Бесплатная электронная книга

учусь gradle

Free unaffiliated eBook created from
Stack Overflow contributors.

#gradle

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: gradle

It is an unofficial and free gradle ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official gradle.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с градиентом

замечания

Gradle - инструмент для создания общего назначения с открытым исходным кодом. Он популярен в сообществе Java и является предпочтительным инструментом сборки для Android .

Выделенные функции Gradle

- Декларативные скрипты сборки - *это* код, написанный в Groovy или Kotlin .
- Множество основных и общинных плагинов, которые используют гибкий, основанный на соглашения подход
- Инкрементальные сборки, такие, что задачи, чьи зависимости не изменились, не повторяются.
- Встроенное разрешение зависимостей для Maven и Ivy . Добавленные плагины обеспечивают разрешение зависимостей от других `repositories` таких как npm .
- Первоклассные многопроектные сборки.
- Интеграция с другими инструментами построения, такими как Maven , Ant и другие.
- Build Scans, которые увеличивают способность разработчиков взаимодействовать и оптимизировать сборки Gradle.

Дополнительная информация

Если вы хотите узнать больше о функциях Gradle, посмотрите на обзорную часть Руководства пользователя Gradle .

Если вы хотите попробовать Gradle, ознакомьтесь с руководствами здесь . Вы можете пройти руководство по быстрому старту Java, узнать, как использовать Gradle в первый раз, и перенести с другого инструмента сборки.

Examples

Установка подставки

Требования: Установленная Java JDK или JRE (версия 7 или выше для версии Gradle 3.x)

Этапы установки:

1. Скачать Gradle распространение с официального сайта
2. Распаковать ZIP
3. Добавьте переменную среды `GRADLE_HOME` . Эта переменная должна указывать на

распакованные файлы с предыдущего шага.
4. Добавьте `GRADLE_HOME/bin` в `PATH` среды `PATH` , чтобы вы могли запускать Gradle из интерфейса командной строки (CLI)
5. Проверьте установку Gradle, набрав `gradle -v` в CLI. Выход должен содержать установленную версию Gradle и текущие настройки конфигурации Gradle

Более подробную информацию можно найти в официальном руководстве пользователя

Установка с homebrew на OS X / macOS

Пользователи доморощенного могут установить град, выполнив

```
brew install gradle
```

Установка с помощью SdkMan

Пользователи SdkMan могут установить Gradle, запустив:

```
sdk install gradle
```

Установка конкретной версии

```
sdk list gradle
sdk install gradle 2.14
```

Версия для переключения

```
sdk use gradle 2.12
```

Установите плагин Gradle для Eclipse

Вот шаги, необходимые для установки плагина Gradle в Eclipse:

1. Открыть Eclipse и перейти в **Help** -> **Eclipse Marketplace**
2. В строке поиска введите **buildship** и нажмите Enter.
3. Выберите **«Buildship Gradle Integration 1.0»** и нажмите « **Установить».**
4. В следующем окне нажмите « **Подтвердить».**
5. Затем **примите** условия и лицензию соглашения, затем нажмите « **Готово»**
6. После установки Eclipse необходимо перезапустить, нажмите **Да**

Привет, мир

Задачи Gradle могут быть записаны с использованием кода Groovy из файла build.gradle проектов. Эти задачи затем могут быть выполнены с использованием `> gradle [taskname]` на терминале или путем выполнения задачи из среды IDE, такой как Eclipse.

Чтобы создать пример Hello World в градиенте, мы должны определить задачу, которая будет печатать строку на консоли с помощью Groovy. Мы будем использовать `printLn` Groovy для вызова метода `System.out.printLn` Java для печати текста на консоли.

build.gradle

```
task hello {
    doLast {
        println 'Hello world!'
    }
}
```

Затем мы выполним эту задачу, используя `> gradle hello` or `> gradle -q hello` . `-q` используются для подавления Gradle лога сообщений , так что только выход задачи будет показан.

Вывод `> gradle -q hello` :

```
> gradle -q hello
Hello world!
```

Подробнее о задачах

Прежде всего: оператор `<<` (leftShift) эквивалентен `doLast {closure}` clos `doLast {closure}` . Из **градации 3.2** она **устарела** . Весь код задачи записывается в **build.gradle** .

> Задача представляет собой некоторый атомный кусок работы, который выполняет сборка. Это может быть компиляция некоторых классов, создание JAR, создание Javadoc или публикация некоторых архивов в репозитории.

Gradle поддерживает два больших типа задач: простой и расширенный.

Давайте рассмотрим некоторые стили определения задачи:

```
task hello {
    doLast{
       //some code
    }
}
```

Или:

```
task(hello) {
    doLast{
       //some code
    }
}
```

Эти задачи выше эквивалентны. Кроме того, вы можете предоставить некоторые

расширения для задачи, такие как: `dependsOn` , `mustRunAfter` , `type` и т. Д. Вы можете расширить задачу, добавив действия после определения задачи, например:

```
task hello {
    doLast{
       println 'Inside task'
    }
}
hello.doLast {
    println 'added code'
}
```

Когда мы выполним это, мы получим:

```
> gradle -q hello
    Inside task
    added code
```

Вопросы о зависимостях задач и упорядочения рассмотрены здесь

Давайте поговорим о двух больших типах задач.

Просто:

Задачи, которые мы определяем с закрытием действия:

```
    task hello {
        doLast{
        println "Hello from a simple task"
        }
    }
```

Повышенная

Усовершенствованный это задача с предварительно настроенным поведением. Все плагины, которые вы используете в своем проекте, являются *расширенными* или **расширенными задачами** . Давайте создадим нашу, и вы поймете, как это работает:

```
task hello(type: HelloTask)

class HelloTask extends DefaultTask {
    @TaskAction
    def greet() {
        println 'hello from our custom task'
    }
```

```
}
```

Кроме того, мы можем передать параметры нашей задаче, например:

```
class HelloTask extends DefaultTask {
    String greeting = "This is default greeting"
    @TaskAction
    def greet() {
        println greeting
    }
}
```

И теперь мы можем переписать нашу задачу так:

```
   //this is our old task definition style
task oldHello(type: HelloTask)
   //this is our new task definition style
task newHello(type: HelloTask) {
    greeting = 'This is not default greeting!'
}
```

Когда мы выполним это, мы получим:

```
> gradle -q oldHello
This is default greeting

> gradle -q newHello
This is not default greeting!
```

Все вопросы о разработке плагинов gradle на официальном сайте

Прочитайте Начало работы с градиентом онлайн: https://riptutorial.com/ru/gradle/topic/894/начало-работы-с-градиентом

глава 2: Auto Increment Version Number Использование скрипта Gradle для приложений для Android

