Android Studio Project Site

Search this site

Preview Releases

Downloads & Preview Channels
Recent Changes
New Build System

Resources & Feedback

Technical docs
Known Issues
Feedback
Filing Bugs
Tips
Android Studio Videos

Open Source

Projects Overview
Build Overview
Contributing

Social Media

@AndroidStudio
+Android Studio

Technical docs > New Build System >

Gradle Plugin User Guide

Contents

翻译

Introduction

DSL reference

If you are looking for a full list of options available in `build.gradle` files, please see the DSL reference.

Goals of the new Build System

The goals of the new build system are:

- Make it easy to reuse code and resources
- Make it easy to create several variants of an application, either for multi-apk distribution or for different flavors of an application
- Make it easy to configure, extend and customize the build process
- Good IDE integration

Why Gradle?

Gradle is an advanced build system as well as an advanced build toolkit allowing to create custom build logic through plugins. Here are some of its features that made us choose Gradle:

- Domain Specific Language (DSL) based on Groovy, used to describe and manipulate the build logic
- Build files are Groovy based and allow mixing of declarative elements through the DSL and using code to manipulate the DSL elements to provide custom logic.
- Built-in dependency management through Maven and/or Ivy.
- Very flexible. Allows using best practices but doesn’t force its own way of doing things.
- Plugins can expose their own DSL and their own API for build files to use.
- Good Tooling API allowing IDE integration

Requirements

- Gradle 2.2
- SDK with Build Tools 19.0.0. Some features may require a more recent version.

翻译

Basic Project Setup

A Gradle project describes its build in a file called *build.gradle* located in the root folder of the project. (See Gradle User Guide for an overview of the build system itself.)

Simple build files

The most simple Android project has the following *build.gradle*:

```
buildscript {
    repositories {
        jcenter()
    }

    dependencies {
        classpath
'com.android.tools.build:gradle:1.3.1'
    }
}

apply plugin: 'com.android.application'

android {
    compileSdkVersion 23
    buildToolsVersion "23.1.0"
}
```

There are 3 main areas to this Android build file:

`buildscript { ... }` configures the code driving the build. In this case, this declares that it uses the jCenter repository, and that there is a classpath dependency on a Maven artifact. This artifact is the library that contains the Android plugin for Gradle in version 1.3.1. ***Note:*** *This only affects the code running the build, not the project. The project itself needs to declare its own repositories and dependencies. This will be covered later.*

Then, the **`com.android.application`** plugin is applied. This is the plugin used for building Android applications.

Finally, **`android { ... }`** configures all the parameters for the android build. This is the entry point for the Android DSL. By default, only the compilation target, and the version of the build-tools are needed. This is done with the **`compileSdkVersion`** and **`buildtoolsVersion`** properties. The compilation target is the same as the **`target`** property in the project.properties file of the old build system. This new property can either be assigned a int (the api level) or a string with the same value as the previous **`target`** property.

Important: You should only apply the **`com.android.application`** plugin. Applying the **`java`** plugin as well will result in a build error.

翻译

Note: You will also need a *local.properties* file to set the location of the SDK in the same way that the existing SDK requires, using the `sdk.dir` property. Alternatively, you can set an environment variable called `ANDROID_HOME`. There is no differences between the two methods, you can use the one you prefer. Example *local.properties* file:

```
sdk.dir=/path/to/Android/Sdk
```

Project Structure

The basic build files above expects a default folder structure. Gradle follows the concept of convention over configuration, providing sensible default option values when possible. The basic project starts with two components called “source sets”, one for the main source code and one for the test code. These live respectively in:

- `src/main/`
- `src/androidTest/`

Inside each of these directories there are subdirectories for each source component. For both the Java and Android plugin, the location of the Java source code and the Java resources are:

- `java/`
- `resources/`

For the Android plugin, extra files and folders specific to Android:

- `AndroidManifest.xml`
- `res/`
- `assets/`
- `aidl/`
- `rs/`
- `jni/`
- `jniLibs/`

This means that `*.java` files for the main source set are located in `src/main/java` and the main manifest is `src/main/AndroidManifest.xml`

Note: `src/androidTest/AndroidManifest.xml` is not needed as it is created automatically.

Configuring the Structure

When the default project structure isn’t adequate, it is possible to configure it. See this page in gradle documentation for information how this can be done for pure-java projects.

