**Gradle 学习笔记**

**Gradle 命令**

1. gradle wrapper 生成 Wrapper
2. gradle -q 打印日志，-s 关键性堆栈信息，-S 全部堆栈信息
3. gradle tasks 查看有哪些 Task 可以执行
4. gradle help --task tasks 查看指定 Task 的帮助
5. gradle --refresh-dependencies <task> 执行特定 Task 前强制刷新依赖
6. gradle clean jar 多任务调用，只需按顺序以空格分开即可，多个任务可以继续添加
7. gradle aR 可以缩写调用基于驼峰命名的 Task，aR 表示 assembleRelease 可以调用的前提是没有两个及以上缩写冲突的 Task

**Groovy 语言基础**

通过 def 关键字定义变量和方法

**1. 字符串的表示**

单引号和双引号都可以定义一个字符串，区别是单引号中的所有表达式都是常量字符串，而双引号可以对字符串里的表达式做运算。一个美元符号加一对花括号，花括号里放表达式，例如：{a+b}，只有一个变量时可以省略花括号，例如：name

**2. 集合**

// List 的定义：

def numList = [1,2,3,4,5,6]

// 访问：

numList[1] // 第二个元素

numList[-1] // 倒数第一个元素

numList[-2] // 倒数第二个元素

numList[1..3] // 访问第二个到第四个元素

// each 方法用来完成遍历操作

task pritList << {

def numList = [1,2,3,4,5,6]

numList.each {

// it 表示遍历到的元素

pritln it

}

}

// Map 的定义

def map = ['width':1024 , 'height':768]

// Map 的访问

map['width']

map.height

// Map 的遍历,遍历到的是一个 Map.Entry 元素

task printMap << {

def map = ['width':1024 , 'height':768]

map.each {

println "Key:${it.key},Value:${it.value}"

}

}

**3. 方法**

方法的调用可以省略(), method(1,2) 可以写成：method 1,2

有返回值的时候 return 语句不是必须的，没有 return 的时候，Groovy 会把最后一句代码作为其返回值

Groovy 支持闭包，如果最后一个参数时闭包，可以将闭包的花括号写到 () 外面，并且如果只有闭包一个参数，() 可以不写

list.each({println it})

// 简化

list.each {

println it

}

**4. JavaBean**

Groovy 中并不一定要定义 getter/setter 方法才能访问成员变量，并且定义了 getter/setter 方法后我们也可以访问没有声明的成员变量

**5. 向闭包传递参数**

如果只有一个参数，直接放入调用闭包时的括号里面，默认使用时就是 it，，如果是多个参数，闭包中就必须要显示声明出来

// 定义一个多个参数的闭包，并传入 eachMap 方法，闭包需要两个参数，通过 k 和 v 区分

eachMap{k,v -> println "$k is $v"}

**6. 闭包委托**

Groovy 闭包支持闭包方法的委托，Groovy 的闭包有 thisObject owner delegate 三个属性，当在闭包中调用方法时，由他们来确定哪个对象来处理，默认情况下 delegate 和 owner 是相等的，但是 delegate 是可以被修改的，Gradle 中的闭包的很多功能都是通过修改 delegate 实现的

thisObject 的优先级最高，默认情况下使用 thisObject 来处理闭包中调用的方法，如果有则执行。thisObject 其实就是这个构建脚本的上下文，和脚本中的 this 是相当的

owner 的优先级比 delegate 高，所以闭包内方法的调用顺序是：thisObject > owner > delegate

在 DSL 中，比如 Gradle，我们一般会指定 delegate 为当前的 it，这样就可以在闭包中对该 it 进行配置，或者调用其方法

task configClosure {

// 设置委托模式优先后，就可以在闭包中直接多该 Person 实例配置以及进行方法调用

person {

personName = "Jerry"

personAge = 20

dumpPerson()

}

}

class Person {

String personName

int personAge

def dumpPerson() {

println "name:$personName age:$personAge"

}

}

def person(Closure<Person> closure) {

Person p = new Person();

closure.delegate = p

// 委托模式优先

closure.setResolveStrategy(Closure.DELEGATE\_FIRST);

closure(p)

}

**Gradle 构建脚本基础**

**1. Setting 文件**

setting.gradle 用于初始化及工程树的配置，放在根目录下，该文件的大多数作用都是为了配置子工程，在 Gradle 中多工程是通过工程树表示的，类比 AndroidStudio 中的 Project 和 Module，根工程相当于 Project，子工程相当于 Module，一个根工程可以有多个子工程

一个子工程只有在 Setting 文件中配置了 Gradle 才会识别，才会在构建时被包含进去。配置一个子工程时可以指定相应的目录，如果不指定，默认目录是 Setting 文件其同级的目录

配置子工程时，子工程的名字可以不与目录相同，只要其他工程添加依赖时使用配置子工程时的名字即可

// 配置一个子工程

include ':example02'

// 指定子工程的目录

project(':example02').projectDir = new File(rootDir,'chapter01/example02')

**2. Project 和 Task**

Project 就是一个个独立的模块，多个 Project 组成了整个 Gradle 的构建，而 Project 是由一个个 Task 构成的，Task 就是一个操作，一个原子性操作，比如打一个 jar 包，复制一个文件等。

**定义一个 Task**

// 调用 Project 的 task 方法，接受一个 name(任务名称) 为参数，返回一个 Task 对象

def Task exTask1 = task(exTask1)

// 以一个任务名字 + 一个对该任务配置的 Map 对象来创建

def Task exTask2 = task(exTask2, group: BasePlugin,BUILD\_GROUP)

