**Gradle For Android**

Abstruct

The build process for an Android app is an incredibly complex procedure that involves many tools. To begin with, all the resource files are compiled and referenced in a R.java file, before the Java code is compiled and then converted to Dalvik bytecode by the dex tool. These files are then packaged into an APK file, and that APK file is signed with a debug or release key, before the app can finally be installed onto a device. Going through all these steps manually would be a tedious and time-consuming undertaking. Luckily, the Android Tools team has continued to provide developers with tools that take care of the entire process, and in 2013, they introduced Gradle as the new preferred build system for Android apps. Gradle is designed in a way that makes it easy to extend builds and plug into the existing build processes. It provides a Groovy-like DSL to declare builds and create tasks, and makes dependency management easy. Additionally, it is completely free and open source. By now, most Android developers have switched to Gradle, but many do not know how to make the best of it, and are unaware of what can be achieved with just a few lines of code. This book aims to help those developers, and turn them into Gradle power users. Starting with the basics of Gradle in an Android context, this book goes on to cover dependencies, build variants, testing, creating tasks, and more.

目录

[1.Getting Started with Gradle and Android Studio 4](#_Toc511051558)

[1.1 Android Studio 4](#_Toc511051559)

[1.2 Understanding Gradle basics 4](#_Toc511051560)

[1.3 Projects and tasks 5](#_Toc511051561)

[1.4 The build lifecycle 5](#_Toc511051562)

[1.5 Getting started with the Gradle Wrapper 6](#_Toc511051563)

[2. Basic Build Customization 6](#_Toc511051564)

[2.1 Understanding the Gradle files 6](#_Toc511051565)

[2.2 The settings file 7](#_Toc511051566)

[2.3 The top-level build file 7](#_Toc511051567)

[2.4 The module build file 8](#_Toc511051568)

[2.5 Plugin 8](#_Toc511051569)

[2.6 Android 9](#_Toc511051570)

[3. Managing Dependencies 10](#_Toc511051571)

[3.1 Repositories 10](#_Toc511051572)

[3.2 Preconfigured repositories 11](#_Toc511051573)

[3.3 Remote repositories 12](#_Toc511051574)

[3.4 Local repositories 12](#_Toc511051575)

[3.5 File dependencies 13](#_Toc511051576)

[3.6 Library projects 13](#_Toc511051577)

[3.7 Creating and using library project modules 13](#_Toc511051578)

[3.8 Using .aar files 14](#_Toc511051579)

[4. Creating Build Variants 14](#_Toc511051580)

1.Getting Started with Gradle and Android Studio

When Google introduced Gradle and Android Studio, they had some goals in mind. They wanted to make it easier to reuse code, create build variants, and configure and customize the build process. On top of that, they wanted good IDE integration, but without making the build system dependent on the IDE. Running Gradle from the command line or on a continuous integration server will always yield the same results as running a build from Android Studio.

1.1 Android Studio

Android Studio was announced and released (as an early access preview) by Google in May 2013, alongside support for Gradle. Android Studio is based on JetBrains' IntelliJ IDEA, but is designed specifically with Android development in mind. It is available for free for Linux, Mac OS X, and Microsoft Windows.

Compared to Eclipse, Android Studio has an improved user interface designer, a better memory monitor, a nice editor for string translation, warnings for possible Android-specific issues and a lot more features aimed at Android developers. It also features a special project structure view for Android projects, besides the regular Project view and Packages view that exist in IntelliJ IDEA. This special view groups Gradle scripts, drawables, and other resources in a convenient way. As soon as the stable version 1.0 of Android Studio was released, Google retired the Android Developer Tools (ADT) for Eclipse and recommended all developers to switch to Android Studio. This means that Google will not provide new features for Eclipse anymore, and all IDE-related tool development is now focused on Android Studio. If you are still using Eclipse, it is time to change if you do not want to be left behind.

1.2 Understanding Gradle basics

In order for an Android project to be built using Gradle, you need to set up a build script. This will always be called build.gradle, by convention. You will notice, as we go through the basics, that Gradle favors convention over configuration and generally provides default values for settings and properties. This makes it a lot easier to get started with a lot less configuration than that found in systems such as Ant or Maven, which have been the de facto build systems for Android projects for a long time. You do not need to absolutely comply with these conventions though, as it is usually possible to override them if needed.

Gradle build scripts are not written in the traditional XML, but in a domain-specific language (DSL) based on Groovy, a dynamic language for the Java Virtual Machine (JVM). The team behind Gradle believes that using a declarative, DSL-style approach based on a dynamic language has significant advantages over using the more procedural, free-floating style of Ant, or any XML-based approach used by many other build systems.

