言，常用于文本和数据的提取和报告。
'{print $1, $4}' 是 awk 的程序部分，指示 awk 打印每行的第一个和第四个字段。在 ps 命令的上下文中，通常第一个字段是进程 ID（PID），而第四个字段可能是命令名或其他信息

Why is grep in there?

Oh yes... when we search for *firefox* we create a new process with *firefox* in its commandline. Lets drop the last line

```
ps -A | grep -i firefox | awk '{print $1, $4}' | head -n -1
```

```
44179   firefox
60731   /usr/local/lib/firefox/firefox
57651   /usr/local/lib/firefox/firefox
78402   /usr/local/lib/firefox/firefox
53121   /usr/local/lib/firefox/firefox
35192   /usr/local/lib/firefox/firefox
46680   /usr/local/lib/firefox/firefox
 9850   /usr/local/lib/firefox/firefox
40081   /usr/local/lib/firefox/firefox
44225   /usr/local/lib/firefox/firefox
 3307   /usr/local/lib/firefox/firefox
```

And really I'd just like a count of the number of processes

```
ps -A | grep -i firefox | awk '{print $1, $4}' | head -n -1 | wc -l
```

11

Other piping techniques

<(是连在一起的

- 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*)

- The `<(echo hello)` pipe is completely magical...
  - It runs the command in the parentheses outputting to a temporary file (descriptor)...
  - It replaces itself with the path to that file

So you can do things like:

```
diff <(echo Hello World) \
    <(echo Hello World | tr r R)
```

```
1c1
<     Hello   World
---
>     Hello   WoRld
```

command | tee file1.txt
这个例子中，command 的输出会被 tee 读取，然后 tee 会做两件事：

将输出显示在终端（或其他标准输出设备）上。
将相同的输出写入 file1.txt。

Different shells

(Just use bash unless you care about *extreme* portability in which case use POSIX sh)

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)

Other shells

dash	simplified faster bash, used for booting on Linux
Busybox sh	simplified bash you find on embedded systems

Weird shells

fish	More usable shell (but different incompatible syntax)
elvish	Nicer syntax for scripting (but incompatible with POSIX)
nushell	Nicer output (but incompatible, and weird)

One last thing...

Suppose you want to run a shellscript as a network service.

- *Normally* you'd have to run a webserver and do some socket programming
- None of which is ever usually available for shell scripts

Inetd

`inetd` is *super server* that can launch any program when someone connects to a socket

- `stdin` will be the input from the network
- `stdout` will be sent back on the same socket

Check out `fingerd` for an ancient social network built like this...

Suppose we wanted to build some code

An *awful* lot of the things we do with a computer are about *format shifting*
We do this when we compile code:

- `cc -c library.c -o library.o`
- `cc hello.c library.o -o hello`

When we archive files:

- `zip -r coursework.zip coursework`

When we draw figures:

- `dot -Tpdf flowchart.dot -O flowchart.pdf`

Can we automate this?

YES!

We *could* write a shellscript and stick all the tasks in one place...

```
#! /usr/bin/env bash
cc -c library.c -o library.o
cc hello.c library.o -o hello
zip -r coursework.zip coursework
dot -Tpdf flowchart.dot -O flowchart.pdf
```

But can we do better than this?

- Do we really need to recompile the C program if only our flowchart has changed?
- Can we generalise build patterns?

Make

Make in an *ancient* tool for automating builds.

- Developed by *Stuart Feldman* in 1976
- Takes *rules* which tell you how to build files
- Then follows them to build the things you need!

Two main dialects of it (nowadays):

BSD Make More old fashioned, POSIX

GNU Make More featureful, default on Linux

In practice, unless your developing a BSD *every one* uses GNU Make

- If you're on a Mac, or BSD box install GNU Make and try `gmake` if things don't work

Makefiles

Rules for Make are placed into a *Makefile* and look like the following:

```
hello: hello.c library.o
	cc -o hello hello.c library.o

library.o: library.c
	cc -c -o library.o library.c

coursework.zip: coursework
	zip -r coursework.zip coursework

flowchart.pdf: flowchart.dot
	dot -Tpdf flowchart.dot -O flowchart.pdf
```

