GNU make — Anregung zum Einsatz

Robert Stanowsky

2019-09-21
Dingfabrik Köln

Inhaltsverzeichnis

1. Was ist GNU make?

GNU make ist ein Werkzeug um eine ganze Gruppe aufeinander aufbauender Dokumente bei Änderung eines Dokuments zu aktualisieren.

- sehr verbreitet zum Kompilieren von Programm-Quelltext, zum Linken und Installieren der Programme

GNU make besteht aus

- dem ausführbaren Programm *make*
- und meist einem sogenannten *makefile*, standardmäßig (nach Priorität)
  1. „`GNUmakefile`“ (nur GNU make)
  2. „`makefile`“
  3. „`Makefile`“

  oder stattdessen abweichend

  `make --file=`*Anderer_Makefile_Name* `[--file=`*Weiteres_Makefile*`]` ...

2. makefile

Das makefile steuert make, ähnlich einer Skript-Sprache

Es enthält

- „rules“
- „variables“
- „directives“
- „conditionals“
- „comments“

2.1. „rules“

Allgemeine Syntax

```
targets : prerequisites
        recipe
        ...
```

oder

```
targets : prerequisites ; recipe
        recipe
        ...
```

Rezeptzeilen beginnen mit einem Tabulator (sofern nicht anders vorgegeben)

Mit davon abweichender Syntax: „double suffix rule“

```
.target-suffix.source-suffix
        recipe
        ...
```

und „single suffix rule“

```
.source-suffix
        recipe
        ...
```

und ferner weitere Sonderformen

- „double-colon rule“
- „static pattern rule“
- „terminal pattern rule“
- oder mit „order-only prerequisites“

2.1.1. „targets“ & „prerequisites“

Zwei reguläre Arten

- „explicit“: Die Datei ist explizit benannt
  - direkt als Klartext
  - oder nach Variablenexpansion
- „implicit“: Der Dateiname ist durch ein „pattern“ beschrieben
  - das „pattern“ enthält ein „%“-Zeichen als Platzhalter und ggf. fixe Zeichen als Präfix und Suffix
  - make muss dazu finden
    - eine passende Datei oder
    - ... eine Regel eine passende Datei zu erzeugen

... außerdem „special targets“

- stehen für keine Datei
- steuern nur den Ablauf
- Bsp.: „.PHONY“, „.SUFFIXES“

... und ferner „order-only prerequisites“

- notwendig für das „target“, begründen aber keine Erneuerung

2.1.2. „recipes“

Rezepte dienen zumeist der Erzeugung des Ziels

- make-Variablen werden von make expandiert
- das Ergebnis wird zeilenweise mit „sh“ (standardmäßig) ausgeführt

Zuweilen erfüllen die Rezepte andere, besondere Aufgaben

- z.B. Aufräumen und Dateien löschen

2.2. „variables“

Zwei „flavors“

- „recursively expanded“: `variable = ...`
- „simply expanded“: `variable := ...`

Aufruf mit `$(variable)` oder `${variable}`

Innerhalb eines Rezeptes

- „automatic variables“ wie z.B. „`@`“, „`<`“, „`*`“, ... ; aufgerufen mit „`$(@)`“ oder kurz „`$@`“, ...

Zu den „built-in rules“ sind i.d.R. zugehörige „built-in variables“ vordefiniert um diese Regeln eigenen Wünschen einfach anzupassen

- z.B. `CFLAGS`, `CPPFLAGS`, ...

Weiter gibt es

- „target-specific variables“ und „pattern-specific variables“ inclusive Vererbung auf die „prerequisites“
- Zusätze `export`, `override`, `private` zur besonderen Regelung des Definitionsbereiches
- Regeln zur Vererbung an sub-makes

2.3. „directives“

„directives“ für spezielle Aktionen beim Einlesen des makefile

- z.B. Einlesen eines sub-makefiles

2.4. „conditionals“

Die Syntax von „conditionals“ ist

```
conditional-directive
text-if-condition-one-is-true
else conditional-directive
text-if-condition-one-is-false-but-two-is-true
...
else
text-if-all-is-false
endif
```

wobei wie üblich nicht alle Zweige vorhanden sein müssen

Bedingt werden ganze Zeilen des makefiles

- gültig oder ungültig
- unabhängig von Syntax und Inhalt der Zeilen!