That does not mean you need to know Groovy to get started with your build scripts. It is easy to read, and if you already know Java, the learning curve is not that steep. If you want to start creating your own tasks and plugins (which we will talk about in later chapters), it is useful to have a deeper understanding of Groovy. However, because it is based on the JVM, it is possible to write code for your custom plugins in Java or any other JVM-based language.

1.3 Projects and tasks

The two most important concepts in Gradle are projects and tasks. Every build is made up of at least one project, and every project contains one or more tasks. Every build.gradle file represents a project. Tasks are then simply defined inside the build script. When initializing the build process, Gradle assembles Project and Task objects based on the build file. A Task object consists of a list of Action objects, in the order they need to be executed. An Action object is a block of code that is executed, similar to a method in Java

1.4 The build lifecycle

Executing a Gradle build is, in its simplest form, just executing actions on tasks, which are dependent on other tasks. To simplify the build process, the build tools create a dynamic model of the workflow as a Directed Acyclic Graph (DAG). This means all the tasks are processed one after the other and loops are not possible. Once a task has been executed, it will not be called again. Tasks without dependencies will always be run before the others. The dependency graph is generated during the configuration phase of a build. A Gradle build has three phases:

• Initialization: This is where the Project instance is created. If there are multiple modules, each with their own build.gradle file, multiple projects will be created.

• Configuration: In this phase, the build scripts are executed, creating and configuring all the tasks for every project object.

• Execution: This is the phase where Gradle determines which tasks should be executed. Which tasks should be executed depends on the arguments passed for starting the build and what the current directory is.

1.5 Getting started with the Gradle Wrapper

Gradle is a tool that is under constant development, and new versions could potentially break backward compatibility. Using the Gradle Wrapper is a good way to avoid issues and to make sure builds are reproducible.

The Gradle Wrapper provides a batch file on Microsoft Windows and a shell script on other operating systems. When you run the script, the required version of Gradle is downloaded (if it is not present yet) and used automatically for the build. The idea behind this is that every developer or automated system that needs to build the app can just run the wrapper, which will then take care of the rest. This way, it is not required to manually install the correct version of Gradle on a developer machine or build server. Therefore, it is also recommended to add the wrapper files to your version control system.

Running the Gradle Wrapper is not that different from running Gradle directly. You just execute gradlew on Linux and Mac OS X and gradlew.bat on Microsoft Windows, instead of the regular gradle command.

2. Basic Build Customization

2.1 Understanding the Gradle files

When creating a new project with Android Studio, three Gradle files are generated by default. Two of those files, settings.gradle and build.gradle, end up on the top level of the project. Another build.gradle file is created in the Android app module. This is how the Gradle files are placed in the project:

MyApp

├── build.gradle

├── settings.gradle

└── app

└── build.gradle

These three files each serve their own purpose, which we will further look into in the upcoming sections.

2.2 The settings file

For a new project containing only an Android app, settings.gradle looks like this:

include ':app'

The settings file is executed during the initialization phase, and defines which modules should be included in the build. In this example, the app module is included. Single module projects do not necessarily require a settings file, but multimodule projects do; otherwise, Gradle does not know which modules to include.

Behind the scenes, Gradle creates a Settings object for every settings file, and invokes the necessary methods from that object. You do not need to know the details of the Settings class, but it is good to be aware of this.

2.3 The top-level build file

The top-level build.gradle file is where you can configure options that need to be applied to all the modules in the project. It contains two blocks by default:

buildscript { repositories { jcenter() } dependencies { classpath 'com.android.tools.build:gradle:1.2.3' } }

allprojects { repositories { jcenter() } }

The buildscript block is where the actual build is configured. We looked at this briefly in Chapter 1, Getting Started with Gradle and Android Studio. The repositories block configures JCenter as a repository. In this case, a repository means a source of dependencies or, in other words, a list of downloadable libraries that we can use in our apps and libraries. JCenter is a well-known Maven repository. The dependencies block is used to configure dependencies for the build process itself. This means that you should not include dependencies that you need for your applications or libraries in the top-level build file. The only dependency that is defined by default is the Android plugin for Gradle. This is required for every Android module, because it is this plugin that makes it possible to execute Android-related tasks. The allprojects block can be used to define properties that need to be applied to all modules. You can take it even further and create tasks in the allprojects block. Those tasks will then be available in all modules.

