РАБОТА С УТИЛИТОЙ make

В статье на конкретных примерах рассматриваются варианты применения утилиты make как пользователями, так и разработчиками.

АНТОН ИВАНОВ

make для пользователей

Начнём с самого начала. Самое главное, но далеко не последнее применение утилиты make – это автоматическая сборка (компиляция) программ. И хотя большинство программ для Linux могут быть получены в двоичном (то есть в скомпилированном виде, в пакетах .deb, .tgz или .rpm), в процессе работы с Linux вам наверняка потребуется собрать программу самостоятельно из исходного кода.

Итак, вы загрузили архив с исходниками программы, скорее всего в виде tarball, т.е. сжатого tar-архива. Распространены две программы сжатия: gzip и bzip2 (последняя сжимает файлы лучше, но медленнее). Имена файлов таких архивов оканчиваются соответственно на .tar.gz (или .tgz) и tar.bz2 (или .tbz). Распакуйте архив командой:

```
tar xzvf имя_файла.tar.gz
или
tar xjvf имя_файла.tar.bz2
```

Другой вид распространения исходных кодов – в пакетах .src.rpm, в этом случае просто установите такой пакет как обычно, и исходники появятся в каталоге /usr/src/RPM/SOURCES.

Первым этапом в сборке программы является настройка командой ./configure (она определяет параметры системы и создает Makefile, наиболее подходящий для данной конфигурации), но нас интересует следующий шаг – сама компиляция при помощи команды make.

В большинстве случаев для сборки программы достаточно ввести только саму команду make, без каких-либо дополнительных параметров. По мере выполнения на экран будут выводиться команды, исполняемые при компиляции. Процесс этот может быть довольно долгим, в зависимости от скорости вашего компьютера.

Когда управление будет возвращено оболочке, т.е. появится приглашение ввода команд, внимательно просмотрите последние несколько строк, выведенных командой make. Если среди них есть «Error», или «Ошибка», это значит, что программа не скомпилировалась. Это может быть вызвано множеством причин: отсутствием каких-либо пакетов, ошибкой в программе, отсутствием прав записи в какой-либо файл или каталог и т. д. Еще раз внимательно перечитайте файлы README и INSTALL в каталоге с исходниками программы.

Обратите внимание, что в процессе компиляции может появляться множество предупреждений («warnings»). Они обычно вызываются небольшими несоответствиями стандарту Си; скорее всего, они не приведут к ошибке компиляции.

Если вы сомневаетесь, нормально ли скомпилировалась программа, сразу же после завершения работы make выполните команду:

```
echo $?
```

На экран будет выведен код возврата программы. Он показывает, успешно ли завершилось выполнение программы. Если он равен нулю, то всё нормально, если отличен от нуля – произошла ошибка.

После команды make можно указывать цели сборки. Цель – это то, что нужно сделать программе make. В большинстве случаев, если цель не указана, то происходит сборка программы. Если выполнить команду:

```
make install
```

то произойдет установка программы. Обратите внимание, что при выполнении этой команды файлы скорее всего будут скопированы в каталоги, в которые доступ на запись для простых пользователей закрыт. Поэтому вам нужно либо получить права суперпользователя (вы ведь не работаете постоянно под root, правда?), либо использовать программу su:

```
su -c make install
```

Иногда, чтобы освободить место на диске, или при перекомпиляции требуется удалить результаты предыдущей сборки – объектные файлы (*.o), двоичные файлы программы и другие временные файлы. Для этого нужно выполнить команду:

```
make clean
```

Эта команда удалит вышеуказанные файлы. Операция затронет только файлы в каталоге исходников, т.е. уже установленные в системные каталоги при помощи команды make install файлы затронуты не будут.

Если вы хотите привести исходники программы к первоначальному состоянию, т.е. удалить не только двоичные файлы, но и config.log, Makefile и другие, созданные скриптом ./configure, введите:

```
make distclean
```

Теперь для повторной сборки программы вам нужно будет снова запустить ./configure.

И наконец, чтобы отменить действие команды make install, т.е. удалить установленную программу, нужно, как нетрудно догадаться, выполнить:

```
make uninstall
```

Как и для make install, этой команде могут потребоваться права root.

