Краткое введение в `make`

Для сдачи домашних заданий требуется использовать инструмент `make`, и в связи с этим у многих появляется два вопроса: (а) как им пользоваться и (б) что нам даёт умение им пользоваться? Данный документ призван ответить на эти вопросы.

Что это за инструмент

`make` — это один из основных инструментов для сборки проектов в ОС Unix. Также он часто используется и под другими операционными системами. Несмотря на многочисленные попытки, пока что полноценной замены ему не предложено.

Помимо огромного количества недостатков, у `make` есть два очень важных достоинства:

1. Он есть везде. Его стандарт установился достаточно давно, поэтому практически под любую распространённую ОС общего назначения он будет доступен. В частности, есть полноценные реализации для Windows, Unix и Mac OS.

2. Он крайне гибок и мощен. С его помощью можно собрать более-менее любой проект на практически любом языке.

Именно поэтому он используется на серверах Яндекс.Контест — альтернативные варианты (stack, maven и т.п.) могут быть удобнее для конкретного языка, но они не универсальны, они требуют специальной настройки, репозиториев, и подходят для узкого набора языков. Если по какой-то причине владельцы сервера не готовы часто обновлять софт, то все эти варианты сразу отсекаются. Опыт проверки решений студентов в ИТМО показывает, что даже при наличии полного доступа к компьютеру всё равно часто оказывается проще собрать проект «руками», полностью отказавшись от использованной «продвинутой» системы сборки.

Скорее всего, вы не раз ещё встретитесь с этим инструментом в своей программистской практике, поэтому логично не откладывать знакомство.

Общая структура

Инструкция для `make` хранится в файле `Makefile`. Файл можно разделить на две части: сперва идут общие определения различных переменных, которые могут потребоваться в ходе работы, а потом идут описания *правил* — особого рода функций.

Если в строке встретился символ `#`, вся строка от этого места до конца считается комментарием.

Правила и цели

Целью в языке `make` называется файл из проекта. Некоторые цели (исходные файлы) не могут быть построены в процессе сборки, для всех же остальных целей (скажем, для исполнимых файлов) должны существовать правила.

Именем правила является имя файла. Тело правила — это последовательность комад ОС, служащая для его построения. Также у правила есть *зависимости* (*реквизиты*) — цели, которые должны быть достигнуты перед вызовом самого правила.

Имя цели *обязательно* начинается с первой колонки. Можно перечислить несколько целей через пробел. После списка целей ставится двоеточие. Зависимости указываются после двоеточия в той же строке, что и имя цели, также разделяются пробелами. Тело правила указывается в последующих строках, обязательно с отступом в один символ *табуляции* — принципиально важно, чтобы это была именно табуляция, а не несколько пробелов.

Следующий пример кажется самоочевидным:

```
hello_world.exe: hello_world.cpp
	g++ hello_world.cpp -o hello_world.exe
```

Цель (файл) `hello_world.exe` зависит от исходного файла `hello_world.cpp`, и не может быть построена при его отсутствии.

Некоторые цели не имеют конкретных соответствующих им файлов. Такие цели называют *абстрактными* (фиктивными, *phony*). Наиболее часто используются следующие абстрактные цели:

- `all` — цель по умолчанию
- `clean` — соответствующее правило должно удалить все файлы, строящиеся компилятором (то есть, всё, кроме исходных кодов)
- `install` — проинсталлировать проект

Чтобы указать, какие цели являются абстрактными, существует директива `.PHONY`:

```
.PHONY: all clean install uninstall
```

В целом, представление о правилах как о функциях довольно точно отражает их суть. Зависимости можно трактовать как вложенные вызовы других функций. Однако, привязка целей (имён функций) к файлам вносит некоторые отличия от традиционных функций. Если некоторый файл уже построен и актуален, повторно правила для его построения вызываться не будут.

Поясним это подробнее. Файл считается устаревшим, если последнее его изменение имело место раньше, чем последние изменения какой-либо из его зависимостей. Перед выполнением какого-либо правила `make` строит дерево зависимостей и проверяет его актуальность, поднимаясь от листьев к корню. Как только он находит устаревший файл, он выполняет соответствующее ему правило и заменяет файл на актуальный.