翻译

The Android plugin uses a similar syntax, but because it uses its own *sourceSets*, this is done within the **android** block. Here's an example, using the old project structure (used in Eclipse) for the main code and remapping the **androidTest** *sourceSet* to the `tests` folder:

```
android {
    sourceSets {
        main {
            manifest.srcFile 'AndroidManifest.xml'
            java.srcDirs = ['src']
            resources.srcDirs = ['src']
            aidl.srcDirs = ['src']
            renderscript.srcDirs = ['src']
            res.srcDirs = ['res']
            assets.srcDirs = ['assets']
        }

        androidTest.setRoot('tests')
    }
}
```

Note: because the old structure put all source files (Java, AIDL and RenderScript) in the same folder, we need to remap all those new components of the *sourceSet* to the same **src** folder.
Note: **setRoot()** moves the whole *sourceSet* (and its sub folders) to a new folder. This moves **src/androidTest/*** to **tests/*** This is Android specific and will not work on Java *sourceSets*.

Build Tasks

General Tasks

Applying a plugin to the build file automatically creates a set of build tasks to run. Both the Java plugin and the Android plugin do this. The convention for tasks is the following:

- **assemble**
  The task to assemble the output(s) of the project.
- **check**
  The task to run all the checks.
- **build**
  This task does both **assemble** and **check**.
- **clean**
  This task cleans the output of the project.

The tasks **assemble**, **check** and **build** don't actually do anything. They are "anchor" tasks for the plugins to add dependent tasks that actually do the work.

This allows you to always call the same task(s) no matter what the type of project is, or what plugins are applied. For instance, applying the *findbugs* plugin will create a new task and make **check** depend on it, making it be called whenever the **check** task is called.

翻译

From the command line you can get the high level task running the following command:

```
gradle tasks
```

For a full list and seeing dependencies between the tasks run:

```
gradle tasks --all
```

Note: Gradle automatically monitor the declared inputs and outputs of a task.

Running the **`build`** task twice without making changes to your project, will make Gradle report all tasks as `UP-TO-DATE`, meaning no work was required. This allows tasks to properly depend on each other without requiring unnecessary build operations.

Java project tasks

Here are the two most important tasks created by the `java` plugin, dependencies of the main anchor tasks:

- **`assemble`**
  - **`jar`**
    This task creates the output.
- **`check`**
  - **`test`**
    This task runs the tests.

The **`jar`** task itself will depend directly and indirectly on other tasks: **`classes`** for instance will compile the Java code. The tests are compiled with **`testClasses`**, but it is rarely useful to call this as **`test`** depends on it (as well as **`classes`**).

In general, you will probably only ever call **`assemble`** or **`check`**, and ignore the other tasks. You can see the full set of tasks and their descriptions for the Java plugin here.

Android tasks

The Android plugin uses the same convention to stay compatible with other plugins, and adds an additional anchor task:

- **`assemble`**
  The task to assemble the output(s) of the project.
- **`check`**
  The task to run all the checks.
- **`connectedCheck`**
  Runs checks that requires a connected device or emulator. they will run on all connected devices in parallel.
- **`deviceCheck`**
  Runs checks using APIs to connect to remote devices. This is used on

翻译

CI servers.
- `build`
  This task does both `assemble` and `check`
- `clean`
  This task cleans the output of the project.

The new anchor tasks are necessary in order to be able to run regular checks without needing a connected device. Note that `build` does not depend on `deviceCheck`, or `connectedCheck`.

An Android project has at least two outputs: a debug APK and a release APK. Each of these has its own anchor task to facilitate building them separately:

- `assemble`
  - `assembleDebug`
  - `assembleRelease`

They both depend on other tasks that execute the multiple steps needed to build an APK. The `assemble` task depends on both, so calling it will build both APKs.

Tip: Gradle support camel case shortcuts for task names on the command line. For instance:

```
gradle aR
```

is the same as typing

```
gradle assembleRelease
```

as long as no other task matches 'aR'

The check anchor tasks have their own dependencies:

- `check`
  - `lint`
- `connectedCheck`
  - `connectedAndroidTest`
- `deviceCheck`
  - This depends on tasks created when other plugins implement test extension points.

Finally, the plugin creates tasks for installing and uninstalling all build types (`debug`, `release`, `test`), as long as they can be installed (which requires signing), e.g.

- `installDebug`
- `installRelease`
- `uninstallAll`
  - `uninstallDebug`
  - `uninstallRelease`
  - `uninstallDebugAndroidTest`

Basic Build Customization

翻译

The Android plugin provides a broad DSL to customize most things directly from the build system.

