Introduction to Make

Последнее изменение: Пт, 27 ноября, 09:56:46 2020

Ты можешь угостить меня кофе, пожалуйста

Я был бы признателен, если бы вы иногда покупали мне кофе *Примечание для читателей - нажатие на заголовок приведет к переходу к индексу и обратно.*

Содержание

Что такое сделать? Что делает команда?

Оригинальная документация Make от Feldman содержала прекрасное описание утилиты make. При редактировании кода - есть четы шага:

Подумать
Редактировать
Сделать
Тест

Другими словами, разве не было бы неплохо иметь программу, которая позаботится обо всех мелких деталях на шаге 3 для вас? *Make* - отличная программа, которая может помочь вам компилировать программы.

Но предположим, вы просто пишете сценарии оболочки? *Make* на самом деле не нужен, верно?

Что ж, пожалуйста, позвольте мне убедить вас в обратном. Я все время использую *make* со сценариями оболочки. И чем сложнее процесс, тем больше вероятность, что я создам *Makefile* для процесса.

Зачем использовать make для сценариев оболочки?

Некоторые из вас могут сказать себе, что «я просто пишу сценарии оболочки. Почему я должен использовать *make*?»Есть много причин, по которым я использую *make* для сценариев оболочки. Вот некоторые из моих причин:

- Мой рабочий процесс состоит из «шагов» или порядка выполнения.
- Сценарии оболочки содержат несколько команд, параметров и / или аргументов.
- Я хочу задокументировать то, что я сделал.
- Я работаю на удаленном компьютере и не имею доступа к среде GUI.
- Я работаю с командой, и я хочу поделиться командами оболочки, связанными с проектом, без необходимости заставлять друг членов команды устанавливать мои сценарии оболочки и псевдонимы.
- Я работаю с несколькими проектами, но хочу иметь единую методику управления.
- Я не хочу или не могу устанавливать дополнительные инструменты.

Всякий раз, когда я создаю каталог для нового проекта, даже если это несколько сценариев оболочки, я создаю *Makefile*. Лучшая документация - это исходный код, и *make* - отличное место для начала. И если я посещаю старый проект, просмотр файла *Makefile* освежает мою память о том, **что** я делал, **как** я использова сценарии и **почему** я делал определенные вещи. В конце концов, как только вы найдете оптимальную команд и аргументы, шаги в *Makefile* могут «увековечить» точные аргументы и последовательность операций.

Иногда я набираю длинные строки, которые выполняют сложный сценарий оболочки. Я хочу задокументировать это, но я не хочу создавать полноценный сценарий оболочки, особенно если это всего лишь одна строка. Я мог бы сделать это псевдонимом, но повторение этого сотни раз делает файл псевдонима громоздким. Я мог бы поместить это в текстовый файл для документирования / запоминания, но также хочу его выполнить. Я мог бы написать сценарий оболочки, но я хочу, чтобы им было легко поделиться членами команды. Я мог бы создать каталог с именем *scripts*, который содержит все мои рецепты, но тогда мне нужно заставить членов команды установить это в своей среде.

Или я мог бы использовать *make*. (римшот)

Отладка сценариев оболочки без make

Давайте рассмотрим простой рабочий процесс программирования без *make*.

Я мог бы просто ввести сценарий оболочки в одну строку, но если это сложно, и я, возможно, захочу сделать это снова, я должен держать его под рукой, поместив в сценарий оболочки.

Обычно у меня открыто два окна - одно редактирует файл, а другое «использует» файл. Например, я мог бы редактировать скрипт awk, bash, sed или python с помощью **моего любимого редактора ™**. А другое окно - это окно терминала, в котором запущена оболочка.

Я редактирую файл в одном окне, затем сохраняю редактирование. Затем я перехожу в другое окно и «запускаю» программу, обычно набирая стрелку вверх или«!!»команду истории. Или я использую редактирование командной строки для внесения изменений. Стандартная процедура работы.

Иногда вам приходится переключаться между двумя или более командами в окне командной строки. Это легк сделать с помощью оболочки, которая понимает вашу историю команд. Однако в вашем рабочем процессе могут быть шаги. И вы можете забыть шаг или, возможно, выполнить их в неправильной последовательности. Сделайте это неправильно, и ваш результат не будет правильным.

Возможно, точная команда оболочки, которую вы вводите, очень длинная и сложная, и вам нужно посмотреть страницу руководства, чтобы запомнить, что нужно. Или предположим, что вы хотите использовать различны комбинации команд при экспериментировании с результатами, например, изменить параметр для одного из аргументов. Это может быть сложным в многоэтапном процессе.

И тогда возникает классическая проблема выполнения вашей программы или скрипта, но вы забыли сказать редактору сохранить изменения. Мы все это делали.

