

Содержание

- make** 3
- Синтаксис Makefile** 3
- Общая структура 3
- Инкрементная компиляция 3
- Фиктивные цели 4
- Использование переменных и комментариев 4
- Автоматические переменные 5
- Примеры** 5
- Демонстрация удобства использования MakeFile 5
- Универсальный MakeFile 6
- Удобство для пользователя 6

make

(<https://www.opennet.ru/docs/RUS/gnumake/>)

NTB: Сборка = компиляция + линковка

Если запустить `make` то программа попытается найти файл с именем по умолчанию `Makefile` в текущем каталоге и выполнить инструкции из него. Если в текущем каталоге есть несколько мейкфайлов, то можно указать на нужный вот таким образом: `make -f MyMakefile`

Работу `make` можно представить себе так: {Из чего делаем? (реквизиты)} → [Как делаем? (команды)] → {Что делаем? (цели)}

Синтаксис Makefile

Общая структура

Makefile состоит из набора правил, которые в свою очередь описываются:

- целями (то, что данное правило делает);
- реквизитами (то, что необходимо для выполнения правила и получения целей);
- командами (выполняющими данные преобразования).

В общем виде синтаксис `makefile` можно представить так:

```
# Индентация осуществляется исключительно при помощи символов табуляции,  
# каждой команде должен предшествовать отступ  
<цели>: <реквизиты>  
  <команда #1>  
  ...  
  <команда #n>
```

Инкрементная компиляция

Если файлов в проекте много, при каждой сборке весь проект компилируется полностью, а это долго. Решение - разделить компиляцию на два этапа: трансляция и линковка.

Теперь, когда внесены изменения в один из исходников, достаточно произвести его трансляцию, а затем произвести линковку всех объектных файлов. Такой подход называется **инкрементной компиляцией**. Для ее поддержки `make` сопоставляет время изменения целей и их реквизитов (используя данные файловой системы), благодаря чему самостоятельно решает какие правила следует выполнить, а какие можно просто проигнорировать:

```
main.o: main.c  
    gcc -c -o main.o main.c  
hello.o: hello.c
```

```
gcc -c -o hello.o hello.c
hello: main.o hello.o
gcc -o hello main.o hello.o
```

При первом запуске `make` попытается сразу получить цель `hello`, но для неё нужны `main.o` и `hello.o`, а их нет, так что `make` ищет правила для них и выполняет. Как только все реквизиты будут получены, `make` вернется к выполнению отложенной цели. (Отсюда следует, что `make` выполняет правила рекурсивно.)

Фиктивные цели

«фиктивные» (phony) цели позволяют осуществлять с помощью `make` не только сборку программы. Вот краткий список стандартных целей: `all` — является стандартной целью по умолчанию. При вызове `make` ее можно явно не указывать. `clean` — очистить каталог от всех файлов полученных в результате компиляции. `install` — произвести инсталляцию `uninstall` — и деинсталляцию соответственно.

Для того чтобы `make` не искал файлы с такими именами, их следует определить в `Makefile`, при помощи директивы `.PHONY`

```
.PHONY: all clean install uninstall

all: hello # сборка

clean: # очистка от файлов, созданных во время сборки
    rm -rf hello *.o
main.o: main.c
    gcc -c -o main.o main.c
hello.o: hello.c
    gcc -c -o hello.o hello.c
hello: main.o hello.o
    gcc -o hello main.o hello.o
install: # инсталяция
    install ./hello /usr/local/bin
uninstall: # деинсталяция
    rm -rf /usr/local/bin/hello
```

ВАЖНО: `clean` необходим, что бы принудительно пересобрать проект с нуля, так как если реквизит `hello` уже существует, `make` его переделывать не будет. Однако, в данном случае достаточно руками удалить `hello` и запустить сборку, что бы не пересобрать все объектики.

