Homepage **Computer-Stoff(Titelseite)**

Eine Einführung in Makefiles

Diese Datei soll eine Einführung in das Erstellen von Makefiles geben. Sie ist für Leute gedacht, die wissen, was das Programm make macht, und wozu man es verwendet, selber aber noch nie ein makefile erstellt haben. Bei den Beispielen liegt der Schwerpunkt auf Makefiles für C-Programme.

Ich sollte noch sagen, daß es mehrere make-Programme gibt, deren Makefiles nicht 100prozentig kompatibel sind. Ich beziehe mich hier auf das GNUmake. Speziell alle Sachen, die mit Pattern zu tun haben, können je nach verwendetem make-Programm leicht unterschiedlich sein.

Targets, Regeln und Abhängigkeiten

Ein Makefile ist dazu da, dem Programm make mitzuteilen, *was* es tun soll (dies ist das "target"), und *wie* es es tun soll (dies ist die zum target gehörige "Regel"). Des weiteren kann man zu jedem target angeben, von welchen anderen targets oder Dateien dieses abhängt.

Am besten sieht man das an einem einfachen Beispiel. Nehmen wir an, daß wir ein kleines C-Programm namens `prog.c` mit zugehöriger Header-Datei `prog.h` haben, welches wir normalerweise mit

```
gcc -o prog prog.c
```

übersetzen. Unser Ziel ist es also, die Datei `prog` zu erzeugen. Diese ist offensichtlich abhängig von `prog.c` und `prog.h`. Im Makefile sieht dies wie folgt aus:

```
prog: prog.c prog.h
        gcc -o prog prog.c
```

In der ersten Zeile teilen wir make mit, daß es ein target namens "prog" gibt, und daß dieses von `prog.c` und `prog.h` abhängt. In der zweiten Zeile sagen wir make, mit welcher Regel (d.h. wie) es dieses target erzeugen kann. Wird make nun aufgerufen, überprüft es, ob eine der beiden Dateien `prog.c` oder `prog.h` neuer ist als das target. In diesem Fall führt es die zweite Zeile aus und erzeugt eine neue ausführbare Datei.

Eine gemeine Falle für Anfänger ist, daß die zweite Zeile mit einem `<tab>` anfangen muß, und nicht mit Leerzeichen.

Viele targets können nicht wie hier durch einen einzigen Befehl erzeugt werden, sondern es sind mehrere nötig. In diesem Fall folgen auf die Zeile mit den Abhängigkeiten einfach mehrere Zeilen, die alle mit `<tab>` anfangen. Auch die Abhängigkeiten für ein target dürfen auf mehrere Zeilen verteilt sein.

Man beachte, daß in unserem Beispiel "`prog`" gleichzeitig ein Name für ein target und für eine Datei ist. make sieht darin keinen Unterschied. Auch die beiden Dateien `prog.c` und `prog.h` sind für make nichts anderes als targets. Diese targets hängen von nichts ab, sind also immer aktuell, und es gibt auch keine Regel, um sie zu erzeugen. Würde nun beispielsweise die Datei `prog.h` nicht existieren, so würde make feststellen, daß es das target `prog.h`, welches für `prog` benötigt wird, nicht erzeugen kann. Die Fehlermeldung lautet dementsprechend:

```
make: *** No rule to make target `prog.h'. Stop.
```

Die Zeile mit der Regel (`gcc ...`) wird von make in einer shell aufgerufen. Es ist also möglich, wildcards oder Kommandosubstitutionen (mit `...`) zu benutzen.

Das Defaulttarget

Beim Aufruf von make kann man das zu erzeugende target angeben:

```
make prog
```

Ruft man make dagegen ohne jegliche Argumente auf, so wird das erste target, welches im Makefile gefunden wird, erzeugt. In vielen Makefiles ist das erste target deshalb eines namens `all`, welches von einer ganzen Reihe von weiteren targets abhängt. Ziel ist es in der Regel, durch einen argumentlosen Aufruf von make ein fertiges, ausführbares Programm zu bekommen. Insbesondere müssen hierzu alle Objektdateien und die ausführbaren Dateien erzeugt werden.