Это проблемы, для которых создан *make*.

> Системные администраторы - вы хотите просмотреть файлы журналов и найти шаблоны и странные действия? Продолжайте читать.

Давайте начнем.

Входной файл по умолчанию для make - Makefile

Если вы просто введете команду

```
make
```

Вы получите сообщение об ошибке «make: ничего не нужно делать для "всех" ..» Это потому, что вы не предоставили «make» достаточно информации, и поскольку прямого ввода в мозг (пока) нет, он обычно получает эту информацию из файла с дьявольским именем «makefile» .

Чтобы быть более точным, команда «make» сначала ищет файл с именем «makefile», и если этот файл не существует, она затем ищет файл «Makefile». Обратите внимание, что второй вариант начинается с заглавно буквы.

> В большинстве случаев, когда программисты создают проект, они предоставляют файл «Makefile» вместо «makefile» по нескольким причинам. Обычно команда *ls* сортирует имена файлов таким образом, чтобы файлы, начинающиеся с заглавных букв, отображались перед файлами, начинающимися со строчных букв. Поэтому этот файл часто отображается в списке перед другими. По этой же причине люди предоставляют файл с именем «README» - чтобы сделать его более заметным.
> Вторая причина, по которой люди используют «Makefile» вместо «makefile», заключается в том, чтобы облегчить другому пользователю переопределение поведения по умолчанию. Если вы получаете исходный пакет с файлом «Makefile», который хотите изменить (например, для изменения каталога назначения), просто скопируйте «Makefile» в «makefile» и отредактируйте его. При этом исходный файл сохраняется в качестве резервной копии. Это также позволяет легко тестировать изменения.

Звучит как хорошая конвенция. Давайте начнем.

Мой первый Makefile

Makefile и целевые объекты

Давайте создадим очень простой *Makefile*. Отредактируйте файл и дайте ему имя «Makefile». Файл должен содержать следующее:

```
all:
        ./script1
```

Синтаксис важен и немного суетлив. Позвольте мне предоставить более точное описание синтаксиса, который в общих чертах является:

```
<target>: <prerequisites>
<tab>   <recipe>
<tab>   ...
```

Давайте разберем это.

Слово "«все»" - это название цели. Рецепт в нашем файле makefile «./script1» Предварительные «условия» необязательны. Я расскажу об этом позже. На данный момент рецепт «создает» цель, или, в нашем случае, вы выполняете «./script1,» чтобы сделать «все». Или, другими словами, когда вы вводите

```
make all
```

программа *make* выполняется «./script1» для вас.

Кстати, поздравляю с вашим первым *makefile*. Да, я знаю. Невпечатляющий. Давайте отталкиваться от этого.

Как следует из спецификаций, у вас может быть более одного рецепта для цели. Каждый рецепт выполняется путем передачи строки в оболочку для выполнения. Однако, и это сложная часть, перед каждым рецептом **должен** быть символ табуляции. *Обратите внимание, что перед рецептами необходимо использовать символ табуляции, а не пробелы.*

Если вы забудете это, *make* сообщит об ошибке, такой как:

```
Makefile:2: *** missing separator.  Stop.
```

В этом примере *make* сообщает имя файла и номер строки ошибки - в данном случае «2».

Если вы используете редактор, который заменяет табуляцию пробелами, это приведет к разрыву вашего файла *Makefile*. И если вы вырезаете и вставляете мои примеры, вам, возможно, придется заменить начальные пробелы на табуляцию.

Разобравшись с основами, мы готовы использовать *make* в более реалистичном смысле. Предположим, у ва есть три независимых сценария с именами script1, script2 и script3.

```
all: step1 step2 step3
step1:
        ./script1
step2:
        ./script2
step3:
        ./script3
```

Я добавил три новые цели и рецепты. Вы можете ввести команду «make step1» и т.д. для выполнения только одной из программ. Обратите внимание, однако, что мы добавили step1, step2 и step3 в качестве предварительного условия для цели «all». Если вы введете команду «make all» - *make*, будут построены все три цели. Это потому, что у цели «all» есть три подцели - step1, step2 и step3.

> Если вы не указываете цель *make*, она создает первую цель в файле. Очень часто первой цели присваивается имя «all», поэтому *make* создает все по умолчанию.

Если, например, вы являетесь системным администратором, которому приходится проверять несколько файлов журналов, и каждый раз вы выполняете одни и те же действия, вы можете поместить их все в Makefile

Вы можете использовать *make*, чтобы помочь запомнить сложные рецепты, связанные с проектом, и быть последовательными в операциях. Например, предположим, что нужно было сгенерировать отчет для нескольких проектов в нескольких каталогах. Команды оболочки могут отличаться, но вы можете использоват «make report» - то есть одну и ту же команду - для всех проектов, если вы настраиваете *Makefile* в каждом каталоге.