Examples

Как вызвать метод автоматического увеличения при сборке

```
gradle.taskGraph.whenReady {taskGraph ->
    if (taskGraph.hasTask(assembleDebug)) {  /* when run debug task */
        autoIncrementBuildNumber()
    } else if (taskGraph.hasTask(assembleRelease)) { /* when run release task */
        autoIncrementBuildNumber()
    }
}
```

Определение метода автоматического увеличения

```
 /*Wrapping inside a method avoids auto incrementing on every gradle task run. Now it runs
only when we build apk*/
ext.autoIncrementBuildNumber = {

    if (versionPropsFile.canRead()) {
        def Properties versionProps = new Properties()
        versionProps.load(new FileInputStream(versionPropsFile))
        versionBuild = versionProps['VERSION_BUILD'].toInteger() + 1
        versionProps['VERSION_BUILD'] = versionBuild.toString()
        versionProps.store(versionPropsFile.newWriter(), null)
    } else {
        throw new GradleException("Could not read version.properties!")
    }
}
```

Чтение и присвоение номера версии из файла свойств переменной

def versionPropsFile = файл ('version.properties') def versionBuild

```
/*Setting default value for versionBuild which is the last incremented value stored in the
file */
if (versionPropsFile.canRead()) {
    def Properties versionProps = new Properties()
    versionProps.load(new FileInputStream(versionPropsFile))
    versionBuild = versionProps['VERSION_BUILD'].toInteger()
} else {
    throw new GradleException("Could not read version.properties!")
}
```

Прочитайте Auto Increment Version Number Использование скрипта Gradle для приложений для Android онлайн: https://riptutorial.com/ru/gradle/topic/10696/auto-increment-version-number-использование-скрипта-gradle-для-приложений-для-android

глава 3: В том числе родной источник - экспериментальный

параметры

параметры	подробности
model.android.ndk.toolchain	встроенный toolchain, найденный в папке ndk-bundle

Examples

Базовая конфигурация JNI Gradle

root: build.gradle

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle-experimental:0.8.0-alpha4'
    }
}

allprojects {
    repositories {
        jcenter()
    }
}
```

app: build.gradle

```
apply plugin: 'com.android.model.application'

dependencies {
    compile "com.android.support:support-v4:23.3.0"
    compile fileTree(dir: 'libs', include: '*.jar')
}

model {
    android {
        compileSdkVersion = 23
        buildToolsVersion = '23.0.3'

        defaultConfig {
            applicationId = 'com.example.hello'
            minSdkVersion.apiLevel = 9
            targetSdkVersion.apiLevel = 23
```

```
            buildConfigFields {
                create() {
                    type "int"
                    name "VALUE"
                    value "1"
                }
            }
        }

        ndk {
            platformVersion = 9
            moduleName "hello"

            toolchain "clang"

            stl "gnustl_static"
            CFlags.add("-DANDROID_NDK")
            cppFlags.add("-std=c++11")

            ldLibs.add("android")
            ldLibs.add("dl")
            ldLibs.add("log")
        }

        sources {
            main {
                jni {
                    exportedHeaders {
                        srcDirs "../../common/headers"
                    }
                    source {
                        srcDirs "../../common/src"
                    }
                }
            }
        }
    }
}
```

Использование готовых библиотек и OpenGL ES 2.0

root: build.gradle

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle-experimental:0.8.0-alpha4'
    }
}

allprojects {
    repositories {
        jcenter()
    }
}
```

app: build.gradle

```
apply plugin: 'com.android.model.application'

dependencies {
    compile "com.android.support:support-v4:23.3.0"
    compile fileTree(dir: 'libs', include: '*.jar')
}

model {
    android {
        compileSdkVersion = 23
        buildToolsVersion = '23.0.3'

        defaultConfig {
            applicationId = 'com.example.glworld'
            minSdkVersion.apiLevel = 9
            targetSdkVersion.apiLevel = 23

            buildConfigFields {
                create() {
                    type "int"
                    name "VALUE"
                    value "1"
                }
            }
        }

        buildTypes {
            release {
                minifyEnabled = false
                proguardFiles.add(file('proguard-rules.txt'))
            }
        }

        ndk {
            platformVersion = 9
            moduleName "glworld"

            toolchain "clang"

            stl "gnustl_static"
            CFlags.add("-DANDROID_NDK")
            CFlags.add("-DDISABLE_IMPORTGL")
            CFlags.add("-DFT2_BUILD_LIBRARY=1")
            cppFlags.add("-std=c++11")

            ldLibs.add("EGL")
            ldLibs.add("android")
            ldLibs.add("GLESv2")
            ldLibs.add("dl")
            ldLibs.add("log")
        }

        sources {
            main {
                jni {
                    dependencies {
                        library "freetype2" linkage "shared"
                    }
                    exportedHeaders {
```

```
                        srcDirs "../../common/headers"
                    }
                    source {
                        srcDirs "../../common/src"
                    }
                }
            }
        }
    }

    repositories {
        prebuilt(PrebuiltLibraries) {
            freetype2 {
                headers.srcDir "../../common/freetype2-android/include"
                binaries.withType(SharedLibraryBinary) {
                    def localLib = "../../common/freetype2-android/Android/libs"
                    sharedLibraryFile =
                            file("$localLib/${targetPlatform.getName()}/libfreetype2.so")
                }
            }
        }
    }
}

// The next tasks compile a freetype library using a make file.
// These `.so`'s are then used as the shared libraries compiled above.
tasks.withType(JavaCompile) {
    compileTask -> compileTask.dependsOn buildNative
}

// Call regular ndk-build (.cmd) script from the app directory
task buildNative(type: Exec) {
    def ndkDir = "/Development/android-sdk-macosx/ndk-bundle"
    commandLine "$ndkDir/ndk-build",
            '-C',
            file('../../common/freetype2-android/Android/jni').absolutePath
}

task cleanNative(type: Exec) {
    def ndkDir = "/Development/android-sdk-macosx/ndk-bundle"
    commandLine "$ndkDir/ndk-build",
            '-C',
            file('../../common/freetype2-android/Android/jni').absolutePath,
            "clean"
}

clean.dependsOn cleanNative
```

Прочитайте В том числе родной источник - экспериментальный онлайн:
https://riptutorial.com/ru/gradle/topic/4460/в-том-числе-родной-источник---экспериментальный

глава 4: Грейд-обертка

Examples

Грейд-обертка и Гит

Как обсуждалось во введении, функциональность оболочки градиента работает, потому что в проект будет `gradlew` который будет использоваться при `gradlew` команды `gradlew` . Однако это может не произойти, и после того, как в следующий раз проект будет извлечен, `gradlew` не будет работать с ошибкой:

```
Error: Could not find or load main class org.gradle.wrapper.GradleWrapperMain
```

Это будет связано с тем, что ваш .gitignore, скорее всего, включает `*jar` для проектов Java. Когда оболочка градиента была инициализирована, она копируется в файл `gradle/wrapper/gradle-wrapper.jar` . Таким образом, вам нужно добавить его в индекс git и зафиксировать его. Сделайте это с помощью:

```
git add -f gradle/wrapper/gradle-wrapper.jar
git ci
```

С `-f` , чтобы заставить его.