Иногда при запуске make вы можете получить такие сообщения об ошибке:

```
make: *** Не заданы цели и не найден make-файл. Останов.
make: *** Нет правила для сборки цели 'uninstall'. Останов.
```

Это связано с тем, что при запуске утилита make ищет в текущем каталоге файл с именем Makefile или makefile (различный регистр символов). Если его нет, выдается первое сообщение. Что делать в таком случае? В первую очередь проверьте, не забыли ли вы запустить скрипт ./configure, ведь именно он создает Makefile. Если при выполнении ./configure произошла ошибка, Makefile не будет создан! Во-вторых, проверьте, нет ли в каталоге с исходниками какого-то другого подобного файла:

```
ls Makefile*
```

Например, вы можете получить список файлов типа Makefile.linux, Makefile.hpux, Makefile.macosx. Это значит, что для компиляции одной и той же программы на разных платформах автор предусмотрел несколько вариантов Makefile. Другой вариант – если вы загрузили исходный код из CVS, файл будет назван Makefile.cvs. Учтите, что Makefile.in и Makefile.am не относятся к Makefile, это всего лишь заготовки для создания Makefile. Итак, если вы нашли нужный Makefile, запустите make с ключом -f, например, так:

```
make -f Makefile.cvs
```

Второе из приведенных выше сообщений об ошибке может появляться, если автор Makefile не предусмотрел такую цель (в данном случае – uninstall, т.е. автоматическое удаление программы невозможно), или, опять же, если отсутствует Makefile.

Вы начали компилировать программу, но вдруг... Ошибка 1. Что делать? Для начала еще раз перечитайте документацию к программе. Если ситуация не прояснилась, попробуйте запустить make с ключом -i, означающим «игнорировать все ошибки»:

```
make -i
```

Вероятность того, что программа соберется правильно, все-таки есть, особенно, если ошибка была при обработке несущественных файлов, таких как файлы документации. Если же ничего не вышло – попробуйте задать вопрос на каком-нибудь форуме, например, на linux.org.ru.

Теперь давайте заглянем внутрь Makefile. Попробуем его немного поизменять. Откройте в своем любимом текстовом редакторе, например, Makefile из исходного кода Linux (если у вас установлены исходники Linux, то это /usr/src/linux/Makefile). Посмотрите на самые первые строчки:

```
VERSION = 2
PATCHLEVEL = 6
SUBLEVEL = 1
EXTRAVERSION =
```

Это переменные, используемые в дальнейшем при сборке ядра. Поменяв их значения, например, так:

```
VERSION = 3
PATCHLEVEL = 0
SUBLEVEL = 0
EXTRAVERSION = topsecret
```

и пересобрав ядро, вы можете потом хвастаться перед друзьями командой uname -r, которая будет показывать доставшуюся вам сверхсекретную версию ядра (3.0.0.topsecret).

Если же говорить о практическом применении, то иногда вам может потребоваться немного изменить Makefile, скажем, для того, чтобы исключить компиляцию какой-либо программы из большого пакета утилит. Пример: вы загрузили свежую версию утилит KDE (пакет kdeutils). Но какая-то программа из этого пакета никак не компилируется, и make -i не помогает. Допустим, программой, на которой происходит ошибка, является kedit. Откройте kdeutils/Makefile. Теперь просто найдите все встречающиеся слова «kedit» (в редакторе vim для этого введите /kedit) и удалите их:

```
TOPSUBDIRS =  <...> kdf kedit kfloppy <...>
```

Это список подкаталогов с исходным кодом различных программ. Утилита make по очереди заходит в каждый из них и выполняет Makefile, находящийся в этом подкаталоге. Удалите слово kedit, сохраните файл и снова запустите make. Вы увидите, что сборка kedit будет пропущена.

make для разработчиков

Пока ваша программа состоит из одного файла, проще всего компилировать ее командой:

```
cc main.c
```

Теоретически можно и всю программу поместить в один файл. Однако принято разделять код на несколько файлов, в больших проектах их десятки и сотни.

Для примера создадим простую программу на C, вычисляющую квадрат числа. Она будет состоять из двух файлов: главного (main.c) и kvadrat.c, содержащего функ-

цию возведения в квадрат. Ниже приведен исходный код этой программы:

```
main.c:

#include <stdio.h>

int main()
{
    int a, rslt;

    printf("Program calculates the square of a number.\n");
    printf("Please enter an integer: ");
    scanf("%d", &a);

    rslt = kvadrat(a);
    printf("%d x %d = %d\n", a, a, rslt);

    return 0;
}

kvadrat.c:

int kvadrat(number)
{
        return number * number;
}
```

Заранее скажу, что и эту программу можно скомпилировать довольно быстро без помощи Makefile командой cc *.c. Но мы все будем делать «по-правильному».