Pattern in Regeln

In der Praxis bestehen Programmierprojekte aus so vielen Dateien und Abhängigkeiten, daß es unpraktikabel ist, für jede einzelne Objekt-Datei eine neue Regel ins Makefile einzubauen, und diese bei jedem neuen `#include` zu ändern. Deshalb gibt es die Möglichkeit, Regeln durch eine Art Wildcard-Pattern zu definieren:

```
%.o: %.c
      gcc -Wall -g -c $<
```

Diese Regel besagt, daß jede `.o`-Datei von der entsprechenden `.c`-Datei abhängt, und wie sie mit Hilfe des Compilers erzeugt werden kann.

Um innerhalb der Regel auf den Namen des targets und die der Abhängigkeiten zugreifen zu können, gibt es einige von make vordefinierte Variablen. Hier die meiner Meinung nach wichtigsten (von sehr vielen):

- `$<` die erste Abhängigkeit
- `$@` Name des targets
- `$+` eine Liste aller Abhängigkeiten
- `$^` eine Liste aller Abhängigkeiten, wobei allerdings doppelt vorkommende Abhängigkeiten eliminiert wurden.

Man beachte, daß bei der Angabe der Abhängigkeiten hier die Abhängigkeit der Objekt-Dateien von diversen Header-Dateien unter den Tisch gefallen ist. Eine Möglichkeit, diese wiederzubekommen, werde ich im Abschnitt über Abhängigkeiten als target besprechen.

Die hier vorgestellte Art, durch Pattern Regeln zu erzeugen, ist nur eine von vielen möglichen. Leider muß man sagen, daß Pattern und Pattern-Substitutionen in Makefiles zu den Sachen gehören, die am meisten durcheinandergeraten und am verwirrendsten sind. Außerdem unterscheiden sie sich bei unterschiedlichen make-Versionen u.U. voneinander.

Regeln, die durch Pattern erzeugt wurden, sind ein Spezialfall sog. "Impliziter Regeln", d.h. Regeln, die man make nicht explizit angegeben hat, sondern die von make durch Anwendung bestimmter Vorschriften deduziert werden.

Suffix-Regeln

Eine andere Art von impliziten Regeln, die man häufig sieht, sind sog. Suffix-Regeln. Die obige Abhängigkeit von `.o`-Dateien von den entsprechenden `.c`-Dateien würde mit ihrer Hilfe wie folgt ausgedrückt werden:

```
.c.o
```

```
	gcc -Wall -g -c $<
```

Es gibt mehrere Arten von Suffix-Regeln. Suffix-Regeln sollten allerdings als obsolet betrachtet werden, und man sollte lieber die besser lesbaren durch Pattern erzeugten Regeln verwenden.

Variablen in Makefiles

Es ist möglich, in Makefiles Variablen zu definieren und zu benutzen. Üblicherweise verwendet man Großbuchstaben. Gebräuchlich sind beispielsweise folgende Variablen:

`CC`	Der Compiler
`CFLAGS`	Compiler-Optionen
`LDFLAGS`	Linker-Optionen

Auf den Inhalt dieser Variablen greift man dann mit `$(CC)`, `$(CFLAGS)` bzw. `$(LDFLAGS)` zurück.