Введение

Gradle имеет возможность добавлять обертку к проектам. Эта оболочка устраняет необходимость использования всеми пользователями или системами непрерывной интеграции Gradle. Он также предотвращает проблемы с версиями, когда существует некоторая несовместимость между версией, которую использует проект, и тем, что пользователи установили. Он делает это, устанавливая версию градиента локально в проекте.

Пользователи проекта просто запускают:

```
> ./gradlew <task> # on *Nix or MacOSX
> gradlew <task>   # on Windows
```

Чтобы настроить проект на использование обертки, разработчики:

1. Выполнение:

```
gradle wrapper [--gradle-version 2.0]
```

Где `--gradle-version X` является необязательной и, если она не указана (или задача

оболочки не включена, как показано ниже), используемая версия является используемой версией градиента.

1. Чтобы заставить проект использовать определенную версию, добавьте следующее в `build.gradle` :

```
task wrapper(type: Wrapper) {
    gradleVersion = '2.0'
}
```

Когда выполняется `gradle wrapper` она создает файлы:

```
the_project/
  gradlew
  gradlew.bat
  gradle/wrapper/
    gradle-wrapper.jar
    gradle-wrapper.properties
```

Официальная документация по этой функции находится по адресу https://docs.gradle.org/current/userguide/gradle_wrapper.html .

Использовать локально поданный Грейдл в Обтекателе Грейдл

Если вы хотите сохранить локальную копию Gradle и позволить Wrapper использовать ее в сборках, вы можете установить `distributionUrl` указывающий на вашу копию в задаче `wrapper` :

```
task wrapper(type: Wrapper) {
    gradleVersion = '2.0'
    distributionUrl = "http\://server/dadada/gradle-${gradleVersion}-bin.zip"
}
```

после выполнения `gradle wrapper` оболочки `gradlew` сценарий оболочки `gradlew` а параметр `gradle/wrapper/gradle-wrapper.properties` настроен на использование предоставленного URL для загрузки Gradle.

Использование оберточной машины за прокси-сервером

В первый раз, когда пользователь запускает `gradlew` проекта, следует понимать, что он будет делать две ключевые вещи:

1. Убедитесь, что версия градиента, используемая оберткой, уже находится в ~ / .gradle / wrapper / dists
2. Если нет, загрузите архив версии из Интернета

Если вы находитесь в среде, которая требует, чтобы весь внешний трафик проходил через прокси-сервер, второй шаг будет терпеть неудачу (если только это не прозрачная среда

прокси). В результате вы должны убедиться, что у вас установлены параметры прокси-сервера *JVM* .

Например, если у вас есть базовая настройка прокси без аутентификации, просто установите переменную среды `JAVA_OPTS` или `GRADLE_OPTS` с помощью:

```
-Dhttps.proxyPort=<proxy_port> -Dhttps.proxyHost=<hostname>
```

Таким образом, завершенный пример для окон:

```
set JAVA_OPTS=-Dhttps.proxyPort=8080 -Dhttps.proxyHost=myproxy.mycompany.com
```

Если, однако, ваша среда также требует аутентификации, вы также захотите просмотреть свои другие параметры на странице https://docs.oracle.com/javase/8/docs/api/java/net/doc-files/net-properties. html .

*ПРИМЕЧАНИЕ. Эта конфигурация прокси-сервера **дополняет** любую конфигурацию прокси-сервера для доступа к репозиториям зависимостей.*

Прочитайте Грейд-обертка онлайн: https://riptutorial.com/ru/gradle/topic/3006/грейд-обертка

глава 5: зависимости

Examples

Добавление локальной зависимости JAR-файла

Одноместный JAR

Иногда у вас есть локальный файл JAR, который нужно добавить в качестве зависимости от вашей сборки Gradle. Вот как вы можете это сделать:

```
dependencies {
    compile files('path/local_dependency.jar')
}
```

Где `path` - `path` к каталогу в вашей файловой системе, а `local_dependency.jar` - это имя вашего локального файла JAR. `path` может быть относительно файла сборки.

Каталог JAR

Также можно добавить каталог jars для компиляции. Это можно сделать так:

```
dependencies {
        compile fileTree(dir: 'libs', include: '*.jar')
}
```

Там, где `libs` будет каталогом, содержащим банки, и `*.jar` будет фильтром для включения файлов.

Каталог JAR как хранилища

Если вы хотите искать банки в хранилище вместо прямого добавления их в качестве зависимости с их путем, вы можете использовать репозиторий flatDir.

```
repositories {
    flatDir {
        dirs 'libs'
    }
}
```

Ищет банки в каталоге *`libs`* и его дочерних каталогах.

Добавить зависимость

Зависимости в Gradle соответствуют тому же формату, что и Maven . Зависимости структурированы следующим образом:

```
group:name:version
```

Вот пример:

```
'org.springframework:spring-core:4.3.1.RELEASE'
```

Чтобы добавить как зависимость времени компиляции, просто добавьте эту строку в блок `dependency` в файле сборки Gradle:

```
compile 'org.springframework:spring-core:4.3.1.RELEASE'
```

Альтернативный синтаксис для этого имени определяет каждый компонент зависимости явно:

```
compile group: 'org.springframework', name: 'spring-core', version: '4.3.1.RELEASE'
```

Это добавляет зависимость во время компиляции.

Вы также можете добавлять зависимости только для тестов. Вот пример:

```
testCompile group: 'junit', name: 'junit', version: '4.+'
```

Зависит от другого проекта «Грейдл»

В случае многопроектной градиловой сборки иногда вам может понадобиться зависеть от другого проекта в вашей сборке. Чтобы выполнить это, вы должны ввести следующие значения в зависимости от вашего проекта:

```
dependencies {
    compile project(':OtherProject')
}
```

Where `':OtherProject'` - это путь градиента для проекта, на который ссылается корень структуры каталогов.

Чтобы сделать `':OtherProject'` доступным в контексте файла `build.gradle` добавьте его в соответствующие `settings.gradle`

```
include ':Dependency'
project(':Dependency').projectDir = new File('/path/to/dependency')
```

Для более подробного объяснения, вы можете ссылаться на официальную документацию Gradle в здесь .

