Системы сборки проектов

Системы сборки проектов представляют собой инструменты и сценарии, которые автоматизируют превращение исходного кода в исполняемые файлы, библиотеки и другие артефакты. По мере роста проекта и увеличения числа модулей рутинные вызовы компилятора и копирование команд в скрипты становятся неэффективными и подвержены ошибкам. Ниже описан эволюционный путь от базовой ручной сборки до полноценных систем сборки с указанием их преимуществ и недостатков.

---

1. Ручная компиляция

```
$ gcc main.c -o main
```

При добавлении новых исходников вручную пропишем каждую команду:

```
$ gcc -c src0.c
$ gcc -c src1.c
$ gcc -c main.c
$ gcc -o main main.o src0.o src1.o
```

Преимущества:

- Простота и прямолинейность — достаточно знать одну команду компилятора.
- Минимум зависимостей — не требует дополнительных утилит или конфигурации.

Недостатки:

- Масштабируемость — при большом количестве файлов управление командами становится невозмутимо трудоемким.
- Полная пересборка — даже небольшое изменение требует пересборки всего проекта.
- Риск человеческой ошибки — опечатки, неправильный порядок команд, забытые файлы.

---

2. Скрипт на Bash

```
#!/bin/bash

gcc -c src0.c
gcc -c src1.c
gcc -c main.c

gcc -o main src0.o src1.o main.o
```

Преимущества:

- Автоматизация повторяющихся команд — все действия сконцентрированы в одном файле.
- Удобство запуска — достаточно запустить `./build.sh`.

Недостатки:

- Отсутствие инкрементальности — скрипт пересобирает всё каждый раз.
- Ограниченная гибкость — для сложных сценариев потребуются дополнительные проверки и ветвления.

---

3. Инкрементальная сборка в скрипте с проверкой времени модификации

```
#!/bin/bash

function modification_time {
    # Возвращает время последней модификации файла в секундах с начала эпохи
    date -r "$1" '+%s'
}

function check_time {
    local name=$1
    # Если объектный файл не существует, нужно собрать
    [ ! -e "$name.o" ] && return 0
    # Если исходник новее объектного файла, нужно собрать
    [ "$(modification_time "$name.c")" -gt "$(modification_time "$name.o")"
] && return 0

    return 1
}
```

```
check_time src0 && gcc -c src0.c
check_time src1 && gcc -c src1.c
check_time main && gcc -c main.c

gcc -o main src0.o src1.o main.o
```

Преимущества:

- Точная инкрементальная сборка — каждая функция самостоятельно определяет, нужен ли перекомпиляция.
- Простота понимания и расширения — логика проверок отделена в функции `check_time`.

Недостатки:

- Локальная логика — нет учёта сложных цепочек зависимостей (например, изменение заголовочных файлов).
- Скрипт остаётся жестко привязанным к конкретным именам и расширениям файлов.

---

4. Make

Make — классическая система сборки в UNIX-подобных системах. Она читает правила из файла `Makefile`, где каждая запись определяет цель, её зависимости и команды для выполнения:

1. Make выбирает первую цель (обычно `all`) и проверяет, какие из её зависимостей устарели.
2. Для каждой устаревшей зависимости выполняется соответствующий рецепт (команды).
3. Процесс повторяется, пока все цели актуализированы.

Также Make использует механизм сравнения времён последней модификации файлов: для каждой цели он проверяет, старше ли её timestamp по сравнению с любыми зависимостями. Если зависимость была изменена позже, чем цель, Make считает цель устаревшей и выполняет соответствующее правило. Таким образом, пересобираются только те части проекта, где исходники обновились, что значительно ускоряет сборку при небольших изменениях.

Структура Makefile

- **Цели и зависимости:**

```
target: dep1 dep2
    <commands>
```

- - **Шаблонные правила (pattern rules):** позволяют описать общее правило преобразования группы файлов с использованием символа `%` в качестве подстановки.

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

    Это правило означает: для каждого файла `имя.o` возьми соответствующий `имя.c` и скомпилируй.
    Внутри рецепта доступны автоматические переменные:
    - `$@` — имя target цели (например, `main.o`).
    - `$<` — имя первой зависимости (например, `main.c`).
- **Переменные:** позволяют задавать параметры компиляции и переиспользовать их: `CC = gcc`, `CFLAGS = -O2 -Wall`.
- **.PHONY:** объявляет виртуальные цели, чтобы Make не искал одноимённые файлы.