Ein einfaches Makefile eines Programmes namens `prog`, welches aus einer Reihe von Objekt-Dateien zusammengelinkt werden soll, könnte also wie folgt aussehen:

```
VERSION = 3.02
CC      = /usr/bin/gcc
CFLAGS  = -Wall -g -D_REENTRANT -DVERSION=\"$(VERSION)\"
LDFLAGS = -lm -lpthread `gtk-config --cflags` `gtk-config --libs` -lgthread

OBJ = datei1.o datei2.o datei3.o datei4.o datei5.o

prog: $(OBJ)
        $(CC) $(CFLAGS) -o prog $(OBJ) $(LDFLAGS)

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

Das Defaulttarget ist hier das ausführbare Programm `prog`. Dieses hängt von allen Objekt-Dateien ab. Beim Linken werden die Mathe-, die pthread, diverse GTK-Libraries, und die gdk-threads-library dazugelinkt.

An diesem Beispiel kann man außerdem sehen, daß man nicht vergessen darf, daß eine Shell die Befehle ausführt: Ohne die backslashs in `\"$(VERSION)\"` würde diese nämlich die Anführungzeichen entfernen. Die Versionsnummer soll dem C-Präprozessor aber als konstante Zeichenkette übergeben werden. Des weiteren sieht man, daß beim Linken ein Programm namens "`gtk-config`" aufgerufen wird, welches die für GTK benötigten Libraries auf der Standard-Ausgabe ausgibt. Hier wird eine Kommandosubstitution durchgeführt.

Da, wie man in den letzten Beispielen gesehen hat, das Dollarzeichen in Makefiles zur Kennzeichnung von Variablen verwendet wird, muß, wenn in einer Regel tatsächlich ein Dollarzeichen auftaucht, dieses maskiert werden: `$$` ist ein Dollarzeichen. (Vergl. das Beispiel im Abschnitt über rekursives make!)

Kommentare in Makefiles

Genau wie die Shell werden von make Zeilen, die mit einem "#" anfangen, als Kommentare angesehen:

```
# auskommentierte Zeile.
```

Phony targets

Bei normalen targets überprüft make, ob sie aktueller sind als alle targets, von denen sie abhängen. Falls dies nicht der Fall ist, werden sie neu erzeugt. Targets, die von keinen weiteren targets abhängen, sind folglich immer aktuell und werden nie erzeugt. Dies trifft in der Regel beispielsweise für die Quellen eines Programmes zu.

Manche targets entsprechen keiner physikalischen Datei. Ein typisches solches target ist das target "clean", welches alle erzeugten Dateien löscht. Es ist von nichts abhängig. Probleme kann es nun geben, wenn es im Verzeichnis eine Datei namens "clean" gibt. Make stellt nun fest, daß das target "clean" bereits erzeugt ist, und von nichts abhängig und deshalb aktuell ist.

Es gibt also targets, die stets(!) neu erzeugt werden sollen, unabhängig davon, von welchen anderen targets sie abhängig sind. Diese nennt man phony targets:

```
OBJ = datei1.o datei2.o datei3.o datei4.o datei5.o
BIN = prog

.PHONY: clean
clean:
	rm -rf $(BIN) $(OBJ)
```

Gibt man "`make clean`" ein, so wird nun der `rm`-Befehl stets durchgeführt, unabhängig davon, ob eine Datei namens "clean" existiert oder nicht.

Phony targets können genau wie alle normale targets von anderen abhängen. Im Allgemeinen ist es nicht nötig, irgendwelche targets als phony zu deklarieren. Man sollte das ganze jedoch im Hinterkopf behalten.

Pattern Substitution

Genau, wie man mit Hilfe von Pattern Regeln erzeugen konnte, kann man auch Variablen erzeugen:

```
OBJ = datei1.o datei2.o datei3.o datei4.o datei5.o
SRC = $(OBJ:%.o=%.c)
HDR = $(OBJ:%.o=%.h) config.h
```

Hier haben die Variablen `SRC` und `HDR` danach die Werte