Или предположим, что вам нужно было выполнить журнал или анализ данных для нескольких проектов / программ. Вы можете использовать команду «make logs» для каждого проекта, без необходимости устанавливать или обмениваться сценариями между товарищами по команде.

И бывают случаи, когда вам приходится экспериментировать с длинной и сложной командой оболочки, чтобы заставить что-то работать, и как только это сделано, это сделано. Да, вы можете забыть об этом, но иногда неплохо записать эти длинные и сложные команды, чтобы вам было легче повторно использовать их в следующий раз. Например, я использовал команду «wget» с 8-10 параметрами командной строки, но я могу н помнить эти точные команды шесть месяцев спустя.

Итак, давайте научимся работать с файлом *Makefile* и добавим некоторые зависимости к рецептам.

Зависимости и порядок выполнения в файлах makefile

Давайте предположим, что эти программы / скрипты должны выполняться в определенном порядке, в данном случае step1, step2 и step3 в этой последовательности. Мы можем сделать это, внеся незначительные изменения в предыдущий *makefile*:

```
all: step3
step1:
        ./script1
        touch step1
step2: step1
        ./script2
        touch step2
step3: step2
        ./script3
        touch step3
clean:
        rm  step1 step2 step3
```

Обратите внимание, что мой «Makefile» создает три файла с именами step1, step2 и step3 с помощью команд *touch* . Это обновляет временную метку в файле, и если файл не существует, создается пустой файл.

Также обратите внимание, что цель *all* имеет только *step3* в качестве предварительного условия или зависимости. И если вы посмотрите на *step3*, у него есть *step2* в качестве зависимости. То есть, прежде чем *script3* будет выполнен, *step2* должен существовать. И *step2* создается при *выполнении* script2. Но этого не произойдет, пока не *будет создан step1*.

Другими словами, мы создали не просто набор шагов, а явную последовательность или порядок действий.

Make понимает эти зависимости и отслеживает их для вас. Если вы выполните команду «make», она выполни следующие команды

```
./script1
touch step1
./script2
touch step2
./script3
touch step3
```

Однако, если вы выполните *make* во второй раз, это ничего не даст, потому что все уже сделано. Другими словами, предварительные условия были выполнены.

Файлы step1, step2 и step3 должны быть удалены, если вы хотите снова выполнить три сценария. Вот почему я добавил цель с именем «clean», которая при выполнении удаляет эти файлы. Вы можете ввести команду «make clean;make», и три сценария будут выполнены снова.

> Цель «make clean» является общим соглашением в *makefiles*.

Использование входных и выходных файлов в качестве целевых объектов и зависимостей

Я использовал файлы *step1* и т. Д. В качестве целей. Это работает и просто, но может вызвать проблемы. Эти файлы могут не синхронизироваться с вашими скриптами и программами. Лучше иметь цели и зависимости, которые являются частью потока данных.

У нас есть три сценария выше. Давайте предположим, что они обычно соединяются вместе, вот так:

```
./script1 <data.txt | ./script2 | ./script3 >script3.out
```

Я собираюсь изменить *makefile*, чтобы выполнить приведенный выше сценарий, поэтапно:

```
all: script3.out
script3.out:  script2.out
        ./script3 <script2.out >script3.out
script2.out:  script1.out
        ./script2 <script1.out >script2.out
script1.out: data.txt
        ./script1 <data.txt >script1.out
clean:
        rm *.out
```

Это устраняет необходимость в этих файлах шагов «бездействия». Вместо этого мы используем файлы, которые содержат реальные данные. Если мы введем команду «make», скрипт будет выполнен

```
./script1 <data.txt >script1.out
./script2 <script1.out >script2.out
./script3 <script2.out >script3.out
```

Это ближе к реальному *makefile*, но в нем отсутствует важная функция. *Make* может работать с файлами данных, но его реальная сила связана с исходным кодом. В этом случае, если *script1* и т. Д. Является сценарием, мы хотим, Чтобы *make* понимал, что если сценарий изменится, выходные данные также изменятся. Таким образом, более полезный *makefile* будет выглядеть так:

```
all: script3.out
script3.out: ./script3 script2.out
```

```
	./script3 <script2.out >script3.out
script2.out: ./script2 script1.out
	./script2 <script1.out >script2.out
script1.out: ./script1 data.txt
	./script1 <data.txt >script1.out
clean:
	rm *.out
```

Если вы посмотрите на целевой «script3.out», вы увидите два предварительных условия. Как и раньше, у нас есть «script2.out», но мы также добавили «script3». То есть, если вы редактируете скрипт или меняете входны данные - вам придется перезапускать скрипт.

