Gradle简介

Gradle 是一个基于Ant和Maven概念的项目自动化建构工具。它使用一种基于Groovy的特定领域语言(DSL)来声明项目设置，这比我们的ANT使用XML构建配置要灵活的多。在编写配置时，你可以像编程一样灵活，Gradle是基于Groovy的DSL语言，完全兼容JAVA

Gradle入门

projects 和 tasks是Gradle中最重要的两个概念，任何一个Gradle构建都是由一个或者多个project组成，每个project可以是一个jar包，一个web应用，或者一个android app等，每个project又由多个task构成，一个task其实就是构建过程中一个原子性的操作，比如编译、拷贝等。

一个build.gradle文件是一个构建脚本，当运行gradle命令的时候会从当前目录查找build.gradle文件来执行构建。下面我们来看下gradle的Hello World。在build.gradle构建文件中输入以下构建脚本：

1

2

3

4

5

task hello {

doLast {

println 'Hello world!'

}

}

task定义了一个任务，这个任务名字是hello。doLast是Task的方法，意思是在该hello任务执行之后作的事情，可以用一个闭包配置它,这里是输出Hello world!字符串。我们在终端里执行如下命令运行查看结果：

1

2

$gradle hello -q

Hello world!

其他关于Gradle的更多介绍请参考Gradle使用指南

Android Studio入门

使用Android Studio新建一个工程之后，其目录结构是这样的：

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

├── 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

settings.gradle用于配置project，标明其下有几个module，比如这里包含一个:app module

1

include ':app'

和settings.gradle在同一目录下的build.gradle是一个顶级的build配置文件，在这里可以为所有project以及module配置一些常用的配置。

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

// Top-level build file where you can add configuration options common to all sub-projects/modules.

buildscript {

repositories {

jcenter()//使用jcenter库

}

dependencies {

// 依赖android提供的1.1.0的gradle build

classpath 'com.android.tools.build:gradle:1.1.0'

// NOTE: Do not place your application dependencies here; they belong

// in the individual module build.gradle files

}

}

//为所有的工程的repositories配置为jcenters

allprojects {

repositories {

jcenter()

}

}

Android Gradle基本配置

下面着重说一下Android的Gradle，毕竟对Android开发来说，这才是重中之重。这里以初始化好的build.gradle为例。

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

apply plugin: 'com.android.application'

android {

compileSdkVersion 21

buildToolsVersion "22.0.1"

defaultConfig {

applicationId "org.flysnow.demo"

minSdkVersion 9

targetSdkVersion 21

versionCode 1

versionName "1.0"

}

buildTypes {

release {

minifyEnabled false

proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'

}

}

}

dependencies {

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

compile 'com.android.support:appcompat-v7:22.0.0'

}

开头第一行apply plugin: ‘com.android.application’，这表示该module是一个app module，应用了com.android.application插件，如果是一个android library，那么这里的是apply plugin: ‘com.android.library’。

其次是基于哪个SDK编译，这里是API LEVEL，是21，buildToolsVersion是基于哪个构建工具版本进行构建的。defaultConfig是默认配置，如果没有其他的配置覆盖，就会使用这里的。看其属性的名字就可以知道其作用，比如applicationId是配置包名的，versionCode是版本号，versionName是版本名称等。

buildTypes是构建类型，常用的有release和debug两种，可以在这里面启用混淆，启用zipAlign以及配置签名信息等。

dependencies就不属于Android专有的配置了，它定义了该module需要依赖的jar，aar，jcenter库信息。

配置应用的签名信息

在android.signingConfigs{}下定义一个或者多个签名信息，然后在buildTypes{}配置使用即可。比如这里

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

android {

signingConfigs {

release {

storeFile file("release.keystore")

keyAlias "release"

keyPassword "123456"

storePassword "123456"

}

debug {

...

}

}

buildTypes {

release {

signingConfig signingConfigs.release

}

debug {

signingConfig signingConfigs.debug

}

}

}

storeFile是签名证书文件，keyAlias是别名，keyPassword是key的密码，storePassword是证书的密码。配好好相关信息即可在buildTypes配置使用。

启用proguard混淆

我们可以为不同的buildTypes选择是否启用混淆，一般release发布版本是需要启用混淆的，这样别人反编译之后就很难分析你的代码，而我们自己开发调试的时候是不需要混淆的，所以debug不启用混淆。对release启用混淆的配置如下：

1

2

3

4

5

6

7

8

9

android {

buildTypes {

release {

minifyEnabled true

proguardFile 'proguard.cfg'

}

}

}

minifyEnabled为true表示启用混淆，proguardFile是混淆使用的配置文件，这里是module根目录下的proguard.cfg文件

启用zipAlign

这个也是比较简单的，同样也是在buildTypes里配置，可以为不用的buildTypes选择时候开启zipAlign

1

2

3

4

5

6

7

8

android {

buildTypes {

release {

zipAlignEnabled true

}

}

}

多渠道打包

东西到了国内就变了，做什么都是一窝蜂，比如Android App市场就是，所以才有了多渠道打包，每次发版几十个渠道包。还好Android Gradle给我们提供了productFlavors，让我们可以对生成的APK包进行定制，所以就有了多渠道。