Зависимости списков

Вызов задачи `dependencies` позволяет увидеть зависимости корневого проекта:

```
gradle dependencies
```

Результатом являются графики зависимостей (с учетом транзитивных зависимостей), с разбивкой по конфигурации. Чтобы ограничить отображаемые конфигурации, вы можете передать параметр `--configuration` а затем одну выбранную конфигурацию для анализа:

```
gradle dependencies --configuration compile
```

Чтобы отобразить зависимости подпроекта, используйте задачу `<subproject>:dependencies` . Например, для перечисления зависимостей подпроекта с именем `api` :

```
gradle api:dependencies
```

Добавление репозиториев

Вы должны указать Gradle на расположение ваших плагинов, чтобы Gradle мог их найти. Сделайте это, добавив `repositories { ... }` в свой `build.gradle` .

Ниже приведен пример добавления трех репозиториев: JCenter , Maven Repository и специализированного репозитория, который предлагает зависимости в стиле Maven.

```
repositories {
  // Adding these two repositories via method calls is made possible by Gradle's Java plugin
  jcenter()
  mavenCentral()

  maven { url "http://repository.of/dependency" }
}
```

Добавить .aar-файл в Android-проект, используя gradle

1. Перейдите в модуль `app` проекта и создайте каталог `libs` .
2. Поместите там свой файл `.aar` . Например, `myLib.aar` .
3. Добавьте код ниже в блок `android app` файла `build.gradle` уровне `app` .

```
  repositories {
        flatDir {
            dirs 'libs'
        }
    }
```

Таким образом, вы определили новый дополнительный репозиторий, который указывает на папку `libs` модуля `app` .

4. Добавьте код ниже в блок `dependencies` или файл `build.gradle` :

```
compile(name:'myLib', ext:'aar')
```

Прочитайте зависимости онлайн: https://riptutorial.com/ru/gradle/topic/2524/зависимости

глава 6: Зависимости задач

замечания

doLast

Обратите внимание, что в градиенте 3.x больше идиоматического определения задачи: использование **явного doLast {закрытия}** вместо оператора «leftShift» (<<) предпочтительнее. (**LeftShift** устарел в градиенте 3.2, планируется удалить в градиенте 5.0 .)

```
task oldStyle << {
    println 'Deprecated style task'
 }
```

эквивалентно:

```
task newStyle {
    doLast {
    println 'Deprecated style task'
    }
 }
```

Examples

Добавление зависимостей с использованием имен задач

Мы можем изменить порядок выполнения задач с `dependsOn` метода `dependsOn` .

```
task A << {
    println 'Hello from A'
}
task B(dependsOn: A) << {
    println "Hello from B"
}
```

Добавление `dependsOn: вызывает:

- задача B зависит от задачи A
- Gradle для выполнения `A` задачи каждый раз **перед** `B` выполнения задачи.

И выход:

```
> gradle -q B
Hello from A
Hello from B
```

Добавление зависимостей из другого проекта

```
project('projectA') {
    task A(dependsOn: ':projectB:B') << {
        println 'Hello from A'
    }
}

project('projectB') {
    task B << {
        println 'Hello from B'
    }
}
```

Чтобы ссылаться на задачу в другом проекте, вы **префикс имени задачи** с указанием пути к проекту, к которому он принадлежит `:projectB:B`

И выход:

```
> gradle -q B
Hello from A
Hello from B
```

Добавление зависимости с использованием объекта задачи

```
task A << {
    println 'Hello from A'
}

task B << {
    println 'Hello from B'
}

B.dependsOn A
```

Это альтернативный способ определения зависимости вместо использования имени задачи .

И результат тот же:

```
> gradle -q B
Hello from A
Hello from B
```

Добавление нескольких зависимостей

Вы можете добавить несколько зависимостей.

```
task A << {
    println 'Hello from A'
}
```

```
task B << {
    println 'Hello from B'
}

task C << {
    println 'Hello from C'
}

task D << {
    println 'Hello from D'
}
```

Теперь вы можете определить набор зависимостей:

```
B.dependsOn A
C.dependsOn B
D.dependsOn C
```

Выход:

```
> gradle -q D
Hello from A
Hello from B
Hello from C
Hello from D
```

Другой пример:

```
B.dependsOn A
D.dependsOn B
D.dependsOn C
```

Выход:

```
> gradle -q D
Hello from A
Hello from B
Hello from C
Hello from D
```

Множественные зависимости с методом dependOn

Вы можете добавить несколько зависимостей.

```
task A << {
    println 'Hello from A'
}

task B(dependsOn: A) << {
    println 'Hello from B'
}
```

```
task C << {
    println 'Hello from C'
}

task D(dependsOn: ['B', 'C'] << {
    println 'Hello from D'
}
```

Выход:

```
> gradle -q D
Hello from A
Hello from B
Hello from C
Hello from D
```

Прочитайте Зависимости задач онлайн: https://riptutorial.com/ru/gradle/topic/5545/зависимости-задач

глава 7: Заказные задания

замечания

> Обратите внимание, что `mustRunAfter` и `shouldRunAfter` обозначаются как «инкубация» (`shouldRunAfter` с Gradle 3.0), что означает, что это экспериментальные функции, и их поведение может быть изменено в будущих выпусках.

Доступны два правила заказа:

- `mustRunAfter`
- `shouldRunAfter`

Когда вы используете `mustRunAfter` упорядочения `mustRunAfter` вы указываете, что taskB должен всегда запускаться после taskA, когда запускаются оба taskA и taskB.

`shouldRunAfter` упорядочения `shouldRunAfter` аналогично, но менее строго, поскольку оно игнорируется в двух ситуациях:

- если использование этого правила вводит цикл упорядочения.
- при использовании параллельного выполнения и всех зависимостей задачи были выполнены отдельно от задачи `shouldRunAfter` , тогда эта задача будет выполняться независимо от того, были ли `shouldRunAfter` ее зависимости `shouldRunAfter` .

Examples

Заказ с помощью метода mustRunAfter

```
task A << {
    println 'Hello from A'
}
task B << {
    println 'Hello from B'
}

B.mustRunAfter A
```

Линия `B.mustRunAfter A` сообщает Gradle запускать задачу после задания задачи в качестве аргумента.

И выход:

```
> gradle -q B A
Hello from A
Hello from B
```

Правило упорядочения не вводит зависимость между задачами A и B, но имеет эффект только тогда, когда **обе задачи запланированы** для выполнения.

Это означает, что мы можем выполнять задачи A и B независимо.