Теперь мы готовы. Эта версия обнаружит, когда программа (или сценарий) изменится, и когда это произойдет она повторно запустит этот шаг.

Не забывайте, что ваши скрипты могут иметь дополнительные зависимости. Например, если «script2» выполнил *awk*-скрипт с именем «script2.awk», вы можете добавить это в качестве зависимости для целевого объекта «script2.out» . *Make* не анализирует ваши файлы, он просто просматривает временные метки, поэтом вы должны добавить эти зависимости самостоятельно.

> Хотя вы могли бы просто объединить три сценария вместе, эта методология очень удобна при отладке, особенно если вы не знаете, в каком скрипте ошибка, и если выполнение скриптов занимает много времени. Он не только просто перезапускает минимальные шаги после изменения сценария, но и фиксирует все данные, которые обычно отбрасывались бы, если бы сценарии передавались вместе.

Как make узнает, когда нужно выполнить рецепт?

Утилита *make* использует временные метки и рецепты, чтобы определить, когда она должна восстановить цель. В приведенном выше примере он просматривает временные метки входного файла и исполняемой программы / скрипта, чтобы определить, требуется ли какое-либо действие. В *Make* также есть некоторые встроенные правила, которые он использует, но мы рассмотрим это в следующем уроке.

Если вы системный администратор, вы можете видеть, что это может быть полезно, потому что «make» обнаружит, если файл журнала изменится, и сгенерирует новые данные только в том случае, если это произойдет.

Интеграция make с вашим редактором

Интеграция вашего редактора с *make* может значительно повысить вашу эффективность. Сколько раз вы пытались исправить код, чтобы обнаружить, что забыли сохранить изменения? Вместо этого ваш редактор автоматически сохранит для вас файлы и, что еще лучше, автоматически проведет вас к строке в файле с ошибкой? *Вы хотите сделать это, банда.* Поехали.

Используя vi / vim и сделать

Прежде всего, я должен извиниться, что это всего лишь вступление к интеграции *vim / make*. Честно говоря, я использовал *vi* примерно за десять лет до выхода *vim 2.0*, и к тому времени я перешел на *Emacs*. Следовательно, я так и не освоил расширенные возможности *vim*. Но я сделаю все, что в моих силах, чтобы вы начали.

Для вас, пользователей *vim*, первый шаг, который вы, возможно, захотите сделать, это сопоставить функциональную клавишу с командой *make*. Один пример - добавить это в ваш файл *~/.vimrc* - в данном случае клавиша F9:

```
" You want to save files when you execute make. You have two choices. Either:
" (1) set autowrite whenever you execute :make
set aw
" or (2) Press F8 to save all open files
:map <f9> :wa
" and to execute make, press the F9 key:
" Press F9 to execute make
:map <f9> :make
```

Это правда, что привязка функциональной клавиши к make экономит всего несколько шагов, так как вы всегда можете ввести «:make» в *vi*.

Когда вы хотите выполнить make, вы должны убедиться, что ваши изменения сохранены в файловой системе Если вам нравится вариант 2, вы должны нажать функциональную клавишу F8. Если у вас есть вариант 1, установив автозапись, то это произойдет автоматически. Теперь, когда вы нажимаете функциональную

клавишу F9, *vim* выполнит команду *make*, позволит вам указать цель и аргументы и ждать, пока вы нажмете клавишу ENTER. Когда это будет сделано, *vim* покажет вам результаты.

Вы можете просмотреть результаты компиляции с помощью команды «:copen» (сокращение «:co»), которая открывает окно быстрого исправления. Если есть ошибки, которые *vim* может проанализировать, вы можете нажать ENTER и перейти к другим ошибкам, а также отредактировать файл. Например, если у вас есть ошибка в вашем *Makefile*, вы можете использовать окно quickfix, чтобы перейти к строке, которая вызвала ошибку. Команда «:cclose» закроет это окно, и вы можете переключить его с помощью команды«:cw» .

Vim позволяет определить переменную makeprg для выполнения make с параметрами командной строки или для использования какой-либо программы, отличной от make. Я видел один пример, когда кто-то изменил *makeprg* для выполнения компилятора Java. Лично я считаю, что это плохая идея. Прежде всего, если второй пользователь хочет повторить ваши шаги, он должен таким же образом изменить свой собственный загрузочный файл *vim*. Во-вторых, они также должны использовать *vim* вместо предпочитаемого редактора. Эти параметры принадлежат Makefile, который документируется как часть исходного кода, а не скрывается в личных предпочтениях.

