Build-системы

Билд-системы (aka система сборки, система автоматизации сборки, далее по тексту БС) — это ПО, которое позволяет вам "собирать" что-то. Иными словами запускать набор команд, чтобы из входных файлов создавать выходные файлы.

Зачем это может понадобится?

Пример 1: Вы пишете текстовый документ с помощью какой-нибудь системы разметки (asciidoc, latex, ...) и вам нужно написать некоторую последовательность команд чтобы из входных файлов (текст, картинки, таблицы) получить выходные файлы (pdf-документ).

Пример 2: Вы разрабатываете ПО и вам из исходных файлов нужно получить исполняемые файлы.

Пример 3: Вы пишете тесты к своему ПО, и вам нужно автоматизировать их запуск после сборки ПО.

Пример 4: Вам нужно автоматизировать процесс установки ПО на целевую систему.

Автоматизировать задачи из примеров выше можно с помощью БС. БС позволяет вам описать все **правила** для сборки тех или иных файлов. Набор **правил** сборки называется сценарием сборки, и описываются в в специальном файле, который будет читать БС и запускать необходимые команды для сборки. **Правила** в данном случае это просто некая последовательность команд, которые нужно запустить. При этом БС берут на себя задачу по отслеживанию зависимостей между файлами, и запуску только тех команд, которые необходимы.

Существует много разных БС для разных задач. Перечислим некоторые из них:

- make — БС общего назначения
- nmake — клон make от Microsoft
- cmake — кроссплатформенная БС, используется для сборки C-подобных языков.
- meson --ещё одна кроссплатформеная БС. Её особенность в том, что она из своего конфига генерит конфиги для других БС (напр ninja или make)
- ninja — низкоуровневая БС, похожая на make.
- MSBuild — БС от Microsoft, основана на XML-конфигах, используется в visual studio.

ВНИМАНИЕ	Запоминать названия БС не обязательно, это просто на погуглить, если очень хочется.

Какие-то БС заточены под конкретную группу языков программирования, например **cmake**, система сборки для C/C++. Какие-то БС более общие и подходят для автоматизации каких угодно задач, например **make**.

Не смотря на то, что синтаксис и детали реализации у каждой отдельной БС могут различаться, все билд системы строятся по похожим принципам и сценарии сборки состоят

из:

- одной или нескольких **целей** сборки, т.е. выходных файлов, которые нужно получить.
- набора **зависимостей**, т.е. входных файлов от которых зависит **цель** сборки.
- **правил** сборки, которые описывают как из **зависимостей** получить **цель** сборки.

Примеры сценариев БС

Для иллюстрации посмотрим на сценарии различных билд-систем. Полностью понимать их не обязательно, важно постараться выделить **цели зависимости** и **правила** сборки в конфигах сборки.

Пример. meson

main.c

```c
#include <stdio.h>

int main()
{
    puts("Hello, World!");
    return 0;
}
```

meson.build

```
project('tutorial', 'c')      # tutorial – имя проекта,
                              # c – язык программирования

executable('demo', 'main.c')  # demo – имя исполняемого файла,
                              # «main.c» – файл исходного кода
```

shell commands:

```
$ meson setup builddir
$ cd builddir
$ meson compile
```

Пример. meson bzip

```
project(
  'bzip2',
  ['c'],
  version : '1.0.7',
  meson_version : '>= 0.50.0',
  default_options : ['c_std=c89', 'warning_level=1'],
)

cc = meson.get_compiler('c')
add_project_arguments(cc.get_supported_arguments([
    # Please keep this list in sync with CMakeLists.txt
    '-Wall',
  ]),
  language : 'c',
)

os_defines = []
if host_machine.system() == 'windows'
  os_defines += '-DBZ_LCCWIN32=1'
else
  os_defines += '-DBZ_LCCWIN32=0'
endif

bz_sources = ['blocksort.c', 'huffman.c', 'crctable.c', 'randtable.c', 'compress.c',
'decompress.c', 'bzlib.c']

libbzip2 = library(
  'bz2',
  bz_sources,
  c_args : c_args,
  install : true,
)

bzip2 = executable(
  'bzip2',
  ['bzip2.c'],
  link_with : [libbzip2],
  install : true,
  c_args : os_defines,
)
```

Это упрощённый сценарий для сборки утилиты bzip2(3). В сценарии описывается 2 цели — исполняемый файл bzip2 и библиотека libbzip2. Они в свою очередь зависят от исходных файлов, описанных в переменных bz_sources.