Manifest entries

Through the DSL it is possible to configure the most important manifest entries, like these:

- `minSdkVersion`
- `targetSdkVersion`
- `versionCode`
- `versionName`
- `applicationId` (the effective packageName -- see ApplicationId versus PackageName for more information)
- `testApplicationId` (used by the test APK)
- `testInstrumentationRunner`

Example:

```
android {
    compileSdkVersion 23
    buildToolsVersion "23.0.1"


    defaultConfig {
        versionCode 12
        versionName "2.0"
        minSdkVersion 16
        targetSdkVersion 23
    }
}
```

Please see the Android Plugin DSL Reference for a complete list of build properties that can be configured and their default values.

The power of putting these manifest properties in the build file is that the values can be chosen dynamically. For instance, one could be reading the version name from a file or using other custom logic:

```


def computeVersionName() {
    ...
}


android {
    compileSdkVersion 23
    buildToolsVersion "23.0.1"


    defaultConfig {
        versionCode 12
        versionName computeVersionName()
        minSdkVersion 16
        targetSdkVersion 23
    }
}
```

Note: Do not use function names that could conflict with existing getters in the given scope. For instance `defaultConfig { ...}` calling

翻译

`getVersionName()` will automatically use the getter of `defaultConfig.getVersionName()` instead of the custom method.

Build Types

By default, the Android plugin automatically sets up the project to build both a debug and a release version of the application. These differ mostly around the ability to debug the application on a secure (non dev) devices, and details of how the APK is signed. The debug version is signed with a key/certificate that is created automatically with a known name/password (to prevent required prompt during the build). The release is not signed during the build, this needs to happen after.

This configuration is done through an object called a **`BuildType`**. By default, 2 instances are created, a **`debug`** and a **`release`** one. The Android plugin allows customizing those two instances as well as creating other *Build Types*. This is done with the **`buildTypes`** DSL container:

```
android {
    buildTypes {
        debug {
            applicationIdSuffix ".debug"
        }


        jnidebug {
            initWith(buildTypes.debug)
            applicationIdSuffix ".jnidebug"
            jniDebuggable true
        }
    }
}
```

The above snippet achieves the following:

- Configures the default **`debug`** *Build Type*:
  - set its package to be `<app appliationId>.debug` to be able to install both *debug* and *release* apk on the same device
- Creates a new *BuildType* called **`jnidebug`** and configure it to be a copy of the **`debug`** build type.
- Keep configuring the **`jnidebug`**, by enabling debug build of the JNI component, and add a different package suffix.

Creating new *Build Types* is as easy as using a new element under the **`buildTypes`** container, either to call **`initWith()`** or to configure it with a closure. See the DSL Reference for a list of all properties that can be configured on a build type.

In addition to modifying build properties, *Build Types* can be used to add specific code and resources. For each Build Type, a new matching *sourceSet* is created, with a default location of `src/<buildtypename>/`, e.g. `src/debug/java` directory can be used to add sources that will only be compiled for the debug APK. This means the *Build Type* names cannot be

翻译

main or *androidTest* (this is enforced by the plugin), and that they have to be unique.

Like any other source sets, the location of the build type source set can be relocated:

```
android {
    sourceSets.jnidebug.setRoot('foo/jnidebug')
}
```

Additionally, for each *Build Type*, a new `assemble<BuildTypeName>` task is created, e.g. **`assembleDebug`**. The **`assembleDebug`** and **`assembleRelease`** tasks have already been mentioned, and this is where they come from. When the **`debug`** and **`release`** *Build Types* are pre-created, their tasks are automatically created as well. According to this rule, *build.gradle* snippet above will also generate an **`assembleJnidebug`** task, and **`assemble`** would be made to depend on it the same way it depends on the **`assembleDebug`** and **`assembleRelease`** tasks.

Tip: remember that you can type `gradle aJ` to run the **`assembleJnidebug`** task.

Possible use case:

- Permissions in debug mode only, but not in release mode
- Custom implementation for debugging
- Different resources for debug mode (for instance when a resource value is tied to the signing certificate).

The code/resources of the *BuildType* are used in the following way:

- The manifest is merged into the app manifest
- The code acts as just another source folder
- The resources are overlayed over the main resources, replacing existing values.

Signing Configurations

Signing an application requires the following (See Signing Your Application for details about signing an APK):

- A keystore
- A keystore password
- A key alias name
- A key password
- The store type

The location, as well as the key name, both passwords and store type form together a Signing Configuration. By default, there is a **`debug`** configuration that is setup to use a debug keystore, with a known password and a default key with a known password.

