Tutorial: Aprenda a criar seu próprio makefile

Darcamo (Forúns Ubuntu)

08 de Junho de 2007

Resumo

Eu estava acostumado a sempre deixar a "IDE" criar o makefile pra mim e nunca liguei muito pra ele, mas recentemente precisei aprender a criar meu próprio makefile. Embora na internet tenha toda a documentação que você possa precisar com milhares de tutoriais sobre o make além do próprio manual dele, é informação demais (quase tudo em inglês) e demorei um dia todo para aprender e criar um makefile que funcione do jeito que eu queria. Dessa forma resolvi escrever esse pequeno tutorial que preferi postar aqui ao invés de no Dicas e Truques por ser algo muito particular a programação.

Capítulo 1

O que é o programa make

O programa `make` é uma maneira muito conveniente de gerir grandes programas ou grupos de programas. Quando se começa a escrever programas cada vez maiores e visível a diferença de tempo necessário para recompilar esses programas em comparação com programas menores. Por outro lado, normalmente trabalha-se apenas em uma pequena parte do programa (tal como uma simples função que se está depurando), e grande parte do resto do programa permanece inalterada.

O programa `make` ajuda na manutenção desses programas observando quais partes do programa foram mudadas desde a última compilação e recompilando apenas essas partes.

Para isso, é necessário que se escreva uma "`makefile`", que é um arquivo de texto responsável por dizer ao programa make "o que fazer" e contém o relacionamento entre os arquivos fonte, objecto e executáveis.

Outras informações úteis colocadas no makefile são as "`flags`" que precisam ser passados para o compilador e o "linkador", como diretórios onde encontrar arquivos de cabeçalho (arquivos `.h`), com quais bibliotecas o programa deve ser ligado, etc. Isso evita que se precise escrever enormes linhas de comando incluindo essas informações para compilar o programa.

Capítulo 2

Escrevendo um makefile

Uma `makefile` contém essencialmente atribuições de variáveis, comentários e regras ("targets"). Comentários são iniciados com o carácter "#", enquanto que as regras possuem a forma

```
target1 target2 ... : dependencia1 dependencia2 ...
   <TAB> comando1
   <TAB> comando2
...
```

Um alvo ("target") é geralmente o nome de um arquivo que será gerado com a execução do(s) comando(s) associados ao target. Exemplos comuns são arquivos executáveis e arquivos objeto. Um target também pode ser o nome de uma ação a ser efetuada, tal como "`clean`" (limpar).

Como exemplo, usarei um programa composto de 3 arquivos `.cpp` (chamados `Fraction.cpp, fractiontest.cpp` e `ftest.cpp`) e 2 arquivos `.h` (`Fraction.h` e `fractiontest.h`). Para compilar esse programa também é necessário linkar com uma biblioteca (cppunit). Para quem estiver curioso esse programa é um exemplo que achei na internet sobre como usar a biblioteca para automação de testes `cppunit`. A organização das dependências do programa é a seguinte: `ftest.cpp` inclui apenas arquivos da biblioteca `cppunit`; `Fraction.h` não inclui ninguém; `fractiontest.h` inclui `Fraction.h` e arquivos da biblioteca `cppunit` e, por fim, os arquivos `.cpp` incluem apenas seus respectivos arquivos `.h`.

Para compilar esse programa é necessário digitar na linha de comando

```
g++ Fraction.h Fraction.cpp fractiontest.h fractiontest.cpp ftest.cpp\
-I/diretorio_onde_se_encontram_os_arquivos_.h_da_biblioteca \
-L/diretorio_onde_se_encontra_a_biblioteca_cppunit -lcppunit -o ftest
```

Para fazer o mesmo com usando o comando `make`, vamos escrever um arquivo chamado `Makefile` no diretório onde estão os programas. O conteúdo do arquivo será[1]

[1]Todo comando em um target deve ser iniciado com um TAB então cuidado com espaços em branco.

```
all
<TAB> g++ Fraction.h Fraction.cpp fractiontest.h fractiontest.cpp ftest.cpp\
<TAB>-I/diretorio_onde_encontrar_os_arquivos_.h_da_biblioteca\
<TAB> -L/diretorio_onde_encontrar_a_biblioteca_cppunit -lcppunit -o ftest
```

Agora podemos compilar o programa apenas digitando `make`[2]

Isso evita a enorme linha de comando para recompilar o programa, mas ainda temos o problema de precisar compilar todos os arquivos com menor modificação em um deles. Isso é porque no nosso makefile ainda não dizemos para o make quem depende de quem.