Использование GNU Emacs с make

GNU Emacs имеет встроенную поддержку *make*. Однако вы можете захотеть сопоставить нажатие клавиши с командой *make*. Я использую следующее для сопоставления «Control-C m» и «Control-C M» с *make*.

```
;;; map Control-C m and Control-C M to make
(global-set-key \C-cm" 'compile)
(global-set-key "\C-cM" 'compile)
```

Иногда у меня случайно включается shift, поэтому я сопоставил как верхний , так и нижний регистр «M» для *make*.

Когда я нажимаю «Control-C m», Emacs спросит меня, хочу ли я сохранить какие-либо файлы, которые не был сохранены. Затем он запрашивает у меня команду *make* по умолчанию с аргументами. Я могу отредактировать это, и он запомнит это в следующий раз. Затем Emacs запустит команду make и передаст результаты в буфер с именем «*compilation *» . Если этот процесс займет некоторое время, вы все равно можете редактировать свои сценарии и данные, просматривая результаты. Статус обновляется в строке состояния, в которой указано «Compilation:exit» после завершения *make*. Я нахожу это удобным для длительных заданий.

Если вы повторите команду, и она все еще выполняется, Emacs спросит вас, хотите ли вы завершить текущую компиляцию. Команда «Control-C Control-K» завершит текущую компиляцию. Я делаю это часто, когда понимаю что я облажался и хочу повторно запустить задание компиляции.

После завершения компиляции вы можете нажать «Control-X `», что означает *command-next-error* . Это позволит прочитать ошибки в буфере компиляции, найти первую ошибку и перейти к строке в файле, которая вызвала ошибку. Если вы повторите эту команду, она перейдет к следующей ошибке, даже если она находится в другом файле. Используя это, вы можете быстро перемещаться по своим ошибкам, а затем перекомпилировать. Это не будет работать очень хорошо в программах, которые не генерируют ошибки, которые могут быть проанализированы.

Если вы имеете дело с несколькими «Make»-файлами, такими как вложенные каталоги или несколько проектов, команда *компиляции* будет выполняться в текущем каталоге файла, который вы активно редактируете.

Если вы используете Emacs в графическом режиме и редактируете «Makefile», в строке меню будут отображаться текущие команды *Makefile*. Я постараюсь описать их в более позднем уроке.

Я хотел убедиться, что вы поняли обычный рабочий процесс. Пришло время узнать, как настроить свой *Makefile*, чтобы быть более продуктивным пользователем.

Переменные в файлах Makefile

Make была ранней программой в истории систем Unix, и поддержка переменных необычна. Переменные обозначаются знаком доллара, но имя переменной состоит либо из одной буквы, либо из более длинного имени переменной, заключенного в круглые скобки. Текущие версии *make* также позволяют использовать фигурные скобки. Вот некоторые примеры переменных в рецепте:

```
all:
        printf "variable B is $B\n"
```

```
        printf "variable Bee is $(Bee)\n"
        printf "variable Bee is also ${Bee}\n"
```

Вы должны быть осторожны, потому что, если вы использовали «$Bee» в файле make, вы ссылаетесь на переменную «$ B» с добавленными к ней буквами «ee». Итак, если переменная $B имела значение «Whoop» то «$Bee» имеет значение «Whoopee» .

Следовательно, я всегда использую круглые скобки или фигурные скобки (не имеет никакого значения, какой из них вы используете) вокруг переменных, чтобы убедиться, что никто не путает «$ (B) ee» с «$ (Bee)»

Существуют специальные переменные, имена которых являются специальными символами. Я пока не собираюсь их освещать.

Включение знака доллара $ в Makefile

Поскольку знак доллара указывает на переменную, что вы делаете, если хотите, чтобы знак доллара остался покое? В этом случае просто используйте «$$» для обозначения одного знака доллара. Например, если вы хотите использовать регулярное выражение, содержащее знак доллара в файле *makefile*, вам нужно будет удвоить знак доллара. Предположим, вы хотите выполнить поиск по всем строкам, в которых последним символом было число, используя регулярное выражение «"[0-9] $"», образец рецепта будет:

```
grepthelines:
        grep '[0-9]$$' data.txt
```

Обратите внимание, что *make* интерпретирует строки и вычисляемые переменные, прежде чем передавать результаты в вашу оболочку. Оболочка видит только один «$». Обычно мета-символы в сильных кавычках оставляются в покое. Но *make* обрабатывает строки перед отправкой их в оболочку.

Установка переменных в файлах Makefile

Существует два способа установки переменных Makefile. Я не говорю о переменных оболочки или среды.

