ЛУЦЕНКО Д. Ю.
СБОРКА (CI / CD) ПРОЕКТОВ, НЕ ИСПОЛЬЗУЮЩИХ JVM С ПОМОЩЬЮ GRADLE / KOTLIN

УДК 004.912:332.62, ВАК 05.13.11, ГРНТИ 50.05.19

Сборка (CI / CD) проектов, не использующих JVM с помощью Gradle / Kotlin

Д. Ю. Луценко

Санкт-Петербургский политехнический университет Петра Великого;
г. Санкт-Петербург

Для JVM проектов Gradle является общепризнанным инструментом. Он также используется в проектах и вне платформы JVM. Например, в официальной документации описаны сценарии использования для Swift и C++ проектов. Также на практике gradle применим для автоматизации сборки, тестирования и развертывания проектов, которые включают в себя модули node.js, golang, terraform.

Ключевые слова: JVM, Gradle, проект, автоматизация сборки, развертывание проектов.

Build (CI/CD) of non-JVM projects using Gradle / Kotlin

D. Yu. Lutsenko

Peter the Great St. Petersburg Polytechnic University,
St. Petersburg

For JVM projects, Gradle is an established tool. It is also used in projects and outside the JVM framework. For example, the official documentation describes use cases for Swift and C ++ projects. Also in practice, gradle is applicable to automate the build, testing and deployment of projects that include modules node.js, golang, terraform.

Keywords: JVM, Gradle, project, build automation, project deployment.

Введение

Каждый из модулей большого проекта разрабатывается отдельной командой в собственном репозитории. При этом также было бы удобно работать с подобными проектами как с единой системой: обеспечить единые настройки для проектов, выполнить интеграционное тестирование, иметь возможность выполнять развертывание в различных конфигурациях и выпускать последовательные релизы. Довольно удобно подключать репозитории проектов к единому, главному репозиторию с помощью подмодулей git. В этот же момент можно работать с одной общей версией всех подпроектов. Проверка каждого подпроекта будет производиться при соответствующем коммите. В случае реализации функционала, который затрагивает несколько подпроектов одновременно, можно создать ветвь в проекте верхнего уровня и указать вложенную ветвь для использования каждого из подпроектов. Таким образом, можно синхронизировано разрабатывать и тестировать новую функциональность локально.

Краткий обзор того, как работает Gradle

Фаза инициализации [1]. Сначала Gradle ищет settings.gradle.kts [2], компилирует и запускает его, чтобы получить список подпроектов и их расположение. Он компилирует только подвергнутые изменениям файлы по мере необходимости. Если зависимости и файл не были изменены, то будет использоваться последняя скомпилированная версия.

Фаза конфигурации [1]. Скрипты сборки определены для всех проектов и некоторые из них выполняются (только те проекты, которые необходимы для целевых задач).

Фаза выполнения [1]. На основе построенного частичного графа зависимостей между задачами определяется подграф, необходимый для достижения выполнения текущих конкретных задач. Для каждой задачи проверяется её актуальность: должна ли задача выполняться или нет. И тогда выполняются лишь необходимые задачи.

Сценарий сборки, основанный только на графе задач, является сложным в обслуживании императивным сценарием. Чтобы организовать похожие списки задач, связанных с разными модулями, в gradle есть концепции проектов и плагинов. Проект - это модуль, который представляет часть исходного кода более крупного решения, а плагин - это многократно используемый набор взаимосвязанных задач, которые создаются для конкретного проекта. Подобные концепции существуют и в maven.

DSL (предметно-ориентированный язык)

Gradle использует гибкий подход к организации скрипта сборки, основанный на идее встроенного предметно-ориентированного языка. В основном языке (Groovy или Kotlin) функции, объекты и классы разработаны специальным образом, так чтобы при их использовании получаются легко воспринимаемые скрипты, аналогичные декларативному описанию проекта. То есть, несмотря на то, что скрипт сборки является императивной программой, он может выглядеть как декларативное описание конфигурации плагинов и структуры проекта.

Встроенный проект buildSrc

Настройка сборки проекта в основном выполняется в скрипте build.gradle.kts [2]. Помимо прочего, данный скрипт позволяет создавать специальные задачи и выполнять произвольный код. Если не следовать рекомендациям, скрипт сборки может довольно быстро превратится в спагетти-код. Поэтому создание задач и использование исполняемого кода внутри сценария сборки следует рассматривать в качестве исключения и временного решения. Следует помнить, что поддержка сборки проекта с императивной логикой в сценариях сборки крайне сложна и не удобна.

Gradle предлагает удобное решение с использованием вспомогательного проекта buildSrc [3]. Этот проект можно рассматривать как основное место для императивной логики и пользовательских задач. Проект buildSrc компилируется автоматически и добавляется как зависимость к скрипту сборки. Так что все, что

в нем указано, будет доступно для использования в скриптах без дополнительных усилий.

Таким образом, в сценарии сборки должны оставаться только декларативные элементы, такие как объявление и конфигурации подключаемых модулей и настройки проекта.