Vamos então modificar nosso makefile para

```
all: ftest

ftest: ftest.o Fraction.o fractiontest.o
<TAB> g++ -o ftest ftest.o Fraction.o fractiontest.o -L/diretorio_onde_encontrar_a_biblioteca_cppunit -lcppunit

Fraction.o: Fraction.cpp Fraction.h
<TAB> g++ -c Fraction.cpp -I/diretorio_onde_encontrar_os_arquivos_.h_da_biblioteca

fractiontest.o: fractiontest.h fractiontest.cpp Fraction.h
<TAB> g++ -c fractiontest.cpp -I/diretorio_onde_encontrar_os_arquivos_.h_da_biblioteca

ftest.o: ftest.cpp
<TAB> g++ -c ftest.cpp -I/diretorio_onde_encontrar_os_arquivos_.h_da_biblioteca

clean
<TAB> -rm -f *.o ftest *~
```

Agora, o target all não possui mais nenhum comando e apenas depende do target ftest, que por sua vez depende dos targets Fraction.o, fractiontest.o e ftest.o. Se modificarmos apenas o arquivo ftest.cpp, então o target ftest.o será refeito seguido do target ftest. Se modificarmos o arquivo fractiontest.h então fractiontest.o será refeito, seguido de ftest e assim por diante. Note que criamos também um target chamado "clean" que não depende de ninguém e não cria nenhum arquivo. Ele apenas executa o comando que "rm -f *.o ftest * " que é bastante conveniente quando queremos apagar esses arquivos (bastando digitar make clean)[3].

Note que repetimos algumas coisas em vários targets diferentes. Isso é trabalhoso e pode levar a erros. No entanto o make nos fornece um recurso interessante que é a utilização de variáveis (ou macros). Assim, podemos reescrever nosso makefile como:

```
CPPUNIT_PATH=/diretorio_onde_esta_o_cppunit
# supondo que os arquivos .h da biblioteca estejam no diretório include e a biblioteca
INCLUDE_DIR=$CPPUNIT_PATH)/include
# esteja em um diretório lib ambos dentro do diretório raiz onde se encontra o cppunit
LIB_DIR=$(CPPUNIT_PATH)/lib
LIBS=-lcppunit

CPPFLAGS=-I$(INCLUDE_DIR)
LDFLAGS=-L$(LIB_DIR) $(LIBS)

all: ftest

ftest: ftest.o Fraction.o fractiontest.o
<TAB> g++ -o ftest ftest.o Fraction.o fractiontest.o $(LDFLAGS)

Fraction.o: Fraction.cpp Fraction.h
<TAB> g++ -c Fraction.cpp $(CPPFLAGS)

fractiontest.o: fractiontest.h fractiontest.cpp Fraction.h
<TAB> g++ -c fractiontest.cpp $(CPPFLAGS)
```

[2]Na verdade é "`make nome_do_alvo`", mas por omissão se não for especificado um alvo o `make` executa o primeiro alvo.

[3]O "-" antes do comando rm diz para o make ignorar erros, como é o caso quando não há arquivos para apagar.

```
ftest.o: ftest.cpp
<TAB> g++ -c ftest.cpp $(CPPFLAGS)

clean
<TAB> -rm -f *.o ftest *~
```

Agora temos um makefile organizado e nosso programa pode ser corretamente compilado observando as dependências entre os arquivos para não compilar o que não é necessário.

2.1 Melhorando o makefile

No makefile anterior podemos ver que o comando para compilar os arquivos .cpp em arquivos .o é sempre o mesmo. E se precisarmos criar mais arquivos no projeto, o que fazer? A resposta curta é: criar outro target e atualizar as dependências dos demais targets se necessário, mas isso está longe de ser a solução ideal. Não é nada agradável ficar criando novos targets para cada arquivo incluído e, principalmente, saber quem depende de quem.

Quando o número de arquivos cresce fica bastante complicado organizar todas as dependências. Você precisa atualizar o makefile não apenas quando inclui ou exclui um arquivo, como também quando inclui ou exclui um "#include" em um dos arquivos.

Vamos resolver o primeiro problema! Para isso, modificamos nosso makefile para:

```
CPPUNIT_PATH=/diretorio_onde_esta_o_cppunit
INCLUDE_DIR=$CPPUNIT_PATH)/include
LIB_DIR=$(CPPUNIT_PATH)/lib
LIBS=-lcppunit

CPPFLAGS=-I$(INCLUDE_DIR)
LDFLAGS=-L$(LIB_DIR) $(LIBS)

CPPSOURCES = $(wildcard *.cpp)

all: ftest

ftest: $(CPPSOURCES:.cpp=.o)
<TAB> g++ -o $@ $^ $(LDFLAGS)

%.o: %.cpp
<TAB> g++ -c $< $(CPPFLAGS) -o $@

clean
<TAB> -rm -f *.o ftest *~

remade:
<TAB> $(MAKE) clean
<TAB> $(MAKE)
```

A linha

```
CPPSOURCES = $(wildcard *.cpp)
```

atribui a variável CPPSOURCES todos os arquivos com a extensão .cpp no diretório atual separados por espaço. Usamos então a variável CPPSOURCES como dependência do target ftest, mas com a substituição da extensão .cpp por .o (já que ftest depende dos arquivos objeto e não dos arquivos fonte).