2.5. „comments“

Das „#“-Zeichen und alles folgende auf der Zeile sind Kommentare

- Ausnahmen: als Zuweisung an eine Variable; innerhalb Variablen- und Funktionsaufrufen
- einzelne „\“ am Zeilenende setzen den Kommentar in die nächste Zeile fort

3. Einstiegshürden

Die Haupthürden sehe ich in

1. Nichtsequentieller Ablauf
2. Doppel-Syntax
3. Eingebaute Regeln und Variablen

3.1. Regelbasierter Ablauf

Der regelbasierte, nichtsequentielle Ablauf ist bei der Programmierung zumeist ungewohnt

Insbesondere bei Fehlern ...

- ist nicht mit dem Abbruch in unmittelbarer Umgebung des Fehlers zu rechnen
- muss daher anders nach dem Fehler gesucht werden

Diese Hürde ist immerhin offensichtlich!

3.2. Doppel-Syntax

Die make-Syntax ist ...

- kompliziert mit vielen Sonderfällen
- teils verwirrend, z.B. optisch teils gleichen „special targets“ („`.SUFFIXES`“) und „predefined variables“ („`.RECIPEPREFIX`“)
- Debug-Informationen in teilweise obsoleter Syntax („suffix rules“)

Die sh-Syntax ist an sich schon sehr kompliziert

- ... und durch eingebettete make-Variablen noch schwieriger zu lesen

Es müssen also zwei Syntaxe beherrscht werden für ein makefile!

Und dies ist nicht unbedingt sofort ersichtlich!

3.3. Eingebaute Regeln und Variablen

Über die eingebauten Regeln und Variablen ist ein Teil des Programms quasi schon geschrieben bevor das makefile auch nur eine Zeile enthält

- dieser Teil muss also erst einmal verstanden werden
- der eigene Teil muss damit harmonieren

Diese Hürde sieht man einem makefile schwerlich an!

4. make beobachten

Zur Fehlersuche ist es unerlässlich den Ablauf nachzuvollziehen

- `make --debug=a` ... zeigt den Ablauf
- `make --print-database` ... zeigt fortlaufend die wirksamen Regeln und Variablenwerte
- ... und `make --print-database --file=/dev/null` zeigt dies nur einmal ohne Aktion
- `make --dry-run` ... zeigt was make täte

5. Endlich: Das Beispiel-makefile

Nach soviel Abschreckung nun ein aufmunterndes Beispiel

Dieses `makefile` ist dazu gemacht vorab etwa mit

```
notangle -t8 -Rmakefile sfd.nw > makefile
```

extrahiert zu werden

Folgende Argumente werden erkannt

-ohne-|`all` wie `make alldoc allprog`

`alldoc` wie `make dvi ps pdf html`

`dvi`|`ps`|`pdf`|`html` erzeugt die Dokumentation `sfd.dvi`|`ps`|`pdf`|`html` in dem Format

`allprog` wie `make allc allpy`

`allc` erzeugt alle C-Programme

`allpy` erzeugt alle Python-Skripte

`allchunks` extrahiert alle „code-chunks“ unverändert

`makefile` erneuert das makefile!

`clean` löscht die meisten Dateien

`cleanall` löscht alle Dateien außer der noweb-Datei

Zusätzliche Optionen für C-Quellcode

`comments=yes` wandelt LaTeX-Dokumentations-Quellcode in C-Kommentare um (als Notlösung)

`lines=no` wählt Verweise auf die Ursprungszeilen in der `noweb`-Datei als C-Präprozessor-Kommentare (zum Debuggen) ab

⟨*makefile* 25a⟩≡ 25b ▷

```
# The noweb file "sfd.nw" as of 2019-09-21 including this makefile and
# full documentation is available at <http://sfd.koelnerlinuxtreffen.de>
# Copyright (C), 2019, R. Stanowsky
# License: GPL v. 2
```