这条规则的意思是，如果 hello.c 或 library.o 有任何更改，make 将使用 cc（C 编译器）来编译和链接这两个文件，生成可执行文件 hello。这里，cc 是C语言编译器的常见别名，-o 选项用于指定输出文件的名称。

这条规则说明当 library.c 文件发生变化时，make 会执行 cc 命令来编译这个文件，但不进行链接（由 -c 选项指定），生成目标文件 library.o。

If you ask make to build `hello` it will figure out what it needs to do:

```
$ make hello
cc -c -o library.o library.c
cc -o hello hello.c library.o
```

First line specifies how to build *what* from *which* source files

- The rest of the **TAB** indented block is a shellscript (ish)

Making changes

If you alter files... Make is smart enough to only rerun the steps you need:
For example if you edit `hello.c` and rebuild:

```
$ make hello
cc -o hello hello.c library.o
```

But if you edit `library.c` it can figure out it needs to rebuild *everything*

```
$ make hello
cc -c -o library.o library.c
cc -o hello hello.c library.o
```

Phony targets

As well as rules for how to make files you can have *phony* targets that don't depend on files but just tell make what to do when they're run

Often a `Makefile` will include a phony:

- `all` typically first rule in a file (or marked `.default`): depends on everything you'd like to build
- `clean` deletes all generated files
- `install` installs the program

```
$ make
cc -c -o library.o library.c
cc -o hello.o hello.c library.o
zip -r coursework.zip coursework
dot -Tpdf flowchart.dot -O flowchart.pdf
```

```
.PHONY: all clean

all: hello coursework.zip flowchart.pdf

clean:
	git clean -dfx

hello: hello.c library.o
	cc -o hello hello.c library.o

library.o: library.c
	cc -c -o library.o library.c

coursework.zip: coursework
	zip -r coursework.zip coursework

flowchart.pdf: flowchart.dot
	dot -Tpdf flowchart.dot -O flowchart.pdf
```

Pattern rules

(So far, everything *should* have worked in GNU and BSD Make... here on out we're in GNU land)
What if we wanted to add an extra library to our `hello` programs? We could go and update the `Makefile` but its better to generalise!

```
CC=clang
CFLAGS=-Wall -O3

.PHONY: all clean

all: hello coursework.zip flowchart.pdf
clean:
	git clean -dfx

hello: hello.c library.o extra-library.o

%.o: %.c
	$(CC) $(CFLAGS) -c -o $@ $<

%: %.c
	$(CC) $(CFLAGS) -o $@ $<

%.zip: %
	zip -r $@ $<

%.pdf: %.dot
	dot -Tpdf $< -O $@
```

$() 主要用于命令替换，让你可以将一个命令的输出用作另一个命令的输入或赋值给变量。
${} 主要用于参数扩展，它提供了一系列操作变量的方法，包括但不限于替换操作、子字符串提取、默认值等。

所以 此处用 $(CC)

%.o: %.c：表示如何从 .c 文件编译 .o 对象文件。$@ 表示目标文件名，$< 表示第一个依赖文件名。例如，如果有一个 library.c 文件，这个规则会执行 clang -Wall -O3 -c -o library.o library.c 来编译它。

%: %.c：表示如何从 .c 文件直接编译出可执行文件，没有指定的输出文件类型。例如，对于 hello.c，这个规则会执行 clang -Wall -O3 -o hello hello.c。

Implicit pattern rules

Actually because Make is so old, it knows about compiling C (and Fortran/Pascal...) code already:

```
.PHONY: all clean

all: hello coursework.zip flowchart.pdf
clean:
	git clean -dfx

hello: hello.c library.o extra-library.o

%.zip: %
	zip -r $@ $<

%.pdf: %.dot
	dot -Tpdf $< -O $@
```

Lets get even more general!

Suppose we wanted to add more figures... we could add dependencies on `all` to build them or...

```
.PHONY: all clean
figures=$(patsubst .dot,.pdf,$(wildcard *.dot))

all: hello coursework.zip ${figures}
clean:
	git clean -dfx

hello: hello.c library.o extra-library.o

%.zip: %
	zip -r $@ $<

%.pdf: %.dot
	dot -Tpdf $< -O $@
```

wildcard *.dot：wildcard 函数会匹配当前目录下所有以 .dot 结尾的文件，并返回一个以空格分隔的文件列表。例如，如果当前目录下有 example1.dot、example2.dot 和 example3.dot 这三个文件，那么 wildcard *.dot 的结果就是 example1.dot example2.dot example3.dot。

patsubst .dot,.pdf,$(wildcard *.dot)：patsubst 函数会对字符串进行模式匹配和替换。在这个例子中，它会将匹配到的 .dot 文件名替换为 .pdf。假设 wildcard *.dot 返回的结果是 example1.dot example2.dot example3.dot，那么经过 patsubst 函数处理之后，就会得到 example1.pdf example2.pdf example3.pdf。

Make is crazy powerful

I love Make...