// 以一个任务名字加闭包的形式，闭包中也可以添加任务配置

task customTask1 {

doFirst {

println "first"

}

doLast {

println "last"

}

}

这里的 task 看着像一个 关键字，其实他是 Project 对象的一个函数，原型为 create(String name,Closure configureClosure) ,customeTask1 为任务的名字，第二个参数是一个闭包，将闭包提到括号外表再省略 () 就成了上面简洁的写法。其中 doFirst 和 doLast 是任务的两个方法，分别在任务执行前后会调用，此外，Task 还有其他方法和属性

还可以通过 TaskContainer 创建任务，其实以上提到的 Task 创建方式最终都是通过这种方式创建的，在 Gradle 里，Project 对象已经定义好了一个 TaskContainer，就是 tasks ，以上几种创建方式的作用是一样的

// TaskContainer 方式创建 Task

tasks.create("customTask2") {

doFirst{println "first"}

doLast{println "last"}

}

**3. 任务依赖**

创建任务的时候，通过 dependsOn 指定其依赖的任务,

task exHello {

println 'hello'

}

// 依赖一个任务

task exMain (dependsOn: exHello) {

doLast { println 'world'}

}

// 依赖多个任务，dependsOn 是 Task 类的一个方法，可以接收多个依赖的任务作为参数

task exMain2 {

dependsOn exHello exMain

doLast {println 'end'}

}

**4. 任务间通过 API 控制、交互**

创建一个任务类似定义一个变量，变量名就是任务名，类型是 Task，使用 Task 的 API 可以访问他的方法和属性或者对任务重新配置，和变量一样，要使用任务名操作任务，必须先定义声明，因为脚本是顺序执行的

Project 在创建任务的时候，会同时把该任务对应的任务名注册为 Project 的一个属性，类型是 Task

task exHello {

println 'hello'

}

exHello.doFirst {

println 'fist'

}

**5. 自定义属性**

Project 和 Task 都允许用户添加额外的自定义属性，通过应用所属对应的 ext 属性来添加额外的属性。添加之后可以通过 ext 属性对自定义属性读取和设置，如果要同时添加多个自定义属性，可以通过 ext 代码块。

相比局部变量，自定义的属性有更广泛的作用域，可以跨 Project，跨 Task 访问这些自定义属性，前提时能访问这些属性所属的对象。项目组一般使用它来自定义版本号和版本名称等

// 自定义一个 Project 的属性

ext.age = 18

// 通过代码库同时自定义多个属性

ext {

phone = '18000000000'

name = 'Hen'

}

**Gradle 任务 - Task**

**1. 多种任务访问方式**

首先，我们创建的任务都会作为 Project 的一个属性，属性名就是任务名，我们可以通过该任务名称访问和操作该任务

其次任务都是通过 TaskContainer 创建的，其实 TaskContainer 就是我们创建任务的集合，在 Project 中可以通过 tasks 属性访问 TaskContainer ，所以就可以以访问集合元素的方式访问我们创建的任务,访问的时候，任务名就是 [] 操作符的参数，[] 在 Groovy 中是一个操作符

task exAccessTask

tasks['exAccessTask'].doLast {...}

然后还有通过路径访问和通过名称访问，这两个都有 find 和 get 两种方式，区别是 get 的时候如果找不到该任务就会抛出 UnkownTaskException 而 find 在找不到任务时会返回 null

task exAccessTask1

tasks['exAccessTask1'].doLast {

// 通过路径访问

tasks.findByPath(':exacple:exAccessTask')

tasks.getByPath(':exacple:exAccessTask1')

// 通过名称访问

tasks.findByName('exAccessTask')

tasks.getByName('exAccessTask1')

}

**2. 任务分组和描述**

任务的分组其实就是对任务的分类，便于对任务进行归类整理。任务的描述就是说明这个任务的作用。

def Task myTask = task example

example.group = BasePlugin.BUILD\_GROUP

example.description = '这是一个构建的引导任务'

**3. << 操作符**

<< 操作符在 Gradle 的 Task 上是 doLast 方法的对标记形式

// 定义一个 exDoLast 的 task 并调用 doLast 方法

task(exDoLast) >> {

...

}

**4. 任务的执行分析**

当我们执行一个 Task 的时候，其实就是遍历执行其拥有的 actions 列表，这个列表保存在 Task 对象实例的 actions 成员变量中，其类型是一个 List。自定义的 Task 类型中可以声明一个被 TaskAction 注解标注的方法，意思是该 Task 本身执行要执行的方法。当我们使用 Task 方法创建任务的时候，Gradle 会解析其中被 TaskAction 注解的方法作为其 Task 执行的 Action，然后把该 Action 添加到 actions List 里，doFirst 和 doList 会将对应的 action添加到第一位和最后一位，最后这个 action List 的执行顺序就确定了。

采用非依赖的形式控制任务的执行顺序，可以通过 shouldRunAfter 和 mustRunAfter 两个方法来实现，这个限制在脚本中添加，通过 gradle 命令执行 task 时起作用

taskB.shouldRunAfter(taskA) 表示 taskB 应该在 taskA 执行之后执行，这里是应该而不是必须，所以又可能执行顺序不会按预设的执行

taskB.mustRunAfter(taskA) 表示 taskB 必须在 taskA 执行之后执行

Task 中有个 enable 属性，用于启动和禁用任务，默认时 true，表示启用，设置为 false 则禁止该任务执行，输出会提示该任务被跳过，调用 **taskName.enable = false** 即可

