

Управление конфигурацией ПО (часть 2)

Алексей Островский

Физико-технический учебно-научный центр НАН Украины

30 апреля 2015 г.

Построение системы

Определение

Построение системы (англ. *system building*) — процесс создания полной исполняемой версии программной системы путем компиляции и связывания (англ. *linking*) компонентов программы, внешних библиотек, файлов конфигурации и т. п.

Входные данные

исходный код;
файлы конфигурации;
файлы данных (напр., локализация);
внешние библиотеки;
компиляторы и другие инструменты;
тесты.

Выходные данные

исполняемые файлы;
документация;
результаты тестирования;
упаковка исполняемых файлов (напр., JAR-архивы при разработке в Java);
развертывание ПО на целевой системе.

Инструменты построения

Требования к инструментам построения:

- ▶ Комплексное использование компиляторов (напр., gcc) и компоновщиков (напр., ld) для создания исполняемого кода, готового к развертыванию.
- ▶ Минимизация количества повторных действий: отслеживание изменившихся исходных файлов, чтобы компилировать / компоновать только изменившиеся модули.
- ▶ Использование промежуточных артефактов (напр., объектных файлов) для повышения модульности и ускорения построения.
- ▶ Отчеты об ошибках: остановка построения при ошибке компиляции или компоновки; возможность быстро локализовать ошибку.

Инструменты построения

Требования к инструментам построения (продолжение):

- ▶ Конфигурация построения: возможность задания параметров, влияющих на особенности построения системы (напр., дополнительные опции для компиляторов / компоновщиков; архитектура целевой системы).
- ▶ Тестирование: автоматическая прогонка модульных / интеграционных тестов после компоновки для определения корректности поведения системы.
- ▶ Интеграция с системой управления версиями: запрос актуальных версий компонентов из хранилища перед началом построения.
- ▶ Дополнительные режимы построения для создания документации и других вспомогательных ресурсов.

Классификация инструментов построения

- ▶ **Самостоятельные** — инструменты, вызываемые вручную (напр., из командной строки).

Достоинства: большая гибкость; четкая спецификация действий при построении.

Недостатки: большой объем конфигурационных файлов.

- ▶ **Интегрированные** — утилиты, встроенные в интегрированные среды разработки (IDE).

NB. Интегрированные инструменты построения часто основаны на самостоятельных (напр., построение в Visual Studio работает на основе MSBuild).

Достоинства: простота использования; минимальная потребность в конфигурации.

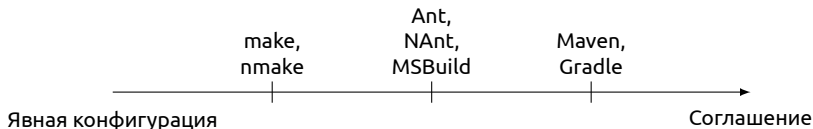
Недостатки: недостаточная гибкость; потребность в дополнительном ПО для построения системы.

Классификация инструментов построения

| Утилита | Язык проекта | Язык конфигурации |
|--------------|----------------------|---|
| make | любой | Makefile |
| Apache Ant | Java | XML |
| NAnt | языки .NET | XML |
| MSBuild | языки .NET, другие | XML |
| Apache Maven | Java | XML / модель проекта (англ. <i>Project Object Model</i>) |
| Gradle | Java, Groovy, другие | DSL на основе Groovy |

Примечание. Большинство инструментов построения в разной степени поддерживают плагины, расширяющие список допустимых действий.

Классификация инструментов построения



- ▶ **make** — отсутствие каких-либо априорных сведений о структуре программной системы; возможно построение произвольной системы с использованием любых инструментов.
- ▶ **Apache Ant** — минимальные сведения о структуре программной системы (проект Java); построение с применением утилит разработки Java (javac, jar, javadoc, ...).
- ▶ **Apache Maven** — неявное соглашение о структуре программной системы (проект Java с размещением исходного кода в фиксированных директориях), заданные наперед цели построения. Конфигурационные файлы описывают *отступления* от соглашения.

Цели построения

Определение

Цель построения (англ. *build target*) — режим работы инструмента построения, который специфицируется при вызове инструмента (например, как параметр командной строки) и означает проведение действий для создания определенных промежуточных или конечных программных артефактов.

Примеры целей построения:

- ▶ компиляция всех исходных файлов;
- ▶ (*зависит от предыдущего*) компоновка полученных объектных файлов в единый исполняемый файл;
- ▶ создание документации;
- ▶ удаление промежуточных артефактов построения (напр., объектных файлов).

Зависимости

Наблюдение

Режим построения может зависеть от успешного выполнения другого режима построения.

Пример: компоновка объектных файлов зависит от компиляции исходного кода.

Организация режимов построения:

- ▶ Организация в виде ациклического ориентированного графа: декларации зависимости (ребра) между режимами (вершины); режимы выполняются согласно топологической сортировке.