Правила для абстрактных целей выполняются всегда, абстрактные цели всегда устаревшие. В самом деле, ведь файлов, имеющих имя абстрактных целей, в проектах обычно не бывает, и их надо заново «строить». Однако, если эти цели не указаны в директиве `.PHONY`, а соответствующие файлы существуют, цели начнут вести себя как обычные (файловые).

Макропеременные и правила работы с ними

Определения переменных даются в начале файла, перед указанием правил, каждое определение указывается в отдельной строке и соответствует следующей грамматике:

⟨идентификатор⟩ '=' ⟨значение⟩

Например, мы могли бы написать что-нибудь такое:

```
SOURCES=src/x.ml src/y.ml
```

Если где-то требуется использовать содержимое переменной, пишут знак доллара и имя переменной в скобках: `$(SOURCES)`. В данном месте произойдёт макроподстановка, и результат будет неотличим от прямого указания значения: `src/x.ml src/y.ml`.

Помимо переменных, определённых пользователем в начале файла, в мэйкфайле доступны все переменные окружения. Например, начиная с Windows NT во всех версиях данной ОС по умолчанию определена переменная окружения `OS`, имеющая значение `Windows_NT`. Её можно использовать для написания переносимых мэйкфайлов (пример см. в `ocaml-solution`).

Также существуют следующие особые автоматически определяемые переменные:

Переменная	Значение
`$<`	Первая зависимость цели
`$^`	Все зависимости цели, разделённые пробелами
`$*`	Базовое имя цели (без расширения и имени каталога)
`$@`	Полное имя цели

Бывает, что одно имя нужно получать из другого путём замены. Для этого есть особый синтаксис:

‘$(’ ⟨переменная⟩ ‘:’ ⟨исх-префис⟩ ‘%’ ⟨исх-суффикс⟩ ‘=’ ⟨итог-префис⟩ ‘%’ ⟨итог-суффикс⟩ ‘)’

Данная команда разбивает содержимое переменной по пробелам, и в каждом полученном слове заменяет начала и концы слов на указанные строчки. Например, `$(SOURCES:src/%.ml=%.o)` значение `src/x.ml src/y.ml out/z.c` подставит в текст как `x.o y.o out/z.c`.

Также, удобной возможность макропроцессора мэйкфайлов являются условные конструкции. Их можно использовать в любой части файла, они имеют относительно традиционный синтаксис:

```
ifeq ($(OS),Windows_NT)
    DEL=del /f
else
    DEL=rm -f
endif
```

В этом примере мы определяем команду удаления, в зависимости от операционной системы. При необходимости её вызова нужно будет указать макропеременную `DEL`. Например, так можно задать абстрактное правило, уничтожающее исполнимый файл и объектные файлы, построенные при компиляции исходников:

```
clean:
        $(DEL) main.exe $(SOURCES:src/%.c=out/%.o)
```

Мэйкфайлы обладают ещё многими возможностями, подробнее они описаны в документации.

Вызов `make` и особенности мэйкфайлов для Яндекс.Контест

Построение проекта осуществляется указанием в командной строке команды `make`. Первым аргументом может идти цель, при её отсутствии предполагается цель `all`. Чтобы команда успешно отработала, требуется в текущем каталоге иметь `Makefile`, из которого и будут браться определения правил.

В условиях сказано, что ваш файл компилируется при помощи `make` и запускается при помощи `make run`. Это значит, что у вас в корне архива должен находиться `Makefile`, в котором должно быть определено как минимум две цели — `all`, и `run`.

Цель `all` должна строить проект, а цель `run` — запускать его.

Помимо этих целей, в эталонных решениях определена абстрактная цель `pack`, осуществляющая частичную сборку проекта (построение исходников для генерируемых парсеров), а также упаковывающая пакет для отправки на сервер надлежащим образом.

Чрезмерная зависимость проекта от конкретной среды, в которой он разрабатывается, сильно мешает его переносимости, делает его неотчуждаемым. Поэтому принципиально важно, чтобы компиляция выполнялась бы на сервере, исключение может быть сделано для инструментов, строящих исходный код (например, для уасс и других генераторов парсеров). Однако и в этом случае требуется, чтобы `Makefile` позволял осуществить полную сборку проекта при наличии на компьютере соответствующих инструментов.

Дальнейшая литература

Данное введение не пытается заменить собой литературу по утилите `make`, за дальнейшими подробностями мы рекомендуем обратиться к документации.

`http://www.crossplatform.ru/documentation/gnu-make/gnu-make-ru.php`