```
datei1.c datei2.c datei3.c datei4.c datei5.c
datei1.h datei2.h datei3.h datei4.h datei5.h config.h
```

Genau wie bei der Erzeugung von Regeln mit Hilfe von Pattern gilt auch hier, daß die Möglichkeiten, Variablen durch solche Substitutionen zu erzeugen, so mannigfaltig sind, daß man sie eigentlich nicht alle kennen kann. Im Zweifelsfall empfiehlt sich, die make-Anleitung durchzulesen. (S. im Abschnitt über weitere Informationen!)

Abhängigkeiten als Target (`make dep`)

Benutzt man implizite oder durch Pattern erzeugte Regeln zur Erzeugung von Objekt-Dateien, so entfallen die Abhängigkeiten dieser von den Header-Dateien. Um dieses Problem zu umgehen, ohne die Abhängigkeiten jeder Objekt-Datei händisch ins Makefile einbauen zu müssen, definiert man meist ein target namens "`dep`" für "dependencies":

```
SRC = datei1.c datei2.c datei3.c datei4.c datei5.c
CC  = /usr/bin/gcc
DEPENDFILE = .depend

dep: $(SRC)
        $(CC) -MM $(SRC) > $(DEPENDFILE)

-include $(DEPENDFILE)
```

Die Option `-MM` läßt den Präcompiler nach `#include`-Direktiven im Quellcode schauen. Er gibt die entsprechenden Abhängigkeiten genau in dem Format, in dem sie make braucht, auf der Standardausgabe aus. Hier werden sie allerdings in eine Datei namens `.depend` umgelenkt, welche dann mittels eines `include` in das Makefile eingefügt wird. Das Minuszeichen vor dem `include` sagt make, daß es nicht mit einer Fehlermeldung abbrechen soll, wenn die Datei nicht existiert - schließlich muß sie ja erst einmal durch einen Aufruf von "`make dep`" erzeugt werden. (Der `include`-Befehl ist eine Spezialität des GNUmake-Programmes. Wie man eine entsprechende Wirkung bei anderen make-Programmen erzielt, weiß ich nicht.)

Ein solches target ist insbesondere dann wichtig, falls es Header-Dateien gibt, die dynamisch generiert werden und anschließend in Quell-Code-Dateien eingebunden werden.

Rekursives Make

Bei großen Projekten sind die Quelldateien über viele Verzeichnisse in mehreren Ebenen verstreut. Aus Gründen der Übersichtlichkeit und auch der Effektivität ist es in solchen Fällen üblich, jedem Verzeichnis sein eigenes Makefile zu geben. Man kann dann z.B. in einem Verzeichnis ein "`make clean`" machen, ohne daß danach der gesamte Verzeichnisbaum neu compiliert werden muß.

Im Top-Level-Verzeichnis befindet sich dann nur noch ein Makefile, welches rekursiv make in allen Unterverzeichnissen aufruft. Des weiteren könnte man dort eine Datei mit allgemeinen Regeln ablegen, die von allen Makefiles in den Unterverzeichnissen eingebunden wird.

Es gibt keine Standard-Methode, dies alles zu machen. Es hängt zu sehr von den besonderen Bedingungen, die beim Projekt herrschen, ab. Hier ein aus Gründen der Übersichtlichkeit äußerst rudimentär gehaltenes Beispiel:

```
# Makefile im Top-Level-Verzeichnis

DIRS = dir1 dir2 dir3

compile:
	for i in $(DIRS); do make -c $$i; done
```

Das `$$i` wird von make zu `$i` evaluiert. Die Shell referenziert dann den Wert der Variablen `i`. Die Option `-c`, die dem make übergeben wird, bewirkt, daß make vor dem Start ins angegebene Verzeichnis wechselt.

```
# Makefile.rules im Top-Level-Verzeichnis

CC     = /usr/bin/gcc
CFLAGS = -Wall -g

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

In dieser Datei werden Regeln und Variablen definiert, die in allen Unterverzeichnissen gelten sollen. Was noch fehlt, sind die Makefiles in den einzelnen Unterverzeichnissen:

```
# Makefile in einem Unterverzeichnis
include ../Makefile.rules
```

```
OBJ = datei1.o datei2.o datei3.o datei4.o datei5.o

all: $(OBJ)
```

Hier werden also nur noch die in diesem Verzeichnis zu erzeugenden Objekt-Dateien aufgelistet und das Default-Target so gesetzt, daß sie bei einem Aufruf von make ohne Argumente erzeugt werden.