1

2

3

4

5

6

7

8

9

10

11

12

13

android {

productFlavors {

dev{

}

google{

}

baidu{

}

}

}

这样当我们运行assembleRelease的时候就会生成3个release包，分别是dev、google以及baidu的。目前看这三个包除了文件名没有什么不一样，因为我们还没有定制，使用的都是defaultConfig配置。这里的flavor和defaultConfig是一样的，可以自定义其applicationId、versionCode以及versionName等信息，比如区分不同包名：

1

2

3

4

5

6

7

8

9

10

11

12

13

android {

productFlavors {

dev{

applicationId "org.flysnow.demo.dev"

}

google{

applicationId "org.flysnow.demo.google"

}

baidu{

applicationId "org.flysnow.demo.baidu"

}

}

}

批量修改生成的apk文件名

在我们打包发版的时候，一次性打几十个包，这时候我们就想让生成的apk文件名有区分，比如一眼就能看出这个apk是哪个版本的，哪个渠道的，是哪天打的包等等，这就需要我们在生成apk文件的时候动态修改生成的apk文件名达到这一目的。这里以我们的产品随手记为例：

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

def buildTime() {

def date = new Date()

def formattedDate = date.format('yyyyMMdd')

return formattedDate

}

android {

buildTypes {

release {

applicationVariants.all { variant ->

variant.outputs.each { output ->

if (output.outputFile != null && output.outputFile.name.endsWith('.apk')

&&'release'.equals(variant.buildType.name)) {

def apkFile = new File(

output.outputFile.getParent(),

"Mymoney\_${variant.flavorName}\_v${variant.versionName}\_${buildTime()}.apk")

output.outputFile = apkFile

}

}

}

}

}

}

以baidu渠道为例，以上的代码会生成一个名字为Mymoney\_baidu\_v9.5.2.6\_20150330.apk安装包。下面我们分析一下，Android Gradle任务比较复杂，它的很多任务都是自动生成的，为了可以更灵活的控制，Android Gradle提供了applicationVariants、libraryVariants以及testVariants，他们分别适用于app、library、app和library都适用。

这里是循环处理每个applicationVariant，当他们的输出文件名以apk结尾并且buildType是release时，重新设置新的输出文件名，这样就达到了我们批量修改生成的文件名的目的。

AndroidManifest里的占位符

AndroidManifest.xml这是一个很重要的文件，我们的很多配置都在这里定义。有时候我们的一些配置信息，比如一个第三方应用的key，第三方统计分析的渠道号等也要在这里进行配置。这里以友盟统计分析平台为例，演示这一功能的使用。在友盟统计分析中，我们需要根据渠道进行统计，比如google，百度，应用宝等渠道的活跃新增等，友盟的SDK是在AndroidManifest里配置一个name为UMENG\_CHANNEL的meta-data，这样这个meta-data的值就表示这个apk是哪个渠道，我们版本发布有几十个渠道，以前ant打包的时候是采用文字替换的办法，现在Gradle有更好的处理办法，那就是manifestPlaceholders，它允许我们动态替换我们在AndroidManifest文件里定义的占位符。

1

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

如上${UMENG\_CHANNEL\_VALUE}就是一个占位符，然后我们在gradle的defaultConfig；里这样定义脚本：

1

2

3

4

5

android {

defaultConfig {

manifestPlaceholders = [UMENG\_CHANNEL\_VALUE: 'dev']

}

}

以前的意思就是我们的默认配置里AndroidManifest的${UMENG\_CHANNEL\_VALUE}占位符会被dev这个字符串所替换，也就说默认运行的版本是一个开发板。以此类推，我们其他渠道的版本就可以这样定义：

1

2

3

4

5

6

7

8

9

10

11

12

android {

productFlavors {

google{

applicationId "org.flysnow.demo.google"

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

}

baidu{

applicationId "org.flysnow.demo.baidu"

manifestPlaceholders.put("UMENG\_CHANNEL\_VALUE",'baidu')

}

}

}

这样有多少个渠道就做多少次这样的定义，即可完成分渠道统计。但是如果上百个渠道，这样一个个写的确太累，很麻烦，我们继续研究，同学们有没有发现，我们的渠道名字和我们的flavorName一样，我们用这个flavorName作为UMENG\_CHANNEL\_VALUE不就好了吗，可以批量的替换吗？当然可以，这又体现了我们Gradle的强大和灵活之处。

1

2

3

productFlavors.all { flavor ->

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

}

循环每个flavor，并把他们的UMENG\_CHANNEL\_VALUE设置为他们自己的name名字，ok，搞定。

自定义你的BuildConfig

BuildConfig.java是Android Gradle自动生成的一个java类文件，无法手动编译，但是可以通过Gradle控制，也就是说他是动态可配置的，有了这个功能就很好玩了，这里以生产环境和测试环境为例来说明该功能的使用。