- I abuse it for compiling everything
- For distributing reproducible science studies
- For building and deploying websites

Pattern rules and the advanced stuff is neat...

- ...but if you never use it I won't be offended
- Make is one of those tools that you'll come back to ***again and again*** over your careers.
- ...and there's a ***bunch*** of tricks I haven't shown you `;-)`

Go and read the *GNU Make Manual*

- Its pretty good for a technical document

Just type `make`

When you get a bit of software... and you find a `Makefile` in there...
Just type `make`!

- (and make sure your projects build in the same way!)

(Actually often you'll have to type `./configure` then `make`)

- No I'm not going to teach you *autotools* don't worry!

Limitations of Make

I love `Make` but it has one *big* weakness

- Modern development makes extensive use of external libraries…

But *Make* is rubbish at dealing with them:

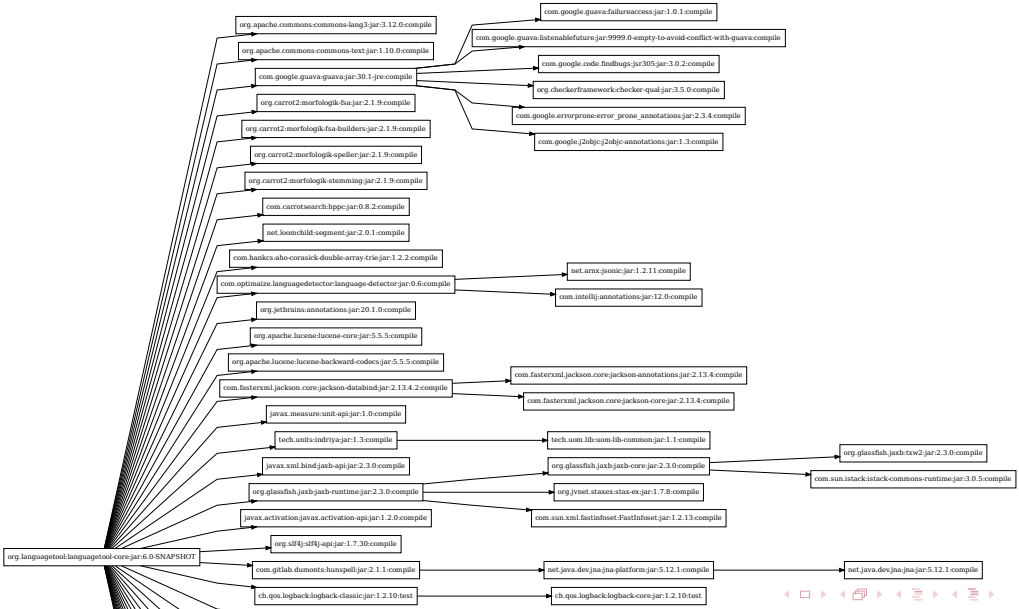
- Doesn't know how to fetch dependencies
- Doesn't track versions beyond *source is newer than object*

LanguageTool is a cool little Java grammar checker:

- How many libraries does *just the core* of the tool make use of?

```
mvn dependency:tree -D outputType=dot | dot -Tpdf
```

This is surely too many?



In the old days...

Traditionally you'd have to go download all the dependencies by hand...

- And then compile and install them
- Very tedious and error prone

So we automated it!

Modern build tooling

(Almost) every language comes with its own library management tooling

- Lets developers specify dependencies
- Tells compiler how to rebuild the project

...which means *for every language you use you need to learn its build tools*...

- Yay?

(Honestly, I still use *Make* but I'm old and cantankerous)

So now we have...

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

...and *many* more.

And they're all different

Very little similarity between *any* of them.