**5. 任务的 onlyIf 断言**

断言就是一个条件表达式。Task 有一个 onlyIf 方法，他接受一个闭包作为参数，如果该闭包返回 true 则该任务执行，否则跳过，可以用来判断控制打包等任务。

命令行中 -P 的意思时为 Project 指定 K-V 格式的属性键值对，使用格式为 -PK-V

task example {

println "执行"

}

example.onlyIf {

def execute = false

if(project.hasProperty("build\_apps"){

Object buildApps = project.property("build\_apps")

if("all".equals(buildApps) {

execute = true

}

}

execute

}

// 命令

// 会执行

gradle -Pbuild\_apps=all example

// 不会执行

gradle example

**6. 任务规则**

TaskContainer 继承了 NamedDomainObjectCopllection, 是一个具有唯一不变名字的域对象的集合，他里面所有的元素都有一个唯一不变的名字，改名字时 String 类型，我们可以通过名字获取该元素，NamedDomainObjectCopllection 的规则就是当想获取的名字的元素不存在时，对应在 TaskContainer 中就是想获取的任务不存在时，调用我们添加的规则来处理这种异常情况。

通过调用 addRule 来添加我们自定义的规则，addRule 有两个重载方法，第一个是直接添加一个 Rule，第二个是通过闭包配置成一个 Rule 在添加

// 通过闭包的形式，对 tasks 这个 Task 添加任务规则

tasks.addRule("对该规则的一个描述，这里是处理任务找不到时的处理") { String taskName ->

task(taskName) << {

println("该 $taskName 任务不存在")

}

}

**Gradle 插件**

Gradle 本身提供一些基本的概念和整体核心的框架，其他用于描述真实使用场景逻辑的都以插件扩展的方式实现

把插件应用到项目中，插件会扩展项目的功能，帮助我们在项目构建的过程中做很多事情，我们只需要按照它约定的方式，使用它提供的任务、方法或者扩展买就可以对我们的项目进行构建。

1. 添加任务，这些任务能帮我们做一下比如测试，编译，打包等的功能
2. 可以添加依赖配置到我们的项目中，通过插件配置我们项目在构建过程中需要的依赖，比如一些第三方库等依赖。
3. 可以向项目中现有的对象类型添加新的扩展属性/方法等，我们可以通过她们帮助我们配置优化构建，比如 android{} 这个配置可就是 AndroidGradle 插件为 Project 对象添加的一个扩展
4. 可以对项目进行一些约定，比如应用 Java 插件后，约定 src/main/java 目录是我们的源代码存放位置，等等

**插件的应用**

1. 应用二进制插件，二进制插件就是实现了 org.gradle.api.Plugin 接口的插件，可以有 plugin id

// 下面三种应用方式效果相同

apply plugin:'java' // 通过唯一的 plugin id

apply plugin:org.gradle.api.plugins.JavaPlugin // 通过插件类型

apply plugin:JavaPlugin // org.gradle.api.plugins 是默认导入的

1. 应用脚本插件，apply from:'version.gradle' ,应用脚本插件就是把这个脚本加载进来，使用 from 关键字，后面可以时本地的脚本文件也可以时一个网络文件，如果是网络文件的话要使用 HTTP URL

apply 的使用方式有三种，我们上面使用到的时第一种，接受一个 Map 类型参数的方式

常用的方式还有接受闭包的方式

apply {

plugin:'java'

}

还用一种 Action 的方式，这个知道即可，需要时再查文档

1. 应用第三方发布的插件，使用第三方发布的插件我们必须要在 buildscript{} 里配置其 classpath 才能使用，buildscript 是 Project 的一个方法，参数类型是一个闭包

// 配置 AndroidGradle 插件

buildscript {

repositories {

jcenter()

}

dependencies {

classpath 'com.android.tools.build.gradle:1.5.0'

}

}

buildscript{} 块是在构建项目之前，为了项目进行前期准备和初始化相关配置依赖的地方，配置好 buildscript 后就可以应用插件了,如果没有进行 buildscript 配置，则会提示找不到这个插件

buildscript{} 通过在 repositories 中配置仓库，通过 dependencies 中配置插件的 classpath

apply plugin: 'com.android.applicaition'

1. 使用 plugin DSL 应用插件,更简洁一下，并且使用这种配置方式，如果插件已经托管到了 <https://plugins.gradle.org/> 网站上我们就不需要在 buildscript 中配置 classpath

plugins {

id 'java'

}

**自定义插件**

自定义插件必须实现 Plugin 接口，这个接口有一个 apply() 方法，该方法在插件被应用的时候执行，所以我们可以实现这个方法，做一些想做的时，比如为 Project 创建新的 Task 等

**Java 插件**

**源码合集**

SoureSet - 源代码集合 - 源集，是 Java 插件用来描述和管理源代码及其资源的一个抽象概念，是一个 Java 源代码文件和资源文件的集合。通过源集，我们可以方便的访问源代码目录，设置源集的属性，更改源集的 Java 目录或者自由目录等。

main 和 test 是 Java 插件为我们内置的两个源代码集合，我们也可以通过 SourceSets 添加自己的源集，也可以通过 SourceSets 更改 main 和 test 这两个内置源集的代码路径

源集有很多的属性，通过这些属性我们可以对源集进行配置，这些属性包括 name，java，java.srcDirs，resources.srcDirs 等

新创建的源集必须配合多渠道使用，创建的源集的名字与匹配的渠道的名字应该是一样的，这样在编译打包时就会自动匹配，否则创建的源集是无法使用的

// 创建新的源集

sourceSets {

vip{

java {

srcDir 'src/java'