Выход:

```
> gradle -q B
Hello from B
```

Прочитайте Заказные задания онлайн: https://riptutorial.com/ru/gradle/topic/5550/заказные-задания

глава 8: Инициализация Gradle

замечания

терминология

- Задача - атомная часть работы, которую выполняет сборка. Задачи имеют `inputs` , `outputs` и зависимости задач.
- `dependencies {}` - объявляет `File` или двоичные зависимости, необходимые для выполнения задач. Например, `org.slf4j:slf4j-api:1.7.21` - сокращенные координаты зависимости от Maven.
- `repositories {}` - Как Gradle находит файлы для внешних зависимостей. Действительно, всего лишь набор файлов, организованных группой, именем и версией. Например: `jcenter()` - метод удобства для `maven { url 'http://jcenter.bintray.com/' } }` , репозитория Bintray Maven .

Examples

Инициализация новой библиотеки Java

Предварительное условие: установка Gradle

После установки Gradle вы можете настроить новый или существующий проект, выполнив

```
cd $PROJECT_DIR
gradle init --type=java-library
```

Обратите внимание, что есть другие типы проектов, такие как Scala, с которыми вы можете начать, но мы будем использовать Java для этого примера.

Вы получите:

```
.
├── build.gradle
├── gradle
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradlew
├── gradlew.bat
├── settings.gradle
└── src
    ├── main
    │   └── java
    │       └── Library.java
    └── test
```

```
        └── java
            └── LibraryTest.java
```

Теперь вы можете запускать `gradle tasks` и видеть, что вы можете создать `jar` , запустить `test` s, создать `javadoc` и многое другое, даже если ваш файл `build.gradle` :

```
apply plugin: 'java'

repositories {
    jcenter()
}

dependencies {
    compile 'org.slf4j:slf4j-api:1.7.21'
    testCompile 'junit:junit:4.12'
}
```

Прочитайте Инициализация Gradle онлайн: https://riptutorial.com/ru/gradle/topic/2247/инициализация-gradle

глава 9: Использование сторонних плагинов

Examples

Добавление стороннего плагина для build.gradle

Gradle (все версии) *Этот метод работает для всех версий gradle*

Добавьте код buildscript в начале файла build.gradle.

```
buildscript {
  repositories {
    maven {
      url "https://plugins.gradle.org/m2/"
    }
  }
  dependencies {
    classpath "org.example.plugin:plugin:1.1.0"
  }
}

apply plugin: "org.example.plugin"
```

Gradle (Версии 2.1+) *Этот метод работает только для проектов с использованием Gradle 2.1 или более поздней* **версии** .

```
plugins {
  id "org.example.plugin" version "1.1.0"
}
```

build.gradle с несколькими сторонними плагинами

Gradle (все версии)

При добавлении нескольких сторонних плагинов вам не нужно разделить их на разные экземпляры кода buildscript (All) или плагина (2.1+), новые плагины могут быть добавлены вместе с уже существующими плагинами.

```
buildscript {
  repositories {
    maven {
      url "https://plugins.gradle.org/m2/"
    }
  }
  dependencies {
    classpath "org.example.plugin:plugin:1.1.0"
    Classpath "com.example.plugin2:plugin2:1.5.2"
```

```
  }
}

apply plugin: "org.example.plugin"
apply plugin: "com.example.plugin2"
```

Gradle (Версии 2.1+)

```
plugins {
  id "org.example.plugin" version "1.1.0"
  id "com.example.plugin2" version "1.5.2"
}
```

Прочитайте Использование сторонних плагинов онлайн:
https://riptutorial.com/ru/gradle/topic/9183/использование-сторонних-плагинов

глава 10: Настройка IntelliJ IDEA

Синтаксис

- groovy.util.Node = node.find {childNode -> return true || ложный }
- node.append (nodeYouWantAsAChild)
- groovy.util.Node parsedNode = (новый XmlParser ()). parseText (someRawXMLString)
- " 'mutli-строка (не интерполированная)' "

замечания

Три основных файла проекта IntelliJ - файлы ipr, iws и iml - могут быть доступны, как в градиенте в идее, через

```
project.ipr
module.iml
workspace.iws
```

использование .withXml позволяет вам получить доступ к xml. Использование .asNode (), которое превращает его в массивный узел xml.

Пример:

```
project.ipr.withXml { provider ->
    def node = provider.asNode()
```

Оттуда это довольно просто - изменить град, чтобы настроить проекты IntelliJ для вас, взять файл по мере его запуска, выполнить действия, которые вы хотите сделать градиентом (внутри IntelliJ), а затем разбить новый файл на старый файл. Вы должны увидеть, какой XML вам понадобится, чтобы настроить идею. Вам также необходимо принять во внимание, где находится xml, в котором он находится.

Еще одна вещь, которую следует учитывать, заключается в том, что вы не хотите дублировать узлы в файлах IntelliJ, если несколько раз запускаете идею gradle. Таким образом, вы захотите найти узел, который хотите сделать, и если его там нет, вы можете его создать и вставить.

Ловушки:

Иногда при использовании == для сравнения строк в методе find он терпит неудачу. При тестировании, и я считаю, что это так, я использую .contains.

При поиске узлов не все узлы имеют атрибут, который вы используете в качестве критерия, поэтому обязательно проверяйте значение null.

Examples

Добавить базовую конфигурацию запуска

Предположения для этого примера:

- У вас есть класс, `foo.bar.Baz` .
- Вы хотите создать конфигурацию запуска, которая запускает основной метод.
- Он находится в модуле `fooBar` .

В вашем файле gradle:

```
idea {
    workspace.iws.withXml { provider ->
        // I'm not actually sure why this is necessary
        def node = provider.asNode()

        def runManager = node.find { it.@name.contains('RunManager')}

        // find a run configuration if it' there already
        def runner = runManager.find { it.find ({ mainClass ->
            return mainClass.@name != null && mainClass.@name == "MAIN_CLASS_NAME" &&
mainClass.@value != null && mainClass.@value.contains('Baz');
        }) != null }

        // create and append the run configuration if it doesn't already exists
        if (runManager != null && runner == null){
            def runnerText = '''
                <configuration default="false" name="Baz" type="Application"
factoryName="Application" nameIsGenerated="true">
                  <extension name="coverage" enabled="false" merge="false" runner="idea">
                    <pattern>
                      <option name="PATTERN" value="foo.bar.Baz" />
                      <option name="ENABLED" value="true" />
                    </pattern>
                  </extension>
                  <option name="MAIN_CLASS_NAME" value="foo.bar.Baz" />
                  <option name="VM_PARAMETERS" value="" />
                  <option name="PROGRAM_PARAMETERS" value="" />
                  <option name="WORKING_DIRECTORY" value="file://$PROJECT_DIR$" />
                  <option name="ALTERNATIVE_JRE_PATH_ENABLED" value="false" />
                  <option name="ALTERNATIVE_JRE_PATH" />
                  <option name="ENABLE_SWING_INSPECTOR" value="false" />
                  <option name="ENV_VARIABLES" />
                  <option name="PASS_PARENT_ENVS" value="true" />
                  <module name="foobar" />
                  <envs />
                  <method />
                </configuration>'''
            runner = (new XmlParser()).parseText(runnerText)
            runManager.append(config);
        }

        // If there is no active run configuration, set the newly made one to be it
        if (runManager != null && runManager.@selected == null) {
            runManager.@selected="${runner.@factoryName}.${runner.@name}"
```

```
            }
        }
}
```

