ammaaim 7 фев 2014 в 21:12

Просто о make

6 мин **441K**

C*

Из песочницы

Меня всегда привлекал минимализм. Идея о том, что одна вещь должна выполнять одну функцию, но при этом выполнять ее как можно лучше, вылилась в создание UNIX. И хотя UNIX давно уже нельзя назвать простой системой, да и минимализм в ней узреть не так то просто, ее можно считать наглядным примером количество-качественной трансформации множества простых и понятных вещей в одну весьма непростую и не прозрачную. В своем развитии make прошел примерно такой же путь: простота и ясность, с ростом масштабов, превратилась в жуткого монстра (вспомните свои ощущения, когда впервые открыли мэйкфайл).

Мое упорное игнорирование make в течении долгого времени, было обусловлено удобством используемых IDE, и нежеланием разбираться в этом 'пережитке прошлого' (по сути — ленью). Однако, все эти надоедливые кнопочки, менюшки ит.п. атрибуты всевозможных студий, заставили меня искать альтернативу тому методу работы, который я практиковал до сих пор. Нет, я не стал гуру make, но полученных мною знаний вполне достаточно для моих небольших проектов. Данная статья предназначена для тех, кто так же как и я еще совсем недавно, желают вырваться из

уютного оконного рабства в аскетичный, но свободный мир шелла.

Make- основные сведения

make — утилита предназначенная для автоматизации преобразования файлов из одной формы в другую. Правила преобразования задаются в скрипте с именем Makefile, который должен находиться в корне рабочей директории проекта. Сам скрипт состоит из набора правил, которые в свою очередь описываются:

1) целями (то, что данное правило делает);
2) реквизитами (то, что необходимо для выполнения правила и получения целей);
3) 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>
```

То есть, правило make это ответы на три вопроса:

```
елаем? (реквизиты)} ---> [Как делаем? (команды)] ---> {Что делаем? (цели)}
```

Несложно заметить что процессы трансляции и компиляции очень красиво ложатся на эту схему:

```
{исходные файлы} ---> [трансляция] ---> {объектные файлы}
```

```
{объектные файлы} ---> [линковка] ---> {исполнимые файлы}
```

Простейший Makefile

Предположим, у нас имеется программа, состоящая всего из одного файла:

```c
/*
 * main.c
 */
#include <stdio.h>
int main()
{
    printf("Hello World!\n");
    return 0;
}
```

Для его компиляции достаточно очень простого мэйкфайла:

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

Данный Makefile состоит из одного правила, которое в свою очередь состоит из цели — «hello», реквизита — «main.c», и команды — «gcc -o hello main.c». Теперь, для компиляции достаточно дать команду make в рабочем каталоге. По умолчанию make станет выполнять самое первое правило, если цель выполнения не была явно указана при вызове:

```
    $ make <цель>
```

Компиляция из множества исходников

Предположим, что у нас имеется программа, состоящая из 2 файлов:
main.c

```
/*
 * main.c
 */
int main()
{
    hello();
```

```
    return 0;
}
```

и hello.c

```
/*
 * hello.c
 */
#include <stdio.h>
void hello()
{
    printf("Hello World!\n");
}
```

Makefile, выполняющий компиляцию этой программы может выглядеть так:

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

Он вполне работоспособен, однако имеет один значительный недостаток: какой — раскроем далее.

Инкрементная компиляция

Представим, что наша программа состоит из десятка- другого исходных файлов. Мы вносим изменения в один из них, и хотим ее пересобрать. Использование подхода описанного в предыдущем примере приведет к тому, что все без исключения

исходные файлы будут снова скомпилированы, что негативно скажется на времени перекомпиляции. Решение — разделить компиляцию на два этапа: этап трансляции и этап линковки.

Теперь, после изменения одного из исходных файлов, достаточно произвести его трансляцию и линковку всех объектных файлов. При этом мы пропускаем этап трансляции не затронутых изменениями реквизитов, что сокращает время компиляции в целом. 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
```

Попробуйте собрать этот проект. Для его сборки необходимо явно указать цель, т.е. дать команду make hello.
После- измените любой из исходных файлов и соберите его снова. Обратите внимание на то, что во время второй компиляции, транслироваться будет только измененный файл.

После запуска make попытается сразу получить цель hello, но для ее создания необходимы файлы main.o и hello.o, которых пока еще нет. Поэтому выполнение

правила будет отложено и make станет искать правила, описывающие получение недостающих реквизитов. Как только все реквизиты будут получены, make вернется к выполнению отложенной цели. Отсюда следует, что make выполняет правила рекурсивно.

Фиктивные цели

На самом деле, в качестве make целей могут выступать не только реальные файлы. Все, кому приходилось собирать программы из исходных кодов должны быть знакомы с двумя стандартными в мире UNIX командами:

```
$ make
$ make install
```

Командой make производят компиляцию программы, командой make install — установку. Такой подход весьма удобен, поскольку все необходимое для сборки и развертывания приложения в целевой системе включено в один файл (забудем на время о скрипте configure). Обратите внимание на то, что в первом случае мы не указываем цель, а во втором целью является вовсе не создание файла install, а процесс установки приложения в систему. Проделывать такие фокусы нам позволяют так называемые фиктивные (phony) цели. 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


```
.PHONY: all clean install uninstall

all: hello

clean:
	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:
	install ./hello /usr/local/bin
uninstall:
	rm -rf /usr/local/bin/hello
```

Теперь мы можем собрать нашу программу, произвести ее инсталлцию/деинсталляцию, а так же очистить рабочий каталог, используя для этого стандартные make цели.