}

resources {

srcDir 'src/resources'

}

}

}

// 新的渠道

productFlavors {

// vip 渠道

vip{}

}

**配置第三方的依赖**

要使用第三方依赖，必须告诉 gradle 如何找到这些依赖，就需要在 Gradle 中配置仓库的依赖，这样 Gradle 就知道在哪儿搜寻我们依赖，Gradle 中通过 repositories{} 块来配置仓库

repositories {

maveCentral()

}

有了仓库配置以后，就可以在 dependencies{} 块中添加依赖，指定 group name version

dependencies {

// 标准写法

compile group: 'com.squareup.okhttp3', name: 'okhttp', version: '3.0.1'

// 简写方式，group，name，version 中间通过冒号分割

compile 'com.squareup.okhttp3:okhttp:3.0.1'

// 项目依赖,依赖后这个项目的 Java 类就会为你所用

compile project(':example')

// 文件依赖，例如 Jar 包依赖，如下所示，依赖了两个 Jar 包

compile files('libs/example01.jar', 'libs/example02.jar')

// Jar 包统一依赖,指定文件夹下的指定扩展名文件都会依赖

compile fileTree(dir: 'libs',include: '\*.jar')

}

**依赖的方式**

| **名称** | **意义** |
| --- | --- |
| compile | 编译时依赖 |
| runtime | 运行时依赖 |
| testCompile | 编译测试用例时依赖 |
| testRuntime | 仅仅在测试用例运行时依赖 |
| archives | 该项目发布构件(JAR 包等)依赖 |
| default | 默认依赖配置 |

Gradle 3.0 添加的新的依赖方式，添加了隔代隔离的概念，即 A 依赖 B，B implementation lib，此时编译期 A 是不能直接访问 lib 中的类的，运行期可以。隔离依赖只对 Java 代码生效，对 resource 无效

| **名称** | **意义** | **隔代隔离效果** |
| --- | --- | --- |
| implementation | 编译期隔代依赖不可见，运行期间可见 | “隔代”编译期间隔离 |
| api | 与 compile 一样，编译期/运行期 都可见 | 无 |
| compileOnly | 依赖项仅编译器对模块可见，并且编译和运行期对模块的消费者不可用 | 无 |
| runtimeOnly | 依赖项仅在运行时对模块及其消费者可用 | 编译期间隔离 |

**为不同源集指定不同的依赖 sourceSet + 依赖指令**

例如：vipCompile 'com.squareup.okhttp:okhttp:2.5.0'

**多项目构建**

Gradle 提供了基于根项目对其所有子项目通用配置的方法，Gradle 的根项目可以理解为是一个所有子项目的容器，subprojects 方法可以对其所有子项目进行配置，如果相对包括根项目在内的所有项目进行统一配置，可以使用 apllprojects 用法同 subprojects

subprojects 和 allprojects 都是 Project 中的方法，接受一个闭包，其中通过 Project 中的方法 repositories 和 dependencies 配置依赖仓库的地址和依赖的项目。

**这里需要注意 buildscript 中和 Project 中的 repositories 和 dependencies 方法是不同的**

subprojects {

// 为所有子项目添加代码仓库

repositories {

mavenCentral()

}

// 为所有子项目添加插件的 classpath

dependencies {

classpath 'com.android.tools.build:gradle:3.0.1'

}

// 为所有子项目添加 java 插件

applu plugin:'java'

}

**发布构件**

我们的库项目开发完毕后就可以配置构件然后上传构件到本地目录/maven 库等里面，上传就是发布。

// 配置

task publishJar(type: Jar)

version '1.0.0'

// artifacts 通过一个 Task 生成 jar 包

artifacts {

archives publishJar

}

// uploadArchives 是一个 upload task 用于上传我们发布的构件到本地/maven库等地方

uploadArchieves {

repositores {

...

}

}

**生产 idea 和 Eclipse 配置**

在一个已经初始化 wrapper 的项目中，通过 apply plugin:'idea' 就可以生产 IDEA 的工程配置文件，可以直接导入到 IDEA 中使用

**Android Gradle 插件**

Android Gradle 插件属于第三方插件，所以需要在 buildscript 中配置 classpath，这部分配置可以写到根工程的 build.gradle 文件中，这样所有的子工程就都可以使用了

AndroidGradle 插件继承 Java 插件，具有所有 Java 插件的特性

Android 插件默认提供了 androidTest，main，test 三种源集和三个渠道

**分类**

1. App 插件，id: com.android.application
2. Library 插件，id: com.android.libraay
3. Test 插件，id: com.android.test

android{} 块是 Android 插件提供的一个扩展方法，参数类型是一个闭包，在 Android 插件的构建任务执行时会调用这个方法，android{} 块中可以让我们自定义自己的 Android 工程。

**android{} 扩展中的配置**

1. compileSdkVersion 编译 Android 工程的 SDK 版本，原型是 android{} 中提功能的一个扩展方法
2. buildSdkVersion 使用的 Android 构建工具的版本
3. defaultConfig 是默认的配置，是一个 ProductFlavor，ProductPlavor 允许我们根据不同的情况同时生成多个不同的 APK，不过不针对自定义的 ProductPlavor 单独配置，会为这个 ProductPlavor 使用默认的 defaultConfig 的配置，有关 ProductPlavor 中的配置下面再说
4. buildTypes 是一个 NamedDomainObjectContainer，是一个域对象，和 sourceSet 类似，buildType 里有 release 和 debug 等，我们可以新增任意多个需要构件的类型，Gradle 会帮我们自动创建一个 BuildType，名字就是我们定义的名字，BuildType 中的属性也可以设置，具体的配置下面再说
5. productFlavors 是 AndroidGradle 提供的一个方法，用来添加不同的渠道，接受域对象类型的 ProductFlavor 闭包作为参数，可以通过 productFlavors{} 闭包添加渠道，每一个都是一个 ProductFlavor 类型的渠道，名字就是渠道名。