No comando do target ftest encontramos uma variável especial, o "$^". Essa variável é substituída por todas as dependências do target. Dessa forma, o target ftest tem o mesmo efeito que no makefile anterior, mas se acrescentarmos algum novo arquivo .cpp no projeto não precisaremos mudar nada para o target ftest.

O target clean continua o mesmo enquanto que o target remade que acrescentei é apenas uma maneira conveniente de recompilar todo o projeto se desejado. Note que eu não uso o comando make diretamente, mas sim a variável especial $(MAKE). Isso garante que se eu usar alguma opção de linha de comando para o make quando executar esse

makefile que estamos criando, os "makes internos" também usaram as mesmas opções. Então, sempre que chamarem o make dentro de algum comando no makefile usem a variável $(MAKE) ao invés de chamar o make diretamente.

Já o target %.o é o que considero o mais interessante. Com esse target estamos "ensinando" ao make como compilar qualquer arquivo .cpp em um arquivo objeto .o. Dessa forma não precisamos nos preocupar quando acrescentamos novos arquivos no projeto pois o make saberá como compilá-los. Note que no comando usamos duas variáveis especiais (também chamadas de variáveis automáticas). A variável $< é substituída pela primeira dependência do target e nosso target genérico %.o depende apenas de seu arquivo .cpp correspondente. Já a variável $@ é substituída pelo nome do target. Com isso, o make compilará o arquivo Fraction.cpp em um arquivo objeto chamado Fraction.o, fractiontest.cpp em fractiontest.o e assim por diante. Como nosso target ftest depende de todos os arquivos.o (um para cada arquivo .cpp), então sempre que o make precisar do ftest ele vai antes recompilar todos os targets .o necessários.

E quais arquivos .o devem ser recompilados? Eis o único problema de nosso target genérico %.o. Ele depende apenas do arquivo .cpp correspondente. Ou seja, se modificarmos Fraction.h, o target Fraction.o não será recompilado pois não estamos dizendo para o make que Fraction.o também depende de Fraction.h. Como podemos deixar então esse gerenciamento de dependências correto e automático?

Para resolver esse problema pediremos ajuda aos universitários..., quer dizer, ao compilador. Afinal, quem melhor que ele para nos dizer de qual arquivo um certo .cpp depende? Para isso usaremos as opções -MM e -MD do g++. Se executarmos o comando:

```
g++ Fraction.cpp -MM $(CPPFLAGS)
```

o `g++` retorna a linha:

```
Fraction.o: Fraction.cpp Fraction.h
```

Olha aí! Já está até com o "look" de um target de makefile. Acrescentando a opção -MD, ao invés do g++ imprimir essa informação no terminal ele criará um arquivo chamado Fraction.d contendo exatamente essa linha. Podemos então incluir esse arquivo no nosso makefile para que o make saiba de quais arquivos o target Fraction.o depende. Com posse dessa nossa nova arma, nosso makefile se torna:

```
CPPUNIT_PATH=/diretorio_onde_esta_o_cppunit
INCLUDE_DIR=$CPPUNIT_PATH)/include
LIB_DIR=$(CPPUNIT_PATH)/lib
LIBS=-lcppunit

CPPFLAGS=-I$(INCLUDE_DIR)
LDFLAGS=-L$(LIB_DIR) $(LIBS)

CPPSOURCES = $(wildcard *.cpp)

all: ftest

ftest: $(CPPSOURCES:.cpp=.o)
<TAB> g++ -o $@ $^ $(LDFLAGS)

%.o: %.cpp
<TAB> g++ -c $< $(CPPFLAGS) -o $@

clean
```

```
    <TAB> -rm -f *.o ftest *~

    remade:
    <TAB> $(MAKE) clean
    <TAB> $(MAKE)

    -include $(CPPSOURCES:.cpp=.d)

    %.d: %.cpp
    <TAB> g++ $< -MM -MD $(CPPFLAGS)
```

Ensinamos então ao make como criar os arquivos .d e incluímos esses arquivos (lembrando que o sinal "-" antes do include diz para o make ignorar erros, ou seja, caso o arquivo .d ainda não tenha sido criado).

Agora o make sabe de quais arquivos cada .o depende e se modificarmos Fraction.h o target Fraction.o será corretamente recompilado.