我们在开发App的时候免不了要和服务器进行通信，我们的服务器一般都有生产和测试环境，当我们处理开发和测试的时候使用测试环境进行调试，正式发布的时候使用生成环境。以前的时候我们通过把不同的配置文件打包进APK中来控制，现在不一样了，我们有更简便的方法，这就是buildConfigField。

1

2

3

4

5

6

7

8

9

10

11

12

13

android {

defaultConfig {

buildConfigField 'String','API\_SERVER\_URL','"http://test.flysnow.org/"'

}

productFlavors {

google{

buildConfigField 'String','API\_SERVER\_URL','"http://www.flysnow.org/"'

}

baidu{

buildConfigField 'String','API\_SERVER\_URL','"http://www.flysnow.org/"'

}

}

}

buildConfigField 一共有3个参数，第一个是数据类型，就是你定义的常量值是一个什么类型，和Java的类型是对等的，这里是String。第二个参数是常量名，这里是API\_SERVER\_URL。第三个参数是常量值。如此定义之后，就会在BuildConfig.java中生成一个常量名为API\_SERVER\_URL的常量定义。默认配置的生成是：

1

public final static String API\_SERVER\_URL = "http://test.flysnow.org/"

当是baidu和google渠道的时候生成的就是http://www.flysnow.org/了。这个常量可以在我们编码中引用。在我们进行打包的时候会根据Gradle配置动态替换。

我们发现一般渠道版本都是用来发布的，肯定用的是生产服务器，所以我们可以使用批处理来搞定这个事情，而不用在一个个渠道里写这些配置。

1

2

3

productFlavors.all { flavor ->

buildConfigField 'String','API\_SERVER\_URL','"http://www.flysnow.org/"'

}

此外，比如Gradle的resValue，也是和buildConfigField，只不过它控制生成的是资源，比如我们在android的values.xml定义生成的字符串。可以用它来动态生成我们想要的字符串，比如应用的名字，可能一些渠道会不一样，这样就可以很灵活的控制自动生成，关于resValue详细介绍请参考相关文档，这里不再举例说明。

插装测试覆盖率代码

代码覆盖率现在已经成为检验单元测试是否覆盖到的一种手段，Android Gradle提供了原生的用于单元测试的代码覆盖率，这个就是jacoco。今天我们不谈这个，我想要的是在我们生成的APK包中已经包含了检测代码覆盖率的代码，这样当我们安装APK后运行进行一些测试的时候，这些检测代码覆盖率的代码就会被执行到，这样最后我们导出一份代码测试覆盖率的文件，然后生成查看测试覆盖率报告看哪些覆盖到，哪些没有覆盖到。这种场景在检测测试工程师测试功能以及Android UI自动化测试是否完全覆盖尤为有效。这里代码覆盖率框架我选择的是emma，一来这个在Ant打包的时候一直在用，二来它具有很方便的插装功能。

emma插装的是class文件，所以我们只能在编译完java文件生成class文件后进行插装，这是我们进行覆盖率代码插装的最好时机。找到了时机，那么具体对应在Gradle脚本上是哪呢？还记不记得我们上面讲的applicationVariants，每一个applicationVariant都有一个javaCompile属性，javaCompile是一个JavaCompile类型的Task，这个就是负责编译java代码的。是Task就有doLast方法，就是在这个任务本身完成之后要做的事情，我们就是在这个方法里进行我们的代码覆盖率的安装。一般我们这个插装只是在特性情况下，那么我们新增一个特殊的flavor好了，专门做这个使用，这里我姑且叫feature。

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

applicationVariants.all { variant ->

//为feature 版本加上代码覆盖率

if('feature'.equals(variant.flavorName)){

variant.javaCompile.doLast {

def coverageFile=file('out/coverage.em')

if(coverageFile.exists()){

coverageFile.delete()

}

javaexec {

main 'emma'

args 'instr','-ip',variant.javaCompile.destinationDir,'-m','overwrite','-out','out/coverage.em'

classpath files(new File(getSdkDirectory(),'tools/lib/emma.jar'))

}

}

}

}

非常简单，我们使用javaexec命令执行java应用程序进程插装，插装模式使用的是overwrite，就是插装后覆盖源文件。getSdkDirectory()函数获取你电脑上的Android SDK目录，这里我们使用SDK自带的emma，保持每个人的统一。另外注意进行代码覆盖率插装的APK不能进行代码混淆，这个很简单，为feature flavor指定不混淆的proguardFile覆盖默认的proguardFile即可。最后该APK需要emma的框架代码，所以要配置feature flavor的特殊依赖信息。

1

2

3

4

dependencies {

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

featureCompile files(new File(android.getSdkDirectory(),'tools/lib/emma\_device.jar'))

}

dexOptions javaMaxHeapSize

在Gradle 进行dex的可能会遇到内存不够用的情况，错误信息大概是java.lang.OutOfMemoryError: GC overhead limit exceeded。这个时候只需要配置dexOptions的javaMaxHeapSize大小即可，我这里配置4g：

1

2

3

dexOptions {

javaMaxHeapSize "4g"

}