**defaultConfig 属性**

1. applicationId 指定生成的 App 的包名，默认为 null(构件时会在 Application 中的 package 获取)
2. minSdkVersion ，可以接受 int 和 String 两种，会统一处理
3. targetSdkVersion
4. versionCode
5. versionName
6. testApplicationId 配置测试 App 包名，有默认值，applicationId + ".test"
7. signingConfig 配置默认签名信息，是 ProductPlavor 中的一个 SigningConfig 属性，下面会详细介绍
8. proguardFile 混淆使用的 ProGuard 配置文件
9. proguardFiles 可以同事接受多个 ProGuard 配置文件
10. ...

**配置签名信息**

Android Gradle 提供了 signingConfigsf 配置块便于我们生成多个签名配置信息,signingConfigs 是 Android 的一个方法,它接受一个域对象作为其参数。前面我们讲过,其类型是 NamedDomainObjcctContainer,这样我们在 signingConfigs{} 块中定义的都是一个 SigningConfig ,一个 SigningConfig 就是一个签名配置,其可配置的元素如下:

storeFile 签名证书文件  
storePassword 签名证书文件的密码  
storeType 签名证书的类型  
keyAlias 签名证书中密钥别名  
keyPassword 签名证书中该密钥的密码

signingConfigs {

debug {

storeFile file('./keystore/debug.keystore')

storePassword "debug"

keyAlias "debug"

keyPassword "debug"

}

release {

storeFile file('./keystore/release.keystore')

storePassword "release"

keyAlias "release"

keyPassword "release"

v2SigningEnabled false

}

}

// 使用一

defaultConfig {

// 在 signingConfigs 中选择需要的签名信息

signingConfig signingConfigs.release

}

// 使用二

buildTypes {

release {

// 对构建类型设置签名信息

signingConfig signingConfigs.release

}

}

**隐藏签名文件**

为了防止签名文件的泄漏，可以在服务器端存储签名文件及其密钥，并在 signingConfigs 中通过在服务端存储的位置获取，如果本地需要调试，就在获取不到服务端的密钥时使用本地 debug 的密钥，保证了签名文件的安全性

**构建的应用类型 BuildType**

Android Gradle 帮我们内置了 relase 和 debug 两个构建类型，差别在于能否在设备上调试和签名不同，其代码和资源是一样的

buildTypes {

release {

}

debug {

}

}

如果需要新增构建类型在 buildTypes{} 中继续添加元素即可，buildTypes 也是 android 中的一个方法，接受一个域对象，添加的每一个都是 BuildType 类型

**构建类型中的配置**

1. applicationSuffix 用于配置默认 applicationId 的后缀
2. debugable 用于配置是否生成一个可供调试的 Apk ，值可以是 true 或者 false
3. jniDebugable 用于配置是否生成一个可供 Jni 调试的 Apk ，值可以是 true 或者 false
4. minifyEnable 是否启用 Proguard 混淆
5. miltiDexEnable 是否启用自动拆分多个 Dex
6. proguardFile 配置混淆文件
7. profuardFiles 配置多个混淆文件
8. shrinkResources 是否自动清理未使用的资源文件，默认 false
9. signingConfig 签名配置
10. zipalignEnable 是否启用 zipaline 优化，是一个优化整理 apk 的工具，可以提高系统和应用的运行效率，更快读写 apk 中的资源，降低内存使用等，默认开启

每一个 BuildType 都会生成一个 SourceSet ，默认位置 src// 这样我们就可以针对不同 BuildType 单独指定期 Java 源代码，res 资源等，构建时，AndroidGradle 会使用它们代替 main 中的相关文件

每一个 BuildType 都会生成一个 assemble 任务，执行相应的 assemble 任务就可以生成对应 BuildType 的 APK

**批量修改生成的 apk 名字**

applicationVariants.all { variant ->

variant.outputs.each { output ->

def outputFile = output.outputFile

def outputName = new String(outputFile.name);

if (outputFile != null && outputName.endsWith('.apk')) {

def fileName = outputFile.name.replace('.apk', "-${versionName}\_.apk");

output.outputFile = new File(outputFile.parent, fileName)

}

}

}

**动态配置 AndroidManifest 文件**

在构建的过程中动态修改 AndroidManifest 中的内容，Android Gradle 提供了 manifestPlaceholder、Manifest 占位符来替换 AndroidManifest 中的内容

ManifestPlaceholder 是一个 Map 类型，所以可以同时定义多个占位符，ManifestPlaceholder 是 BaseConfigImpl 中的一个属性，ProductFlavor 继承了 BaseConfigImpl，所以可以在 BaseConfigImpl 中访问，在 BuildType 中也直接访问 BaseconfigImpl 中的 ManifestPlaceholder

// BuildType 中定义

buildTypes {

debug {

// 替换原来的 ManifestPlaceholder

manifestPlaceholders = ["UMENG\_CHANNEL","google"]

// 添加新的元素

manifestPlaceholders.put("UMENG\_CHANNEL","google")