Примеры утилит: make, Apache Ant.

- ▶ Линейная организация (жизненный цикл): режимы построения выполняются в фиксированном порядке.

Примеры утилит: Apache Maven.

Make

Определение

Make — утилита для автоматического построения произвольных проектов из исходных файлов, использующая сценарии построения (Makefile).

Поддерживаемые ОС: *NIX; Windows (через Cygwin; nmake — аналог из MS Visual Studio).

Достоинства:

- ▶ построение произвольных программных проектов;
- ▶ отсутствие необходимости в плагинах для задействования внешних инструментов;
- ▶ возможность динамического добавления правил во время построения.

Недостатки:

- ▶ большой объем файлов конфигурации, затраты на их создание и поддержку (решается использованием генераторов);
- ▶ проблемы переносимости между ОС.

Составляющие Makefile

- ▶ **Макросы** — ~ переменные в языках программирования, подставляются в правила и действия; позволяют конфигурировать построение (напр., используемые компиляторы).

Синтаксис:

- 1 `macro=value` # определение
- 2 `$(macro)` # использование

- ▶ **Включения** других конфигурационных файлов Makefile.

Синтаксис:

- 1 `include filename`

Составляющие Makefile

- ▶ **Правила** — *зависимости* (другие режимы построения и файлы), которые влияют на *цель* (режим построения или выходной файл / файлы).
- ▶ **Действия** — произвольные команды, выполняемые в оболочке sh для создания выходных файлов.

Синтаксис:

```
1  target [target...]: [dependency...]  
2      command1  
3      command2  
4      ...  
5      commandN
```

Пример Makefile для построения программы на C

```
1  # Декларации макросов.
2  CC=gcc
3  # Первая цель построения (используется по умолчанию).
4  all: main
5
6  # Создание исполняемого файла main на основе объектных файлов.
7  # $@ — встроенный макрос, заменяющийся на текущую цель построения;
8  # $< — встроенный макрос, заменяющийся на первую из зависимостей;
9  # $^ — встроенный макрос, заменяющийся на список зависимостей.
10 main: main.o other.o
11     $(CC) $(LDFLAGS) -o $@ $^
12 # Макросы LDFLAGS (дополнительные опции компоновщика) и CFLAGS (опции компилятора)
13 # не заданы в файле и по умолчанию равны пустой строке.
14 # Их можно переопределить при вызове make: make CFLAGS="-m64" ...
15
16 # Создание объектных файлов на основе исходного кода.
17 main.o: main.c main.h
18     $(CC) -c $(CFLAGS) -o $@ $<
19 other.o: other.c main.h
20     $(CC) -c $(CFLAGS) -o $@ $<
```

Дополнительные возможности make

- ▶ **Правила на основе суффиксов** (англ. *suffix rules*) — шаблон правил для построения определенного типа файлов (напр., объектных файлов *.o на основе исходных файлов *.c).
- ▶ **Шаблонные правила** (англ. *pattern rules*) — обобщение правил на основе суффиксов с шаблонным видом выходных файлов.
- ▶ Использование **подстановочных знаков** (* и ?) в списках зависимостей.
- ▶ **Функции** для преобразования макросов.
- ▶ **Директивы ветвления** для условного выполнения команд или объявления макросов.
- ▶ **Интерпретация кода**, напр., для динамического добавления правил.

Построение с помощью make

```
make && [sudo] make install
```

- ▶ **make:** компилирует и компоует составляющие приложения;
- ▶ **make install:** устанавливает полученные выполняемые файлы / библиотеки в конечный пункт назначения.

Проблема: Определение конфигурации построения (используемых компиляторов, архитектуры целевой системы, ...) средствами make невозможно или чрезвычайно сложно.

Построение с помощью make

```
./configure && make && [sudo] make install
```

- ▶ **configure:** определяет конфигурацию построения и создает файлы, включаемые в основной Makefile.
- ▶ **make:** компилирует и компоунет составляющие приложения;
- ▶ **make install:** устанавливает полученные выполняемые файлы / библиотеки в конечный пункт назначения.

Проблема: Создание сценария конфигурации вручную занимает много усилий.

Построение с помощью make

```
autoconf && ./configure && make && [sudo] make install
```

- ▶ **autoconf:** Создает сценарий конфигурации `configure` на основе списка проверок и т. п.; генерирует сценарии `Makefile` с помощью шаблонов.
- ▶ **configure:** определяет конфигурацию построения и создает файлы, включаемые в основной `Makefile`.
- ▶ **make:** компилирует и компоует составляющие приложения;
- ▶ **make install:** устанавливает полученные выполняемые файлы / библиотеки в конечный пункт назначения.

Apache Ant

Определение

Apache Ant — утилита для автоматического построения проектов, написанных на языке программирования Java.

