参考：<http://blog.csdn.net/baidu_26352053/article/details/52852363>

[Gradle基本知识点与常用配置](http://blog.csdn.net/baidu_26352053/article/details/52852363)

2016-10-18 20:37 963人阅读 [评论](http://blog.csdn.net/baidu_26352053/article/details/52852363#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/baidu_26352053/article/details/52852363#report)

 分类：

Android studio（58）

目录[(?)[+]](http://blog.csdn.net/baidu_26352053/article/details/52852363)

http://blog.csdn[**.NET**](http://lib.csdn.net/base/dotnet)/u010818425/article/details/52268126

查看原文：<http://blog.csdn.net/u010818425/article/details/52268126>

本文篇幅较长，文中系统地讲解了Gradle的基本知识点以及一些常用的命令和配置，适合于刚接触Gradle的同学；

在此基础上，可以阅读以下的实战经验文章：   
[《Gradle实战：Android多渠道打包方案汇总》](http://blog.csdn.net/u010818425/article/details/52319382)   
[《Gradle实战：不同编译类型的包同设备共存》](http://blog.csdn.net/u010818425/article/details/52335844)   
[《Gradle实战：发布aar包到maven仓库》](http://blog.csdn.net/u010818425/article/details/52441711)   
[《Gradle实战：执行sql操作hive数据库》](http://blog.csdn.net/u010818425/article/details/52490628)

（一）基本概念

1. Groovy

Domain Specific Language，领域相关语言

2. Gradle

Gradle是一个框架，它负责定义流程和规则；

每一个待编译的工程都叫一个Project；

每一个Project在构建的时候都包含一系列的Task。

3. Closure（闭包）

Closure是一段单独的代码块，它可以接收参数，返回值，也可以被赋值给变量：

//无参数

def Closure1 = {

println 'Hello world'

}

Closure1() // 执行闭包，输出Hello world

//接收一个参数

def Closure2 = {

String str -> println str //箭头前面是参数定义，后面是执行代码，str为外部传入的参数

} //如果只有一个参数可以用it代替，也可写作：

def Closure2 = {

println it

}

Closure2('Hello world')// 执行闭包，输出Hello world

//接收多个参数

def Closure3 = {

String str , int n -> println "$str : $n" //参数前加$

} //也可以写作：

def Closure3 = {

str , n -> println "$str : $n"

}

Closure3('Hello world’, 1) // 执行闭包，输出Hello world : 1

//使用变量

def var = "Hello world"

def Closure4 = {

println var

}

Closure4() // 执行闭包，输出Hello world

//改变上下文

def Closure5 = {

println Var //这时还不存在

}

MyClass m = new MyClass()

Closure5.setDelegate(m) // 改变上下文，这时Var已经有了，在执行之前改变了

Closure5() //执行闭包，输出Hello world

class MyClass {

def Var = 'Hello world'

}

4. Property

Gradle在默认情况下已经为Project定义了很多Property，如下：

project：Project本身

name：Project的名字

path：Project的绝对路径

description：Project的描述信息

buildDir：Project构建结果存放目录

version：Project的版本号

通过ext来自定义Property：

ext.property1 = "this is property1"

或

ext {

property2 = "this is property2"

}

task showProperties << {

println property1 //直接访问

println property2

}

任何实现了ExtensionAware接口的Gradle对象都可以通过这种方式来添加额外的Property，比如Task也实现了该接口。

通过“-p”命令行参数定义Property：

task showCommandLineProperties << {

println propertyTest

}

gradle -P propertyTest ="this is propertyTest" showCommandLineProperties

通过JVM系统参数定义Property（需要以“org.gradle.project”为前缀）：

gradle -D org.gradle.project.propertyTest="this is another propertyTest" showCommandLineProperties

另一种方式：

写入参数：gradle -DpropertyTest="this is another propertyTest"

读取参数：def propertyTest = System.properties['propertyTest']

通过环境变量设置Property（需要以“ORG\_ GRADLE\_ PROJECT\_ ”为前缀）：

export ORG\_GRADLE\_PROJECT\_propertyTest = "this is yet another propertyTest"

gradle showCommandLineProperties

5. Bean

> Groovy会为每一个字段自动生成getter和setter，我们可以通过像访问字段本身一样调用getter和setter，如：

class GroovyBeanExample {

private String name

}

def bean = new GroovyBeanExample()

bean.name = 'this is name' //Groovy动态地为name创建了getter和setter

println bean.name

6. delegate机制

delegate机制可以使我们将一个闭包中的执行代码的作用对象设置成任意其他对象（me:类似js中的callee和bind-对象冒充）

class Child {

private String name

}

class Parent {

Child child = new Child();

void configChild(Closure c) {

c.delegate = child

c.setResolveStrategy Closure.DELEGATE\_FIRST //默认情况下是OWNER\_FIRST，即它会先查找闭包的owner（这里即parent）

c()