翻译

The debug keystore is located in `$HOME/.android/debug.keystore`, and is created if not present. The **`debug`** *Build Type* is set to use this **`debug`** *SigningConfig* automatically.

It is possible to create other configurations or customize the default built-in one. This is done through the **`signingConfigs`** DSL container:

```
android {
    signingConfigs {
        debug {
            storeFile file("debug.keystore")
        }


        myConfig {
            storeFile file("other.keystore")
            storePassword "android"
            keyAlias "androiddebugkey"
            keyPassword "android"
        }
    }


    buildTypes {
        foo {
            signingConfig signingConfigs.myConfig
        }
    }
}
```

The above snippet changes the location of the debug keystore to be at the root of the project. This automatically impacts any *Build Types* that are set to using it, in this case the **`debug`** *Build Type*. It also creates a new *Signing Config* and a new *Build Type* that uses the new configuration.

Note: Only debug keystores located in the default location will be automatically created. Changing the location of the debug keystore will not create it on-demand. Creating a *SigningConfig* with a different name that uses the default debug keystore location will create it automatically. In other words, it's tied to the location of the keystore, not the name of the configuration.

Note: Location of keystores are usually relative to the root of the project, but could be absolute paths, thought it is not recommended (except for the debug one since it is automatically created).

Note: If you are checking these files into version control, you may not want the password in the file. The following Stack Overflow post shows ways to read the values from the console, or from environment variables.

Dependencies, Android Libraries and Multi-project setup

Gradle projects can have dependencies on other components. These components can be external binary packages, or other Gradle projects.

Dependencies on binary packages

翻译

Local packages

To configure a dependency on an external library jar, you need to add a dependency on the **`compile`** configuration. The snippet below adds a dependency on all jars inside the `libs` directory.

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

android {
    ...
}
```

Note: the **`dependencies`** DSL element is part of the standard Gradle API and does not belong inside the **`android`** element.

The **`compile`** configuration is used to compile the main application. Everything in it is added to the compilation classpath **and** also packaged in the final APK. There are other possible configurations to add dependencies to:

- **`compile`**: main application
- **`androidTestCompile`**: test application
- **`debugCompile`**: debug Build Type
- **`releaseCompile`**: release Build Type.

Because it's not possible to build an APK that does not have an associated *Build Type*, the APK is always configured with two (or more) configurations: **`compile`** and `<buildtype>Compile`. Creating a new *Build Type* automatically creates a new configuration based on its name. This can be useful if the debug version needs to use a custom library (to report crashes for instance), while the release doesn't, or if they rely on different versions of the same library (see Gradle documentation on details of how version conflicts are handled).

Remote artifacts

Gradle supports pulling artifacts from Maven and Ivy repositories. First the repository must be added to the list, and then the dependency must be declared in a way that Maven or Ivy declare their artifacts.

```
repositories {
     jcenter()
}

dependencies {
    compile 'com.google.guava:guava:18.0'
}

android {
    ...
}
```

翻译

Note: `jcenter()` is a shortcut to specifying the URL of the repository. Gradle supports both remote and local repositories.
Note: Gradle will follow all dependencies transitively. This means that if a dependency has dependencies of its own, those are pulled in as well.

For more information about setting up dependencies, read the Gradle user guide here, and DSL documentation here.

Multi project setup

Gradle projects can also depend on other gradle projects by using a multi-project setup. A multi-project setup usually works by having all the projects as sub folders of a given root project. For instance, given to following structure:

```
MyProject/
 + app/
 + libraries/
    + lib1/
    + lib2/
```

We can identify 3 projects. Gradle will reference them with the following name:

```
:app
:libraries:lib1
:libraries:lib2
```

Each projects will have its own *build.gradle* declaring how it gets built. Additionally, there will be a file called *settings.gradle* at the root declaring the projects. This gives the following structure:

```
MyProject/
 | settings.gradle
 + app/
    | build.gradle
 + libraries/
    + lib1/
       | build.gradle
    + lib2/
       | build.gradle
```

The content of settings.gradle is very simple. It defines which folder is actually a Gradle project:

```
include ':app', ':libraries:lib1', ':libraries:lib2'
```

The `:app` project is likely to depend on the libraries, and this is done by declaring the following dependencies:

```
dependencies {
     compile project(':libraries:lib1')
```

翻译

```
}
```

More general information about multi-project setup [here](#).

Library projects

In the above multi-project setup, **`:libraries:lib1`** and **`:libraries:lib2`** can be Java projects, and the **`:app`** Android project will use their *jar* output. However, if you want to share code that accesses Android APIs or uses Android-style resources, these libraries cannot be regular Java project, they have to be Android Library Projects.

Creating a Library Project

A Library project is very similar to a regular Android project with a few differences. Since building libraries is different than building applications, a different plugin is used. Internally both plugins share most of the same code and they are both provided by the same `com.android.tools.build.gradle` jar.