Mas e se modificarmos o Fraction.h e acrescentarmos a linha "#include nova_dependencia.h"? Claro que o target Fraction.o será recompilado, mas o arquivo Fraction.d não será atualizado com a nova dependência, pois na nossa regra para criar os arquivos .d dizemos que ele depende apenas do arquivo .cpp correspondente. Para resolver esse último problema vamos novamente pedir ajuda ao compilador. O ideal mesmo seria se ao invés de o arquivo Fraction.d conter a linha:

```
Fraction.o: Fraction.cpp Fraction.h
```

ele contivesse a linha:

```
Fraction.o Fraction.d : Fraction.cpp Fraction.h
```

Ou seja, dizer que o arquivo .d também depende de todas as dependências do arquivo .o. No manual do make existe um exemplo que usa o sed para fazer essa modificação, mas descobri que existe uma opção do g++ bem mais prática. Vamos modificar nosso target para arquivos .d como segue:

```
%.d: %.cpp
    <TAB> g++ $< -MM -MT '$*.o $*.d ' -MD $(CPPFLAGS)
```

A opção -MT nos permite especificar o nome do target que o g++ gera (ao invés do padrão arquivo.o), enquanto que a variável especial $* é substituída pelo que foi "casado" pelo % na nossa regra no makefile (ou seja, o nome do target sem a extensão .o). Com isso os nossos arquivos .d gerados terão a forma:

```
arquivo.o arquivo.d : dependências
```

e nosso problema estará resolvido.

Agora temos um makefile bastante genérico que pode ser usado em outro projeto modificando apenas umas poucas linhas e que gerencia automaticamente todas as dependências.

Nosso makefile completo é:

```
CPPUNIT_PATH=/diretorio_onde_esta_o_cppunit
INCLUDE_DIR=$CPPUNIT_PATH)/include
LIB_DIR=$(CPPUNIT_PATH)/lib
LIBS=-lcppunit
```

```
CPPFLAGS=-I$(INCLUDE_DIR)
LDFLAGS=-L$(LIB_DIR) $(LIBS)

CPPSOURCES = $(wildcard *.cpp)

all: ftest

ftest: $(CPPSOURCES:.cpp=.o)
<TAB> g++ -o $@ $^ $(LDFLAGS)

%.o: %.cpp
<TAB> g++ -c $< $(CPPFLAGS) -o $@

clean
<TAB> -rm -f *.o ftest *~

remade:
<TAB> $(MAKE) clean
<TAB> $(MAKE)

-include $(CPPSOURCES:.cpp=.d)

%.d: %.cpp
<TAB> g++ $< -MM -MT '$*.o $*.d ' -MD $(CPPFLAGS)
```

Tudo isso é apenas uma parte do poder do make, mas é o suficiente pra se fazer muita coisa. É possível, por exemplo, criar um target para gerar a documentação do programa com o doxygen, ou usar o make para trabalhar com o latex e bibtex, etc..

Agora você não precisa de uma IDE completa para programar. Mesmo usando apenas o gedit será possível compilar facilmente um programa composto de vários arquivos de dentro do gedit (ele apenas chama o make para compilar).

Para quem ainda não cansou, tem mais uma dica que pode ser útil. Por padrão quando se roda o make ele imprime ("echoa") na tela os comandos que estão sendo executados, o que pode resultar em mais informação do que você gostaria. Para resolver esse problema basta acrescentar @ antes dos comandos que você não quer que sejam impressos. Podemos então reescrever o makefile como:

```
CPPUNIT_PATH=/diretorio_onde_esta_o_cppunit
 INCLUDE_DIR=$CPPUNIT_PATH)/include
 LIB_DIR=$(CPPUNIT_PATH)/lib
 LIBS=-lcppunit

 CPPFLAGS=-I$(INCLUDE_DIR)
 LDFLAGS=-L$(LIB_DIR) $(LIBS)

 CPPSOURCES = $(wildcard *.cpp)

 all: ftest

 ftest: $(CPPSOURCES:.cpp=.o)
 <TAB> @echo Criando arquivo executável: $@
 <TAB> @g++ -o $@ $^ $(LDFLAGS)
```

```
%.o: %.cpp
<TAB> @echo Compilando arquivo objeto: $@
<TAB> @g++ -c $< $(CPPFLAGS) -o $@

clean
<TAB> @echo Limpando arquivos
<TAB> -@rm -f *.o ftest *~

remade:
<TAB> $(MAKE) clean
<TAB> $(MAKE)

-include $(CPPSOURCES:.cpp=.d)

%.d: %.cpp
<TAB> @g++ $< -MM -MT '$*.o $*.d ' -MD $(CPPFLAGS)
```

Assim, ao compilar o programa inteiro o make apenas escreverá:

```
Compilando arquivo objeto: Fraction.o
Compilando arquivo objeto: fractiontest.o
Compilando arquivo objeto: ftest.o
Criando arquivo executável: ftest
```