Прочитайте Настройка IntelliJ IDEA онлайн: https://riptutorial.com/ru/gradle/topic/2297/настройка-intellij-idea

глава 11: Плагины Gradle

Examples

Простой плагин gradle от `buildSrc`

Простой пример создания настраиваемого плагина и DSL для вашего проекта gradle.
В этом примере используется один из трех возможных способов создания плагинов.
Три способа:

- в соответствии
- buildSrc
- автономные плагины

В этом примере показано создание плагина из папки **buildSrc** .

Этот образец создаст пять файлов

```
// project's build.gradle
build.gradle
// build.gradle to build the `buildSrc` module
buildSrc/build.gradle
// file name will be the plugin name used in the `apply plugin: $name`
// where name would be `sample` in this example
buildSrc/src/main/resources/META-INF/gradle-plugins/sample.properties
// our DSL (Domain Specific Language) model
buildSrc/src/main/groovy/so/docs/gradle/plugin/SampleModel.groovy
// our actual plugin that will read the values from the DSL
buildSrc/src/main/groovy/so/docs/gradle/plugin/SamplePlugin.groovy
```

build.gradle:

```
group 'so.docs.gradle'
version '1.0-SNAPSHOT'

apply plugin: 'groovy'
// apply our plugin... calls SamplePlugin#apply(Project)
apply plugin: 'sample'

repositories {
    mavenCentral()
}

dependencies {
    compile localGroovy()
}

// caller populates the extension model applied above
sample {
    product = 'abc'
    customer = 'zyx'
```

```
}

// dummy task to limit console output for example
task doNothing <<{}
```

buildSrc / build.gradle

```
apply plugin: 'groovy'

repositories {
    mavenCentral()
}

dependencies {
    compile localGroovy()
}
```

buildSrc / SRC / Основной / заводной / так / документы / Gradle / плагин / SamplePlugin.groovy:

```
package so.docs.gradle.plugin

import org.gradle.api.Plugin
import org.gradle.api.Project

class SamplePlugin implements Plugin<Project> {
    @Override
    void apply(Project target) {
        // create our extension on the project for our model
        target.extensions.create('sample', SampleModel)
        // once the script has been evaluated the values are available
        target.afterEvaluate {
            // here we can do whatever we need to with our values
            println "populated model: $target.extensions.sample"
        }
    }
}
```

buildSrc / SRC / Основной / заводной / так / документы / Gradle / плагин / SampleModel.groovy:

```
package so.docs.gradle.plugin

// define our DSL model
class SampleModel {
    public String product;
    public String customer;

    @Override
    public String toString() {
        final StringBuilder sb = new StringBuilder("SampleModel{");
        sb.append("product='").append(product).append('\'');
        sb.append(", customer='").append(customer).append('\'');
        sb.append('}');
        return sb.toString();
    }
```

```
}
```

buildSrc / SRC / основные / ресурсы / META-INF / Gradle-плагин / sample.properties

```
implementation-class=so.docs.gradle.plugin.SamplePlugin
```

Используя эту настройку, мы можем видеть значения, предоставленные вызывающим абонентом в вашем блоке DSL

```
 $ ./gradlew -q doNothing
SampleModel{product='abc', customer='zyx'}
```

Как написать автономный плагин

Чтобы создать собственный автономный подключаемый модуль Gradle с использованием java (вы также можете использовать Groovy), вам необходимо создать такую структуру:

```
plugin
|-- build.gradle
|-- settings.gradle
|-- src
    |-- main
    |   |-- java
    |   |-- resources
    |       |-- META-INF
    |           |-- gradle-plugins
    |-- test
```

Настройка конфигурации градиента

В файле `build.gradle` вы определяете свой проект.

```
apply plugin: 'java'
apply plugin: 'maven'

dependencies {
    compile gradleApi()
}
```

Плагин `java` будет использоваться для написания Java-кода.
`gradleApi()` даст нам весь метод и правильность, необходимые для создания плагина Gradle.

В файле `settings.gradle` :

```
rootProject.name = 'myplugin'
```

Он определит **идентификатор артефакта** в Maven.

Если файл `settings.gradle` отсутствует в каталоге плагина, значением по умолчанию будет имя каталога.

Создание плагина

Определите класс в `src/main/java/org/sample/MyPlugin.java` реализующем интерфейс `Plugin` .

```
import org.gradle.api.Plugin;
import org.gradle.api.Project;

public class MyPlugin implements Plugin<Project> {

    @Override
    public void apply(Project project) {
         project.getTasks().create("myTask", MyTask.class);
    }

}
```

Определите задачу, расширяющую класс `DefaultTask` :

```
import org.gradle.api.DefaultTask;
import org.gradle.api.tasks.TaskAction;

public class MyTask extends DefaultTask {

    @TaskAction
    public void myTask() {
        System.out.println("Hello World");
    }
}
```

Объявление класса плагина

В папке `META-INF/gradle-plugins` вам необходимо создать файл свойств, определяющий свойство `implementation-class` которое идентифицирует класс реализации Plugin.

В `META-INF/gradle-plugins/testplugin.properties`

```
implementation-class=org.sample.MyPlugin.java
```

Обратите внимание, что **имя файла свойства соответствует идентификатору плагина** .