Использование переменных и комментариев

Переменные - удобный способ учесть возможность того, что проект будут собирать другим компилятором или с другими опциями.

```
CC=g++ # Это комментарий, который говорит, что переменная CC указывает компилятор,
используемый для сборки
```

```
CFLAGS=-c -Wall # Это еще один комментарий. Он поясняет, что в переменной CFLAGS лежат флаги, которые передаются компилятору
```

```
hello: main.o
    $(CC) main.o -o prog

main.o: main.cpp
    $(CC) $(CFLAGS) main.cpp
```

По умолчанию make станет выполнять самое первое правило, если цель выполнения не была явно указана при вызове:

```
$ make <цель>
```

Автоматические переменные

- \$@ Имя цели обрабатываемого правила
- \$< Имя первой зависимости обрабатываемого правила
- \$^ Список всех зависимостей обрабатываемого правила

подробнее [тут](#)

Примеры

Демонстрация удобства использования MakeFile

Пример компиляции руками:

```
# main.cpp - главный файл
# functions.h - прототипы всех ф-ий
# factorial.cpp - инcludes functions.h и определяет реализацию тамошних ф-ий
# hello.cpp -
g++ main.cpp hello.cpp factorial.cpp -o prog
```

Долго каждый раз писать. Да и при разрастании проекта можно запутаться. Автоматизируем. Для нашего примера мейкфайл будет выглядеть так:

```
all:
    g++ main.cpp hello.cpp factorial.cpp -o hello
```

Использовать несколько целей в одном мейкфайле полезно для больших проектов. Это связано с тем, что при изменении одного файла не понадобится пересобирать весь проект, а можно будет обойтись пересборкой только измененной части. Пример:

```
all: hello

hello: main.o factorial.o hello.o
```

```
g++ main.o factorial.o hello.o -o hello

main.o: main.cpp
g++ -c main.cpp factorial.o: factorial.cpp g++ -c factorial.cpp

hello.o: hello.cpp
g++ -c hello.cpp

clean:
rm -rf *.o hello
```

Теперь у цели `all` есть только зависимость, но нет команды. В этом случае `make` при вызове последовательно выполнит все указанные в файле зависимости этой цели.

Еще добавилась новая цель `clean`. Она традиционно используется для быстрой очистки всех результатов сборки проекта. Очистка запускается так: `make -f Makefile-2 clean`

Универсальный MakeFile

```
CC=g++
CFLAGS=-c -Wall
LDFLAGS=
SOURCES=main.cpp hello.cpp factorial.cpp
OBJECTS=$(SOURCES:.cpp=.o)
EXECUTABLE=prog

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
$(CC) $(LDFLAGS) $(OBJECTS) -o $@

.cpp.o:
$(CC) $(CFLAGS) $< -o $@
```

Удобство для пользователя

установка при помощи исходников (любые, программы и библиотеки). Допустим, наш целевой пакет называется «hello».

1) Получаем код:

можно клонировать репозиторий:

```
git clone hello
cd hello
```

ИЛИ используем тарболл:

```
tar -xf hello-1.0.tar.xz
```

2) Сборка:

если хотим использовать голый make:

```
make PREFIX=/префикс-который-вы-хотите
```

если используем autotools:

```
./configure --prefix=/префикс-который-вы-хотите  
make
```

если используем cmake:

```
cmake -DCMAKE_INSTALL_PREFIX=/префикс-который-вы-хотите .  
make
```

Откуда точка в конце? Дело в том, что cmake'у нужно передавать путь к сорцам. А поскольку у нас не out of tree build, мы собираем здесь же. Сорцы находятся там же, где находимся мы. Поэтому точка, т. е. текущий каталог.

3) Установка:

Используя make:

```
make PREFIX=/префикс-который-вы-хотите install
```

Используя autotools или cmake:

```
make install
```

Сборка всегда осуществляется с обычными правами, а вот установка осуществляется с теми правами, которые нужны, чтобы записать в prefix.

From:

<https://wiki.radi0.cc/> - radi0wiki

Permanent link:

<https://wiki.radi0.cc/soft:make>

Last update: **2025/11/09 12:07**