}

}

在 gradle 配置文件中定义了 manifestPlaceholder 后，在构建时，它会把 AndroidManifest 文件中所有的占位符变量为 manifestPlaceholder 中定义的占位符替换为 manifestPlaceholder 中对应的 value 值

// AndroidManifest 文件

<meta-data android:value="${UMENG\_CHANNEL}" android:name=""UMENG\_CHANNEL" />

如果在每一个 Flavor 中都需要配置相同占位符的不同 value，可以通过 productFlavors.all 方法遍历所有的 ProductFlavor 完成批量修改,name 为 ProductFlavor 的名字

productFlavors.all { flavor ->

manifestPlaceholders.put("UMENG\_CHANNEL",name)

}

**BuildConfig 文件**

BuildConfig 文件时 Gradle 编译后生成的，自动生成的包含包名、是否 Debug、构建类型、Flavor、版本号、版本名称的常量，开发中我们可以很方便的访问

Android Gradle 提供了 buildConfigField(String type,String name,String value) 方法让我们添加自己的常量到 BuildConfig 文件中，type 是要生成的字段类型，name 是要生成常量的名字，value 是生成字段的常量值，在 Gradle 脚本中就可以添加

注意 value 中的值，是单引号中的部分，是什么就写什么，要原封不动放入单引号，如果是 String ，双引号不能省略

// 比如在不同 productFlavor 中配置同一常量的不同值,这样在编译不同的 ProductFlavor 时就会生成不同的值

productFlavors {

google {

buildConfigField 'String','WEB\_URL','"http://google.com"'

}

baidu {

buildConfigField 'String','WEB\_URL','"http://baidu.com"'

}

}

**动态添加自定义的资源**

开发中遇到的资源包括图片、动画、字符串等，这些资源我们可以在 res 文件中定义，除了这种方式，针对 res/values 类型，里面有 arrays、ids、attrs、colors、dimens、strings 等类型，Gradle 提供了 resValue(String type,String name,String vlue) 方法在 ProductPlavor 和 BuildType 中可以定义这些资源的方法，这样我们就能根据不同渠道不同构建类型来自定义特有资源,也可以替换 res/valuse 文件夹中已经定义的资源,可以定义 id、bool、dimen、integer、color 等类型

// 以 productFlavors 为例，这样在构建时不同渠道生成的 App 名字就会不同

productFlavors {

google {

resValue 'String','app\_name','GoogleApp'

}

baidu {

resValue 'String','app\_name','BaiduApp'

}

}

**Java 编译选项**

Android Gralde 提供了一个 compileOptions 方法，接受一个 CompileOptions 类型的闭包为参数，来对构建过程中 Java 编译选项进行配置

compileOptions {

// 配置源文件编码为 UTF-8

encoding = 'utf-8'

// 配置 Java 源代码的编译版本

sourceCompatibility JavaVersion.VERSION\_1\_8

// 配置生成的字节码的版本

targetCompatibility JavaVersion.VERSION\_1\_8

}

**adb 操作选项配置**

Gradle 对 Android 手机的操作，最终还是调用了 adb 命令，Android Gralde 只不过是对 adb 命令的调用做了一层封装，可以通过 android 提供的 adbOptions 方法配置 Gradle 调用 adb 时的一些配置，该方法接受一个 AdbOptions 类型的闭包，AdbOptions 可以对 adb 操作选项添加配置

android {

adbOptions {

// 配置操作超时时间，单位毫秒

timeOutInMs = 5\* 1000\_0

// adb install 命令的选项配置

installOptions '-r','-s'

}

}

**DEX 选项配置**

Android 中 Java 代码被编译成 class 字节码，生成 apk 时又通过 dx 指令优化成 Android 虚拟机可执行的 DEX 文件，AndroidGradle 插件会调用 SDK 中的 dx 命令，dx 命令最终调用了 Java 编写的 dx.jar 库，是 Java 编写的，所以当调用 dx.jar 库是如果内存不足就会出现异常，默认给 dx 分屏的内存是一个 G8，也就是 1024M

android 提供了 dexOptions 方法用来添加对 dx 操作的配置，接受一个 DexOptions 类型的闭包，配置由 DexOptions 提供

android {

dexOptions {

// 配置是否启用 dx 增量模式，默认 false，增量模式速度块，但是有很多限制，不建议开启

increment true

// 配置执行 dx 命令是为其分配的最大堆内存

javaMaxHeapSize '4g'

// 配置是否开启 jumbo 模式，代码方法是超过 65535 需要强制开启才能构建唱歌

jumboMode true

// 配置是否预执行 dex Libraries 工程，开启后会提高增量构建速度，不过会影响 clean 构建的速度，默认 true

preDexLibraries true

// 配置 Andriod Gradle 运行 dx 命令时使用的线程数量

threadCount 8

}

}

**自动清理未使用的资源**

Android Gradle 提供了构建时自动清理未使用的资源的方法，会将项目中的和第三方依赖中的都清理

Resource Shrinking -- 资源清理 -- shrinkResource true  
Code Shrinking -- ProGuard 混淆代码清理 monifyEnabled true

shrinkResource 是在 BuildType 中配置的，默认不启用

Android Gradle 提供了 keep 方法来配置我们不希望被清理的资源

新建一个 xml 文件，res/raw/keep.xml 然后通过 tools:keep 属性来配置，可以通过都好分割配置资源列表，支持通配符 \*

keep 中还有一个 tools:shrinkMode 属性，用来配置自动清理资源的模式，默认是 safe 是安全的，这种情况下代码中通过 getResource() 方法引用的资源也会被保留，如果设置成 shrink 模式，这些资源也会被清理

keep 文件在构建时也会自动被清理，如果想保留也要添加到 keep 属性中

<?xml version="1.0" encoding="utf-8"?>

<resources xmlns:tools="http://schemas.android.com/tools"

tools:shrinkMode="safe"

tools:keep="@layout/getui\_notification,@drawable/h\_good,@drawable/h\_ok"/>

除了 shrinkResource 外，Android Gradle 还提供了 resConfig 配置，属于 ProductFlavor 的一个方法，用来配置哪些类型的资源会被打包到 apk 中，比如只打包中文资源，只打包 hdpi 中的图片等，它的参数就是我们 Android 开发时用到的资源限定符，这样在打包时可以清理大量无用资源

android {

defaultConfig {

// 接受一个

resConfig 'zh'