Weise vorab den Namensstamm der noweb-Datei der make-Variable `src` fest zu

⟨*makefile* 25a⟩+≡ ◁ 25a 26a ▷

```
src = sfd
```

Deklariere „phony targets“ um Konflikte mit eventuell gleichnamigen Dateien auszuschließen

- bei mir funktionierte auch ein Tabulator am Anfang der Fortsetzungszeile

⟨*makefile* 25a⟩+≡ ◁25b 26b▷

```
.PHONY : all allc allchunks alldoc allprog allpy \
         dvi ps pdf html autopdf autopdf2 clean cleanall
```

Lösche hier generell am Ende keine „Zwischen“-Dateien

Bsp.: `sfd.nw` → *sfd.tex* → `sfd.pdf`

⟨*makefile* 25a⟩+≡ ◁26a 27a▷

```
.SECONDARY :
```

Standardmäßig vorbelegte „suffix rules“ besitzen mindestens eine vom „special target“ „`.SUFFIXES`“ abhängige Endung

Entferne die standardmäßige Vorbelegung (`.c`, `.cpp`, `.tex`, ... ca. 35 Endungen)

⟨*makefile* 25a⟩+≡ ◁26b 27b▷

```
.SUFFIXES :
```

Nimm die hier relevanten Endungen (zumeist wieder) auf

⟨*makefile* 25a⟩+≡ ◁27a 28a▷

```
.SUFFIXES : .c .dvi .html .nw .o .pdf .ps .py .sh .tex
```

Die erste „(simple) rule“ wird ausgeführt falls make ohne Parameter aufgerufen wird

⟨*makefile* 25a⟩+≡ ◁27b 28b▷

```
all : alldoc allprog
```

Es definiert nur die Abhängigkeit des „phony target“ „`all`“ von zwei weiteren „phony targets“

. . . und so fort eine Ebene tiefer

⟨*makefile* 25a⟩+≡ ◁28a 29▷

```
alldoc : dvi html pdf ps

allprog : allc allpy
```

Kompiliere alle C-Programme aus der noweb-Datei

1. Extrahiere mit grep alle C-Chunk-Namen
2. Entferne mit sed die umgebenden spitzen Doppelklammern und das Gleichheitszeichen
   - Maskiere sie mit Hochkommata um sie vor noweb zu verbergen
   - sh fügt die Teile wieder nahtlos zusammen
3. Rufe make mit den extrahierten Chunk-Namen rekursiv auf!

⟨*makefile* 25a⟩+≡ ◁28b 30▷

```
allc : $(src).nw
        grep '^<''<.*\.c>''>=$$' $< |sed 's/^<''<//' \
            |sed 's/.c>''>=$$//' |xargs $(MAKE)
```

Analog für die Python-Skripte

⟨*makefile* 25a⟩+≡ ◁29 31▷

```
allpy : $(src).nw
        grep '^<''<.*\.py>''>=$$' $< |sed 's/^<''<//' \
            |sed 's/>''>=$$//' |xargs $(MAKE)
```

Erzeuge auch das makefile selbst aus der noweb-Datei

Eine Regel um das makefile selbst zu erstellen wird automatisch vorrangig behandelt

⟨*makefile* 25a⟩+≡ ◁30 32▷

```
makefile : $(src).nw
        notangle -t8 -R$@ $< |cpif $@
```

Extrahiere alle noweb-Code-Chunks (unverändert)

- „`--no-builtin-rules`“ unterdrückt alle „implicit rules“ damit die Auffangregel zur Extraktion (s.u.) zur Anwendung kommt

⟨*makefile* 25a⟩+≡ ◁31 33▷

```
allchunks : $(src).nw
        grep '^<''<.*>''>=$$' $< |sed 's/^<''<//' \
            |sed 's/>''>=$$//' |xargs $(MAKE) --no-builtin-rules lines=no
```

Signalisiere mit dem „empty recipe“, dass für dieses „target“ nichts zu tun ist

Verhindere eine Behandlung durch die Auffangregel (s.u.)

