在 Android Studio 构建的项目中，基于 Gradle 进行项目的构建，同时使用 Android DSL 进行 Android 项目的配置，而 Gradle 是基于 Groovy 脚本语言进行开发，所以熟练掌握 Android Studio 中的项目配置，就需要掌握以下知识点：

- Groovy 基础
- Gradle DSL 语法
- Android DSL 语法

1. Groovy 基础

Groovy 和 Java 一样也是一门 JVM 语言，最终都会编译成 .class 文件然后运行在 JVM 上，Groovy 语言类似脚本，语法简单更灵活，所以在编写项目脚本构建上优势更加明显。

1.1 Groovy 环境配置

工欲善其事必先利其器，编码之前环境先行。网上有很多配置 Groovy 开发的环境，针对 Android 程序员来说，使用我们的 Android Studio 就能满足。

第一步：创建一个 Application 或 Library 工程

工程创建好后，除了 build.gradle 文件和 src/main 目录文件夹保留外，其它的都可以删除。

第二步：在 src/main 目录下创建一个 groovy 文件夹

在创建的 Module 工程中的 build.gradle 文件中添加以下依赖：

```
apply plugin: 'groovy'

dependencies {
    compile gradleApi()
    compile localGroovy()
}

repositories {
    mavenCentral()
}
```

第三步：在 groovy 文件夹中创建 .groovy 文件

Android Studio 中会自动识别 groovy 文件夹下的 .groovy 文件。

第四步：配置运行环境

- 点击 Edit Configurations

1.2 Groovy 基本语法

1.2.1 基本数据类型

Groovy 是弱化类型语言，但是实现上还是强类型相关，如果类型不对，还是会报错。Groovy 中的基本数据类型有：

- byte -这是用来表示字节值。例如2。
- short -这是用来表示一个短整型。例如10。
- int -这是用来表示整数。例如1234。
- long -这是用来表示一个长整型。例如10000090。
- float -这是用来表示32位浮点数。例如12.34。
- double -这是用来表示64位浮点数，这些数字是有时可能需要的更长的十进制数表示。例如12.3456565。
- char -这定义了单个字符文字。例如“A”。
- Boolean -这表示一个布尔值，可以是true或false。
- String -这些是以字符串的形式表示的文本。例如，“Hello World”的。

```
int a = 4;
float b = 1.0;
double c = 2.1;
byte  te = 4;
char ch = 'c';
String str = "abcd";
```

1.2.2 变量和方法的声明

在 Groovy 中通过 **def** 关键字进行变量和方法的声明。

```
def a = 1
def b = 1.0
def str = "Hello World"

def output() {
    print 'Hello World'
}
```

Groovy 类似脚本，所以有很多都可以省略：

- 语句后的分号;可以省略
- 变量的类型可以省略
- 方法返回值 return 语句可以省略

方法的声明

Groovy 也是一门 JVM 语言，所以在语法上与 Java 有很多相通的地方，这样在方法的声明时候格式也比较随意，所以作为 Android 程序员，我们可以选择靠拢 Java 语法格式的风格。

```
def methodName() {
    print("HelloWorld")
}

def int sum2Number(a, b) {
    return a + b
}

def sum(a, b) {
    return a+b
}

int add(a ,b) {
    return a + b
}
```

1.2.3 循环

Groovy 中循环控制语句与 Java 中的类似，有以下三种：

- for 语句
- while 语句
- for-in 语句

```
for(int i = 0 ;i < 9; i++) {
    println(i)
}

int i = 0
while(i++ < 9) {
    println(i)
}

for(int j in 1..9) {
    println(j)
}
```

同样，针对循环也有循环控制语句：

- break：break语句用于结束循环和switch语句内的控制流。
- continue：结束本次循环，进行下次循环，仅限于while和for循环。

```
for(int i = 0 ;i < 9; i++) {
    println(i)
    continue
}

for(int j in 1..9) {
    println(j)
    break
}
```

1.2.4 条件判断语句

Groovy 中的条件判断语句与 Java 中的类似，有：

- if
- if...else
- if...else if...else
- switch

例子就不演示了，语法跟 Java 相同。

在上面的 Groovy 基础介绍中，形式上跟 Java 语言非常相似，没有太大的变化，针对 Java 、Android 程序员来说应该非常容易上手。

1.3 Groovy 中的集合

Java 中集合主要有：List、Map 衍生出来的类，在 Groovy 中同样存在集合，名称跟 Java 中的相同，只不过形式发生改变了，更加简单容易操作。Groovy 中的集合：

- List：亦称为列表，列表是用于存储数据项集合的结构。
- Map：亦称为映射，映射（也称为关联数组，字典，表和散列）是对象引用的无序集合。

1.3.1 List 列表

基本语法：List 列表使用[ ] 进行声明，并通过索引进行区分。

```
def listEmpty = []    //空列表
def list = [1,2,3,4,5] //整数值列表
def listInt = [1,[2,3],4,5] //列表嵌套列表
def listString = ["andoter","note"] //字符串列表
def listNone = ["andoter",1,4]  //异构对象列表
`
```

列表中的方法：

- boolean add(E e)
- void add(int index, E element)
- boolean addAll(Collection<? extends E> c)
- void clear()
- boolean contains(Object o)
- Iterator iterator()
- Object[] toArray()
- int lastIndexOf(Object o)
- E set(int index, E element)

这些方法都跟 Java 中的类似，打开对应的类型查看后，发现通过 def 声明的列表竟然是 java.util.List 下面的。

```
def listEmpty = []    //空列表
def list = [1,2,3,4,5] //整数值列表
def listInt = [1,[2,3],4,5] //列表嵌套列表
def listString = ["andoter","note"] //字符串列表
def listNone = ["andoter",1,4]  //异构对象列表

listEmpty.add(1)
listEmpty << 6
println(listEmpty.size())
list.clear()
println(listInt.contains([2,3]))
println(listString.lastIndexOf("note"))
println(listNone.indexOf(1))
```

> 需要注意，在 groovyjarjarantlr.collections.List 包下同样存在 List，所以使用的时候需要注意。

关于列表 List 的遍历，我们可以参照 Java 中的 Iterator 接口去遍历，或者使用 Groovy 系统提供的 each 方法进行遍历。在 Groovy 中提供 DefaultGroovyMethods 类，该类定义很多快捷使用方法：