2.4 The module build file

The module-level build.gradle file contains options that only apply to the Android app module. It can also override any options from the top-level build.gradle file. The module build file looks like this:

apply plugin: 'com.android.application'

android { compileSdkVersion 22 buildToolsVersion "22.0.1"

defaultConfig { applicationId "com.gradleforandroid.gettingstarted" minSdkVersion 14 targetSdkVersion 22 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-v7:22.2.0' }

2.5 Plugin

The first line applies the Android application plugin, which is configured as a dependency in the top-level build file, which we discussed earlier. The Android plugin is written and maintained by the Android Tools team at Google, and provides all tasks needed to build, test, and package Android applications and libraries.

2.6 Android

The biggest part of the build file is the android block. This block contains the entire Android-specific configuration, which is available through the Android plugin we applied earlier.

The only properties that are required are compileSdkVersion and buildToolsVersion:

• The first one, compileSdkVersion, is the API version of Android that you want to use to compile your app

• The second one, buildToolsVersion, is the version of build tools and compilers to use

The build tools contain command-line utilities, such as aapt, zipalign, dx, and renderscript; which are used to produce the various intermediate artifacts that make up your application. You can download the build tools through the SDK Manager.

The defaultConfig block configures core properties for the app. The properties in this block override the corresponding entries in the AndroidManifest.xml manifest file:

defaultConfig { applicationId "com.gradleforandroid.gettingstarted" minSdkVersion 14 targetSdkVersion 22 versionCode 1 versionName "1.0" }

The first property in this block is applicationId. This overrides the package name from the manifest file, but there are some differences between applicationId and the package name. Before Gradle was used as the default Android build system, the package name in AndroidManifest.xml had two purposes: it served as the unique identifier of an app, and it was used as the name for the package in the R resource class. Gradle makes it easier to create different versions of your app, using build variants. For example, it is very easy to make a free version and a paid version. These two versions need to have separate identifiers, so they appear as different apps on the Google Play Store, and can both be installed at the same time. The source code and generated R class, however, must retain the same package name at all times. Otherwise, all your source files would need to change, depending on the version you are building. That is why the Android Tools team has decoupled these two different usages of package name. The package, as defined in the manifest file, continues to be used in your source code and your R class, while the package name that is used by the device and Google Play as the unique identifier is now referred to as application id. This application ID will become a lot more interesting as we start experimenting with build types.

The next two properties in defaultConfig are minSdkVersion and targetSdkVersion. Both of these should look familiar because they have always been defined in the manifest as part of the <uses-sdk> element. The minSdkVersion setting is used to configure the minimum API level required to run the app. The targetSdkVersion setting informs the system that the app is tested on a specific version of Android, and that the operating system does not need to enable any forward-compatibility behavior. This has nothing to do with compileSdkVersion that we saw earlier. The versionCode and versionName also have the same function as in the manifest file, and define a version number and a user-friendly version name for your app.

All values in the build file will override the values in the manifest file. It is therefore not required to define them in the manifest file if you define them in build.gradle. In case the build file does not contain a value, the manifest values will be used as a fallback. The buildTypes block is where you define how to build and package the different build types of your app.

3. Managing Dependencies

Dependency management is one of the areas where Gradle really shines. In the best case scenario, all you need to do is add one line to your build file, and Gradle will download the dependency from a remote repository and make sure its classes are available to your project. Gradle even goes a step further. In case a dependency for your project has dependencies of its own, Gradle will resolve those, and take care of everything. These dependencies of dependencies are called transitive dependencies.

3.1 Repositories

When we discuss dependencies, we usually mean external dependencies, such as libraries that are provided by other developers. Manually managing dependencies can be a big hassle. You have to locate the library, download the JAR file, copy it into your project, and reference it. Often these JAR files have no version in their name, so you need to remember to add it yourself, in order to know when to update. You also need to make sure the libraries are stored in a source control system, so that the team members can work with the code base without manually downloading the dependencies themselves.

Using repositories can solve these issues. A repository can be seen as a collection of files. Gradle does not define any repositories for your project by default, so you need to add them to the repositories block. If you use Android Studio, this is done for you. We have mentioned the repositories block briefly in the previous chapters; it looks like this:

repositories { jcenter() }

Gradle supports three different kinds of repositories: Maven, Ivy, and static files or directories. Dependencies are fetched from the repositories during the execution phase of the build. Gradle also keeps a local cache, so a particular version of a dependency only needs to be downloaded to your machine once. A dependency is identified by three elements: group, name, and version. The group specifies the organization that created the library and is usually a reverse domain name. The name is a unique identifier for the library. The version specifies which version of the library you want to use. Using these three elements, a dependency can be declared in the dependencies block with the following structure:

dependencies { compile 'com.google.code.gson:gson:2.3' compile 'com.squareup.retrofit:retrofit:1.9.0' }