⟨*makefile* 25a⟩+≡ ◁32 34▷

```
$(src).nw : ;
```

Kompiliere und linke C-Programme

- mit „`CC`“ als „predefined variable“ für den voreingestellten C-Kompiler
- und Optionen in einer neuen Variable „`CCFLAGS`“
  - (erhalte die zu „`CC`“ vorgesehene „predefined variable“ mit inkonsistentem Namen „`CFLAGS`“ unverändert)
- ... und ermögliche so abweichende Kompilation von der Kommandozeile
  z.B. `make CCFLAGS=-std=c89 hallo`

⟨*makefile* 25a⟩+≡ ◁33 35▷

```
CCFLAGS = -std=c99 -Wall -lm
% : %.o
        $(CC) $(CCFLAGS) $< -o $@
        chmod 744 $@

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

Extrahiere schließlich den C-Quelltext

- standardmäßig mit Zeilenverweisen auf die noweb-Datei zum Debuggen aber ohne zusätzlichen Kommentar
- rufe „`make lines=no ...`“ für C-Quellcode ohne Zeilenverweise
- ... und „`make comments=yes ...`“ für zusätzlich in den C-Quellcode eingebetteten Kommentar
- ergänze ggf. den Namen der C-Quellcodedatei um „`_noLines`“ bzw. „`_comments`“ (rufe `make lines=no hallo.c` für Ergebnis `hallo_noLines.c`)

⟨*makefile* 25a⟩+≡ ◁34 36a▷

```
%.c : $(src).nw
ifeq ($(comments),yes)
        nountangle -c -R$@ $< |cpif $*_comments.c
else ifeq ($(lines),no)
        notangle -R$@ $< |cpif $*_noLines.c
else
        notangle -L -R$@ $< |cpif $@
endif
```

Erzeuge die LaTeX-Datei

⟨*makefile* 25a⟩+≡ ◁35 36b▷

```
%.tex : $(src).nw
        noweave -delay -index $< |cpif $@
```

... und extrahiere Python-Skripte

⟨*makefile* 25a⟩+≡ ◁36a 37▷

```
%.py : $(src).nw
        notangle -R$@ $< |cpif $@
```

Erzeuge (klassische) shell-Skripte

1. Extrahiere sie
2. Ersetze mit sed Variablen analog zu make
3. Mache sie ausführbar

⟨*makefile* 25a⟩+≡ ◁36b 38▷

```
%.sh : $(src).nw
        notangle -R$@ $< |sed --posix 's/\$$(src)/$(src)/g' \
            |sed --posix 's/\$$(MAKE)/$(MAKE)/g' |cpif $@
        chmod 744 $@
```

Erzeuge die DVI- und PDF-Dokumentation mit einem einfachen shell-Skript

Definiere vorab „`LATEX`“ als „pattern-specific variable“ die abhängig davon definiert ist ob das aktuelle „target“ auf das „pattern“ passt

Rufe (PDF)LaTeX nach empfohlener Praxis über Variablen auf

- dies ermöglicht z.B. auf der Kommandozeile
  `make LATEXFLAGS=-interaction=errorstopmode pdf`

⟨*makefile* 25a⟩+≡ ◁37 39▷

```
dvi : $(src).dvi
pdf : $(src).pdf
%.dvi : LATEX = latex
%.pdf : LATEX = pdflatex
LATEXFLAGS = -halt-on-error
```

Eine Regel mit mehreren „targets“ entspricht mehreren dieser Regel mit je einem der „targets“

Solange der grep-Befehl das Muster findet und den Exit-Status „0“ liefert wiederhole den (PDF)LaTeX-Lauf

⟨*makefile* 25a⟩+≡ ◁38 40▷