- abs：取绝对值计算
- addAll(Collection)
- each：遍历
- eachWithIndex：带 index 的遍历
- grep：符合条件的element会被提取出来，形成一个list
- every：所有的element都满足条件才返回true，否则返回false
- any：只要存在一个满足条件的element就返回true，否则返回false
- join：用指定的字符连接collection中的element
- sort：根据指定条件进行排序
- find：查找collection中满足条件的‘第一个’element
- findAll：查找collection中满足条件的‘所有’element

很多使用的方法，可参照源码查看。

```
def listString = ["andoter","note"]

listString.each {
    println(it)
}

listString.each {
    value -> println(value)
}
```

1.3.2 Map 映射

Map集合中的元素由键值访问。 Map中使用的键可以是任何类。当我们插入到Map集合中时，需要两个值：键和值。

```
def mapEmpty = [ : ]
def mapString = ["name":"andoter","email" : "andoter0504@gmail.com"]
def mapInt = ["name" : 123, "age" : 26]
```

映射中的方法：

- void clear()
- boolean containsValue(Object value)
- Map.Entry eldest()
- Set> entrySet()
- void forEach(BiConsumer<? super K, ? super V> action)
- V get(Object key)
- Set keySet()
- Collection values()

总体上方法与 Java 中的 Map 相同。

```
def mapEmpty = [ : ]
def mapString = ["name":"andoter", "email" : "andoter0504@gmail.com"]
def mapInt = ["name" : 123, "age" : 26]

mapEmpty.put("name","andoter")
mapEmpty.values()
mapString.get("name")
mapInt.containsValue("123")
mapString.each {key, value ->
    if(key == null || key.length() == 0) {
        println("Null Object")
    }
```

```
    if(key.equals("name")){
        println(key + "=" + value)
    }else{
        println(key + ":" + value)
    }
}
```

1.4 Groovy 中的 IO 操作

Java 提供了 java.io.* 一系列方法用于文件的操作，这些方法在 Groovy 中也适用。Groovy 针对 Java 提供的方法做了增强处理，更方便使用。

```
def file = new File("/Users/dengshiwei/WorkProject/GradlePlugin/groovydemo/src/main/groovy/T
if (file.exists()) {
   file.eachLine {
       line ->
           println(line)
   }
} else {
    print("File not exist")
}
```

这里简单的示例下，更多的内容请参照官方 API 接口。

1.5 闭包

闭包作为 Groovy 中非常重要的特性，它使得 Groovy 语言更加灵活，在 Gradle 项目构建中，更是在 DSL 中大量被使用，所以掌握闭包的使用对掌握 Android 项目构建有非常重要的作用。

1.5.1 闭包的定义

闭包的定义格式：

```
{
    parameters -> statements
}
```

从形式上来看与 Lambda 表达式非常类似，所以熟悉 Lambda 表达式的同学上手闭包非常简单。如果闭包没有定义参数，它隐含一个参数 it,类似 Java 中的 this，假设你的闭包不需要接受参数，但是还是会生成一个隐式参数 it，只不过它的值为 null,也就是说，闭包至少包含一个参数。

无参数的闭包

```
def closure = {
    println("No Parameters")
}
```

一个参数的闭包

```
def closureOneParameters = {
    key -> println(key)
}
```

两个参数的闭包

```
def closure2Parameter = {
    key,value->
        if (key == 1) {
            key = key + 1
            println(key + ":" + value)
        } else if (key == 2)
            println(key + ":" + value)
}
```


1.5.2 闭包的特性

闭包的引入让 Groovy 语言更加简单、方便，比如作为函数的最后一个参数，闭包可以单独写在函数，本小节中介绍一下闭包常见的使用形式。

闭包特性：

- 闭包可以访问外部的变量，方法是不能访问外部变量的。
- 闭包中可以包含代码逻辑，闭包中最后一行语句，表示该闭包的返回值，不论该语句是否冠名return关键字，如果最后一行语句没有不输入任何类型，闭包将返回null。
- 闭包的参数声明写在‘->’符号前，调用闭包的的标准写法是：闭包名.call(闭包参数)。
- 闭包的一些快捷写法，当闭包作为闭包或方法的最后一个参数。可以将闭包从参数圆括号中提取出来接在最后，如果闭包是唯一的一个参数，则闭包或方法参数所在的圆括号也可以省略。对于有多个闭包参数的，只要是在参数声明最后的，均可以按上述方式省略。

闭包作为函数参数

闭包作为函数参数时，跟普通的变量参数使用方式相同。

```
def checkKey = {
    map ->
        if (map.size() == 0) {
            println("Parametes is Null or Empty")
        }
```

```
        println(map)
}

def enqueue(key, value, closure) {
    def map = [:]
    map.put(key, value)
    closure(map)
}

enqueue(1, 2, checkKey)
```

通常情况下，在函数具有闭包作为参数的时候，会将闭包放在最后一个参数的位置，**当闭包作为最后一个参数的时候，闭包可以抽离到函数体之外，提高函数的简洁性。**

关于 Groovy 比较好的文章

- 深入理解Android之Gradle
- Groovy进阶之函数、闭包和类

2. Gradle DSL 语言

上面我们针对 Groovy 语言进行简单的学习，接下来就是 Gradle DSL 语言的学习。Gradle 是 Android Studio 中采用的全新项目构建方式。

> Gradle 是一个开源的自动化构建工具，提供更高的灵活和体验。Gradle 脚本采用 Groovy 或 Kotlin 进行编写。官方文档

2.1 基本概念

Gradle 是一种脚本配置，所以当它执行的时候，它需要跟对应的类型相对应。在 Gradle 中存在以下三种类型：

脚本类型	委托的实例
Build script	Project
Init script	Gradle
Settings script	Settings

Gradle 围绕项目 Project，所以 Project 是我们最重要的接口,通过 Project 接口，我们可以获取整个 Gradle 的属性。通常我们的项目在 Project 模式的下结构是：

```
├── app #Android App目录
│   ├── app.iml
│   ├── build #构建输出目录
│   ├── build.gradle #构建脚本
│   ├── libs #so相关库
│   ├── proguard-rules.pro #proguard混淆配置
│   └── src #源代码，资源等
├── build
│   └── intermediates
├── build.gradle #工程构建文件
├── gradle
│   └── wrapper
├── gradle.properties #gradle的配置
├── gradlew #gradle wrapper linux shell脚本
├── gradlew.bat
├── LibSqlite.iml
├── local.properties #配置Androod SDK位置文件
└── settings.gradle #工程配置
```

2.2 Project

2.2.1 生命周期 Lifecycle

Project 与 build.gradle 文件是一对一的关系，在初始化脚本构建的过程中，Gradle 为每一个项目创建 Project对象。步骤如下：

初始化阶段

在初始化阶段，构建工具根据每个 build.gradle 文件创建出每个项目对应的 Project，同时会执行项目根目录下的 settings.gradle 分析需要参与编译的项目。