Пример. cmake

Аналогичный пример для cmake.

```
set(BZ2_SOURCES
    blocksort.c
    huffman.c
    crctable.c
    randtable.c
    compress.c
    decompress.c
    bzlib.c)

add_library(bz2_ObjLib OBJECT)
target_sources(bz2_ObjLib
    PRIVATE   ${BZ2_SOURCES}
    PUBLIC    ${CMAKE_CURRENT_SOURCE_DIR}/bzlib_private.h
    INTERFACE ${CMAKE_CURRENT_SOURCE_DIR}/bzlib.h)


if(ENABLE_SHARED_LIB)
    # The libbz2 shared library.
    add_library(bz2 SHARED ${BZ2_RES})
    target_sources(bz2
        PRIVATE   ${BZ2_SOURCES}
                  ${CMAKE_CURRENT_SOURCE_DIR}/libbz2.def
        PUBLIC    ${CMAKE_CURRENT_SOURCE_DIR}/bzlib_private.h
        INTERFACE ${CMAKE_CURRENT_SOURCE_DIR}/bzlib.h)
    set_target_properties(bz2 PROPERTIES
        COMPILE_FLAGS "${WARNCFLAGS}"
        VERSION ${LT_VERSION} SOVERSION ${LT_SOVERSION})
    install(TARGETS bz2 DESTINATION ${CMAKE_INSTALL_LIBDIR})
    install(FILES bzlib.h DESTINATION ${CMAKE_INSTALL_INCLUDEDIR})
endif()


if(ENABLE_APP)
    # The bzip2 executable.
    add_executable(bzip2)
    target_sources(bzip2
        PRIVATE   bzip2.c)
    target_link_libraries(bzip2
        PRIVATE   bz2_ObjLib)
    if(WIN32)
        target_compile_definitions(bzip2 PUBLIC BZ_LCCWIN32 BZ_UNIX=0)
    else()
        target_compile_definitions(bzip2 PUBLIC BZ_LCCWIN32=0 BZ_UNIX)
    endif()
    install(TARGETS bzip2 DESTINATION ${CMAKE_INSTALL_BINDIR})
```

make

Подробно рассмотрим одну из систем сборки — make. Это достаточно древнючая система сборки (первая версия появилась в 1976), она подходит для сборки чего угодно (т.е. не ограничивается какой-то конкретной группой языков).

Почему мы выбрали её?

- чрезвычайно распространена на Unix-системах.
- хорошо подходит для маленьких и средних проектов

Рецепты сборки описываются в специальных make-файлах (имя файла по умолчанию — Makefile). Синтаксис make-файлов довольно странный и необычный, однако в нём не очень сложно разобраться.

Сборка происходит с помощью одноимённой утилиты — make(1).

Синтаксис вызова:

```
make [ -f Makefile ] [ цель ] ...
```

Параметры:

- **-f** — опциональный параметр, с помощью которого можно указать путь к make-файлу (если он лежит не в текущей директории или назван как-то по другому)
- **цель** — опциональный параметр, указывает **цель** сборки

В общем случае make-файл выглядит следующим образом

```
#комментарий
переменная=значение

цель: зависимости
    правила_сборки
```

ВАЖНО!	формат файла требует чтобы правила сборки писались после символа табуляции (\t)!

Правила сборки — это простые shell команды, можно использовать специальные символы shell.

Для каждой цели мы указываем список зависимостей.