Сборка программы у нас будет разбита на два этапа: препроцессинг и компиляция, т.е. преобразование исходного кода в объектный код (файлы *.o), и сборка и линковка, т.е. объединение .o-файлов в один исполняемый файл и подключение динамических библиотек. Первый этап выполняется командой «gcc -c -o», второй – «gcc -o». gcc при запуске пытается осуществить все этапы сборки программы; параметр -c означает, что нужно только получить объектные файлы.

Итак, как вы уже знаете, утилита make ищет файл с именем Makefile. Создайте его.

Makefile состоит из набора т.н. правил. Они, в свою очередь, состоят из трех частей: цель (то, что должно быть получено в результате выполнения правила); зависимости (что требуется для получения цели); команды, которые должны быть выполнены для сборки цели. Общий синтаксис правил таков:

```
цель: зависимость_1 зависимость_2 зависимость_N
      команда_1
      команда_N
```

Давайте рассуждать логически. Нам требуется собрать программу, пусть имя ее исполняемого файла будет «kv». У нас два файла с исходным кодом, каждый из них должен быть преобразован в .o-файл (принято называть объектные файлы так же, как и исходные файлы, но с расширением .o; в нашем случае это будут main.o и kvadrat.o). Именно они необходимы для сборки исполняемого файла, значит, они являются зависимостями. Ну а про команды уже было сказано выше. Итак, Makefile должен выглядеть так:

```
kv: main.o kvadrat.o
    cc -o kv main.o kvadrat.o
    strip kv

main.o: main.c
    cc -c -o main.o main.c

kvadrat.o: kvadrat.c
    cc -c -o kvadrat.o kvadrat.c
```

Здесь нужно обратить внимание на несколько особенностей:

- параметр -o у cc указывает имя файла, получаемого в результате операции;
- в качестве зависимостей у целей main.o и kvadrat.o указаны соответствующие им файлы исходного кода, но это вовсе не обязательно делать. Если, скажем, файл main.c будет отсутствовать, cc и так выведет сообщение об этом. Но главная причина, по которой вам не нужно лениться и всегда указывать эту зависимость, описана чуть ниже;
- при сборке конечной цели вызывается не только cc, но и программа strip. Она позволяет зачастую значительно сократить размер выполняемого файла, удалив из него отладочную информацию.

Для понятности я опишу логику работы утилиты make с этим Makefile: сначала проверяется наличие файла main.o – первой зависимости у цели kv. Если его нет, то выполняется его компиляция, если есть, то проверяется, не был ли изменен файл main.c с момента последней его компиляции (вот зачем нужно указывать main.c как зависимость для main.o!). Если он был изменен, он компилируется заново. Если же он не был изменен, то просто пропускается; таким образом при повторной сборке можно значительно сократить затрачиваемое время.

Далее та же проверка происходит и для kvadrat.o, и как только будут получены main.o и kvadrat.o, включающие все изменения в исходном коде, начнется сборка исполняемого файла.

Итак, сохраните Makefile и запустите make. Должны появиться два .o-файла и сам исполняемый файл kv. Для проверки запустите его:

```
./kv
```

Теперь давайте добавим в Makefile еще одну цель: clean. При вводе make clean должны удаляться три полученных в результате компиляции файла. Допишите в конец Makefile такие строки:

```
clean:
        rm -f *.o kv
```

Сохранив Makefile, введите make clean. Все три двоичных файла будут удалены.

Теперь создадим последнюю цель – install. Она будет служить для копирования двоичного файла программы в системный каталог.

Тут есть несколько вариантов. Во-первых, можно сделать так, чтобы при выполнении make install проверялось, была ли уже скомпилирована программа, и если нет, то перед установкой выполнялась бы ее компиляция (логично, не правда ли?). Это можно сделать, указав kv в качестве зависимости. Во-вторых, для копирования программы в системный каталог можно использовать или традиционный метод – утилиту cp, а можно – специально предназначенную для этого программу install. Она может при

установке заодно изменять права доступа к программе, удалять из нее отладочную информацию (но мы это уже предусмотрели при помощи команды strip); нам же понадобится такая возможность, как автоматическое создание каталогов по указанному пути. Возможно, это звучит запутанно, так что я поясню. Мы установим программу в каталог /opt/kv/ (каталог /opt предназначен для хранения пользовательских программ, необязательных для функционирования системы). Разумеется, этого каталога еще не существует, поэтому, если бы мы использовали для установки команду «cp kv /opt/kv/», это привело бы к ошибке. Если же использовать команду install с ключом -D, она автоматически создаст все отстутствующие каталоги. Кроме того, нам нужно будет сделать так, чтобы никто, кроме root, не мог выполнять программу, а смог только считывать ее файл (права доступа r-xr--r--, или 544 в восьмеричном виде).