This is shorthand for the full Groovy map notation, which looks like this:

dependencies { compile group: 'com.google.code.gson', name: 'gson', version: '2.3' compile group: 'com.squareup.retrofit', name: 'retrofit' version: '1.9.0' }

3.2 Preconfigured repositories

For your convenience, Gradle has preconfigured three Maven repositories: JCenter, Maven Central, and the local Maven repository. To include them in your build script, you need to include these lines:

repositories { mavenCentral() jcenter() mavenLocal() }

Maven Central and JCenter are two well-known online repositories. There is no reason to use both of them at the same time, and it is always recommended to use JCenter, which is also the default repository in Android projects created with Android Studio. JCenter is a superset of Maven Central, so when you make the switch, you can leave your already defined dependencies intact. On top of that, it supports HTTPS, unlike Maven Central.

The local Maven repository is a local cache of all the dependencies you have used, and you can add your own dependencies as well. By default, the repository can be found in the home directory in a folder called .m2. On Linux or Mac OS X, the path is ~/.m2. On Microsoft Windows, it is %UserProfile%\.m2.

Besides these preconfigured repositories, you can also add other public, or even private repositories.

3.3 Remote repositories

Some organizations create interesting plugins or libraries, and prefer to host them on their own Maven or Ivy server, instead of publishing them to Maven Central or JCenter. To add those repositories to your build, all you need to do is to add the URL to a maven block.

repositories { maven { url "http://repo.acmecorp.com/maven2" } }

The same goes for Ivy repositories. Apache Ivy is a dependency manager that is popular in the Ant world. Gradle supports these repositories in a format that is identical to the one that is used for Maven repositories. Add the repository URL to an ivy block, and you are good to go:

repositories { ivy { url "http://repo.acmecorp.com/repo" } }

If your organization is running its own repository, chances are that it is secured, and you need credentials to access it. This is how you add credentials for a repository:

repositories { maven { url "http://repo.acmecorp.com/maven2" credentials { username 'user' password 'secretpassword' } } }

The approach for Maven and Ivy is the same here as well. You can add a credentials block with the same format to the configuration of your Ivy repository.

3.4 Local repositories

In some cases, you might still need to manually download a JAR file or a native library. Perhaps you want to create your own library that you can reuse in several projects, without publishing it to a public or private repository. In those cases, it is impossible to use any of the online resources, and you will have to use different ways to add the dependencies. We will describe how to use file dependencies, how to include native libraries, and how you can include library projects in your project.

3.5 File dependencies

To add a JAR file as a dependency, you can use the files method that Gradle provides. This is what it looks like:

dependencies { compile files('libs/domoarigato.jar') }

This can get tedious if you have a lot of JAR files, so it might be easier to add an entire folder at once:

dependencies { compile fileTree('libs') }

By default, a newly created Android project will have a libs folder, and declare it to be used for dependencies. Instead of simply depending on all files in the folder, there is a filter that makes sure that only JAR files are used:

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

This means that in any Android project that is created in Android Studio, you can drop a JAR in the libs folder, and it will automatically be included in the compile classpath and the final APK.

3.6 Library projects

If you want to share a library that uses Android APIs, or includes Android resources, you need to create a library project. Library projects generally behave the same as application projects. You can use the same tasks to build and test library projects, and they can have different build variants. The difference is in the output. Where an application project generates an APK that can be installed and run on an Android device, a library project generates a .aar file. This file can be used as a library for Android application projects.

3.7 Creating and using library project modules

Instead of applying the Android application plugin, the build script applies the Android library plugin:

apply plugin: 'com.android.library'

There are two ways to include a library project in your application. One is to have it as a module inside your project; another is to create a .aar file, which can be reused in multiple applications. If you set up a library project as a module in your project, you need to add the module to settings.gradle and add it as a dependency to the application module. The settings file should look like this:

include ':app', ':library'

In this case, the library module is called library, and this corresponds to a folder with the same name. To use the library in the Android module, a dependency needs to be added to the build.gradle file of the Android module:

dependencies { compile project(':library') }

This will include the output of the library in the classpath of the application module.

3.8 Using .aar files

If you create a library that you want to reuse in different Android applications, you can build a .aar file, and add it to your project as a dependency. The .aar file will be generated in the build/output/aar/ folder of the module's directory when building the library. To add the .aar file as a dependency, you need to create a folder in the application module, copy the .aar file to it, and add the folder as a repository:

repositories { flatDir { dirs 'aars' } }

This will make it possible to add any file inside that folder as a dependency. You can reference the dependency as follows:

dependencies { compile(name:'libraryname', ext:'aar') }

This tells Gradle to look for a library with a certain name that has the .aar extension.

4. Creating Build Variants