Установка переменной внутри файла Makefile

Настройка переменных проста; просто поместите их в строку (не начинающуюся с символа табуляции) со знаком равенства. Переменные могут ссылаться на другие переменные. Примером может быть

```
# Example of a Makefile with variables
OUTFILES = script1.out script2.out script3.out
TMPFILES = script1.tmp script2.tmp script3.tmp
TEMPORARYFILES = $(OUTFILES) ${TMPFILES)
# Which programs am I going to install?
PROGS = script1 script2 script3
# Note I am using a shell environment variable here
INSTALL = /usr/local/bin
clean:
        rm $(TEMPORARYFILES)
install:
        cp -i ${PROGS} ${INSTALL}
```

Переменная TEMPORARYFILES - это значение двух других переменных, объединенных в виде строк. Имеет смысл, что переменные *make* основаны на строках.

Установка переменных makefile в командной строке

В последнем примере, если вы выполнили команду «make install», *make* скопирует файлы в каталог */usr/local/bin*. Вы можете переопределить это в командной строке. Синтаксис таков

```
make target [variable=value ...]
```

Например, если вы ввели

```
make install INSTALL=~/bin
```

Затем *make* установит программы в ~/bin.

Вычисление переменных оболочки с помощью переменных Makefile

Когда вы определяете переменные, оболочка используется для оценки строки перед передачей результатов в *make*. Например, можно использовать следующие определения:

```
OUTFILES = *.out
INSTALL = $${HOME}/bin
```

ИСХОДЯЩИЕ ФАЙЛЫ будут равны всем файлам, которые соответствуют шаблону «*.out», в то время как переменная INSTALL основана на переменной среды HOME .

Не путайте переменные оболочки с переменными Makefile

Обратите внимание, что я использовал «$${HOME}» в предыдущем примере. Двойной знак доллара указывае *make* передать один знак доллара в оболочку, которая использует его для получения переменной среды.

Не совершайте ошибку, пытаясь установить переменную в рецепте и предполагая, что она работает так же, как определение. Вот пример с обоими:

```
#Define a variable in a Makefile
A = 123
all:
        A=456; printf "A = ${A}\n";
        A=456; printf "A = $${A}\n";
        printf "A = $${A}\n";
```

Это выведет

```
% make
A=456; printf "A = 123\n";
A = 123
A=456; printf "A = ${A}\n";
A = 456
printf "A = ${A}\n";
A =
```

При первом выполнении printf используется один $, поэтому используется определение в *Makefile* . Во второй раз используется двойной знак доллара, поэтому используется переменная оболочки, и печатается «456».

Третий printf выводит переменную оболочки «A» в виде пустой строки, поскольку она не определена. Каждая строка рецепта выполняется новой оболочкой. В этом примере выполняются три оболочки - по одной для каждой из строк, содержащих *printf*. И каждая оболочка имеет свой собственный набор переменных представления.

> Пожалуйста, обратите внимание, что когда рецепт состоит из нескольких строк, каждая строка выполняется со своей собственной оболочкой. Переменные, заданные в одной строке, не передаются во вторую строку. Также обратите внимание, что вы можете выполнять рецепты, которые запускают фоновое задание, заканчивая рецепт символом амперсанда.

Ошибки при выполнении делают

Есть несколько способов, которыми вы можете совершить ошибку и *заставить* пожаловаться.

Синтаксические ошибки в файле Makefile

Прежде всего, Make будет жаловаться на ошибки в файле makefile. Если у вас нет вкладки перед рецептом, о будет жаловаться

```
Makefile:linenumber: *** missing separator.  Stop.
```

Если вы попросите его создать цель, а он не знает, как это сделать, он скажет вам, что правила нет. Он может обнаруживать циклы в зависимостях как несколько других ошибок.

Ошибки во время выполнения

Второй тип ошибок возникает, когда *make* выполняет другие программы.

Если вы выполняете сложную серию шагов или рецептов и возникает ошибка, *make* остановится. Например, в моем предыдущем примере:

```
all: script3.out
script3.out: ./script3 script2.out
        ./script3 <script2.out >script3.out
script2.out: ./script2 script1.out
        ./script2 <script1.out >script2.out
script1.out: ./script1 data.txt
        ./script1 <data.txt >script1.out
clean:
        rm *.out
```

Если в скрипте script1 есть ошибка, то *make* остановит выполнение сразу после того, как попытается создать «script1.out» Если для цели существует более одного рецепта, *make* остановится после первой ошибки. Если нужно было выполнить четыре строки (или рецепты), а третья была прервана из-за ошибки, четвертая строка не будет выполнена. Это интуитивно понятно, потому что это то, что вы хотите, чтобы произошло. Но есть несколько тонких моментов. Предположим, у вас есть следующий рецепт:

```
test:
        false;true
        exit 1;exit 0
```

Программа «false» - это стандартная команда Unix, которая просто завершается с ошибкой. Если это была единственная команда в этой строке, то *make* остановит выполнение. Однако *make* запрашивает оболочку выполнить несколько команд в строке («false» и «true»), и последняя команда не завершилась ошибкой. Итак, оболочка сообщает *make*, что строка выполнена успешно. Это «игнорирует» предыдущую ошибку в строке. Итак, *make* затем выполняет вторую строку.