比如我们常见的 settings.gradle 配置文件：

```
include ':app', ':groovydemo'
```

指明了需要编译的项目。

配置阶段

配置阶段为每个 Project 创建并配置 Task，配置阶段会去加载所有参与构建项目的 build.gradle 文件，将每个 build.gradle 文件转换为一个 Gradle 的 Project 对象，分析依赖关系，下载依赖。

执行阶段

Gradle 根据 Task 之间的依赖关系，决定哪些 Task 需要执行，以及 Task 之间的先后顺序。

2.2.2 Task

Task 是 Gradle 中的最小执行单元，所有的构建、编译、打包、debug、test 等都是执行了某一个 task，一个 Project 可以有多个 Task，Task 之间可以互相依赖。例如我有两个 Task，TaskA 和 TaskB，指定 TaskA 依赖 TaskB，然后执行 TaskA，这时会先去执行 TaskB，TaskB 执行完毕后在执行 TaskA。

同时，我们也可以自定义 Task，也可以查找 Task 是否存在。站在编程的角度来看 Task 同样是一个类，核心方法：

- String getName()：获取任务名称
- Project getProject()：获取任务所在的 Project 对象
- List> getActions()：获取 Action
- TaskDependency getTaskDependencies()：获取任务依赖
- Task dependsOn(Object... var1)：任务依赖关系函数
- void onlyIf(Closure var1)
- TaskState getState()：获取任务的状态
- Task doFirst(Closure var1)：任务先执行..
- Task doLast(Closure var1)：任务后执行..
- String getDescription()：获取任务描述
- String getGroup()：获取任务分组

1. 任务的创建

Task 是 Gradle 中最小执行基本单元，创建任务的方式有以下几种：

- Project.task(String name)
- Project.task(String name, Closure configureClosure)
- Project.task(Map args, String name, Closure configureClosure)
- Project.task(Map args, String name)
- TaskContainer.create(String name)
- TaskContainer.create(Map options)

前面三种都是基于 Project 提供的 task 重载方法进行创建。这里需要着重介绍下里面的 Map 参数，Map 参数选项用于控制 Task 的创建以及属性。

配置项	描述	默认值
type	任务创建的类型	DefaultTask
overwrite	是否重写已存在的任务	false
dependsOn	添加任务的依赖	[]
action	添加任务中的 Action	null
description	任务的描述	null
group	配置任务的分组	null

第一种：直接以任务名称创建任务

```
Task copyTask = task("copyTask")
copyTask.description = "Copy Task"
copyTask.group = "custom"
copyTask.doLast {
    print("Copy Task Create")
}
```

这种方式跟 Java 中的创建对象的方式非常相似，这种方式的本质是调用 Project 类中的 task(String name) 方法进行对象的创建。

第二种：task + 闭包的方式

```
Task taskClosure = project.task("taskClosure"){
    print("Task Closure")
}
```

这种写法利用闭包是最后一个参数的时候，可以抽取到外部写。上面的这种写法也可以精简：

```
task taskClousre {
    print("Task Closure")
}
```

这种形式的在 .gradle 脚本文件中用的非常多，所以大家也写这种吧！

第三种：task + Map

```
Task mapTask = project.task(dependsOn: copyTask,description: "mapTask",group: "mapTask",
        "mapTask"){
    println("Map Task Create")
}
```

这里通过 Map 进行 Task 的一些设置，这里我们可以同样以方式一一样，单独进行设置。

第四种：TaskContainer 创建任务

```
project.tasks.create("TaskContainer") {
    description "TaskContainer"
    group "TaskContainer"
    doLast {
        println("TaskContainer")
    }
}
```

在上面的演示例子中，我们也介绍了任务的分组和描述的使用，可以在 Gradle Pojects 栏中进行查看任务的分组和描述。

2. 任务之间的关系

1. dependsOn(Task task) 任务依赖,通过 dependsOn 可以建立任务之间的执行依赖关系，先执行依赖的任务。

```
def name = "Hello World from"

task checkName {
    if (name.length() > 0){
        name = name.replace("from", "")
    }
}

task printName() {
    println(name)
}

printName.dependsOn(checkName)
```

2. mustRunAfter(Task task) 必须在添加的任务之后执行。

```
def name = "Hello World from"

task checkName {
    if (name.length() > 0){
        name = name.concat(" China")
    }
}

task printName() {
    println(name)
}

printName.mustRunAfter(checkName)
```

3 任务类型

在 1 节中，我们提到了创建任务时可以通过 Map 配置任务的依赖属性关系，里面涉及到 任务类型（type），默认值是 DefaultTask 类型，关于 type，我的理解是 Groovy 系统的 Task 类型，我们查看官方文档，可以看到有很多 Task 类的子类，这些应该都可以作为 type 值进行设置？那么常见的任务类型有哪些呢？

> 官方 Task API

1. Copy 类型 Task，Copy 任务的方法：

- eachFile：遍历文件
- exclude(Closure excludeSpec)：去除包含的内容
- filter(Closure closure)：过滤
- from(Object... sourcePaths)：源目录
- include(Closure includeSpec)：包含内容
- into(Object destDir)：目标目录
- rename(Closure closure)：重命名
- with(CopySpec... sourceSpecs)

通过 Copy 任务，我们可以更方面实现文件的拷贝复制类操作，因为它提供了一些封装好的方法，不需要我们在通过 File 的操作进行。常见的就是创建的 makeJar 任务,拷贝系统编译的 jar 包到指定目录。

```
task makeJar(type: Copy) {
    def jarName = 'SensorsAnalytics-Android-SDK-Release-' + version + '.jar';
    delete 'build/libs/' + jarName
    from('build/intermediates/bundles/release/')
    into('build/libs/')
    include('classes.jar')
    rename('classes.jar', jarName)
}
```

makeJar 的任务执行步骤：首先设置 type 是 Copy 类型任务，定义拷贝目标的 jar 名称，接着删除 目标目录已存在的 jar 文件，from 从源目录拷贝到 into 的目标目录，包含 include 的 classes.jar，最后给文件重命名。

> 在 Android Studio 的 2.X 版本中自动生成的 jar 包路径在：build/intermediates/bundles/release/ 目录，在 3.X 中目录：build/intermediates/packaged-classes/release/。