Поддерживаемые ОС: произвольные (требуется Java Runtime).

Достоинства:

- ▶ портируемость (базовые задания не зависят от операционной системы);
- ▶ широкие возможности по фильтрации файлов для передачи в команды;
- ▶ интеграция с системой тестирования JUnit.

Недостатки:

ограниченность встроенных команд (для расширения требуются внешние плагины).

Формат файлов конфигурации Ant

- ▶ **Переменные** (могут подставляться в аргументы команд).

Синтаксис: `<property name="имя" value="значение" />`

- ▶ **Цели построения.**

Синтаксис:

```
<target name="имя" description="описание"
        depends="зависимости (цели построения)">
    команды
</target>
```

- ▶ **Команды.**

Синтаксис: XML-тег с атрибутами и / или внутренними элементами, зависящими от типа команды.

Пример файла конфигурации Ant

```
1  <project name="Test" default="compile">
2    <property name="dir.src" value="src"/>
3    <property name="dir.dest" value="classes"/>
4    <property name="jar" value="test.jar"/>
5    <target name="compile" description="Compile Java sources">
6      <mkdir dir="${dir.dest}"/>
7      <javac srcdir="${dir.src}" destdir="${dir.dest}"/>
8    </target>
9    <target name="jar" depends="compile" description="Create JAR archive">
10     <jar destfile="${jar}">
11       <fileset dir="${dir.dest}" includes="**/*.class"/>
12       <manifest>
13         <attribute name="Main-Class" value="test.Program"/>
14       </manifest>
15     </jar>
16   </target>
17   <target name="clean" description="Remove intermediate files">
18     <delete dir="${dir.dest}"/>
19   </target>
20 </project>
```

Непрерывная интеграция

Определение

Непрерывная интеграция (англ. *continuous integration, CI*) — метод разработки ПО, основанный на частой (несколько раз в день) фиксации всех изменений, вносимых в систему разработчиками.

Автор: Гради Буч (Grady Booch).

Связанные технологии:

- ▶ экстремальное программирование (англ. *extreme programming, XP*);
- ▶ разработка через тестирование (англ. *test-driven development, TDD*).

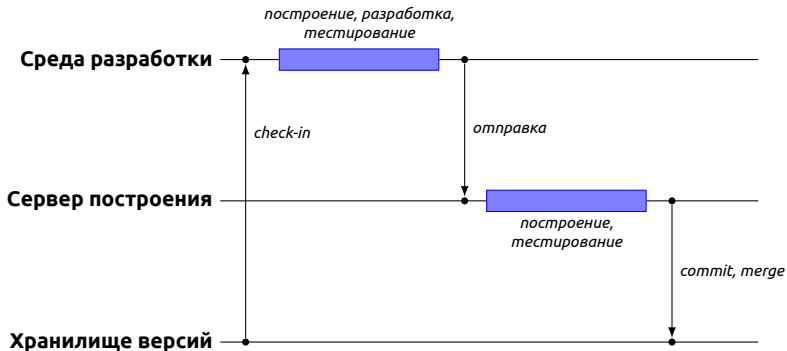
Цели:

- ▶ предотвращение проблем интеграции между компонентами программной системы;
- ▶ проверка изменений с помощью систем автоматического тестирования.

Принципы непрерывной интеграции

- ▶ **Система управления версиями**, охватывающая все артефакты, необходимые для построения проекта;
- ▶ **инструменты построения** для автоматического создания промежуточных выпусков;
- ▶ **автоматическое тестирование** системы при построении;
- ▶ **регулярное фиксирование изменений** (не реже раза в день) с построением проекта после фиксирования;
- ▶ **выделенный сервер** для построения / тестирования (копия целевой системы);
- ▶ **прозрачность и доступность результатов построения** для разработчиков и заказчиков.

Разработка согласно CI



При разработке согласно CI после внесения изменений в программную систему и успешного построения на компьютере разработчика приложение строится на тестовом сервере.

Преимущества и недостатки CI

Преимущества:

- ▶ быстрая локализация дефектов интеграции;
- ▶ более равномерное распределение внесения изменений в систему;
- ▶ постоянная доступность актуальной рабочей версии программной системы для тестирования, демонстрирования или выпуска;
- ▶ возможность быстрого отката дефектного кода без существенной потери функциональности.

Недостатки:

- ▶ необходимость устройства выделенного сервера тестирования с параметрами, сходными с целевой системой;
- ▶ для больших систем построение может занимать много времени.

Выпуски ПО

Определение

Выпуск ПО (англ. *software release*) — версия программной системы, предназначенная для использования вне отдела разработки.

Аспекты управления выпусками:

- ▶ идентифицируемость — определение версий исходных файлов, инструментов, среды, задействованных в построении системы;
- ▶ повторяемость — возможность повторного построения системы (\Rightarrow сохранение версий исходных файлов и т. п.);
- ▶ согласованность — автоматизация создания выпусков и их составляющих (напр., программ установки ПО).