Typische Probleme mit make

Es gibt einige Sachen, die make nicht besonders gut kann, und die deshalb öftermal Probleme bereiten, welche man nur schlecht lösen kann. Die meisten rühren daher, daß das feststellen von Abhängigkeiten zu kompliziert und unsicher ist (vergl. den Abschnitt über Abhängigkeiten!) Typisch hierfür wären:

Abhängigkeiten beim rekursiven make

Nehmen wir an, wir haben unsere Sourcen über mehrere Verzeichnisse verteilt, in die `make` rekursiv absteigt (wie im vorigen Abschnitt beschrieben). Darunter gibt es auch zwei Verzeichnisse `dir1` und `dir2`. In `dir1` gebe es ein target, welches von einer Datei `D1` im Verzeichnis `dir2` abhängt. Und diese wiederum werde aus einer Datei `D2` erzeugt, die sich ebenfalls im Verzeichnis `dir2` befindet.

Wird nun ein `make` im Verzeichnis `dir1` aufgerufen, so kann dieses `make` nicht feststellen, ob die Datei `D1` aktuell ist, da es keine Regel und keine Abhängigkeiten für diese besitzt, denn diese befinden sich ja im Makefile des Verzeichnisses `dir2`. Und was noch schlimmer ist: Es kann die Datei `D1` auch nicht erzeugen,, falls sie fehlt. Insbesondere muß man in diesem Fall dafür sorgen, daß das rekursive `make` zunächst das `make` in `dir2`, und dann erst in `dir1` aufruft. Und auch das hilft nichts, wenn gleichzeitig auch noch der umgekehrte Fall auftritt.

Das einzige was man dagegen machen kann, ist die Sourcen so sauber zu strukturieren, daß alle Abhängigkeiten eines targets im selben Verzeichnis oder in einem Unterverzeichnis sind, aber nicht in einem benachbarten oder höhergelegenen. Bei einem rekursiven `make` müssen dann zunächst die `makes` in den Unterverzeichnissen, und dann im Verzeichnis selbst durchgeführt werden.

Neu-Compilieren bei Änderung von Compiler-Flags

Ein weiteres Problem ist, daß `make` nicht feststellen kann, wenn im makefile beispielsweise einige Compiler-Flags geändert wurden. Hierdurch ist u.U ja die Neu-Compilierung einiger Dateien erforderlich. Da `make` dies nicht alleine feststellt, bleibt einem nichts als ein `make clean; make` übrig. Alternativ könnte man alle targets abhängig vom makefile machen und ev. unnötige Neu-

Compilierungen in Kauf nehmen.

Neu-Compilieren beim Verschieben von Dateien

Wenn mit `mv` alte Dateien wieder zurückgeholt werden, so bleibt das Datum der letzten Änderung auch alt. `make` kann nicht feststellen, daß in Wirklichkeit eine Neu-Compilierung notwendig ist. Hier hilft nur ein `touch`.

Weitere Informationen

erhält man durch

```
info make
```

Die Info-Seiten des GNU-Makes sind sehr verständlich geschrieben und ich kann sie guten Gewissens jedem empfehlen, der mehr über das Schreiben von Makefiles wissen will.

Es gibt diese Dokumentation auch in anderen Formaten (html, ascii, dvi, ps und texinfo). Man kann sie sich von den Online-Manual-Seiten von GNU herunterladen.

Für Ergänzungen, Korrekturen oder Kommentare bin ich übrigens immer dankbar.

---

by *Michael Becker*, 6/2001. Letzte Änderung: 8/2004.