- You need to learn the ones you use.
- We'll play in the labs with *Maven* for Java a little bit

Maven Quickstart

```
mkdir /tmp/src
cd /tmp/src
mvn archetype:generate \
   -DgroupId=uk.ac.bristol.cs \
   -DartifactId=hello \
   -DarchetypeArtifactId=maven-archetype-quickstart \
   -DinteractiveMode=false
```

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------< org.apache.maven:standalone-pom >-------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] --------------------------------[ pom ]---------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:3.2.1:generate (default-cli) > generate-sources @ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:3.2.1:generate (default-cli) < generate-sources @ standalone-pom <<<
[INFO]
[INFO]
[INFO] --- maven-archetype-plugin:3.2.1:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Batch mode
[INFO] ----------------------------------------------------------------------------
[INFO] Using following parameters for creating project from Old (1.x) Archetype: maven-archetype-quickstart:1.0
[INFO] ----------------------------------------------------------------------------
[INFO] Parameter: basedir, Value: /tmp/src
[INFO] Parameter: package, Value: uk.ac.bristol.cs
[INFO] Parameter: groupId, Value: uk.ac.bristol.cs
[INFO] Parameter: artifactId, Value: hello
[INFO] Parameter: packageName, Value: uk.ac.bristol.cs
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] project created from Old (1.x) Archetype in dir: /tmp/src/hello
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  4.256 s
```

...and after spewing all that...

这个命令是使用 find 工具，在 /tmp/src 目录下查找文件，并只显示普通文件（即排除目录和其他特殊类型的文件）。

```
find /tmp/src -type f
```

find: 这是一个用于查找文件的命令。
/tmp/src: 这是查找的起始路径，即查找从这个路径开始。
-type f: 这是 find 命令的选项之一，表示只匹配普通文件。 f 代表普通文件。

- /tmp/src/hello/pom.xml
- /tmp/src/hello/src/main/java/uk/ac/bristol/cs/App.java
- /tmp/src/hello/src/test/java/uk/ac/bristol/cs/AppTest.java
- /tmp/src/hello/target/maven-status/maven-compiler-plugin/compile/default-compile/createdFiles.lst
- /tmp/src/hello/target/maven-status/maven-compiler-plugin/compile/default-compile/inputFiles.lst
- /tmp/src/hello/target/maven-status/maven-compiler-plugin/testCompile/default-testCompile/createdFiles.lst
- /tmp/src/hello/target/maven-status/maven-compiler-plugin/testCompile/default-testCompile/inputFiles.lst
- /tmp/src/hello/target/classes/uk/ac/bristol/cs/App.class
- /tmp/src/hello/target/test-classes/uk/ac/bristol/cs/AppTest.class
- /tmp/src/hello/target/surefire-reports/uk.ac.bristol.cs.AppTest.txt
- /tmp/src/hello/target/surefire-reports/TEST-uk.ac.bristol.cs.AppTest.xml
- /tmp/src/hello/target/maven-archiver/pom.properties
- /tmp/src/hello/target/hello-1.0-SNAPSHOT.jar

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instanc
       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd"
  <modelVersion>4.0.0</modelVersion>
  <groupId>uk.ac.bristol.cs</groupId>
  <artifactId>hello</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>hello</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

And if we try and build...

```
mvn package
```

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------< uk.ac.bristol.cs:hello >------------------------
[INFO] Building hello 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ hello ---
[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory /tmp/src/hello/src/main/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ hello ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ hello ---
[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory /tmp/src/hello/src/test/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ hello ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ hello ---
[INFO] Surefire report directory: /tmp/src/hello/target/surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running uk.ac.bristol.cs.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.018 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ hello ---
```

Other useful commands

`mvn test` run the test suite

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

`mvn clean` delete everything

And if I'm being a bit snarky…

`https://gradle.org` A *better* Java build tool

(That doesn't work everywhere and is much worse than Maven when you try and do more complex things…)

Wrap up

Language specific build tools exist

- You should probably use them
- (but I still use good ol' `make` a lot more)

Makefiles let you shift things between different filetypes

- And avoid rebuilding things that don't need to be rebuilt

Shellscripts let you automate *everything*

- You're gonna end up writing them in anger
- Laziness is good
- Run everything through shellcheck

Aside

Sometimes you'll find you pull a project and it uses a certain build system and you just know you're going to have to spend a day fighting it.
...please don't use CMake.