```
%.dvi %.pdf : %.tex
        while $(LATEX) $(LATEXFLAGS) $< \
            && echo "\n$(MAKE) $@: Check for 'Rerun' warning(s) in $*.log" \
            && grep --line-number "Rerun" $*.log ; \
        do \
          echo "$(MAKE) $@: Will now rerun $(LATEX)!\n" ; \
        done ; \
        echo "$(MAKE) $@: No 'Rerun' warning(s) in $*.log"
```

Und noch die Regeln für die restlichen Dokumentationen

⟨*makefile* 25a⟩+≡ ◁39 41▷

```
html : $(src).html
%.html : %.nw
        noweave -filter l2h -delay -index -html $< |htmltoc |cpif $@

ps : $(src).ps
%.ps : %.dvi
        dvips -f < $< > $@
```

Das autopdf-Shellskript kann aufgerufen werden um im Hintergrund periodisch die PDF-Ausgabe zu erneuern soweit erforderlich

⟨*makefile* 25a⟩+≡ ◁40 42▷

```
autopdf : autopdf.sh
autopdf.sh : $(src).nw
        echo '#Skript erzeugt aus $<\n\n' \
            'modTime=0\n' \
            'while true; do\n' \
            '  if [ $$(stat --format=%Y $<) -gt $$modTime ]; then\n' \
            '    modTime=$$(stat --format=%Y $<)\n' \
            '    $(MAKE) pdf\n' \
            '  fi\n' \
            '  sleep 0.1\n' \
            'done' |sed 's/^ //' > $@
        chmod u+x $@
```

Noch zwei Argumente zum Aufräumen

⟨*makefile* 25a⟩+≡ ◁41 43▷

```
clean :
        rm -f $(src).aux $(src).log $(src).out $(src).tex $(src).toc \
             $(src).dvi $(src).html $(src).pdf $(src).ps \
             hallo hallo.c hallo_noLines.c hallo.o \
             wurzel2 wurzel2.c wurzel2_noLines.c wurzel2.o \
             ggT.py primzahlen.py

cleanall : clean
        rm -f makefile \
             autopdf.sh autopdf2.sh
```

Schließlich eine Auffang-Regel

- extrahiere einfach nur falls keine andere Regel passt
- sie kommt insbesondere beim rekursiven Aufruf (s.o.) von make zur Extraktion aller Chunks zum Zug

⟨*makefile* 25a⟩+≡ ◁42

```
.DEFAULT :
        notangle -R$@ $< |cpif $@
```

6. Ein sh-Skript

Und hier eine alternative Variante für ein gleichwertiges „autopdf“-Skript.

- Vorteil: Das Skript schreibt sich schöner, nicht als „echo-Wurm“
- Nachteil: make-Variablen müssen nachträglich und separat behandelt werden

⟨*autopdf2.sh* 44⟩≡

```
#Skript erzeugt aus $(src).nw

modTime=0
while true; do
  if [ $(stat --format=%Y $(src).nw) -gt $modTime ]; then
    modTime=$(stat --format=%Y $(src).nw)
    $(MAKE) pdf
  fi
  sleep 0.1
done
```

A. Python-Programme

Anmerkung: beginnend mit 341 würde das Programm auch einige Nicht-Primzahlen liefern

⟨*primzahlen.py* 45a⟩≡

```
print 2
for p in range(3, 100, 2) :
  if pow(2, p-1, p) == 1 :
    print p
```

⟨*ggT.py* 45b⟩≡

```
def ggT (x, y) :
  tmp = x % y
  while tmp <> 0 :
    x = y
    y = tmp
    tmp = x % y
  return y
```

B. C-Programme

⟨*hallo.c* 46a⟩≡

```
#include <stdio.h>

int main()
{
    printf("Hallo, Schöne Grüße aus dem C-Programm\n");
    return 0;
}
```

⟨*wurzel2.c* 46b⟩≡

```
#include <math.h>
#include <stdio.h>

int main()
{
    printf("Wurzel von 2: %.20g\n", sqrt(2));
    return 0;
}
```

C. Literatur & Lizenz

Richard M. Stallman, Roland McGrath, Paul D. Smith:
„GNU Make — A Program for Directing Recompilation; GNU make Version 3.82“,
Juli 2010, → http://www.gnu.org/software/make/