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.

Сегодня Make остается незаменимым инструментом в индустрии программной разработки, широко используемым для сборки проектов на различных языках программирования, включая C, C++, Java, Python и многие другие. Его гибкость, надежность и простота в использовании делают его основой для многих других инструментов и систем сборки, что подтверждает его значимость и актуальность даже спустя десятилетия после первоначального создания.

Примеры использования

Давайте рассмотрим пример использования утилиты Make для компиляции программы на языке Python, состоящей из нескольких модулей. В данном случае мы можем использовать Make для автоматической сборки проекта Python. Создадим пример makefile для этой цели:

```
.PHONY: all clean

all: program

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

clean:
    rm -rf __pycache__
```

В этом примере мы определяем три цели: `all`, `program` и `clean`. Цель `all` по умолчанию компилирует программу, `program` запускает главный скрипт Python `main.py`, а `clean` удаляет созданные файлы

pycache.

Чтобы запустить созданный makefile, необходимо ввести команду `make` в той же директории, где находится makefile. Это выполнит все необходимые действия в makefile, включая запуск программы Python.

Make также позволяет определять переменные и использовать их в makefile, что облегчает настройку процесса сборки проекта. Кроме того, условные операторы и циклы могут быть использованы для более сложных сценариев сборки.

```
.PHONY: all clean

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

all: program

program: $(SOURCES)
    $(PYTHON) $(SRC_DIR)/$(MAIN_FILE)

clean:
    rm -rf __pycache__
```

В этом примере переменная `PYTHON` используется для определения версии Python, которую следует использовать при запуске программы. Переменная `SRC_DIR` определяет директорию, содержащую исходные файлы, а переменная `SOURCES` содержит список всех файлов с расширением `.py` в данной директории. Переменная `MAIN_FILE` указывает на основной исполняемый файл программы.

Примеры для C++ и Java:

```
.PHONY: all clean

CXX = g++
CXXFLAGS = -std=c++11 -Wall

all: program

program: main.cpp functions.cpp
    $(CXX) $(CXXFLAGS) -o program main.cpp functions.cpp

clean:
    rm -f program
```

Этот пример демонстрирует создание исполняемого файла `program` из файлов `main.cpp` и `functions.cpp`. Переменные `CXX` и `CXXFLAGS` используются для определения компилятора и флагов компиляции соответственно.

```
.PHONY: all clean

JAVAC = javac
JAVAFLAGS = -g

SOURCES = Main.java Utility.java
CLASSES = $(SOURCES:.java=.class)

all: program

program: $(CLASSES)

%.class: %.java
    $(JAVAC) $(JAVAFLAGS) $<

clean:
    $(RM) *.class
```

В данном случае, переменные `JAVAC` и `JAVAFLAGS` определяют компилятор Java и флаги компиляции соответственно. Цель `all` компилирует все исходные файлы, а `clean` удаляет все скомпилированные классы.

```
.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.