

make

Marc Chantreux

disclaimer

- ▶ je débute, je n'ai pas lu le manuel, j'ai
 - ▶ lu des articles
 - ▶ lu des makefiles
- ▶ 20h effect: ce que je sais me suffit

Exemple: construire son site web avec pandoc et zsh

```
jade.js  template.jade  # gives template.html  
lsc      behave.ls     # gives behave.js  
stylus   look.us       # gives look.css  
dot -Tsvg -o deps.svg  deps.dot
```

```
chapters=( chap{01..04}.md )
```

```
pandoc=(  
    pandoc --template template.html  
    -t markdown -f beamer )
```

```
for c ($chapters)  
    $pandoc -o $c:r.html $c
```

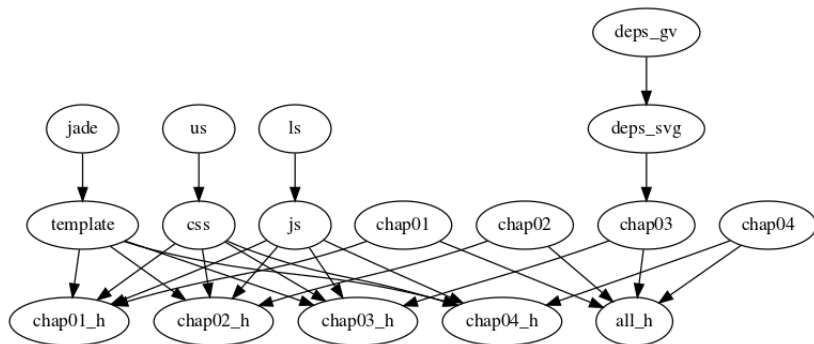
```
cat intro.md $chapters |  
    $pandoc > all.html
```

problème

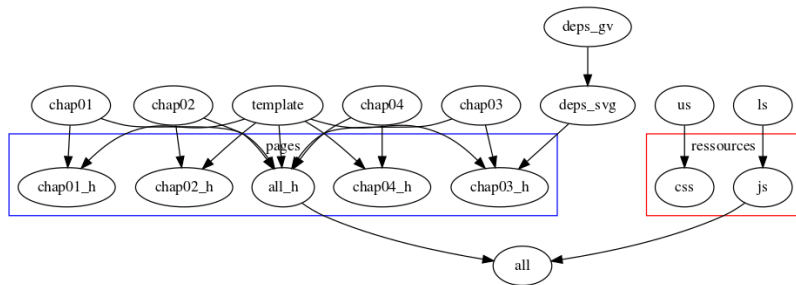
- ▶ le site est reconstruit intégralement à chaque execution.
- ▶ réduire au minimum \implies graphe de dépendances.

problème

- ▶ le site est reconstruit intégralement à chaque execution.
- ▶ réduire au minimum \implies graphe de dépendances.



phony rules



Build automation tools

`man mk`

maintain (make) related files

- ▶ make (mk, pmake, GNU make ...)
- ▶ ses “successeurs”
(cmake, scons, rake, daiku, grunt, ant, gradle, ...)
- ▶ liste sur wikipedia
- ▶ GNU make, standard de facto sous linux.

GNU make

les make-alike ont

- ▶ largement et anciennement établis
- ▶ jamais détronés
- ▶ simples pour les problèmes simples
- ▶ “extensible dans n’importe quel langage”

exemples d’usages connus

- ▶ autotools
 - ▶ m4, m4sh, make
 - ▶ utilisé par de nombreux projets OSS
- ▶ paquets debian:
 - ▶ archives ar
 - ▶ rules est un Makefile

makefile

- ▶ le fichier source
- ▶ un langage pour décrire les graphes de dépendance
- ▶ sous la forme de règles

les règles (rules)

Cible (target)

état à atteindre

- ▶ fichier à construire
- ▶ phony: “fichier virtuel”

Dépendance (dependency)

fichier nécessaire à la construction ou l'utilisation de la cible

Commandes (command)

commandes externes à exécuter pour construire la cible

la commande make

- ▶ s'assure que *les cibles* passées en argument (la première du makefile par défaut) sont *à jour*.
- ▶ être à jour c'est être *plus récent* que toutes ses *dépendances*
- ▶ *contruire* une cible, c'est executer les commandes qui permettent de la mettre à jour

Construction

- ▶ les commandes de construction sont exécutées par des subshells
 - ▶ `/bin/sh` par défaut
 - ▶ `SHELL= /usr/bin/zsh` pour plus de plaisir
- ▶ la construction s'arrête si
 - ▶ une dépendance ne peut être construite
 - ▶ une commande ne s'est pas exécutée correctement (`$?`)

syntaxe

- ▶ Attention: les commandes sont indentées avec une tabulation
 - ▶ `syn on`
 - ▶ `set hlsearch`
 - ▶ `set list listchars=tab:>-`
- ▶ les `\` protègent tous les caractères, fin de ligne y compris
- ▶ les `$` du shell doivent être doublées dans les commandes
- ▶ les `#` en début de ligne introduisent des commentaires

syntaxe

```
# comments
```

```
targets: dependencies; modifiers commands
```

```
targets: dependencies  
        modifiers commands
```

```
targets: dependencies \  
other-dependencies  
        a-very-long command with \  
        a lot of arguments  
        another command
```

hello world, makefile

```
# $$USER instead of $USER
# $$HOME instead of $HOME
# shell command variables needs two
# ...
```

```
hello.txt:
    false
    echo greetings, $$USER
    touch hello.txt
```

hello world, make produit

```
false
```

```
makefile:2: recipe for target 'hello.txt' failed
```

```
make: *** [all] Error 1
```


modifieurs de commande

- ▶ @ annule l'echo
- ▶ - la commande est validée même en cas d'échec

hello.txt:

```
@- false
```

```
@ echo greetings, $$USER
```

```
touch hello.txt
```