В этом случае оболочка выполняет «выход 1» напрямую, точно так же, как команда *false* , но на этот раз оболочка, которая выполняет программы, сама завершает работу, что приводит к остановке *make*. «Выход 0» никогда не отображается.

циклы выполнения в файле Makefile

Иногда вам нужно написать скрипт, который обрабатывает файл, а он обрабатывает только один файл за раз Тем не менее, вам нужно повторить это для нескольких файлов.

В качестве примера я хотел выполнить поиск в каталоге файлов PDF и вычислить хэш SHA1 для каждого из них. Я использовал это, чтобы увидеть, изменились ли какие-либо файлы PDF или были созданы новые. Я использовала следующий рецепт:

```
genpdfsums:
        ./FindPDFFiles | while IFS= read line;do sha1sum "$$line";done >PDFS.hashs
```

Обратите внимание, что я устанавливаю IFS в пустую строку на случай, если какое-либо имя файла содержит пробел в качестве части имени. Предполагается, что каждая строка ввода является именем файла.

Продолжение строки в Makefiles

Поскольку *make* выполняет оболочку, вы можете использовать те же соглашения для продолжения строки, что и оболочка. Например, вот рецепт, который я использовал, когда изучал результаты команды *wget* для извлечения URL-адресов:

```
TRACE = cat
#TRACE = tee /tmp/trace.out
all_urls.out:  Makefile
        cat */wget.err | grep '^--' | \
        grep -v '(try:' | awk '{ print $$3 }'  | ${TRACE} |\
        grep -v '\.\(png\|gif\|jpg\)$$' | sed 's:?.*$$::' | grep -v '/$$' | sort | uniq >all_urls.ou
```

Когда мои *make*-файлы становятся сложными, я часто помещаю «Makefile» в качестве зависимости в цель. В приведенном выше примере, если я отредактирую Makefile, то «all_urls.out» устарел.

То есть мой «исходный код» - это мой *Makefile*.

Вы также можете заметить, что я включил пример того, как я интегрирую *make* с отладкой сценариев оболочки В этом случае я создал переменную с именем «TRACE», которую я могу использовать для отладки моего сценария оболочки. Изменив определение «ТРАССИРОВКИ», я могу получить вывод в середине сложного сценария оболочки.

Условное выполнение в Makefiles

Иногда вам нужно проверить условие и выполнить часть сценария оболочки. Оболочка содержит команды "||" и "&&", которые можно использовать для выполнения простого управления потоком. Символы "&&" выполняю остальную часть строки, если условие истинно. Чтобы проверить условие false, используйте "||".

Предположим, вы проводите тестирование, чтобы убедиться, что две программы выдают одинаковые результаты, но вы хотите проверить это только при изменении программ.

Вот один из способов сделать это. Если "script1" или "script2" изменятся, выполните их. Если результаты отличаются, выполните команду "отчет"

```
diff: script1.out script2.out
        diff -b script1.out script2.out >/dev/null || ./Report
script1.out: script1
        ./script1 >script1.out
script2.out: script2
        ./script2 >script2.out
```

Я перенаправил вывод *diff* в /dev/null, чтобы скрыть информацию при запуске команды make . Меня просто интересует статус команды, а не данные.

Вот еще один пример, в котором я проверяю каталог на наличие файлов в поисках новых файлов, которые я раньше не сканировал. Я отслеживаю все файлы, которые я отсканировал, в файле "scanned.log". Если я нахожу файл, которого нет в этом файле, я проверяю его с помощью программы "Сканирование". Эта программа добавит имя файла к файлу "scanned.log". Этот рецепт *Makefile* делает это с помощью двух циклов while . Первому выдается список имен файлов с помощью *find*, а второй видит только имена файлов, которых нет в файле "scanned.log". Я использую *grep* с быстрой опцией "-q" и беззвучной опцией "-s" для подавления сообщений об ошибках о нечитаемых файлах. Повторяю, мне просто нужен статус задания *grep*, а не данные или ошибки.

Чтобы уточнить, первый цикл *while* выполняет проверку, находится ли имя файла в отсканированном файле.файл журнала. Если его нет в файле, команда echo отправляет имя файла в стандартный вывод, который в данном случае является вторым циклом *while*. Второй цикл сканирует файл.

Таким образом, сложный makefile будет выполнять команды, только если он не выполнялся ранее, но использует данные, управляемые вашей собственной программой.

```
all:
        find . -name \*.txt | while read f;do grep -sq "$$f" scanned.log || echo  "$$f" ;done | \
        while IFS= read n; \
        do Scan "$$n"; \
        done
```

Общие аргументы командной строки

Я рассмотрю некоторые из распространенных аргументов командной строки, которые я использую с *make* . Обратите внимание, что некоторые из этих параметров имеют несколько аргументов для указания одного и того же параметра (например, «-f *filename*» или «--file=*filename*»). Проверьте страницу руководства для получения более подробной информации.

