Практика 8. Make

Введение

Make - это одна из наиболее мощных и широко используемых утилит в мире разработки программного обеспечения. Созданная в конце 1970-х годов, эта утилита стала стандартным инструментом автоматизации сборки проектов и управления зависимостями между файлами. Основная цель Make заключается в автоматизации процесса компиляции программ, что позволяет разработчикам избежать необходимости повторной компиляции уже скомпилированных частей программы.

Make оперирует с помощью специальных файлов, известных как makefile, в которых описываются правила сборки проекта. Эти правила определяют, какие файлы необходимо скомпилировать и какие зависимости между ними существуют. При вызове Make, утилита анализирует содержимое makefile и автоматически определяет, какие части проекта требуют перекомпиляции на основе изменений в исходных файлах.

Кроме того, Make позволяет значительно упростить процесс сборки сложных проектов, предоставляя разработчикам возможность создания множества правил и целей, таких как компиляция исходного кода, линковка объектных файлов, создание исполняемых файлов и многое другое. Это позволяет значительно повысить производительность разработки, особенно в проектах с большим количеством файлов и сложными зависимостями между ними.

Формат

Makefile можно воспринимать как скрипт, он состоит из трёх элементов:

1. target (цель) - файл, который мы хотим получить
2. prerequisites (зависимости) - файлы, которые мы будем преобразовывать
3. commands (команды) - то, как мы будем преобразовывать файлы

Структурно makefile выглядит так:

```
*target*: *prerequisites*
    *commands*
```

Команды выделяются табуляциями
Если сильно притянуть, то можно провести аналогию с функцией в питоне:

```
def target(prerequisites):
    temp = command1(prerequisites)
    temp = command2(temp)
    return command3(temp)
```

где у нас есть функция `target`, которая принимает аргумент `prerequisites`, последовательно вызывает три функции `command1`, `command2` и `command3` и возвращает результат

Начнём с классического hello world:

```
hello:
    echo "Hello world"
```

В данном случае у нашей цели `hello` нет зависимостей и она выполняет только одну команду `echo`
Можно заметить, что и `hello` не является файлом, но об этом позже

Для запуска скрипта достаточно запустить `make` в директории с `makefile`

Если явно не указать цель, то будет выполнено первое правило

```
hello:
    echo "Hello world"

test:
    echo "Test"
```

```
$ make
>>> Hello world

$ make test
>>> Test
```

Напишем пример запуска простого файла на Python:

```
hello: main.py
    python3 main.py
```

В данном случае у нас появилась зависимость от файла main.py, который мы будем запускать

Инкрементная сборка

Перейдём к примерам на Си. Он является компилируемым языком и при запуске make хотелось бы иметь возможность пересобирать файлы только с изменениями.
Это возможно при помощи инкрементной сборки. Для этого необходимо разделить этапы линковки и сборки.

```
result: file1.o file2.o
    gcc file1.o file2.o -o result
```

```
file1.o: file1.s
    gcc -c file1.s -o file1.o

file2.o: file2.s
    gcc -c file2.s -o file2.o

file1.s: file1.c
    gcc -S file1.c -o file1.s

file2.s: file2.c
    gcc -S file2.c -o file2.s
```

При первой сборке у нас выполнится цель `result`, в качестве зависимостей подтянутся `file1.o` и `file2.o`, а у них в качестве зависимостей подтянутся `file1.s` и `file2.s`, которые скомпилируются из файлов `file1.c` и `file2.c`

Если мы изменим любой из исходных файлов, то во время повторного запуска транслироваться будет только он.
Можно убедиться, что Make не хранит никаких историй изменений, он просто сравнивает время изменения файла

PHONY

Как мы заметили выше, не всегда цели являются файлами. Это называется ложной целью (phony). Есть несколько стандартных целей:

1. all - цель по умолчанию, вызывается если цель не была указана явно
2. clean - очистка каталога от файлов компиляции
3. install
4. uninstall

Чтобы Make отличал такие цели от файлов их необходимо явно определить в Makefile:

```
.PHONY: all clean

all: program

program: main.py module.py
    python3 main.py

clean:
    rm -rf __pycache__
```

Переменные, условные операторы, циклы

```
.PHONY: all clean
```

```
PYTHON = python3
SRC_DIR = src
SOURCES = $(wildcard $(SRC_DIR)/*.py)
MAIN_FILE = main.py

ifdef DEBUG
    RUN_COMMAND = $(PYTHON) -m pdb $(SRC_DIR)/$(MAIN_FILE)
else
    RUN_COMMAND = $(PYTHON) $(SRC_DIR)/$(MAIN_FILE)
endif

all: program

program: $(SOURCES)
    $(RUN_COMMAND)

clean:
    rm -rf __pycache__

debug:
    make all DEBUG=1
```

В этом примере мы добавили условный оператор `ifdef` для определения переменной `RUN_COMMAND`, которая определяет команду запуска программы. Если установлен флаг `DEBUG`, то программа будет запускаться с использованием отладчика pdb.

Цель `all` по-прежнему запускает программу по умолчанию, а цель `clean` удаляет временные файлы pycache. Также добавлена новая цель `debug`, которая перезапускает программу с флагом `DEBUG=1` для запуска в отладочном режиме.

```
.PHONY: all clean

PYTHON = python3
SRC_DIR = src
BUILD_DIR = build
SOURCES = $(wildcard $(SRC_DIR)/*.py)
MAIN_FILE = main.py

all: program

program: $(SOURCES)
    @for file in $(SOURCES); do \
        $(PYTHON) $$file; \
    done

clean:
    rm -rf $(BUILD_DIR) __pycache__
```

В этом примере мы добавили цикл `for`, который выполняет запуск каждого файла из переменной `SOURCES` поочередно. Переменная `SRC_DIR` определяет директорию, содержащую исходные файлы,

а переменная `BUILD_DIR` используется для хранения сгенерированных файлов.

Пример использования

```
SRC_DIR = src
MAIN_FILE = $(SRC_DIR)/main.py
PYTHON = python3

coverage:
    coverage erase
    coverage run -m pytest $(SRC_DIR)
    coverage report -m

deps:
    pip install -r requirements.txt

push:
    git push && git push --tags

lint:
    pylint $(SRC_DIR)

test:
    pytest $(SRC_DIR)

all:
    ${PYTHON} ${MAIN_FILE}
```