Exercice

- ▶ retranscrire la règle suivante ou une règle analogue
- ▶ ajoutez et supprimez des modifieurs et assurez-vous d'en avoir compris le fonctionnement
- ▶ lancez make jusqu'à ce qu'il vous indique que tout est à jour.

plusieur règles

hello.txt:

```
@- false
@ echo greetings, $$USER
touch hello.txt
```

hello2.txt:

```
@- false
@ echo greetings, $$USER
touch hello.txt
```

factoriser

```
hello.txt hello2.txt:
```

```
@- false
```

```
@ echo greetings, $$USER
```

```
touch $@
```

- ▶ target par default: hello.txt
- ▶ \$@ est une variable automatique

variables automatiques

- ▶ y'en a plein!
- ▶ celles à connaître sont
 - ▶ `$@` (la cible)
 - ▶ `$<` (la première dépendance). facile à retenir: pensez au redirecteur d'entrée (`<`)

usage des variables automatiques

```
behave.js:      behave.ls      ; lsc $<
components.js:  components.ls  ; lsc $<
bus.js:         bus.ls         ; lsc $<
foo.svg:        foo.dot        ; dot -Tsvg -o $@ $<
bar.svg:        bar.dot        ; dot -Tsvg -o $@ $<
bang.svg:       bang.dot       ; dot -Tsvg -o $@ $<
```

pattern matching

```
behave.js:      behave.ls      ; lsc $<
components.js:  components.ls  ; lsc $<
bus.js:         bus.ls         ; lsc $<
foo.svg:        foo.dot        ; dot -Tsvg -o $@ $<
bar.svg:        bar.dot        ; dot -Tsvg -o $@ $<
bang.svg:       bang.dot       ; dot -Tsvg -o $@ $<
```

peut s'écrire

```
%.js      : %.ls      ; lsc $<
%.svg:    : %.dot     ; dot -Tsvg -o $< $@
```

stem

le pattern match.

- ▶ une règle peut trouver une cible avec
 - ▶ un préfixe (fixe et optionnel)
 - ▶ un “stem” (%)
 - ▶ un suffixe (fixe et optionnel)
- ▶ le stem trouvé est disponible
 - ▶ par % dans les dépendances
 - ▶ par la variable \$* dans la construction

exemples

target	fichier	stem
id_%_key.pub	id_rsa_key.pub	rsa
mod.%	mod.enigma	enigma
%.mod	enigma.mod	enigma

Exercice

soit cible `funk-%-funky` dans votre `makefile`. vous écrirez le reste de la règle telle que

```
make funk-to-funky
```

soit une copie de `ashes-to-ashes` que vous aurez préalablement créé. au passage, vous affichez le stem sur la `stdout`.

variables

affectation

depuis le makefile ou en paramètre de make

affectation	=
append	+=
bind	:=
valeur par défaut	?=

depuis la ligne de commande

```
make all env=prod
```

interpolation

```
$(rule): $(dep)  
    echo $(test)
```

variables (exemple)

```
template= template.beamer.latex
chapters = \
    index.md \
    touch.md
components= $(chapters) $(template) prelude.latex \
    deps.pdf deps_with_phony.pdf

mk-slides = pandoc \
    -f markdown -t beamer+escaped_line_breaks \
    --template $(template)

slides.pdf slides.latex: $(components)
    cat $(chapters) | $(mk-slides) -o $@
@echo DONE
```

conditions

```
ifeq ($(env),prod)
    cc=gcc -O3
else
    cc=gcc
endif
```

```
ifndef env
error.txt:; echo env not set > $@
end
```

phony rules

```
.PHONY: clean
```

```
clean:: rm *
```

exemples courant: all, dist, clean

variables d'environnement et inclusions

a.mk

```
ifeq ($(wowo),HAHA)
    wowo=cool
endif
all:
    make -f b.mk
```

b.mk

```
b.mk:all:
b.mk:    echo $(wowo)
```

dans le shell

```
wowo=HAHA make -f a.mk
```

```
make -f b.mk
```

```
make[1]: Entering directory '/tmp'
```

```
echo cool
```

```
cool
```

```
make[1]: Leaving directory '/tmp'
```


substitutions

```
md-files=$(wildcard *.md)
md-files=$(shell print *.md(/u:$$USER:) )
html-files=$(md-files:%.md=%.html)
```

make dans vim

```
set autowrite
set aw
set makeprg=make
set makeprg=python3\ %
set makeprg=make %:r.html
nnoremap ,x :make<cr>
make
```

aller plus loin avec quickfix.

make dans vim + tmux

```
vim ~/.zshenv
```

```
promise/tmux/tailf/open () {  
    tmux split-window -d -p 20 \  
        'tail -f ${PROMOUT:=/tmp/promise.mix}' }  
promise/new () {  
    : ${1:=zsh}  
    "$@" &>> ${PROMOUT:=/tmp/promise.mix} & }  
}
```

```
vim ~/.tmux.conf
```

```
bind-key P run-shell "zsh -c promise/tmux/tailf/open"
```

dans vim

```
:set makeprg=promise/new\ make
```

Questions?

merci ...