The Build Process: Makefiles And Beyond

Arash Bakhtiari

2012-11-20 Tue

Introduction

- compiling your source code files can be tedious
- specially when you want to include many source files
- have to type the compiling command everytime you want to do it
- for big projects very time consuming

Make utility

- Make: a utility that help you to automatically build and manage your projects
- Makefile: are special format files which specify how to derive the target program
  - A makefile consists of rules

Rules

A Rule :

- specifies when and how to remake certain files(**targets**)
- lists the other files that are the prerequisites of the target (**dependencies**)
- **commands** to use to create or update the target
- Rule syntax:

```
targets : dependencies
        command
```

DEMO

```
prog: prog.o a.o b.o c.o
        g++ a.o b.o c.o prog.o -o prog

a.o: a.c
        g++ -c -g -Wall -o a.o a.c

b.o: b.c
        g++ -c -g -Wall -o b.o b.c

c.o: c.c
        g++ -c -g -Wall -o c.o c.c

prog.o: prog.c
        g++ -c -g -Wall -o prog.o prog.c
```

Macros

- Defining Macros in the Makefile
  - syntax: name = value
  - expressions of the form $(name) or ${name} are replaced with the value
  - example

```
CC = g++
CFLAGS = -c -Wall
```

- Defining Macros on the Command Line

```
make CC=g++ "CFLAGS= -c -Wall"
```

- make arguments overwrite the inside value

DEMO

```
CC = g++
CFLAGS = -c -g -Wall
SRCS = a.c b.c c.c prog.c
OBJS = a.o b.o c.o prog.o

prog: $(OBJS)
        $(CC) $(OBJS) -o prog

a.o: a.c
      $(CC) $(CFLAGS) -o a.o a.c

b.o: b.c
      $(CC) $(CFLAGS) -o b.o b.c

c.o: c.c
      $(CC) $(CFLAGS) -o c.o c.c

prog.o: prog.c
        $(CC) $(CFLAGS) -o prog.o prog.c
```

Substitution References

- substitutes the value of a variable with alterations that you specify
- syntax:

```
$(var:a=b)
${var:a=b}
```

- take the value of the variable *var*, replace every *a* at the end of a word with *b*
- example:

```
SRCS = a.c b.c c.c prog1.c
OBJS = $(SRCS:.c=.o)
```

Inference Rules

- many rules have the following form:

```
<filename>.o: <filename>.c
	$(CC) $(CFLAGS) <filename>.c
```

- use inference rules to avoid all these lenthy entries in a Makefile

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

- $< is a dependent file out-of date with the target file

DEMO

```
CC = g++
CFLAGS = -c -g -Wall
SRCS = a.c b.c c.c prog.c
OBJS = $(SRCS:.c=.o)

prog: $(OBJS)
        $(CC) $(OBJS) -o prog

.c.o:
        $(CC) $(CFLAGS) $<

clean:
  rm -rf *~ *o *d prog
```

Automatic Generation Of Makefile Dependencies

- the inference rule only covers part of the source code dependency (only knows that program.o depends on program.c)
- to solve this problem $\Rightarrow$ list the dependencies separately in make file
- gcc compiler with the flag "-MMD" automatically creates the dependecy files(file.d)
- add the dependecy files to Makefile with include directive
  - include directive suspends reading of the current makefile and read one or more other makefiles before continuing

DEMO

```
CC = g++
CFLAGS = -c -g -Wall -MMD
SRCS = a.c b.c c.c prog.c
OBJS = $(SRCS:.c=.o)

prog: $(OBJS)
	$(CC) $(OBJS) -o prog

.c.o:
	$(CC) $(CFLAGS) $<

clean:
	rm -rf *~ *o *d prog

-include $(OBJS:%.o=%.d)
```

Parallel Compiling

- take advantage of systems that have multiple processors, or multiple-core processors
- a separate build process is created for each available processor
- flag -j: specifies the number of jobs to run simultaneously

```
make -j [jobs]
```