Как создать и опубликовать его

Измените файл `build.gradle` добавив некоторую информацию для загрузки плагина в

репозиторий maven:

```
apply plugin: 'java'
apply plugin: 'maven'

dependencies {
    compile gradleApi()
}

repositories {
    jcenter()
}


group = 'org.sample'
version = '1.0'

uploadArchives {
    repositories {
        mavenDeployer {
        repository(url: mavenLocal().url)
        }
    }
}
```

Вы можете создать и опубликовать подключаемый модуль Gradle к `plugin/build.gradle` Maven, определенному в файле `plugin/build.gradle` используя следующую команду.

```
$ ./gradlew clean uploadArchives
```

Как это использовать

Чтобы использовать плагин, добавьте в `build.gradle` вашего проекта:

```
buildscript {
     repositories {
         mavenLocal()
     }
 dependencies {
    classpath group: 'org.sample',    // Defined in the build.gradle of the plugin
              name: 'myplugin',       // Defined by the rootProject.name
              version: '1.0'
    }
 }

apply plugin: 'testplugin'            // Defined by the properties filename
```

Затем вы можете вызвать задачу, используя:

```
 $ ./gradlew myTask
```

Прочитайте Плагины Gradle онлайн: https://riptutorial.com/ru/gradle/topic/1900/плагины-gradle

глава 12: Скрипты инициалов Gradle

Examples

Добавить репозиторий по умолчанию для всех проектов

Добавьте init.gradle в папку пользователя. Init.gradle распознается в каждом проекте.

```
Unix: ~/.gradle/init.gradle
```

> Это также альтернативные места, где сценарий инициализации можно разместить и загрузить автоматически:
>
> - Любой файл * **.gradle** в **USER_HOME / .gradle / init.d**
> - Любой * **.gradle** файл в каталоге **init.d** установочного Gradle в

init.gradle с mavenLocal как хранилище во всех проектах.

```
allprojects {
    repositories {
        mavenLocal()
    }
}
```

При этом у вас есть ваш локальный кеш-кеш, доступный во всех репозиториях. Вариант использования может состоять в том, чтобы использовать банку, которую вы положили в нее с помощью «gradle install» в другом проекте, без добавления репозитория mavenLocal в build.gradle или добавления сервера nexus / artifactory.

Прочитайте Скрипты инициалов Gradle онлайн: https://riptutorial.com/ru/gradle/topic/4234/скрипты-инициалов-gradle

глава 13: Эффективность колышков

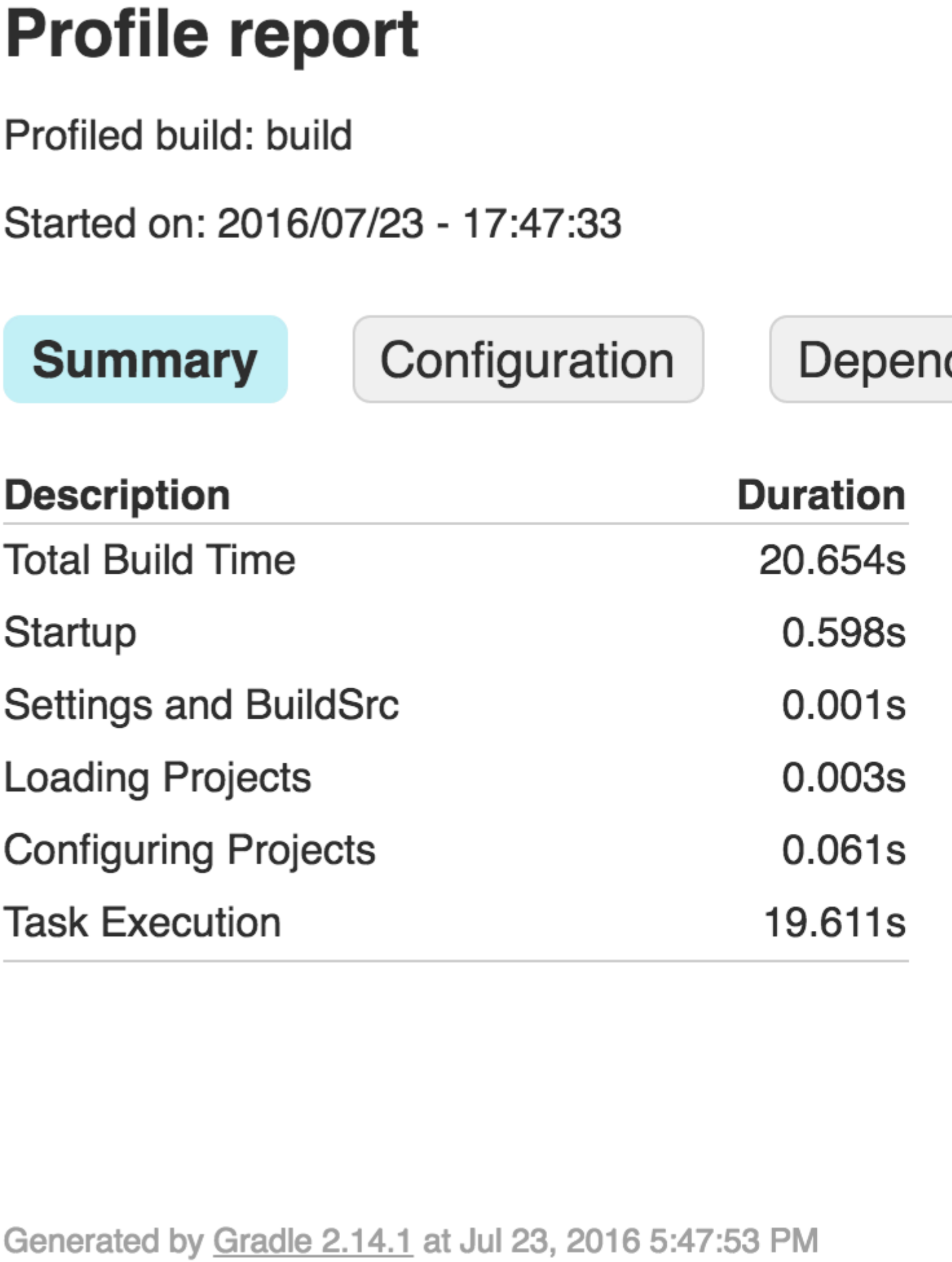
Examples

Профилирование сборки

Прежде чем приступить к настройке сборки Gradle для производительности, вы должны установить базовый уровень и выяснить, какие части сборки занимают больше всего времени. Для этого вы можете профилировать свою сборку , добавив аргумент `--profile` в команду Gradle:

```
gradle --profile
./gradlew --profile
```

После завершения сборки вы увидите отчет профиля HTML для сборки в `./build/reports/profile/` , выглядя примерно так:



Нажав на вкладки рядом с « **Сводкой»** ,

вы увидите более подробную разбивку времени, в которое было потрачено время.

Настройка по требованию

Если профилирование вашей сборки показывает значительные затраты времени на **настройку проектов** , опция «Настройка по требованию» может повысить вашу производительность.

Вы можете включить режим Configure on Demand, `$GRADLE_USER_HOME/.gradle/gradle.properties` (`~/.gradle/gradle.properties` по умолчанию) и установив `org.gradle.configureondemand` .

```
org.gradle.configureondemand=true
```

Чтобы включить его только для конкретного проекта, отредактируйте файл `gradle.properties` этого проекта.

Если параметр «Настроить по требованию» включен, вместо того, чтобы настраивать все проекты спереди, Gradle будет настраивать только проекты, которые необходимы для выполняемой задачи.