Состав выпуска

Обязательные элементы:

- ▶ выполняемый код (двоичные выполняемые файлы, библиотеки, сценарии, ...)

или

- ▶ исходные файлы для построения программы в среде пользователя.

Необязательные элементы:

- ▶ программа установки;
- ▶ файлы конфигурации;
- ▶ файлы данных (напр., локализация сообщений, используемых в программе);
- ▶ электронная и печатная документация.

Жизненный цикл выпусков ПО

Технические (внутренние) выпуски:

- ▶ рабочий выпуск (англ. *development release*, *pre-alpha*, *nightly build*) — для использования разработчиками: создания архитектуры, проектирования и кодирования компонентов системы и т. п.;
- ▶ альфа-выпуск (англ. *alpha release*) — для тестирования по методу белого ящика;
- ▶ бета-выпуск (англ. *beta release*) — для тестирования по методу черного ящика;
- ▶ кандидат (англ. *release candidate*, *RC*) — бета-выпуск, в котором устранено большинство дефектов, потенциально готовый для публичного использования.

Публичные выпуски:

- ▶ RTM (*release to manufacturing*) — выпуск, предназначенный для распространения и развертывания на системах потребителей.

Планирование выпусков

Факторы, влияющие на расписание выпусков:

- ▶ обнаруженные дефекты, требующие исправление в виде отдельного выпуска или патча;
- ▶ изменение среды выполнения (напр., новая версия операционной системы или используемых внешних библиотек);
- ▶ законы эволюции ПО:
 - ▶ необходимость добавления новой функциональности (в т. ч. для успешной конкуренции);
 - ▶ исправление ошибок, связанных с новыми функциями;
- ▶ расписание, составленное отделом маркетинга;
- ▶ требования заказчика.

Нумерация версий ПО

Цели:

- ▶ идентификация выпусков;
- ▶ определение совместимости API различных выпусков;
- ▶ определение совместимости между программами и зависимостей в репозиториях приложений;
- ▶ маркетинг.

Схемы нумерации:

- ▶ целые числа (1, 2, 3, ...);
- ▶ десятичные дроби (1.0, 1.01, 1.1, 1.2, ...);
- ▶ последовательность чисел (1.0.0, 1.0.1, 1.0.2, 1.1.0, ...);
- ▶ даты;
- ▶ произвольные строки.

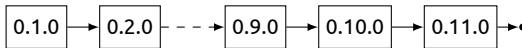
Семантическая нумерация

Формат версии: `major.minor[.patch[-suffix]]`

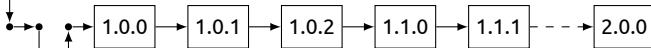
- ▶ **major** — главная версия, целое неотрицательное число, увеличивающееся при внесении значительных изменений (напр., несовместимых модификаций API); **major = 0** означает бета-выпуск;
- ▶ **minor** — второстепенная версия, целое неотрицательное число, увеличивающееся при внесении обратно совместимых изменений;
- ▶ **patch** — версия построения, целое неотрицательное число, увеличивающееся при каждом построении системы (напр., при исправлении ошибок). По умолчанию равна нулю.
- ▶ **suffix** — алфавитно-цифровая последовательность для обозначения внутренних выпусков, напр., `alpha`, гс.1.

Пример нумерации версий ПО

Бета-выпуски



**Публичные
выпуски**



**Технические
выпуски**



Примечание. Технические выпуски создаются для каждого или почти каждого публичного выпуска ПО; они не показаны на схеме из-за нехватки места.

Выводы

1. Управление версиями ПО связано с двумя другими аспектами управления конфигурацией — построением программной системы и управлением выпусками.
2. Инструменты автоматического построения ПО различаются областью применения. Наиболее универсальная утилита построения — `make` — может выполнять построение произвольных проектов. Для Java-проектов одним из популярных средств построения является `Apache Ant`.
3. Автоматическое построение и управление версиями используются в непрерывной интеграции (*continuous integration*) для устранения ошибок интеграции при коллективной разработке.
4. Выпуски ПО — это версии программной системы, предназначенные для тестирования (технические выпуски) или использования потребителями (публичные выпуски). Для идентификации выпусков используются различные способы нумерации версий, напр., семантическая нумерация (*semantic versioning*).

Материалы



Sommerville, Ian

Software Engineering.

Pearson, 2011. — 790 p.



GNU

GNU Make Manual.

<https://www.gnu.org/software/make/manual/make.html>



Fowler, Martin

Continuous Integration.

<http://martinfowler.com/articles/continuousIntegration.html>



Preston-Werner, Tom

Semantic versioning.

<http://semver.org/>

Спасибо за внимание!