1. Jar 类型 Task

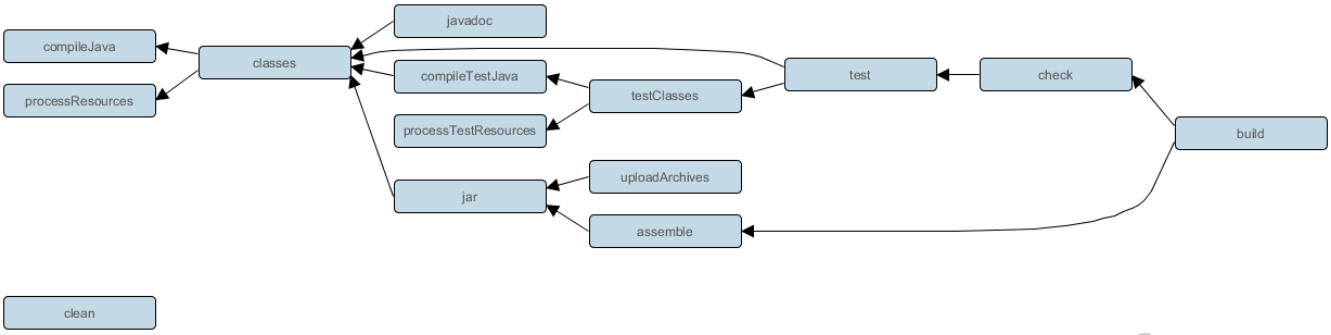
```
generated by haroopad



)
    }
}
```

">task makeJar(type: Jar, dependsOn: build) { delete 'build/libs/sensorsdata.jar' from('build/intermediates/classes/release') into('build/libs/') exclude('android/', 'androidx/','BuildConfig.class', 'R.class') exclude { it.name.startsWith('R @@footer) } }

Jar 任务的作用就是打包 jar 包，比如我们通过 from 指定工程目录中的源码进行打包，这样我们就可以实现高度的定制化，不像通过 Copy 任务复制系统根据整个目录生成的 Jar 包一样。比如下面的 task 生成 Jar 包。

```
task makeJars(type: org.gradle.jvm.tasks.Jar) {
    from(android.sourceSets.main.java.srcDirs)
}
```

在 Gradle 中提供了很多常见的任务：



2.3. 常见脚本块

Gradle 支持 Groovy 语言进行编写，非常灵活支持各种插件。比如你想在脚本中使用一些第三方的插件、类库等，就需要自己手动添加对这些插件、类库的引用，而这些插件、类库又不是直接服务于项目的，而是支持其它 build 脚本的运行，所以你应当将这部分的引用放置在 buildscript 代码块中。gradle 在执行脚本时，会优先执行buildscript代码块中的内容，然后才会执行剩余的build脚本。所以需要我们了解常见的脚本块配置。

2.3.1 allprojects { }

配置整个 Project 和子项目的配置。

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

比如我们在 allprojects 内部定义的 Task 任务，就会用于根目录和子项目。比如下面的例子，我们执行：./gradlew print 任务，打印的结果如下：

```
allprojects {
    task print {
        println project.name
    }
}
```

输出结果：

```
GradlePlugin(根目录)
app
groovydemo
sensorsdatalibrary
```

2.3.2 buildscript { }

buildscript 中的声明是 gradle 脚本自身需要使用的资源。

```
buildscript {

    repositories {
        google()
        jcenter()
        mavenCentral()
    }
    //格式为-->group:module:version
    dependencies {
        classpath 'com.android.tools.build:gradle:3.1.2'
        classpath 'com.qihoo360.replugin:replugin-plugin-gradle:2.2.4'
    }
}
```

2.3.3 configurations { }

配置整个 Project 的 dependency 属性，与之对应的是 ConfigurationContainer，在 Project 项目中可以通过以下方法获取 ConfigurationContainer 对象：

- Project.getConfigurations()
- configurations

通常使用最多的就是去除依赖，比如我们添加的依赖中也依赖某个库，这种间接依赖的冲突，
transitive dependencies 被称为依赖的依赖，
称为“间接依赖”比较合适。

```
configurations {
    compile.exclude module: 'commons'
    all*.exclude group: 'org.gradle.test.excludes', module: 'reports'
}
```

2.3.4 dependencies { }

配置项目的依赖库，与之对应的是 DependencyHandler 类。
在 dependencies{} 脚本块中有不同的依赖方式，这里在 Android Studio 的 2.X 版本与 3.X 版本中差别还是挺大的，Android Studio3.0 中，compile 依赖关系已被弃用，被 implementation 和 api 替代，provided 被 compile only 替代，apk 被 runtime only 替代。为了比较方便，前面写是 3.X 版本，括号是 2.X。

- implementation：依赖的库只能在本项目使用，外部无法使用。比如我在一个 libiary 中使用 implementation 依赖了 gson 库，然后我的主项目依赖了 libiary，那么，我的主项目就无法访问 gson 库中的方法。这样的好处是编译速度会加快，推荐使用 implementation 的方式去依赖，如果你需要提供给外部访问，那么就使用 api 依赖即可
- api(compile)：使用该方式依赖的库将会参与编译和打包
- testImplementation(testCompile)：只在单元测试代码的编译以及最终打包测试 Apk 时有效
- debugImplementation(debugCompile)：只在 debug 模式的编译和最终的 debug Apk 打包时有效
- releaseImplementation(releaseCompile)：仅仅针对 Release 模式的编译和最终的 Release Apk 打包
- compileOnly(provided)：只在编译时有效，不会参与打包，可以在自己的moudle中使用该方式依赖。比如 com.android.support，gson 这些使用者常用的库，避免冲突。
- runtimeOnly(apk)：只在生成 Apk 的时候参与打包，编译时不会参与，很少用。

下面是一些常见的依赖使用方式：

```
apply plugin: 'java'
//so that we can use 'compile', 'testCompile' for dependencies

dependencies {

  //for dependencies found in artifact repositories you can use
  //the group:name:version notation
  compile 'commons-lang:commons-lang:2.6'
  testCompile 'org.mockito:mockito:1.9.0-rc1'

  //map-style notation:
  compile group: 'com.google.code.guice', name: 'guice', version: '1.0'

  //declaring arbitrary files as dependencies
  compile files('hibernate.jar', 'libs/spring.jar')

  //putting all jars from 'libs' onto compile classpath
  compile fileTree('libs')
}
```