Пример расширенного Makefile

```
# Компилятор и флаги
CC = gcc
CFLAGS = -Wall -Wextra -O2

# Исходники и объектные файлы
SRCS = main.c src0.c src1.c
OBJS = $(SRCS:.c=.o)

# Имя итогового приложения
TARGET = app

.PHONY: all clean

# Основная цель
all: $(TARGET)

# Линковка приложения из объектных файлов
$(TARGET): $(OBJS)
    $(CC) $(CFLAGS) -o $@ $^

# Компиляция каждого .c в .o
```

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

# Очистка скомпилированных файлов
clean:
    rm -f $(OBJS) $(TARGET)
```

Возможности Make

- **Инкрементальная сборка:** Make автоматически пересобирает только устаревшие цели.
- **Параллельная сборка:** флаг `-j N` позволяет выполнять N задач одновременно.
- **Условные конструкции:** `ifeq`, `ifneq`, `ifdef`, `ifndef` для разных окружений.
- **Include:** директива `include` разбивает конфигурацию на несколько файлов.
- **Встроенные функции:** `$(wildcard *.c)`, `$(patsubst pattern,replacement,text)` и др. для динамических списков.

Преимущества:

- Широкая доступность и простота установки на UNIX-подобных системах.
- Высокая скорость и экономия времени при инкрементальной сборке.
- Гибкость через переменные и шаблоны.

Недостатки:

- Чувствительный к отступам и синтаксису язык Makefile, что затрудняет чтение.
- Ограниченная кроссплатформенность без дополнительных патчей.
- Отсутствие встроенного управления сторонними библиотеками и сложными цепочками зависимостей.

---

5. CMake

CMake — кроссплатформенный фронтенд для систем сборки и интеграции с IDE, сама по себе сборкой не занимается, а генерирует конфигурации для выбранного бэкенда.

Задача: стандартный Make работает только в UNIX-средах и не учитывает разные форматы проектов IDE (Visual Studio `.vcxproj` / `.sln`, Xcode, CodeBlocks и др.).

Решение:

- В одном файле `CMakeLists.txt` описываете структуру проекта, зависимости и опции сборки.
- CMake генерирует конфигурации для разных инструментов:
  - **Бэкенды сборки:** Make, Ninja и др.;
  - **IDE-проекты:** Visual Studio, Xcode, CodeBlocks, Eclipse, KDevelop3 и др.;
- Позволяет задавать условия сборки, искать внешние библиотеки (`find_package`), настраивать флаги компиляции и линковки для разных платформ.

Ключевые возможности:

- `cmake_minimum_required(VERSION X.Y)` — указывает минимальную версию CMake.
- `project(Name VERSION a.b LANGUAGES C CXX)` — настраивает имя, версию и языки проекта.
- `add_executable` / `add_library` — создание исполняемых файлов и библиотек.
- `target_include_directories` , `target_link_libraries` — указание заголовков и зависимостей.
- `option(NAME "description" ON/OFF)` — флаги включения/отключения фич.
- `find_package(Pkg REQUIRED)` — поиск и подключение внешних пакетов.

Пример `CMakeLists.txt` :

```
cmake_minimum_required(VERSION 3.15)
project(MyApp VERSION 1.0 LANGUAGES C CXX)

# Опция включения тестов
enable_testing()

# Добавляем исполняемый файл из исходников
add_executable(app main.cpp src0.cpp src1.cpp)

target_include_directories(app PRIVATE include)