Первая цель, определённая в make-файле является целью по умолчанию, и будет собираться если пользователь запустил make без аргументов.

Пример make-файла:

```
CC=gcc
CFLAGS=-Wall
OBJS=main.o lib.o
BIN=mybinary

$(BIN): $(OBJS)
	$(CC) $(CFLAGS) $(OBJS) -o $(BIN)

clean:
	rm $(OBJS) $(BIN)
```

Использовать ранее определённую переменную, можно, если запихать её имя внутрь `$()`.

Мы определили цель `mybinary`, и сказали что она зависит от файлов main.o и lib.o. Это объектные файлы, которые получаются при компиляции из исходых файлов.

Если мы создадим Makefile с таким содержимым и запустим `make` без аргументов, то он попытается собрать `mybinary`.

В данном make-файле мы определили вторую цель `clean` (видно что у цели нет никаких входных файлов, от которых она зависит). Как следует из названия она должна почистить рабочую директорию проекта от временных файлов. Чтобы запустить сборку этой цели необходимо запустить в команду `make clean`.

Как make догадывается, какие правила нужны, чтобы собрать `.o` файлы из `.c`? В make предопределено много неявных правил для сборки разных типов файлов (особо любопытным сюда). Но мы могли указать это правило явно, например так:

```
%.o: %.c
    $(CC) $(CFLAGS) -o $@ $<
```

В правиле выше используются шаблоны имён файлов. `%` шаблон, который совпадает с любым количеством произвольных буковок. Т.е. шаблон `%.o` описывает любой файл, который имеет суффикс `.o` (например файлы `main.o lib.o` совпадают с этим шаблоном, а файл `main.o.c` нет).

Таким образом правило выше читается так: все файлы с расширением `.o` зависят от таких же файлов с расширением `.c`. Для их сборки используется команда `$(CC) $(CFLAGS) -o $@ $<`.

Автоматические переменные

При написании правил можно использовать автоматические переменные. Они нужны для упрощения make-файлов. Содержимое этих переменных зависит от цели зависимостей текущего правила.

Таблица 1. Специальные переменные

Переменная	описание переменной
$@	**цель** правила
$<	первая **зависимость** правила.
$^	все **зависимости** правила.

Таким образом команда `$(CC) $(CFLAGS) -o $@ $<`. преобразуется в `$(CC) $(CFLAGS) -o ЦЕЛЬ_СБОРКИ ЗАВИСИМОСТЬ`

Больше подробностей тут.

Цели сборки

В make-файле можно явно определить цели для сборки по умолчанию (те, которые будут собираться при запуске make(1) без аргументов). Для этого используется встроенная цель `all`.

Например для make-файла, указанного ниже при запуске `make` без аргументов будут собраны и `ЦЕЛЬ1` и `ЦЕЛЬ2`.

```
all: ЦЕЛЬ1 ЦЕЛЬ2

ЦЕЛЬ1: ЗАВИСИМОСТИ1
    КОМАНДЫ1

ЦЕЛЬ2: ЗАВИСИМОСТИ2
    КОМАНДЫ2
```

В make-файлах часто можно встретить другие стандартные цели

- install — производит установку ПО
- uninstall — деинсталлирует ПО
- clean — чистит текущую директорию от временных файлов.

Шаблон Makefile для проекта

```
OBJ=main.o lib.o
CFLAGS=-Wall
TARGET=myprog

all: $(TARGET)

$(TARGET): $(OBJ)
    $(CC) $(CFLAGS) -o $@ $^

%.o: %.c
    $(CC) $(CFLAGS) -o $@ $<
```

```
clean:
    rm $(TARGET) $(OBJ)
```

Что почитать

- Ещё один тутор про make на хабре. Настоятельно рекомендую.
- Полная дока по make (eng)
- Перевод доки на русском.
- ...

задачки

- Установить linux, поиграться с make.
- Попробовать написать Makefile для helloworld C-проекта