在实际项目开发中，我们会引入很多第三方开源库，自然就会造成依赖冲突，这里就涉及到在 dependencies 提供的配置字段：

- force = true：即使在有依赖库版本冲突的情况下坚持使用被标注的这个依赖库版本

- transitive = true：依赖的依赖是否可用，举个例子，使用的三方库中可能也依赖别的库，我们称之为“间接依赖”
- exclude：用于排除指定版本库，通常用于排除冲突依赖库

```
dependencies {
  compile('com.sensorsdata.analytics.android:SensorsAnalyticsSDK:2.0.2') {
    //强制使用我们依赖的 2.0.2 版本库
    force = true

    //剔除间接依赖的库,可以通过这三种方式，后面再讲解自定义插件的时候就能看懂这三种方式了。
    exclude module: 'cglib' //by artifact name
    exclude group: 'org.jmock' //by group
    exclude group: 'org.unwanted', module: 'iAmBuggy' //by both name and group

    //禁用所有的间接依赖库
    transitive = false
  }
}
```

2.3.5 repositories { }

配置 Project 项目所需的仓库地址，Gradle 必须知道从哪里下载外部依赖，这是由仓库配置来指定的，比如 google()、jcenter() 或 mavenCentral()。通常在 buildscript 脚本块中也能看到配置的 repositories 属性，buildscript 中的声明是 gradle 脚本自身需要使用的资源，可以声明的资源包括依赖项、 第三方插件、 maven 仓库地址等。而在 build.gradle 文件中直接声明的依赖项、仓库地址等信息是项目自身需要的资源。

```
repositories {
    //Maven本地仓库，寻找本地仓库的逻辑与Maven相同
    mavenLocal()
    //Maven中心仓库
    mavenCentral()
    //JCenter仓库
    jcenter()
    //其它Maven远程仓库
    maven {
        //可以指定身份验证信息
        credentials {
            username 'user'
            password 'password'
        }
        url "http://repo.mycompany.com/maven2"
        //如果上面的URL找不到构件，则在下面找
```

```
        artifactUrls "http://repo.mycompany.com/jars"
    }
    //Ivy远程仓库
    ivy {
        url "http://repo.mycompany.com/repo"
    }
    //Ivy本地仓库
    ivy {
        url "../local-repo"
    }
    //扁平布局的文件系统仓库
    flatDir {
        dirs 'lib'
    }
    flatDir {
        dirs 'lib1', 'lib2'
    }
}
```

2.3.6 sourceSets { }

配置项目的源码目录结构。

```
sourceSets {
    main {
        java {
            srcDirs = ['src/java']
        }
        resources {
            srcDirs = ['src/resources']
        }
    }
}
```

2.3.7 subprojects { }

用于配置子项目的脚本块。比如我们在 subprojects 中配置 print 任务，则只会作用于子目录。

```
subprojects {
    task print {
        println project.name
    }
}
```

输出结果：

```
app
groovydemo
sensorsdatalibrary
```

2.3.8 publishing { }

用于发布构建。

```
publishing {
  publications {
    myPublication(MavenPublication) {
      from components.java
      artifact sourceJar
      pom {
        name = "Demo"
        description = "A demonstration of Maven POM customization"
        url = "http://www.example.com/project"
        licenses {
          license {
            name = "The Apache License, Version 2.0"
            url = "http://www.apache.org/licenses/LICENSE-2.0.txt"
          }
        }
        developers {
          developer {
            id = "johnd"
            name = "John Doe"
            email = "john.doe@example.com"
          }
        }
        scm {

          connection = "scm:svn:http://subversion.example.com/svn/project/trunk/"
          developerConnection = "scm:svn:https://subversion.example.com/svn/project/trunk/"
          url = "http://subversion.example.com/svn/project/trunk/"
        }
      }
    }
  }
}
```

2.4 Gradle 插件

通过使用插件可以扩展项目的功能，帮助我们做很多任务，比如编译、打包，Gradle 插件可以分为两类：

- 二进制插件：继承 org.gradle.api.Plugin 接口实现的插件
- 脚本插件：直接在 build.gradle 配置文件

2.4.1 二进制插件

常见的二进制插件 com.android.application，这里的 ‘com.android.application’ 就是插件的 plugin id，二进制插件的使用形式:

```
apply plugin： plugin id
```

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

2.4.2 脚本插件

通常脚本插件用于本地的配置存储，使用格式：

```
apply from： ‘fileName’
```

```
// config.gradle
rootProject.ext {
    android = [
        compileSdkVersion : 28,
        buildToolsVersion : "28.0.0",
        applicationId : "sw.andoter.com.gradleplugindemo",
        minSdkVersion : 18,
        targetSdkVersion : 28,
        versionCode : 1,
        versionName : "1.0"
    ]

    sdkVersion = 13

    apkPath = [
       apkPath : "/Users/dengshiwei/Desktop/*.apk"
    ]
}

apply from: "../config.gradle"
```

2.5 Gradle 自定义插件

在基于 Gradle 的项目构建中插件的使用非常常见，比如 com.android.application、com.android.library 等，如何自定义自己的插件呢？在 Gradle 中提供了 Plugin 接口用于自定义插件，本小节只做简单介绍自定义插件的步骤：

1. 创建一个 module，什么样的都可以，不管是 Phone&Tablet Module 或 Android Libraty 都可以，然后只留下 src/main 和 build.gradle，其他的文件全部删除。

2. 在main 目录下创建 groovy 文件夹，然后在 groovy 目录下就可以创建我们的包名和 groovy 文件了,记得后缀要以 .groovy 结尾。在这个文件中引入创建的包名，然后写一个 Class 继承于 Plugin< Project > 并重写 apply 方法。

```
class MyPlugin implements Plugin<Project> {

    @Override
    void apply(Project project) {
        System.out.println("-----------插件开始-----------")
        System.out.println("---这是我们的自定义插件---")
        System.out.println("-----------插件结束-----------")
    }
}
```

3. 在 main 目录下创建 resources文件夹，继续在 resources 下创建 META-INF 文件夹，继续在 META-INF 文件夹下创建 gradle-plugins 文件夹，最后在 gradle-plugins 文件夹下创建一个 xxx.properties 文件，注意：这个 xxx 就是在 app 下的 build.gradle 中引入时的名字，例如：apply plugin: ‘xxx’。在文件中写 implementation-class=implementation-class=com.andoter.customplugin.MyPlugin。

```
implementation-class=com.andoter.customplugin.MyPlugin
```

4. 打开 build.gradle 删除里面所有的内容。然后格式按这个写，uploadArchives 是上传到 maven 库，然后执行 uploadArchives 这个 task，就将我们的这个插件打包上传到了本地 maven 中，可以去本地的 Maven 库中查看。

```
apply plugin: 'groovy'
apply plugin: 'maven'

dependencies {
    compile gradleApi()
    compile localGroovy()
}

repositories {
    mavenCentral()
}
```

```
group = 'com.andoter.customplugin'
version = '1.0'
uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: uri('../repo'))
        }
    }
}
```