}

}

def parent = new Parent()

parent.configChild {

name = "child name"

}

println parent.child.name

7. task

task有两个生命周期，配置阶段和执行阶段。   
gradle在执行task时，都会先对task进行配置，task中最顶层的代码就是配置代码，在配置阶段执行，其他代码是在执行阶段执行的；   
task关键字实际上是一个方法调用，我们不用将参数放在括号里面。

task Task1 {

println "hello” // 这段代码是在配置阶段执行的

}

task Task2 {

def name = "hello” // 这段代码是在配置阶段执行的

doLast {

println name

} // 这段代码是在执行阶段执行的，相当于：

// doLast({

// println 'Hello world!'

// })

}

task Task3 << {

println name

}

注意： “<<”语法糖，表示追加执行过程，相当于doLast，因此整个代码都是在执行阶段执行的；与之相反的是doFirst。

//如果代码没有加“<<”，则这个任务在脚本initialization的时候执行（也就是你无论执行什么任务，这个任务都会被执行，“hello”都会被输出）；

//如果加了“<<”，则在输入命令gradle Task3后才执行

通过TaskContainer的create()方法创建Task

tasks.create(name: 'hello') << {

println 'hello'

}

8. 自定义task

class HelloWorldTask extends DefaultTask {

@Optional

String message = 'I am davenkin'

@TaskAction

def hello(){

println "hello world $message"

}

}

task hello(type:HelloWorldTask)

task hello1(type:HelloWorldTask){

message ="I am a programmer"

}

@TaskAction表示该Task要执行的动作，@Optional表示在配置该Task时，message是可选的

9. 任务配置

方法一：在定义Task的时候对Property进行配置

task hello1 << {

description = "this is hello1"

println description

}

方法二：通过闭包的方式来配置一个已有的Task

task hello2 << {

println description

}

hello2 {

description = "this is hello2"

} //Gradle会为每一个task创建一个同名的方法，该方法接受一个闭包

或

hello2.description = "this is hello2"//Gradle会为每一个task创建一个同名的Property，所以可以将该Task当作Property来访问

注：对hello2的description的设置发生在定义该Task之后，在执行gradle hello2时，命令行依然可以打印出正确的“this is hello2”，这是因为Gradle在执行Task时分为两个阶段：配置阶段、执行阶段。   
所以在执行hello2之前，Gradle会扫描整个build.gradle文档，将hello2的description设置为“this is hello2”，再执行hello2。

方法三：通过Task的configure()方法完成Property的设置

task hello3 << {

println description

}

hello3.configure {

description = "this is hello3"

}

10. 任务依赖

task A << {

println 'Hello from A'

}

task B << {

println 'Hello from B'

}

B.dependsOn A

或

task A << {

println 'Hello from A'

}

task B {

dependsOn A

doLast {

println 'Hello from B'

}

}

或

task B(dependsOn: A) {

println 'Hello from B'

}

11. 顺序执行

指之间无依赖关系的任务

task unit << {

println 'Hello from unit tests'

}

task ui << {

println 'Hello from UI tests'

}

task tests << {

println 'Hello from all tests!'

}

task mergeReports << {

println 'Merging test reports'

}

tests.dependsOn unit //单元测试

tests.dependsOn ui //ui测试

ui.mustRunAfter unit //ui测试必须在单元测试之后执行

tests.finalizedBy mergeReports //表示tests执行完后，再执行mergeReports，等价于mergeReports.dependsOn tests

12. 增量式构建

为一个Task定义输入（inputs）和输出（outputs），在执行该Task时，如果它的输入和输出与前一次执行时没有变化，那么Gradle便会认为该Task是最新的（日志会输出“UP-TO-DATE“），因此不会重复执行

task combineFileContent {

def sources = fileTree('sourceDir')

def destination = file('destination.txt')

inputs.dir sources // 将sources声明为该Task的inputs

outputs.file destination // 将destination声明为outputs

doLast {

destination.withPrintWriter { writer ->

sources.each {source ->

writer.println source.text

}

}

}

}

当首次执行combineFileContent时，Gradle会完整地执行该Task，但是紧接着再执行一次，命令行显示：

:combineFileContent UP-TO-DATE //被标记为UP-TO-DATE，表示该Task是最新的，不执行

BUILD SUCCESSFUL

Total time: 2.104 secs

如果修改inputs（上述即sourceDir文件夹）中的任何一个文件或删除destination.txt，再次调用“gradle combineFileContent”时，该Task又会重新执行