Использование другого файла Makefile

Как я упоминал ранее, *make* сначала ищет файл «makefile», и если он не найден, он ищет «Makefile» . Вы можете переопределить это с помощью параметра «-f»:

```
# execute make with the default name file
make
# Execute make using the file Makefile.new
make -f Makefile.new
```

Вы можете использовать это для отладки ваших *make*-файлов или управления вариантами.

Продолжайте использовать аргумент -k

Обычно *make* останавливается при возникновении ошибки. Вы можете использовать аргумент «-k», чтобы указать *make* игнорировать ошибку и продолжать работу. Вы можете быть осторожны с этим, если ошибки приводят к недействительности данных. Buf если у вас длительный процесс сборки, вы можете продолжать его как можно дольше.

Ничего не делать - Make с аргументом -n

Когда вы начнете использовать *make*, вы можете захотеть посмотреть, что будет делать *make*, прежде чем выполнять его. У меня часто возникает эта проблема, когда я просматриваю чужой *Makefile*. Ввод «sudo make install» в чужой код должен заставить вас очень нервничать. Помимо проверки содержимого файла *Makefile*, есть еще один вариант: выполнить «make -n» . Опция «-n» - это опция бездействия, она просто печатает команды, которые будут выполняться, без их выполнения. Ничего не меняется, и файлы не создаются.

Префиксы рецепта

Определенные символы можно использовать в начале рецепта, чтобы управлять тем, как *make* выполняет рецепт. Это может сделать *make* (я не мог удержаться) менее раздражающим.

Игнорирование ошибок с префиксом -

Вы можете использовать опцию «-k» для игнорирования ошибок, но это не всегда то, что вам нужно. Предположим, например, что вы хотите удалить все файлы *.out . Если файлы существуют, проблем нет. Но если вы выполните рецепт «rm *.out», когда не существует файлов, соответствующих этому шаблону, это приведет к ошибке, и *make* остановится. Вы можете поставить префикс «-» в начале строки, чтобы указать *make* игнорировать любые ошибки во время выполнения в этой строке. Вот пример:

```
all:
        ....
clean:
        # The - in the next line is a prefix
        -/bin/rm *.out
        printf "this executes even if no files are deleted"
```

Тем не менее, вы все равно увидите ошибку. Это просто не приведет к выходу *make*.

Скрытие рецептов с помощью префикса @

Я использовал «printf» в некоторых из этих примеров для документирования моих *make*-файлов. Однако, когд вы выполняете *make*, мы в конечном итоге видим, что *make* показывает нам команду printf перед ее выполнением, а затем выполняется команда printf - так что в итоге мы видим это уведомление дважды. Вы можете запретить пересылку строк *make* на терминал с помощью префикса «@»:

```
all:
        ....
clean:
        # The - in the next line is a prefix
        -/bin/rm *.out
        @printf "All clean now"
```

Вы можете комбинировать эти префиксы.

```
test:
        @-/bin/rm *.out >/dev/null 2>1
        @printf "done"
```

Однако это все равно может привести к ошибкам. Например, я все еще получаю сообщение:

```
Makefile:2: recipe for target 'test' failed
make: [test] Error 1 (ignored)
```

Если я добавлю в рецепт «немного мусора.»:

```
test:
        @touch junk.out
        @-/bin/rm *.out >/dev/null 2>1
        @printf "done"
```

Затем, когда я выполняю этот рецепт, единственное, что я вижу, «готово»

Заключение к части 1 - Использование Make с помощью сценариев оболочки

Я думаю, что я дал вам достаточно, чтобы начать комбинировать *make* с вашими сценариями оболочки. Наибольшая мощь *Make* используется при компиляции исходного кода, но я нахожу его полезным при написании сложных сценариев оболочки. Например, мне приходилось создавать десятки длинных и сложных команд оболочки, когда я работал над проектом, в котором мне приходилось искать конфиденциальную информацию на внешнем веб-сервере

Я использовал дюжину различных инструментов для сканирования сервера, извлечения и загрузки всех файлов, поиска дубликатов, извлечения метаданных с этих серверов, а затем сканирования метаданных на предмет ключевых слов. Мне нужно было сделать это как можно быстрее и эффективнее, и я хотел отслеживать все, что я пробовал, чтобы я мог повторно использовать это в будущем. Моим главным инструментом был *make*.

В будущем руководстве будут рассмотрены расширенные функции, используемые для программистов.