在上面的实现中，我们也可以把 group、version 字段配置在内部:

```
uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: uri('../repo'))
            pom.groupId = "com.andoter.customplugin"
            pom.artifactId = "groovydemo"
            pom.version = "1.0"
        }
    }
}
```

5. 应用 gradle 插件：在项目下的 build.gradle（也可以在 module 中）中的 repositories 模块中定义本地 maven 库地址。在 dependencies 模块中引入我们的插件的路径。

```
// 根目录 .gradle 文件配置插件的地址
buildscript {

    repositories {
        google()
        jcenter()
        mavenCentral()
        maven {
            url './repo'
        }
    }
    //格式为-->group:module:version
    dependencies {
        classpath 'com.android.tools.build:gradle:3.1.2'
        classpath 'com.andoter.customplugin:groovydemo:1.0'
    }
}

// 子项目使用插件
apply plugin: 'com.andoter.customplugin'
```

这样就完成一个自定义插件的使用步骤，自定义插件的核心开发一个什么样的插件，比如结合 Transform 开发一个编译时框架。

3. Android DSL

Android 提供对应的 android 插件用于项目的构建配置，在 Android 中项目的类型有以下四种：

- AppExtension：对应 com.android.application。
- LibraryExtension：对应 com.android.library。
- TestExtension：对应 com.android.test。
- FeatureExtension：对应 com.android.feature，及时应用。

3.1 Android 插件脚本块配置

Android 构建系统编译应用资源和源代码，然后将它们打包成可供测试、部署、签署和分发的 APK。Android Studio 使用 Gradle 这一高级构建工具包来自动化执行和管理构建流程，同时也允许您定义灵活的自定义构建配置。每个构建配置均可自行定义一组代码和资源，同时对所有应用版本共有的部分加以重复利用。Android Plugin for Gradle 与这个构建工具包协作，共同提供专用于构建和测试 Android 应用的流程和可配置设置。

3.1.1 aaptOptions { }

配置 Android 资源打包工具 AAPT 选项。

```
aaptOptions {
    additionalParameters '-S',
            '/Users/yifan/dev/github/Testapp/app/src/main/res3',
            '-S',
            '/Users/yifan/dev/github/Testapp/app/src/main/res2',
            '--auto-add-overlay'
    noCompress 'foo', 'bar'
    ignoreAssetsPattern '!.svn:!.git:!.ds_store:!*.scc:.*:<dir>_*:!CVS:!thumbs.db:!picasa.i
}
```

这个选项用于配置 AAPT 资源打包时的一些处理，比如资源替换，这块内容可参照编译时替换资源。

3.1.2 adbOptions { }

配置 Android 调试工具 ADB 选项。通常我们通过 adb 指令来进行 Apk 的安装或卸载，或者一些文件拷贝工作，通过 adbOptions {} 脚本块同样可以在 Android Plugin 中进行配置，adbOptions {} 脚本块对应 AdbOptions 类，该类有两个属性：

- installOptions：adb 配置选项，是 List 类型。
- timeOutInMs：是设置超时时间的，单位是毫秒，这个超时时间是执行adb这个命令的超时时间，int 类型。

```
android {
    adbOptions {
        timeOutInMs 10 * 1000
        installOptions '-r','-s'
    }
}
```

- -l：锁定该应用程序
- -r：替换已存在的应用程序，也就是我们说的强制安装
- -t：允许测试包
- -s：把应用程序安装到 SD 卡上
- -d：允许进行降级安装，也就是安装的比手机上带的版本低
- -g：为该应用授予所有运行时的权限

3.1.3 buildTypes { }

当前项目的构建类型配置，对应的配置类型 BuildType 类，在 Android Studio 的中已经给我们内置了 release 和 debug 两种构建类型，这两种模式的主要差异在于能否在设备上调试以及签名不一样，这里会涉及到很多属性可以配置。

- applicationIdSuffix：配置基于应用默认的 applicationId 的后缀，常用于构建变体应用。
- consumerProguardFiles：配置 .aar 文件中是否使用 Proguard 混淆。
- crunchPngs：针对 png 的优化，设置为 true 的时候会增加编译时间。
- debuggable：配置构建的 apk 是否能够进行 debug。
- javaCompileOptions：配置 Java 编译的配置
- jniDebuggable：配置构建类型的 apk native code 是否能够进行 debug。
- minifyEnabled：是否启用 Proguard 混淆。
- multiDexEnabled：是否使用分包。
- multiDexKeepFile：指定放到主 dex 中的文件。
- multiDexKeepProguard：配置指定的文件使用 Proguard。
- proguardFiles：混淆文件。
- shrinkResources：用于配置是否自动清理未使用的资源，默认为 false。
- signingConfig：签名文件。
- useProguard
- versionNameSuffix：类似于 applicationIdSuffix。
- zipAlignEnabled：zipAlign 优化 apk 文件的工具。

```
buildTypes {

    release {
        minifyEnabled false
        crunchPngs true
        debuggable false
        shrinkResources true
        multiDexEnabled true
        multiDexKeepProguard file('proguard-rules.pro') // keep specific classes using progu
```

```
        multiDexKeepFile file('multiDexKeep.txt')
        minifyEnabled true
        proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        signingConfig
        zipAlignEnabled true
        applicationIdSuffix '.release'
        versionNameSuffix '.release'
    }

    debug {
        applicationIdSuffix '.debug'
        versionNameSuffix '.debug'
    }
}
```

3.1.4 compileOptions { }

Java 编译选项，通常是针对 JDK 进行编码格式修改或者指定 JDK 的版本,对应的类是 CompileOptions。

```
compileOptions {
    encoding = 'utf-8'
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
}
```

3.1.5 dataBinding { }

DataBinding 配置选项，查看源码可以发现对应 DataBindingOptions 类，该类包含四个属性：

- version：版本号。
- enabled：是否可用。
- addDefaultAdapters：是否使用默认的 Adapter。
- enabledForTests：是否用于 Test。

```
dataBinding {
    enabled true
    version 1.0
}
```

3.1.6 defaultConfig { }

defaultConfig 也是 Android 插件中常见的一个配置块，负责默认的所有配置。同样它是一个 ProductFlavor，如果一个 ProductFlavor 没有被特殊配置，则默认使用 defaultFlavor 的配置，比如报名、版本号、版本名称等。常见的属性有：