BuildSrc - это старый JVM проект. В нем существует возможность добавлять ресурсы, писать тесты и реализовывать любую логику, необходимую для сценариев сборки. Проект buildSrc также имеет собственный сценарий сборки, соответственно можно назвать это «рекурсивной сборкой». Результатом сборки данного вспомогательного проекта являются классы, которые будут автоматически добавлены в classpath всех проектов. То есть, при объявлении плагина в buildSrc [3], этот плагин будет доступен для использования во всех проектах и подпроектах без необходимости дополнительной настройки.

Также следует отметить, что buildSrc не содержит скриптов, которые будут выполняться в фазе конфигурации. То есть, при необходимости создания каких-либо задач, нужно обратиться к коду: либо путем прямого вызова функции, либо с помощью плагина.

Плагины

Для различных типов проектов (go, node.js, terraform, ...) может возникнуть необходимость создания плагинов. Можно использовать уже существующие плагины (к примеру, kosogor для terraform), но функциональности данных плагинов может быть недостаточно.

Плагин может быть реализован непосредственно в buildSrc [3] или как отдельный проект для возможности его повторного использования. Если используются отдельные проекты, то необходимо либо подключить эти проекты в качестве импортированной сборки, либо выложить артефакты в уже развернутый репозиторий.

Плагин можно рассматривать как набор следующих элементов:

1. декларативная конфигурация;
2. скрипт / процедура создания задач на основе конфигурации;
3. возможность привязки синглтона к отдельному проекту и его настройка.

Некоторым неудобством плагинов является необходимость создания как задач, так и расширений (объектов конфигурации). Только после этого появляется возможность настройки плагина. Эта процедура не очень удобна, так как на момент создания задач конфигурация еще отсутствует. Следовательно, необходимо использовать более сложный механизм свойств и провайдеров. Такой подход позволяет работать с будущими значениями, которые будут доступны только на этапе выполнения. В то же время важно не использовать значения свойств на этапе конфигурации, поскольку они будут иметь значения по умолчанию.

Пользовательский DSL

Помимо самих плагинов, аналогичного результата можно было бы достичь с помощью вызова функций, которые создают задачи. В качестве примера рассмотрим, как задачи добавляются в библиотеку kosogor с помощью DSL [4]:

```
terraform {
    config {
        tfVersion = "0.10.10"
    }
    root("ex", File(projectDir, "terraform"))
}
@TerraformDSLTag
fun Project.terraform(configure: TerraformDsl.() -> Unit) {
    terraformDsl.project = this
    terraformDsl.configure()
}
```

Код, который пользователь пишет внутри { }, будет выполняться для объекта типа TerraformDsl. Например, метод root создает задачи, используя имя и конфигурацию, переданные методу:

```
@TerraformDSLTag
fun root(name: String, dir: File, enableDestroy: Boolean = false,
    targets: LinkedHashSet<String> = LinkedHashSet(), workspace: String? =
null) {
    val lint = project!!.tasks.create("$name.lint", LintRootTask::class.java) {
task ->
        task.group = "terraform.$name"
        task.description = "Lint root $name"
        task.root = dir
    }
    // ...
}
```

Использование методов в Kotlin, которые принимают последним параметром функции типа Type. () -> Unit, позволяет создать DSL, который выглядит удобно и элегантно. В некоторой степени это обеспечивает большую гибкость и удобство, чем плагины. Например, когда запущен метод «root», предыдущий метод «config» уже был завершен, и все параметры конфигурации доступны напрямую.

Почему важно добиваться инкрементальной сборки?

Сборка проекта может запускаться сотни раз в день. Любая лишняя работа, выполняемая скриптом сборки, может привести к заметной потере времени. В крайних случаях, когда процесс сборки занимает 10-30 минут, сложность работы значительно возрастает. Если сборка выполняется в облаке и результат необходимо развернуть в нескольких конфигурациях, то длительная работа скрипта также может привести к увеличению временных затрат.

«Инкрементность» не появляется сама по себе. Сборка становится более инкрементальной, когда все задачи поддерживают это свойство. В идеале, при повторном запуске одной и той же команды Gradle, она должна завершиться за доли секунды, так как все задачи будут пропущены.

Автоматические зависимости между задачами на основе свойств и файлов

Если задача B зависит от результата выполнения задачи A, то существует возможность настроить эти задачи таким образом, чтобы gradle мог догадаться, что необходимо выполнить задачу A, даже без явного указания зависимости.

Для этого gradle предоставляет механизм свойств и провайдеров [5]. На этапе конфигурации значения инкапсулируются в провайдерах и не доступны напрямую. Если существует функциональная зависимость одного значения от другого в поставщике значений, а внутри лямбда-выражения значение будет доступно в качестве аргумента, и можно будет работать с будущим значением свойства. В результате будет создан новый «провайдер», который вычисляет значение выражения по запросу на этапе выполнения. Пример:

```
class Task1: DefaultTask() {
    @OutputFile
    val res = project.objects.fileProperty()
    init {
        result.convention(project.buildDir.file("res.txt"))
    }
}
class Task2: DefaultTask() {
    @InputFile
    val input = project.objects.fileProperty()
    @Action
    fun task2() {
        println(input.get().asFile.absolutePath)
    }
}
```

В скрипте нет необходимости явно объявлять зависимость, при условии, что существует связь между свойствами:

```
val task1 = tasks.register<Task2>("task1") {
    output.set(file("out.txt"))
}
tasks.register<Task2>("task2") {
    input.set(task1.res)
}
```

Теперь при вызове task2 задача task1 будет рассматриваться как зависимость и при необходимости выполняться.