// 接受多个

resConfigs 'zh','hdpi'

}

}

**Android Gradle 多项目构建**

Android 的项目有三种，库项目、应用项目、测试项目，对应三种项目也有三种插件，com.android.library,com.android.applistion,com.android.test，开发特定的项目需要依赖特定的插件

**Android 多项目设置**

通过根项目中的 setting.gradle 进行配置

**库项目的引用和配置**

引用库项目也是通过在项目的 build.gradle 中的 dependencies 中指定，库项目引用时，库项目会生成一个 aar 包，如果时 Java 项目就会生成一个 jar 包提供给其他项目引用

dependencies {

compile project(':lib')

}

库项目默认发布出来的是一个 aar 包，默认发布的是 release 版本，可以在库项目中通过 android 提供的 defaultPublishConfig 配置来修改,这个方式可以配置不同的版本，只要名字合法即可，例如 "flavor1Debug" ，这样就可以针对不同的 Flavor 和 BuildType 来配置

android {

defaultPublishConfig "debug"

}

android {

defaultPublishConfig "flavor1Debug"

}

如果想同时发布多个版本的 aar 供不同项目引用，可以配置同时发布多个 aar 包,首先子项目进行配置，然后在其他项目引用该项目的生活，通过 path 指定子项目，通过 configration 指定 Flavar 和 BuildType，BuildType 又可以指向匹配 SourceSets ，这样就可以同时发布多个版本的 aar ，且这些 aar 的内容可以不一样

// 子项目配置

android {

publishNonDefault true

}

// 其他项目依赖子项目时的配置

dependencies {

flavor1Compile project(path: ':lib1', configration: 'flavor1Release')

flavor1Compile project(path: ':lib1', configration: 'flavor2Release')

}

**库项目的单独发布**

对于一些公共组件库、工具库等，我们可以将它们单独发布，供其他项目引用，我们也可以将它们发布到 Maven 或者 jcenter 中

**库项目发布到 Maven 仓库**

首先要搭建 Maven 私服，使用 Nexus Repository Manager 并部署启动

Maven 库部署以后，就可以把项目发不到中心库了，想要通过 Maven 发布，首先要在 build.gradle 中应用 Maven 插件，这个插件提供给 Product 一些扩展

应用 Maven 插件以后，需要配置 Maven 构建的三要素，group：artifact：version

版本号可以指定成快照形式，比如 1.0.0-SNAPSHOT ，这时候会发不到 snapshot 库里，每次发布版本号不会变化，只会在版本号后按顺序 + 1，例：1.0.0-1，1.0.0-2

引用的时候版本号写成 1.0.0-SNAPSHOT 即可，Maven 会帮我们下载最新也就是序号最大的快照版本，在联调测试的时候可以使用这种方式，当调试结束，就可以发布正式 release 版本

apply plugin: 'com.android.library'

apply plugin: 'maven'

version '1.0.0'

group 'org.snow.widget'

配置好 version 和 group 后，进行发布配置，比如发不到哪个 Maven 库，用户名密码是什么，发布的格式是什么，它的 artifact 即项目 name 是什么等

uploadArchives {

repositories {

mavenDeployer {

// 快照发布, URL 用户名 密码

snapshotRepository(url: mavenServer + mavenSnapshots) {

authentication(userName: repoUsername, password: repoPassword)

}

// 正式发布

repository(url: mavenServer + mavenReleases) {

authentication(userName: repoUsername, password: repoPassword)

}

// 项目 name

pom.artifactId = "pull-view"

// 发布格式

pom.packaging = 'aar'

}

}

}

**引用发布的项目库**

引用自己发布的库项目，首先需要在 Gradle 中添加对应的 classapth，然后在 dependencies 中添加引用

repositories {

maven {

// 项目库的 url

url ".../groups/relases"

}

}

dependencies {

compile 'org.snow.widget:pull-view:1.0.0'

}

为了使用快照版本，需要单独配置 classpath，因为快照库和 relase 库地址不同,引用也要单独添加

repositories {

maven {

// 项目库的 url

url ".../groups/snapshots"

}

}

dependencies {

compile 'org.snow.widget:pull-view:1.0.0-SNAOSHOTS'

}

可以通过 group 类型的配置，同时将 relase 和 snapshots 类型的库地址添加到 classpath, 引用还是区分 relase 和快照即可

repositories {

maven {

url ".../groups/public"

}

}

**Andriod Gradle 多渠道构建**

在 Android Gradle 中，定义了一个叫 Build Variant 的概念，即 构建变体，即不同的构件，也就是不同的 apk，一个 Build Variant 由一个 BuildType 和一个 ProductFlavor 构成，比如 release 和 debug，goole 和 baidu ，就可以构成四个变体。