Обратите внимание на то, что в цели all не указаны команды; все что ей нужно — получить реквизит hello. Зная о рекурсивной природе make, не сложно предположить как будет работать этот скрипт. Так же следует обратить особое внимание на то, что если файл hello уже имеется (остался после предыдущей компиляции) и его реквизиты не были изменены, то команда make **ничего не станет пересобирать**. Это классические грабли make. Так например, изменив заголовочный файл, случайно не включенный в список реквизитов, можно получить долгие часы головной боли. Поэтому, чтобы гарантированно полностью пересобрать проект, нужно предварительно очистить рабочий каталог:

```
$ make clean
$ make
```

Для выполнения целей install/uninstall вам потребуются использовать sudo.

Переменные

Все те, кто знакомы с правилом DRY (Don't repeat yourself), наверняка уже заметили неладное, а именно — наш Makefile содержит большое число повторяющихся фрагментов, что может привести к путанице при последующих попытках его расширить или изменить. В императивных языках для этих целей у нас имеются переменные и константы; make тоже располагает подобными средствами. Переменные в make представляют собой именованные строки и определяются очень просто:

```
<VAR_NAME> = <value string>
```

Существует негласное правило, согласно которому следует именовать переменные в верхнем регистре, например:

```
SRC = main.c hello.c
```

Так мы определили список исходных файлов. Для использования значения переменной ее следует разименовать при помощи конструкции $(<VAR_NAME>); например так:

```
gcc -o hello $(SRC)
```

Ниже представлен мэйкфайл, использующий две переменные: TARGET — для определения имени целевой программы и PREFIX — для определения пути установки программы в систему.

```
TARGET = hello
PREFIX = /usr/local/bin

.PHONY: all clean install uninstall

all: $(TARGET)

clean:
        rm -rf $(TARGET) *.o
```

```
main.o: main.c
            gcc -c -o main.o main.c
hello.o: hello.c
            gcc -c -o hello.o hello.c
$(TARGET): main.o hello.o
            gcc -o $(TARGET) main.o hello.o
install:
            install $(TARGET) $(PREFIX)
uninstall:
            rm -rf $(PREFIX)/$(TARGET)
```

Это уже посимпатичней. Думаю, теперь вышеприведенный пример для вас в особых комментариях не нуждается.

Автоматические переменные

Автоматические переменные предназначены для упрощения мейкфайлов, но на мой взгляд негативно сказываются на их читабельности. Как бы то ни было, я приведу здесь несколько наиболее часто используемых переменных, а что с ними делать (и делать ли вообще) решать вам:

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

Если кто либо хочет произвести полную обфускацию своих скриптов — черпать вдохновение можете здесь:

Автоматические переменные

Заключение

В этой статье я попытался подробно объяснить основы написания и работы мэйкфайлов. Надеюсь, что она поможет вам приобрести понимание сути make и в кратчайшие сроки освоить этот провереный временем инструмент.

Все примеры на GitHub

Тем, кто вошел во вкус:
Makefile mini HOWTO на OpenNET
GNU Make Richard M. Stallman и Roland McGrath, перевод © Владимир Игнатов, 2000
Эффективное использование GNU Make

Теги: make, makefile

Хабы: C

45 Карма 0 Рейтинг

Подписаться

Alex Balan @ammaaim
Пользователь

Комментарии 111 +111

Публикации

ЛУЧШИЕ ЗА СУТКИ ПОХОЖИЕ

BiktorSergeev 20 часов назад

Подключаемся к BBS через Amstrad NC100 из 1992 года

4 мин 2K

+30 11 8 +8

oldadmin 22 часа назад

HDD, SSD или NVMe: что выбрать для виртуального сервера (тесты внутри)

Средний 6 мин 6.2K

Обзор

+28 36 43 +43

Martynov_M 15 часов назад

Основание кулера выпуклое?

Простой 4 мин 5K

Recovery Mode

+26 13 16 +16

valisak 2 часа назад

Может ли во Вселенной не быть тёмной материи? 5 фактов, которые нельзя отрицать

Средний 8 мин 1.4K

Обзор

+19 4 3 +3

devops_ht 19 часов назад

ClickHouse как бэкенд для Prometheus

Средний 8 мин 3.1K

Туториал

+19 44 0

Mortyre 23 часа назад

Учите матчасть: почему стоит изучать туториалы перед работой с облаками и кому это особенно важно

5 мин 1.7K

+19 26 2 +2

konstantin-s-yakovlev 22 часа назад

Много-агентное планирование траекторий в децентрализованном режиме: эвристический поиск и обучение с подкреплением

Средний 17 мин 988

+17 31 3 +3

ux_designer 3 часа назад

Как сделать универсальный компонент List Cell в Figma

Простой 3 мин 287

Кейс

+13 2 0

arheo_pterix 12 часов назад

Гармония танцующих линий

Простой 9 мин 1.7K

+13 17 24 +24

SLY_G 21 час назад

Закат эпохи пара, часть 1: Внутреннее сгорание

Простой 16 мин 3.2K

Перевод

+13 24 13 +13

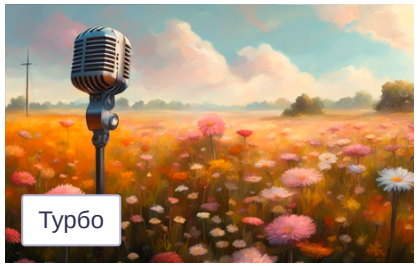
Нейросети в авторитете: вы не угадаете, сколько статей про нейронки и ML было в 2013 году на Хабре

Интересно

Показать еще

МИНУТОЧКУ ВНИМАНИЯ

Как бессонница в час ночной, меняет промокодище облик твой



Ивент моей мечты: опрос среди айтишников

Глупым вопросам и ошибкам — быть! IT-менторство на ХК

ВАКАНСИИ