Использование файлов в качестве сигналов, после обращений к ним

При выполнении операций, которые приводят только к побочным эффектам (например, развертывание в облаке) и не отражаются естественным образом в файловой системе, Gradle не может проверить, необходимость выполнения задачи [5]. В результате соответствующая задача будет каждый раз выполняться.

Чтобы помочь Gradle, можно создать task2.done по завершении задачи и указать, что этот файл является выходным файлом для задачи. В этом файле желательно в сжатом виде отразить описание того, в каком состоянии находится облачная конфигурация [6]. Можно, например, указать SHA развернутой конфигурации или текст со списком развернутых компонентов и их версий.

Если несколько задач изменяют состояние общего облака, то полезно представить это состояние в виде одного или нескольких файлов, совместно используемых этими задачами (cloud.state). Каждая задача, которая изменяет состояние в облаке, также изменяет локальные файлы. Таким образом, Gradle поймет, какие задачи могут потребовать перезапуска.

Централизованная установка номеров портов

Для тестирования может потребоваться запуск конфигурации с другим набором служб. В паре сервисов, зависящих друг от друга, порт взаимодействия нужно указывать дважды - в самом сервисе и в клиенте этого сервиса. Понятно, что по принципу единственности версии (SVOT / SSOT) порт необходимо настраивать только единожды, а в других местах он должен ссылаться на этот доверенный источник. Одна конфигурация службы должна быть доступна как для службы, так и для клиента.

Рассмотрим пример того, как это можно сделать в gradle.

```
data class Service1Config(val port: Int, val path: String) {
   fun localUrl(): URL = URL("http://localhost:$port/$path")
}
```

В основном скрипте build.gradle.kts создается конфигурация и она помещается в extra.

```
val service1Config: Service1Config by extra(Service1Config(8081, "service1/test"))
```

Также в других скриптах существует возможность получить доступ к этой конфигурации, объявленной в RootProject:

```
val service1Config: Service1Config by rootProject.extra
```

Таким образом, можно связать службы и централизовать настройку.

Полезные советы

1. Чтение документации.
2. Изучение модели Gradle.
3. Использование buildSrc. При построении проектов часто возникает необходимость добавления отдельных вспомогательных задач. Такие задачи следует помещать в buildSrc.

4. Стараться делать каждую задачу инкрементальной. В таком случае изменение любой строчки кода приведет только к выполнению строго необходимых задач. Сборка будет выполнена максимально быстро.

Заключение

В данной статье были рассмотрены некоторые особенности системы сборки проектов на основе gradle / kotlin. Gradle является довольно удобным инструментом для сборки проектов, не использующих с JVM. Даже для проектов, не связанных с JVM, сохраняются почти все преимущества - модульность, производительность и защита от ошибок. При соблюдении рекомендуемых правил и подходов к разработке сборки проекта и общих инженерных принципов, Gradle позволяет получить гибкую и удобную в поддержке систему.

Список использованных источников и литературы

1. Мушко Б. и Доктер Г. Gradle в действии. – Издательство Manning Publications, 2013. — 456 p.
2. Праймер Gradle Kotlin DSL Primer, документация Gradle [Электронный ресурс]. – Режим доступа: https://docs.gradle.org/current/userguide/kotlin_dsl.html (дата обращения: (10.05.2021).
3. Организация проектов Gradle [Электронный ресурс]. – Режим доступа: https://docs.gradle.org/current/userguide/organizing_gradle_projects.html (дата обращения: (12.05.2021).
4. Разработка собственных плагинов Gradle [Электронный ресурс]. – Режим доступа: https://docs.gradle.org/current/userguide/custom_plugins.html (дата обращения: (12.05.2021).
5. Берглунд, Т. Грейдл. За пределами основ. - O'Reilly Media, 2013. – 80 с.
6. Иккинк, Х. К. Грейдл. Руководство по эффективному внедрению. – Packt Publishing, 2012. – 350 p.

List of references

1. Muschko, B. and Dockter, H. Gradle in Action. — Manning Publications Company, 2013. — 456 p.
2. Gradle Kotlin DSL Primer. URL: https://docs.gradle.org/current/userguide/kotlin_dsl.html, accessed May 10, 2021.
3. Organizing Gradle Projects. URL: https://docs.gradle.org/current/userguide/organizing_gradle_projects.html, accessed May 12, 2021.
4. Developing Custom Gradle Plugins. URL: https://docs.gradle.org/current/userguide/custom_plugins.html, accessed May 12, 2021.
5. Berglund, T. Gradle Beyond the Basics. — O'Reilly Media, 2013. — 80 p.
6. Ikkink, H. K. Gradle Effective Implementation Guide. — Packt Publishing, 2012. — 350 p.