- applicationId：applicationId 是 ProductFlavor 的一个属性，用于配置 App 生成的进程名，默认情况下是 null。
- minSdkVersion：指定 Apk 支持的最低 Android 操作系统版本。
- targetSdkVersion：用于配置 Apk 基于的 SDK 哪个版本进行开发。
- versionCode：同样是 ProductFlavor 的一个属性，配置 Apk 的内部版本号。
- versionName：配置版本名称。
- testApplicationId：配置测试 App 的报名，默认情况下是 applicationId + “.test”。
- testInstrumentationRunner：配置单元测试使用的 Runner，默认是 android.test.InstrumentationTestRunner,或者可以使用自定义的 Runner。
- signingConfig：配置默认的签名信息，对生成的 App 签名。
- proguardFile：用于配置使用的混淆文件。
- proguardFiles：配置混淆使用的文件，可以配置多个。

```
defaultConfig {
    applicationId 'com.andoter.dsw'
    minSdkVersion 15
    targetSdkVersion 28
    versionCode 1
    versionName "1.0"
    testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    signingConfigs signingConfigs.release
}
```

3.1.7 dexOptions { }

dex 的配置项，通常在开发的过程中，我们可以通过配置 dexOptions {} 提高编译速度，与之对应的是 DexOptions 接口，该接口由 DefaultDexOptions 默认实现，DefaultDexOptions 类中包含以下属性：

- preDexLibraries：默认 true
- jumboMode：默认 false
- dexInProcess：默认为 true，所有的 dex 都在 process 中，提高效率
- javaMaxHeapSize：最大的堆大小
- maxProcessCount：最大的 process 个数
- threadCount：线程个数

```
dexOptions {
    incremental true //是否增量，如果开启multi-dex，此句无效
    preDexLibraries true
    javaMaxHeapSize "4g" //java 编译的 Heap 大小
    jumboMode true
    threadCount 8 //gradle输就输在了并行上，都是串行，增加线程数没用
    // 设置最大的进程数，Memory = maxProcessCount * javaMaxHeapSize
```

```
    // 设置最大的进程数: Memory = maxProcessCount * javaMaxHeapSize
    maxProcessCount 8
}
```

3.1.8 lintOptions { }

Lint 是Android Studio 提供的 代码扫描分析工具，它可以帮助我们发现代码结构/质量问题，同时提供一些解决方案，而且这个过程不需要我们手写测试用例。Lint 发现的每个问题都有描述信息和等级（和测试发现 bug 很相似），我们可以很方便地定位问题同时按照严重程度进行解决。

```
android {
    lintOptions {
        // true--关闭lint报告的分析进度
        quiet true
        // true--错误发生后停止gradle构建
        abortOnError false
        // true--只报告error
        ignoreWarnings true
        // true--忽略有错误的文件的全/绝对路径(默认是true)
        //absolutePaths true
        // true--检查所有问题点，包含其他默认关闭项
        checkAllWarnings true
        // true--所有warning当做error
        warningsAsErrors true
        // 关闭指定问题检查
        disable 'TypographyFractions','TypographyQuotes'
        // 打开指定问题检查
        enable 'RtlHardcoded','RtlCompat', 'RtlEnabled'
        // 仅检查指定问题
        check 'NewApi', 'InlinedApi'
        // true--error输出文件不包含源码行号
        noLines true

        // true--显示错误的所有发生位置，不截取
        showAll true
        // 回退lint设置(默认规则)
        lintConfig file("default-lint.xml")
        // true--生成txt格式报告(默认false)
        textReport true
        // 重定向输出；可以是文件或'stdout'
        textOutput 'stdout'
        // true--生成XML格式报告
        xmlReport false
```

```
        // 指定xml报告文档(默认lint-results.xml)
        xmlOutput file("lint-report.xml")
        // true--生成HTML报告(带问题解释，源码位置，等)
        htmlReport true
        // html报告可选路径(构建器默认是lint-results.html )
        htmlOutput file("lint-report.html")
        //  true--所有正式版构建执行规则生成崩溃的lint检查，如果有崩溃问题将停止构建
        checkReleaseBuilds true
        // 在发布版本编译时检查(即使不包含lint目标)，指定问题的规则生成崩溃
        fatal 'NewApi', 'InlineApi'
        // 指定问题的规则生成错误
        error 'Wakelock', 'TextViewEdits'
        // 指定问题的规则生成警告
        warning 'ResourceAsColor'
        // 忽略指定问题的规则(同关闭检查)
        ignore 'TypographyQuotes'
    }
}
```

3.1.9 packagingOptions { }

Android 打包配置项，可以配置打包的时候哪些打包进 Apk。当项目中依赖的第三方库越来越多时，有可能会出现两个依赖库中存在同一个
（名称）文件。如果这样，Gradle在打包时就会提示错误（警告）。那么就可以根据提示，然后使用以下方法将重复的文件剔除。

```
packagingOptions {
    //这个是在同时使用butterknife、dagger2做的一个处理。同理，遇到类似的问题，只要根据gradle
    exclude 'META-INF/services/javax.annotation.processing.Processor'
}
```

3.1.10 productFlavors { }

用于构建不同的产品风味，在上面我们提到 defaultConfig{} 也是一种产品风味，可作为所有产品风味的“基类”共同部分。风味(Flavor) 对应 ProductFlavor 类，该类的属性与配置属性相匹配。

```
android {
    ...
    defaultConfig {...}
    buildTypes {...}
    productFlavors {
        demo {
```

```
            applicationIdSuffix ".demo"
            versionNameSuffix "-demo"
        }
        full {
            applicationIdSuffix ".full"
            versionNameSuffix "-full"
        }
    }
}
```

在创建和配置您的产品风味之后，在通知栏中点击 Sync Now。同步完成后，Gradle会根据您的构建类型和产品风味自动创建构建变体，
并按照 的格式命名这些变体。例如，如果您创建了“演示”和“完整”这两种产品风味并保留默认的“调试”和“发布”构建类型，Gradle 将创建以下构建变体：