13. 自定义Plugin

在当前工程中的buildSrc/src/main/groovy/davenkin目录下创建DateAndTimePlugin.groovy文件和DateAndTimePluginExtension.groovy文件

// DateAndTimePlugin.groovy

package com.gradle.test

import org.gradle.api.Plugin

import org.gradle.api.Project

class DateAndTimePlugin implements Plugin<Project> {

void apply(Project project) {

//每个Gradle的Project都维护了一个ExtenionContainer，我们可以通过project.extentions访问额外的Property和定义额外的Property

project.extensions.create("dateAndTime", DateAndTimePluginExtension)

project.task('showTime') << {

println "Current time is " + new Date().format(project.dateAndTime.timeFormat)

}

project.tasks.create('showDate') << {

println "Current date is " + new Date().format(project.dateAndTime.dateFormat)

}

}

}

// DateAndTimePlugin.groovy

package com.gradle.test

class DateAndTimePluginExtension {

String timeFormat = "MM/dd/yyyyHH:mm:ss.SSS"

String dateFormat = "yyyy-MM-dd"

}

build.gradle文件中，再apply该Plugin

apply plugin: com.gradle.test.DateAndTimePlugin

// 可以通过以下方式对这两个Property进行重新配置

dateAndTime {

timeFormat = 'HH:mm:ss.SSS'

dateFormat = 'MM/dd/yyyy'

}

（二）常用命令和配置

1. 编译、打包

注：执行“./gradlew xxx”等同于执行“gradle xxx”，但执行“gradle xxx”需配置环境变量

* 清除build文件夹
* ./gradlew clean
* 检查依赖并编译打包
* ./gradlew build
* 编译并打Debug包
* ./gradlew assembleDebug
* 编译并打Release包
* ./gradlew assembleRelease
* 获取gradle版本号
* ./gradlew -v
* 查看所有任务
* ./gradlew tasks 或 gradle tasks
* 查看所有工程
* gradle projects
* 查看所有属性
* gradle properties

2. 常用任务

* 执行任务
* task A << {
* println 'Hello from A'
* }
* 终端输入：gradle A
* 拷贝
* task copyFile(type: Copy) {
* from 'source'
* into 'destination'
* }
* 将source文件夹中的所有内容拷贝到destination文件夹中，
* 这两个文件夹都是相对于当前Project而言的，即build.gradle文件所在的目录
* 删除文件或文件夹
* task deleteTest(type: Delete) {
* delete 'file' ,'dir'
* }
* 文件和文件夹是相对于当前Project而言的，即build.gradle文件所在的目录
* 执行shell命令
* task runShell1(type: Exec) {
* executable "sh"
* args "-c", "rm ./app/libs/test.jar" //路径是相对于当前build.gradle文件
* }
* 或者
* def cmd = 'date +%Y-%m-%d'
* task shellTest << {
* String date = cmd.execute().text.trim().toString() //带返回值
* print date //打印系统日期
* }
* 执行[**Java**](http://lib.csdn.net/base/javaee)代码
* task runJava(type: JavaExec) {
* classpath = sourceSets.main.runtimeClasspath //执行文件所在路径
* main = 'com.example.MyClass' // 执行方法所在类
* // arguments to pass to the application
* args 'haha','xixi' //多个参数用逗号隔开
* }
* 生成jar包
* 方法一：
* task deleteDes(type: Delete) {
* delete 'build/intermediates/bundles/release/class.jar', 'libs/mylib.jar'
* }
* task makeJar(type: Copy) {
* from('build/intermediates/bundles/release/') {
* include '\*.jar' // 只拷贝jar格式的文件
* }
* into('libs')
* //include('classes.jar') // 只拷贝classes.jar这个文件
* rename('classes.jar', 'mylib.jar') // 重命名为mylib.jar
* }
* task releaseLib(type: Copy, dependsOn: [deleteDes, build, makeJar]) { // 依赖多个任务
* into "../app/libs"
* from 'libs'
* }
* // 手动打包，gradle releaseLib
* 方法二：
* // 以下是在编译流程中插入生成jar包的task
* android.libraryVariants.all { variant ->
* variant.outputs.each { output ->
* def file = output.outputFile
* def fileName = 'classes.jar'
* def name = variant.buildType.name
* task "makeJar${variant.name.capitalize()}" << {
* copy {
* from("${projectDir}/build/intermediates/bundles/"+"${name}") {
* include(fileName)
* }
* into(file.parent) // 可自定义存放路径
* rename (fileName, "${project.name}"+"-${name}.jar")
* }
* }
* }
* }
* project.tasks.whenTaskAdded { task ->
* android.libraryVariants.all { variant ->
* if (task.name == "bundle${variant.name.capitalize()}") {
* task.finalizedBy "makeJar${variant.name.capitalize()}"
* }
* }
* }
* 文件读取
* //以读取properties文件中保存的签名信息为例
* def File propFile = new File('signing.properties')
* if (propFile.canRead()) {
* def Properties props = new Properties()
* props.load(new FileInputStream(propFile))
* if (props != null && props.containsKey('RELEASE\_STORE\_FILE') && props.containsKey('RELEASE\_STORE\_PASSWORD') &&
* props.containsKey('RELEASE\_KEY\_ALIAS') && props.containsKey('RELEASE\_KEY\_PASSWORD')) {
* android.signingConfigs.release.storeFile = file(props['RELEASE\_STORE\_FILE'])
* android.signingConfigs.release.storePassword = props['RELEASE\_STORE\_PASSWORD']
* android.signingConfigs.release.keyAlias = props['RELEASE\_KEY\_ALIAS']
* android.signingConfigs.release.keyPassword = props['RELEASE\_KEY\_PASSWORD']
* println 'all good to go'
* } else {
* android.buildTypes.release.signingConfig = null
* println 'signing.properties found but some entries are missing'
* }
* } else {
* println 'signing.properties not found'
* android.buildTypes.release.signingConfig = null
* }

