Introduction of make

汪杰

Contents

Overview

In software development, `make` is a build automation tool that automatically builds executable programs and libraries from source code by reading files called makefiles which specify how to derive the target program.

Though integrated development environments and language-specific compiler features can also be used to manage a build process, `make` remains widely used, especially in Unix and Unix-like operating systems.

The make program uses the **makefile data base** (the file named `Makefile`) and the **last-modification times** of the files to decide which of the files need to be updated. For each of those files, it issues the recipes recorded in the data base.

How to use

To prepare to use `make`, you must write a file called the **Makefile** that describes the relationships among files in your program and provides commands for updating each file.

Once a suitable makefile exists, each time you change some source files, this simple shell command:

```
make
```

It suffices to perform all necessary recompilations.

The `make` utility **automatically** determines which pieces of a large program need to be recompiled, and issues commands to recompile them.

A Makefile looks like...

A simple makefile consists of "rules" with the following shape:

```
target: prerequisites
<-Tab-> recipe
<-Tab-> ...
<-Tab-> ...
```

A **target** is usually the name of a file that is generated by a program, eg. executable or object files. *A target can also be the name of an action to carry out (Phony Targets), eg.* `clean`.

A **prerequisite** is a file that is used as input to create the target. A target often depends on several files. You can list multiple prerequisites with spaces between them.

A **recipe** is an action (eg. shell command) that make carries out. **Please note**: you need to put a tab character (i.e. `'\t'`) at the beginning of every recipe line, NOT spaces!

Example

Here is a makefile that describes the way an executable file called **edit** depends on three object files which, in turn, depend on three C source and two header files.

```
edit: main.o display.o utils.o
    gcc -o edit main.o display.o utils.o

main.o: main.c defs.h
    gcc -c main.c
display.o: display.c defs.h buffer.h
    gcc -c display.c
utils.o: utils.c defs.h
    gcc -c utils.c
clean:
    rm edit main.o display.o utils.o
```

To use this makefile to create the executable file called **edit**, type: `make` or `make edit`

To use this makefile to delete the executable file and all the object files from the directory, type: `make clean`

Contents

Variables

```
foo = bar
${foo} # bar (just like a shell variable)
$(foo) # bar (commonly used in Makefile)
f = a
$foo   # aoo (one character variable by default)
```

Use variables to simplify your Makefile.

```
objs = main.o display.o utils.o
edit: $(objs)
    gcc -o edit $(objs)
```

Automatic Variables

Var	Meanings
$@	The name of the target.
$<	The name of the first prerequisite.
$^	The names of all the prerequisites, with spaces between them.
$?	The names of all the prerequisites that are newer than the target, with spaces between them.

You can also write `$(@)` or `${@}`, but that would not be necessary.

See more automatic variables here.

Predefined Variables

Variable	Meanings	Default
CC	Program for compiling C programs.	cc
RM	Command to remove a file	rm -f
CFLAGS	Extra flags to give to the C compiler. eg. `-g` , `-fPIC` , `-Iinclude/path`	*empty*
LDFLAGS	Extra flags to give to the linker `ld` . eg. `-shared` , `-Llink/path`	*empty*
LDLIBS	Extra flags to give to the linker. eg. `-lpthread`	*empty*

Predefined Variables

Here are some variables involving (or seems to involve) C++, if you are interested.

Variable	Meanings	Default
CXX	Program for compiling C++ programs.	g++
CPP	Program for running the C preprocessor. (NOT C++ compiler!)	$(CC) -E
CXXFLAGS	Extra flags to give to the C++ compiler.	*empty*

See more predefined variables here

Contents

Implicit Rules

Implicit rules tell `make` how to use customary techniques so that you **DO NOT** have to specify them **in detail** when you want to use them.

For example, there is an implicit rule for C compilation. **File names** determine which implicit rules are run. For example, C compilation typically takes a `.c` file and makes a `.o` file. So `make` applies the implicit rule for C compilation when it sees this combination of file name endings.

The **built-in** implicit rules use several **predefined variables** in their recipes so that, by changing the values of the **predefined variables**, you can change the way the implicit rule works. For example, the variable `CFLAGS` controls the flags given to the C compiler by the implicit rule for C compilation.

See more infomation here

Pattern Rules

You can define your own implicit rules by writing pattern rules. A pattern rule looks like an ordinary rule, except that its target contains the character `%` (exactly one).

The target is considered a pattern for matching file names; the `%` can match any nonempty substring. The prerequisites likewise use `%` to show how their names relate to the target name.

```
%.o: %.c
    gcc -c $< -o $@
```

Built-In Rules

Makefile provides built-in rules for `.o` files, executables, and many other file formats.

```
# built-in rule for compiling
%.o: %.c
    $(CC) $(CFLAGS) -c -o $@ $<

# built-in rule for linking
%: %.o
    $(CC) $(LDFLAGS) $^ $(LDLIBS) -o $@
```

Run `make -p` in a directory with no makefiles to see a full list of **predefined variables** and **implicit rules** available in your environment (i.e. for your instance of `make`).

See more built-in rules here

Simplified Example

Practice of **Variables**, **Built-In Rules** and **Phony Targets**.

```makefile
target = edit
objects = main.o display.o utils.o
# CFLAGS = --whatever --flags --you --need
# LDFLAGS = --whatever --flags --you --need
# LDLIBS = -lwhatever -llibraries -lyou -lneed

$(target): $(objects)
    # if you rename main.o to edit.o, the following line can be omitted
    $(CC) $(LDFLAGS) $^ $(LDLIBS) -o $@

main.o: defs.h
display.o: defs.h buffer.h
utils.o: defs.h

.PHONY: clean
clean:
    $(RM) $(target) $(objects)
```

Reference

- GNU make manual
  - https://www.gnu.org/software/make/manual/make.html