- 演示调试
- 演示发布
- 完整调试
- 完整发布

您可以将构建变体更改为您要构建并运行的任何变体，只需转到 Build > Select Build Variant，然后从下拉菜单中选择一个变体。

产品风味组合

通常在适配多个渠道的时候，需要为特定的渠道做部分特殊的处理，这里就会涉及到结合 buildTypes{} 组合不同的产品风味组合。

```
android {
  ...
  buildTypes {
    debug {...}
    release {...}
  }

  // Specifies the flavor dimensions you want to use. The order in which you
  // list each dimension determines its priority, from highest to lowest,
  // when Gradle merges variant sources and configurations. You must assign
  // each product flavor you configure to one of the flavor dimensions.
  flavorDimensions "api", "mode"

  productFlavors {
    demo {
      // Assigns this product flavor to the "mode" flavor dimension.
      dimension "mode"
      ...
```

```
        ...
    }

    full {
      dimension "mode"
      ...
    }

    // Configurations in the "api" product flavors override those in "mode"
    // flavors and the defaultConfig {} block. Gradle determines the priority
    // between flavor dimensions based on the order in which they appear next
    // to the flavorDimensions property above--the first dimension has a higher
    // priority than the second, and so on.
    minApi24 {
      dimension "api"
      minSdkVersion '24'
      // To ensure the target device receives the version of the app with
      // the highest compatible API level, assign version codes in increasing

      // value with API level. To learn more about assigning version codes to
      // support app updates and uploading to Google Play, read Multiple APK Support
      versionCode 30000 + android.defaultConfig.versionCode
      versionNameSuffix "-minApi24"
      ...
    }

    minApi23 {
      dimension "api"
      minSdkVersion '23'
      versionCode 20000  + android.defaultConfig.versionCode
      versionNameSuffix "-minApi23"
      ...
    }

    minApi21 {
      dimension "api"
      minSdkVersion '21'
      versionCode 10000  + android.defaultConfig.versionCode
      versionNameSuffix "-minApi21"
      ...
    }
  }
}
```

Gradle 创建的构建变体数量等于每个风味维度中的风味数量与您配置的构建类型数量的乘积。在 Gradle 为每个构建变体或对应 APK 命名时，属于较高优先级风味维度的产品风味首先显示，之后是较低优先级维度的产品风味，再之后是构建类型。以上面的构建配置为例，Gradle 可以使用以下命名方案创建总共 12 个构建变体：

```
构建变体：[minApi24, minApi23, minApi21][Demo, Full][Debug, Release]
对应 APK：app-[minApi24, minApi23, minApi21]-[demo, full]-[debug, release].apk
```

同样我们在 Terminal 中通过 gradle 指令执行构建变体的任务：**变体名称(flavorDimensions 的第一个维度的 Flavor) + flavorDimensions 的别的维度的 Flavor + buildTypes**，例如：minApi24DemoDebug，构建出来的对应 Apk：app-minApi24-demo-debug.apk

过滤变体

Gradle 会自动根据设置的 buildTypes{} 和 flavorDimensions{} 创建很多变体的组合体，但是有些是我们不需要的，这里就涉及到过滤变体的操作。Gradle 提供了 variantFilter {} 脚本块来过滤，脚本块对应 VariantFilter 接口，接口中包含：

- setIgnore(boolean ignore)：设置是否忽略某个变体
- getBuildType：获取变体的构建类型
- List getFlavors()：返回所有变体的列表
- getName：获取变体的 name 名称


```
variantFilter { variant ->
    def names = variant.flavors.name
    // To check for a certain build type, use variant.buildType.name == "<buildType>"
    if (names.contains("minApi23") && names.contains("demo")) {
        // Gradle ignores any variants that satisfy the conditions above.
        setIgnore(true)
    } else {
        setIgnore(false)
    }
}
```

更多关于构建变体内容参照官方：构建变体

3.1.11 signingConfigs { }

配置签名信息，常用于 BuildType 和 ProductFlavor 配置，在构建变体的过程中，会出现很多种类，所以针对不同类别的变体所使用的签名可能也是不同的，这就需要使用 signingConfigs{} 配置签名信息合集，然后按需所取。

```
signingConfigs {
    release {//发布版本的签名配置
      storeFile file(props['KEYSTORE_FILE'])
      keyAlias props['KEY_ALIAS']
      storePassword props['KEYSTORE_PWD']
      keyPassword props['KEY_PWD']
    }
    debug {//调试版本的签名配置
      storeFile file(props['DEBUG_KEYSTORE'])
      keyAlias props['DEBUG_ALIAS']
      storePassword props['DEBUG_KEYSTORE_PWD']
      keyPassword props['DEBUG_KEY_PWD']
    }
}

buildTypes {
    signingConfig signingConfigs.release
}
```

3.1.12 sourceSets { }

在 AndroidStudio 中，在 src/main/java 目录下创建我们的 .java 文件，这些都是系统通过 sourceSet{} 设置好的，比如我们在外部创建一个文件夹，选中文件夹右键就无创建 .java 文件的选项。就需要我们通过 sourceSets 进行配置，脚本块对应 AndroidSourceSet 接口，接口中有：

- AndroidSourceSet java(Closure configureClosure)：配置 java 文件存放路径
- AndroidSourceSet resources(Closure configureClosure)：配置 resource 目录
- AndroidSourceSet jniLibs(Closure configureClosure)：配置 jniLibs 目录
- AndroidSourceSet jni(Closure configureClosure)：配置 jni 文件目录
- AndroidSourceSet renderscript(Closure configureClosure)：配置 renderscript 目录
- AndroidSourceSet aidl(Closure configureClosure)：配置 aidl 文件目录
- AndroidSourceSet assets(Closure configureClosure)：配置 assets 目录
- AndroidSourceSet res(Closure configureClosure)：配置 res 目录
- AndroidSourceSet manifest(Closure configureClosure)：配置 manifest 目录

```
sourceSets {
    main {
        java {
            srcDir 'src/main/testjava'
        }
        resources {
            srcDir 'src/resources'
        }
```

```
    }
}
```

3.1.13 splits { }

拆分机制比起使用 flavors，能让应用程序更有效地构建一些形式的多个apk。
多 apk 只支持以下类型:

- 屏幕密度
- ABI

脚本块对应 Splits 类，该类中有三个属性：

- density：像素密度
- abi：abi 类型
- language：语言

```
splits {
    density {
        enable true
        exclude "ldpi", "tvdpi", "xxxhdpi"
        compatibleScreens 'small', 'normal', 'large', 'xlarge'
    }

    abi {
        enable true
        reset()
        include 'x86', 'armeabi-v7a', 'mips'
        universalApk true
    }
}
```


4. 发现好资料

Gradle 的功能非常强大，这里也仅仅针对 Android 项目构建常用的进行总结，Gradle 还可用于单元测试，或者用于别的项目构建，上传构建很多部分。具体可以参照官方 API 文档。

- Gradle Guides 官方文档
- Android DSL 官方文档
- Gradle学习笔记
- 迁移到 Android Plugin for Gradle 3.0.0