Android Gradle 通过了 productFlavors 扩展来添加不同渠道，它接受域对象类型的 ProductFlavor 闭包作为参数，可以添加很多渠道，每一个 ProductFlavor 都是一个渠道，在 NamedDomainObjectContainer 中的名字就是渠道，比如 google ，baidu 等

渠道配置以后，Android Gradle 就会生成很多 Task ，基本都是基于 BuildType + ProductPlavor 的方法生成的，如 assembleGoogle assembleGoogleRelase assembleGoogleDebug 等,assembleRelease 运行后就会生成所有渠道的 release 包，assembleGoogle 运行后就会生成 google 渠道的 release 和 debug 包。出了 assemble 还有 install 系列的等

每个 ProductPlavor 还可以有自己的 SourceSet 和自己的 dependencies 依赖，这样，每个渠道都可以定义自己的资源，源代码以及第三方类库

**多渠道构建定制**

通过配置 ProductFlavor 达到灵活控制每一个渠道

1. applicationId 配置该渠道包名
2. consumerProguardFiles ，对 Android 库项目生效，配置混淆文件列表，会被打包到 aar，当应用项目开启混淆时，会使用 aar 包里的混淆文件对 aar 包里的代码进行混淆，这样应用项目就不用对该 aar 进行混淆配置了
3. manifestPlaceholders 上面已经介绍了
4. multiDexEnabled 开启多个 dex 的配置
5. proguardFiles 混淆文件
6. signingConfig 签名文件
7. versionCode 版本号
8. versionName 版本名
9. userJack 是否启用 Jack 和 Jill 编译器，Google 开发的编译器，速度快性能高，但不支持注解、 JDK8 的特性，处于试验阶段
10. testApplicationId 单元测试的 apk 的包名
11. testFunctionalTest 是否为功能测试，testHandleprofiling 是否启用分析功能
12. testInstrumentationRunner 测试使用...
13. testInstrumentationrunnerArguments 测试使用 ...
14. dimension ...

**dimension**

开发中如果遇到这样的情况，比如：项目区分收费和免费版并且代码不同，还区分不同的机器架构 x86 和 arm 并且代码不同，甚至还有其他区分纬度的区分，这时候，我们在配置 ProductFlavor 时，就会出现 收费版的配置在 x86 和 arm 中都各写一份，如果纬度更多，那么需要写的地方就更多，修改和增加时就更容易出错

为了解决这个问题，Android Gradle 为我们提供了 dimension 来解决这个问题

dimension 是 ProductFlavor 的一个属性，接受一个字符串，为该 ProductFlavor 的纬度，多个 ProductFlavor 可以同属一个纬度，这个纬度必须先定义出来才能使用，在 android{} 中，productFlavors{} 之前通过 flavorDimensions 声明纬度，定义后就可以在 ProductFlavor 中使用，在一个 ProductFlavor 中添加了 dimension 属性后，就相当于把这个 ProductFlovar 添加进了这个纬度的分组中。

只要定义了一个纬度，并且这个纬度分组中有了内容，其他所有非当前纬度的 ProductFlavor 都会与这个纬度中每一个 ProductFlavor 进行组合。就比如 收费和付费是一个纬度 'version'，我们在 paid 和 free 两个 ProductFlavor 指定了 demision 为 version，padi 和 free 中还可增加其他配置。这样我们再添加其他 ProductFlavor 后，比如添加了 arm ，Android Gradle 会以 BuildType+arm+version 的格式生成一系列 Task，这里就会生成 ArmPaidDebug ArmPaidRelease ArmFreeDebug ArmFreeRelease

这样，我们只用纬度去分组，去配置，剩下的 AndroidGradle 来组合，实现了共性配置，维护起来也很方便

**新版本的 AndroidGradle 插件，强制要求，必须为每一个 ProductFlavor 添加 dimension，否则编译会报错**

android {

flavorDimensions "abi","version"

productFlavors {

free {

dimension 'version'

}

paid {

dimension 'version'

}

x86 {

dimension 'api'

}

arm {

dimension 'api'

}

}

}

以上配置中，定义了两个分组，加上默认的两种 BuildType 后两两结合以后就会生成八种组合方式

**lint 检查**

Android 为我们提供了针对代码、资源的优化工具 lint，它可以帮助我们检查出哪些资源没有被使用，使用了哪些新的 API，哪些资源没有被国际化等。并且会生成一份报告，告诉我们哪些需要优化。

Lint 是一个命令行工具，是 Android Tool 目录下的一个工具，可以在命令行中直接执行。Android Gradle 中也对 Lint 做了很好的支持，提供了 lintOptions{} 闭包来配置 lint，达到做检查的目的

lintOptions 是 Android 对象的一个方法，接受一个类型为 LintOptions 的闭包，用于配置 Lint,Lint 提供了非常多的配置，以后可以根据项目配置想要的 Lint 检查规则

执行 gradle lint 命令即可运行 lint 检查，默认生成的报告在 outputs/lint-results.html 中

android {

lintOptions {

// 遇到 lint 检查错误会终止构建，一般设置为 false

abortOnError true

// 将警告当作错误来处理

warningAsErros true

// 检查新 API

check 'NewApi'

}

}

**多渠道构建的效率**

通过在 APK 包中的 METE-INF 文件夹下新增文件，文件名为有意义的渠道名字，通过 Python 脚本复制原始 APK 包，解压后在 METE-INF 中添加文件，然后再打包

在使用时，Android 程序的 APP 中通过对 apk 进行读取，就可以读到 METE-INF 中文件的名字，这样就得到了渠道标示，取到标示后可以存入 SP 中，这样就不用每次都读取 APK