3. 常用配置

* 设置全局参数（同额外属性设置）
* ext {
* compileSdkVersion = 22
* buildToolsVersion = "22.0.1"
* }
* 在module中引用全局参数：
* android {
* compileSdkVersion rootProject.ext.compileSdkVersion
* buildToolsVersion rootProject.ext.buildToolsVersion
* }

将属性或方法放入ext{}就可以被全局引用

* 设置全局编码
* allprojects {
* tasks.withType(JavaCompile) {
* options.encoding = "UTF-8"
* }
* }
* 设置全局编译器的版本
* allprojects {
* tasks.withType(JavaCompile) {
* sourceCompatibility = JavaVersion.VERSION\_1\_7
* targetCompatibility = JavaVersion.VERSION\_1\_7
* }
* }
* 去掉重复依赖
* compile 'com.alibaba.fastjson.latest.integration' { //latest.integration 获取服务器上最新版本
* exclude module: 'annotations', group: 'com.google.android'
* }
* 本地aar包依赖
* allprojects {
* repositories {
* jcenter()
* flatDir {
* dirs 'libs'
* }
* }
* }
* dependencies {
* compile(name:'本地库aar的名字，不带后缀', ext:'aar')
* }

4. java Plugin

Gradle的每个source set都包含有一个名字，并且包含有一个名为[**Java**](http://lib.csdn.net/base/java)的Property和一个名为resources的Property，他们分别用于表示该source set所包含的Java源文件集合和资源文件集合。在实际应用时，我们可以将他们设置成任何目录值，以下将资源文件按功能分包，使得与java文件分包保持一致

sourceSets {

main {

manifest.srcFile 'src/main/AndroidManifest.xml'

java.srcDirs = ['src/main/java','.apt\_generated']

aidl.srcDirs = ['src/main/aidl','.apt\_generated']

assets.srcDirs = ['src/main/assets']

res.srcDirs =

[

'src/main/res/bindmobile', //绑定手机资源

'src/main/res/bycaptcha', //验证码登录资源

'src/main/res/bypwd', //密码登录资源

'src/main/res/pwdmodify', //密码修改资源

'src/main/res/pwdreset', //密码重置资源

'src/main/res/resource', //其他资源

'src/main/res/'

]

}

}

（三）配置gradle，加快编译速度

1. ~/.gradle/gradle.properties中添加如下配置（没有该文件则新建一个）：
2. org.gradle.daemon=true //独立进程，停止后台进程命令：gradle --stop
3. org.gradle.parallel=true //并行构建，需要将项目拆分成多个子项目，通过aar引用才能起效
4. org.gradle.configureondemand=true //按需配置，目前还在试验孵化阶段，默认是关闭的
5. 设置离线编译：
6. 打开settings->Build,Execution,Deployment->Build Tools->Gradle，
7. 选中Offlie Work //更新依赖包时要取消它
8. 命令行构建时在命令后面加上如下参数
9. --daemon
10. --parallel
11. --offline

--dry-run