# Линковка с внешней библиотекой
find_package(Threads REQUIRED)
target_link_libraries(app PRIVATE Threads::Threads)
```

Преимущества:

- **Кроссплатформенность:** одна конфигурация для UNIX, Windows и macOS.
- **Генерация под разные инструменты:** поддержка Make, Ninja, IDE-проектов.
- **Модульность:** легко организовать многомодульные проекты через `add_subdirectory` .

- **Мощный язык сценариев:** условия, циклы, функции и макросы.

Недостатки:

- **Крутая кривая обучения:** синтаксис CMakeLists порой непредсказуем.
- **Неинтуитивные ошибки:** сложнее отлаживать конфигурацию по сравнению с простыми Makefile.
- **Дополнительный этап генерации:** перед сборкой нужно запускать `cmake` и только затем — `make` или `ninja`.

---

6. Maven

Maven — декларативная система сборки для Java, основанная на Project Object Model (`pom.xml`). Вместо явного описания шагов вы определяете конфигурацию, а Maven сам управляет процессом.

Основные принципы:

- **Declarative:** вы описываете *что* нужно сделать, а не *как*.
- **Convention over Configuration:** соглашения по структуре каталогов и стандартным жизненным циклам проекта.
- **Lifecycle:** встроенные фазы сборки: `validate` → `compile` → `test` → `package` → `verify` → `install` → `deploy`.
- **Plugins:** расширяют функциональность (анализ кода, генерация документации, тесты и др.).
- **Coordinates (координаты):** `groupId`, `artifactId`, `version` в `pom.xml` определяют идентификатор и версию артефакта.
- **Repositories:** загрузка и кэширование зависимостей из Maven Central или других репозиториев.

Пример `pom.xml`:

```
<project xmlns="http://maven.apache.org/POM/4.0.0">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
```

```
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Преимущества:

- Единая стандартизованная структура упрощает поддержку больших команд.
- Чёткие фазы сборки и мощная экосистема плагинов покрывают большинство задач «из коробки».
- Автоматическое управление транзитивными зависимостями.

Недостатки:

- Жёсткие соглашения могут ограничивать гибкость при нетипичных сценариях.
- XML-конфигурации `pom.xml` становятся громоздкими в крупных проектах.
- Разрешение зависимостей и запуск сборки может занимать больше времени по сравнению с более современными системами.

---

7. Gradle

Gradle сочетает декларативный и программируемый подход с помощью Kotlin или Groovy DSL, обеспечивая гибкость и производительность.

Основные характеристики:

- **DSL вместо XML:** настраиваете сборку через код на Kotlin (`.kts`) или Groovy (`.gradle`), а не через громоздкие XML-файлы.
- **Tasks:** каждая задача (`task`) представляет одно действие, задачи связываются в ацикличный граф зависимостей.
- **Plugins:** широкий набор плагинов (Java, Kotlin, Application, Android и др.) для расширения функциональности.
- **Modules:** поддержка многомодульных проектов — каждый модуль собирается в свой артефакт.

- **Repositories:** возможность подключения Maven, Ivy, FlatDir хранилищ для управления зависимостями.
- **Dependency Management:** лаконичное объявление зависимостей, управление транзитивными связями.
- **Language Agnostic:** поддержка JVM-языков (Java, Kotlin, Scala), C/C++, JavaScript, COBOL и др.
- **Incremental build & cache:** автоматическая проверка входов/выходов задач (UP-TO-DATE, FROM-CACHE) для ускорения повторных сборок.
- **Wrapper:** скрипт `gradlew` гарантирует, что на всех машинах используется одна и та же версия Gradle.
- **Version Catalog:** централизованное управление версиями зависимостей и плагинов в одном месте.

Пример `build.gradle.kts`:

```
plugins {
    kotlin("jvm") version "1.8.0"
    application
}

group = "org.example"
version = "1.0-SNAPSHOT"

repositories {
    mavenCentral()
}

dependencies {
    implementation("io.ktor:ktor-server-core:2.0.0")
    testImplementation("org.jetbrains.kotlin:kotlin-test:1.8.0")
}

tasks.test {
    useJUnitPlatform()
}

application {
    mainClass.set("MainKt")
}
```

Преимущества:

- Максимальная гибкость конфигурации и расширения через код.

- Высокая скорость инкрементальных сборок и использование кэша.
- Единый механизм управления версиями зависимостей и плагинов.
- Поддержка различных языков и платформ.

Недостатки:

- Более высокий порог вхождения по сравнению с Maven или Make.
- Зависимость от Kotlin/Groovy-знаний для написания DSL.
- Иногда бывает сложнее отлаживать скрипты и понимать внутреннюю модель проекта.