Добавьте в конец Makefile следующие строки:

```
install: kv
    install -D -m 544 kv /opt/kv/kv
```

Запустите make install, затем введите команду:

```
ls -gh /opt/kv/kv
```

чтобы убедиться в правильности установки.

После параметра -m у команды install указываются права доступа, предпоследним параметром – файл, который нужно скопировать, последним параметром является каталог назначения с именем файла (т.е. исполняемый файл можно установить под другим именем, введя, например, /opt/kv/program_name в качестве последнего параметра).

Теперь поговорим о переменных. Представьте, что вам нужен Makefile для более сложного проекта, где в каталог назначения копируется одновременно несколько файлов. Вы укажете путь установки для каждого из них, например, так:

```
install <параметры> /opt/kv/file1
install <параметры> /opt/kv/file2 и т.д.
```

Но вдруг вам потребовалось изменить путь установки. Вам придется поменять параметр у каждой команды. И хотя это можно сделать относительно быстро при помощи команды «заменить» в текстовом редакторе, проще всего определить переменную один раз в начале Makefile и потом использовать ее (при необходимости изменяя только ее значение). Добавьте в начало Makefile определение переменной:

```
INSTALL_PATH = /opt/kv/
```

И затем измените команду install таким образом:

```
install -D -m 544 kv $(INSTALL_PATH)/kv
```

На что здесь нужно обратить внимание:

- в объявлении переменной указывается просто ее имя, а когда вы ее используете – знак $ и имя;
- если имя переменной состоит более чем из одного символа, при ее использовании нужно заключать такое имя в скобки;
- нужно или нет ставить косую черту в значении переменной (/opt/kv/ или /opt/kv)? Ответ: лучше перестраховаться и поставить. Как вы можете заметить, в команде установки косая черта идет подряд дважды: /opt/kv//kv (одна косая черта из значения переменной, другая – из записи $(INSTALL_PATH)/kv). Это не приведет к ошибке – вы можете поставить символ «/» хоть десять раз подряд. А вот если забудете поставить ее хотя бы один раз, то результаты, ясное дело, будут отличаться от ожидаемых.

Теперь рассмотрим такой момент: допустим, нужно выполнить некоторую команду и присвоить переменной выводимое этой командой значение. Например, если для разных версий ядра Linux существуют разные версии вашей программы. В таком случае при выполнении установки программы на системе с ядром 2.6.1 можно установить ее в каталог /opt/kv/2.6.1/, а если система запущена с ядром 2.4.24 – в каталог /opt/kv/2.4.24/. Версию ядра можно определить при помощи команды uname -r (попробуйте выполнить ее, чтобы посмотреть вашу версию ядра). Вопрос в том, как же передать полученное значение переменной? Очень просто:

```
INSTALL_PATH = /opt/kv/`uname -r`/
```

Теперь установка будет производиться в каталог /opt/kv/версия_вашего_ядра/. Обратите внимание на косые кавычки: они вводятся при помощи Shift+~ (крайняя левая клавиша в ряду кнопок с цифрами).

И последнее: а ведь make может использоваться не только для компиляции программ! Например, можно использовать Makefile для сборки документации, да и вообще для автоматизации любой работы. Для примера можно создать программу, которая по команде make автоматически определяет, какие из трех файлов с именами «1», «2», «3» изменились, и при необходимости выполняет их резервное копирование в каталог backup, а по окончании работы выводит сообщение «Backup completed»:

```
backup: backup/1 backup/2 backup/3
        : Backup completed

backup/1: 1
        cp 1 ~/backup

backup/2: 2
        cp 2 ~/backup

backup/3: 3
        cp 3 ~/backup
```

Домашнее задание для вас: разберитесь, как работает этот пример, а к первому добавьте цель uninstall, так, чтобы при выполнении команды make uninstall установленная программа удалялась.

Если вас заинтересовала эта тема, прочитайте руководства «man install», «man make», а также GNU Make Manual по адресу http://www.gnu.org/software/make/manual/html_mono/make.html.gz (150 Кб). И главное – больше экспериментируйте!