Из руководства Gralde :

> Режим конфигурации по требованию пытается сконфигурировать только проекты, которые имеют отношение к запрошенным задачам, т. `build.gradle` Он выполняет `build.gradle` файл `build.gradle` проектов, участвующих в сборке. Таким образом, время конфигурации большой многопроектной сборки может быть уменьшено. В долгосрочной перспективе этот режим станет режимом по умолчанию, возможно единственным режимом для выполнения сборки Gradle.

Настройка параметров использования памяти JVM для Gradle

Вы можете установить или увеличить пределы использования памяти (или другие аргументы JVM), используемые для построения Gradle и Gradle Daemon, путем редактирования `$GRADLE_USER_HOME/.gradle/gradle.properties` (`~/.gradle/gradle.properties` по умолчанию) и установки `org.gradle.jvmargs` .

Чтобы настроить эти ограничения только для конкретного проекта, отредактируйте файл `gradle.properties` этого проекта.

Параметры использования памяти по умолчанию для Gradle builds и Gradle Daemon:

```
org.gradle.jvmargs=-Xmx1024m -XX:MaxPermSize=256m
```

Это позволяет общее максимальное распределение памяти (размер кучи) 1 ГБ и максимальное распределение памяти для постоянных «внутренних» объектов размером

256 МБ. Когда эти размеры достигнуты, происходит сборка мусора, которая может значительно снизить производительность.

Предполагая, что у вас есть запасная память, вы можете легко удвоить их так:

```
org.gradle.jvmargs=-Xmx2024m -XX:MaxPermSize=512m
```

Обратите внимание, что вы перестанете видеть выгоду от увеличения `XX:MaxPermSize` раньше, чем при увеличении `Xmx` .

Использование демона Gradle

Вы можете включить Gradle Daemon для повышения производительности ваших сборников.

Gradle Daemon поддерживает инициализацию и запуск Gradle Framework и кэширует данные проекта в памяти для повышения производительности.

Для единой сборки

Чтобы включить Daemon для одной сборки, вы можете просто передать `--daemon` аргумент вашей `gradle` команды или Gradle Упаковочный сценария.

```
gradle --daemon
./gradlew --daemon
```

Для всех проектов проекта

Чтобы включить Daemon для всех построек проекта, вы можете добавить:

```
org.gradle.daemon=true
```

В файл `gradle.properties` вашего проекта.

Для всех сборщиков

Чтобы включить Gradle Daemon по умолчанию, для каждой сборки, сделанной вашей учетной записью пользователя в вашей системе, по `$GRADLE_USER_HOME/.gradle/gradle.properties` (`~/.gradle/gradle.properties`) и добавьте эту строку:

```
org.gradle.daemon=true
```

Вы также можете сделать это в одной команде в системах Mac / Linux / * nix:

```
touch ~/.gradle/gradle.properties && echo "org.gradle.daemon=true" >>
~/.gradle/gradle.properties
```

Или в Windows:

```
(if not exist "%USERPROFILE%/.gradle" mkdir "%USERPROFILE%/.gradle") && (echo
org.gradle.daemon=true >> "%USERPROFILE%/.gradle/gradle.properties")
```

Отключение демона

Вы можете отключить Daemon для конкретной сборки с использованием аргумента `--no-daemon` или отключить его для конкретного проекта, явно установив `org.gradle.daemon=false` в файле `gradle.properties` проекта.

Остановка демона

Если вы хотите остановить процесс Daemon вручную, вы можете либо убить процесс через диспетчер задач операционной системы, либо выполнить команду `gradle --stop` . Переключатель `--stop` заставляет Gradle запрашивать, чтобы все запущенные процессы Daemon той же версии Gradle, используемые для запуска команды, завершались. Обычно процессы Daemon автоматически прекращают себя * через * *3 часа бездействия или меньше* .

Параллельные сборки Gradle

По умолчанию Gradle запускает только одну задачу, независимо от структуры проекта. С помощью `--parallel` switch вы можете заставить Gradle выполнять независимые подпроекты - те, у которых нет явных или явных зависимостей проекта между собой, - параллельно, позволяя ему запускать несколько задач одновременно, пока эти задачи находятся в разные проекты.

Для создания проектов в параллельном режиме:

```
gradle build --parallel
```

Вы также можете сделать построение параллельно по умолчанию для проекта, добавив следующий параметр в файл gradle.properties проекта:

```
org.gradle.parallel=true
```

Используйте последнюю версию Gradle

Команда Gradle регулярно работает над улучшением производительности различных аспектов сборки Gradle. Если вы используете старую версию Gradle, вам не хватает преимуществ этой работы. Попробуйте обновиться до последней версии Gradle, чтобы узнать, какое влияние это имеет. Это низкий риск, поскольку между младшими версиями Gradle нарушается очень мало вещей.

Файл свойств для оболочки Gradle можно найти в папке проекта в разделе `gradle/wrapper/` и называется `gradle-wrapper.properties` . Содержимое этого файла может выглядеть так:

```
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
distributionUrl=https\://services.gradle.org/distributions/gradle-X.X.X.zip
```

Вы можете вручную изменить номер версии `XXX` (текущая версия) на `YYY` (более новая версия), а при следующем запуске оболочки новая версия будет загружена автоматически.

Прочитайте Эффективность колышков онлайн: https://riptutorial.com/ru/gradle/topic/3443/эффективность-колышков

кредиты

S. No	Главы	Contributors
1	Начало работы с градиентом	Afterfield, bassim, Community, Emil Burzo, Eric Wendelin, Hamzawey, Hillkorn, Matthias Braun, Nikem, Pepper Lebeck-Jobe, Sergey Yakovlev, Stanislav, user2555595, vanogrid, Will
2	Auto Increment Version Number Использование скрипта Gradle для приложений для Android	Jayakrishnan PM
3	В том числе родной источник - экспериментальный	iHowell
4	Грейд-обертка	ajoberstar, Fanick, HankCa, I Stevenson
5	зависимости	Afshin, Andrii Abramov, GameScripting, Hillkorn, leeor, Matthias Braun, mcarlin, mszymborski, Will
6	Зависимости задач	Gabriele Mariotti, Sergey Yakovlev, Stanislav
7	Заказные задания	Gabriele Mariotti
8	Инициализация Gradle	Eric Wendelin, Will
9	Использование сторонних плагинов	Afterfield
10	Настройка IntelliJ IDEA	IronHorse, Sam Sieber, Will
11	Плагины Gradle	Gabriele Mariotti, JBirdVegas
12	Скрипты инициалов Gradle	ambes, Hillkorn
13	Эффективность колышков	ambes, Sergey Yakovlev, Will