```
buildscript {
    repositories {
        jcenter()
    }

    dependencies {
        classpath
'com.android.tools.build:gradle:1.3.1'
    }
}

apply plugin: 'com.android.library'

android {
    compileSdkVersion 23
    buildToolsVersion "23.0.1"
}
```

This creates a library project that uses API 23 to compile. *SourceSets, build types* and dependencies are handled the same as they are in an application project and can be customized the same way.

Differences between a Project and a Library Project

A Library project's main output is an [*.aar* package](#) (which stands for Android archive). It is a combination of compile code (as a jar file and/or native .so files) and resources (manifest, res, assets). A library project can also generate a test apk to test the library independently from an application. The same anchor tasks are used for this (**`assembleDebug`**, **`assembleRelease`**) so

翻译

there's no difference in commands to build such a project. For the rest, libraries behave the same as application projects. They have build types and product flavors (see below), and can potentially generate more than one version of the aar. Note that most of the configuration of the *Build Type* do not apply to library projects. However you can use the custom *sourceSet* to change the content of the library depending on whether it's used by a project or being tested.

Referencing a Library

Referencing a library is done the same way any other project is referenced:

```
dependencies {
    compile project(':libraries:lib1')
    compile project(':libraries:lib2')
}
```

Note: if you have more than one library, then the order will be important. This is similar to the old build system where the order of the dependencies in the project.properties file was important.

Library Publication

By default a library only publishes its *release* variant. This variant will be used by all projects referencing the library, no matter which variant they build themselves. This is a temporary limitation due to Gradle limitations that we are working towards removing. You can control which variant gets published:

```
android {
    defaultPublishConfig "debug"
}
```

Note that this publishing configuration name references the full variant name. *Release* and *debug* are only applicable when there are no flavors. If you wanted to change the default published variant while using flavors, you would write:

```
android {
    defaultPublishConfig "flavor1Debug"
}
```

It is also possible to publish all variants of a library. We are planning to allow this while using a normal project-to-project dependency (like shown above), but this is not possible right now due to limitations in Gradle (we are working toward fixing those as well).
Publishing of all variants are not enabled by default. The snippet below enables this feature:

```
android {
    publishNonDefault true
}
```

翻译

It is important to realize that publishing multiple variants means publishing multiple aar files, instead of a single aar containing multiple variants. Each aar packaging contains a single variant. Publishing a variant means making this aar available as an output artifact of the Gradle project. This can then be used either when publishing to a maven repository, or when another project creates a dependency on the library project.

Gradle has a concept of default" artifact. This is the one that is used when writing:

```
dependencies {
    compile project(':libraries:lib2')
}
```

To create a dependency on another published artifact, you need to specify which one to use:

```
dependencies {
    flavor1Compile project(path: ':lib1', 
configuration: 'flavor1Release')
    flavor2Compile project(path: ':lib1', 
configuration: 'flavor2Release')
}
```

Important: Note that the published configuration is a full variant, including the build type, and needs to be referenced as such.
Important: When enabling publishing of non default, the Maven publishing plugin will publish these additional variants as extra packages (with classifier). This means that this is not really compatible with publishing to a maven repository. You should either publish a single variant to a repository OR enable all config publishing for inter-project dependencies.

Testing

Building a test application is integrated into the application project. There is no need for a separate test project anymore.

Unit testing

For the unit testing support added in 1.1, please see this separate page. The rest of this section describes "instrumentation tests" that can run on a real device (or an emulator) and require a separate, testing APK to be built.

Basics and Configuration

As mentioned previously, next to the **`main`** *sourceSet* is the **`androidTest`** *sourceSet*, located by default in `src/androidTest/`. Using this *sourceSet* a test APK gets built, which can be deployed to a device to test the application using the Android testing 翻译

framework. This can contain android unit tests, instrumentation tests, and uiautomator tests. The `<instrumentation>` node of the manifest for the test app is generated but you can create a `src/androidTest/AndroidManifest.xml` file to add other components to the test manifest.

There are a few values that can be configured for the test app, in order to configure the `<instrumentation>` node (see the DSL reference for details)

- **`testApplicationId`**
- **`testInstrumentationRunner`**
- **`testHandleProfiling`**
- **`testFunctionalTest`**

As seen previously, those are configured in the **`defaultConfig`** object:

```
android {
    defaultConfig {
        testApplicationId "com.test.foo"
        testInstrumentationRunner
"android.test.InstrumentationTestRunner"
        testHandleProfiling true
        testFunctionalTest true
    }
}
```

The value of the `targetPackage` attribute of the `instrumentation` node in the test application manifest is automatically filled with the package name of the tested app, even if it is customized through the **`defaultConfig`** and/or the *Build Type* objects. This is one of the reason this part of the manifest is generated automatically.

Additionally, the *androidTest* source set can be configured to have its own dependencies. By default, the application and its own dependencies are added to the test app classpath, but this can be extended with the snippet below:

```
dependencies {
    androidTestCompile 'com.google.guava:guava:11.0.2'
}
```

The test app is built by the **`assembleAndroidTest`** task. It is not a dependency of the main **`assemble`** task, and is instead called automatically when the tests are set to run.

Currently only one *Build Type* is tested. By default it is the **`debug`** *Build Type*, but this can be reconfigured with:

```
android {
    ...
    testBuildType "staging"
}
```

Resolving conflicts between main and test APK

翻译

When instrumentation tests are run, both the main APK and test APK share the same classpath. Gradle build will fail if the main APK and the test APK use the same library (e.g. Guava) but in different versions. If gradle didn't catch that, your app could behave differently during tests and during normal run (including crashing in one of the cases).

To make the build succeed, just make sure both APKs use the same version. If the error is about an indirect dependency (a library you didn't mention in your build.gradle), just add a dependency for the newer version to the configuration ("compile" or "androidTestCompile") that needs it. You can also use Gradle's resolution strategy mechanism. You can inspect the dependency tree by running `./gradlew :app:dependencies` and `./gradlew :app:androidDependencies.`

Running tests

As mentioned previously, checks requiring a connected device are launched with the anchor task called **`connectedCheck`**. This depends on the task **`connectedDebugAndroidTest`** and therefore will run it. This task does the following:

- Ensure the app and the test app are built (depending on **`assembleDebug`** and **`assembleDebugAndroidTest`**).
- Install both apps.
- Run the tests.
- Uninstall both apps.

If more than one device is connected, all tests are run in parallel on all connected devices. If one of the test fails, on any device, the build will fail.

Testing Android Libraries

Testing Android Library project is done exactly the same way as application projects. The only difference is that the whole library (and its dependencies) is automatically added as a Library dependency to the test app. The result is that the test APK includes not only its own code, but also the library itself and all its dependencies. The manifest of the Library is merged into the manifest of the test app (as is the case for any project referencing this Library). The **`androidTest`** task is changed to only install (and uninstall) the test APK (since there are no other APK to install.) and everything else is identical.

Test reports

When running unit tests, Gradle outputs an HTML report to easily look at the results. The Android plugins build on this and extends the HTML report to aggregate the results from all connected devices. All test results are stored as XML files under `build/reports/androidTests/` (this is similar to

翻译

regular jUnit results that are stored under `build/reports/tests`). This can be configured with:

```
android {
    ...

    testOptions {
        resultsDir = "${project.buildDir}/foo/results"
    }
}
```

The value of **`android.testOptions.resultsDir`** is evaluated with **`Project.file(String)`**

Multi-projects reports

In a multi project setup with application(s) and library(ies) projects, when running all tests at the same time, it might be useful to generate a single reports for all tests.

To do this, a different plugin is available in the same artifact. It can be applied with:

```
buildscript {
    repositories {
        jcenter()
    }

    dependencies {
        classpath
'com.android.tools.build:gradle:0.5.6'
    }
}

apply plugin: 'android-reporting'
```

This should be applied to the root project, ie in *build.gradle* next to *settings.gradle*

Then from the root folder, the following command line will run all the tests and aggregate the reports:

```
     gradle deviceCheck mergeAndroidReports --continue
```

Note: the `--continue` option ensure that all tests, from all sub-projects will be run even if one of them fails. Without it the first failing test will interrupt the run and not all projects may have their tests run.

Lint support

You can run lint for a specific variant (see below), e.g. `./gradlew lintRelease`, or for all variants (`./gradlew lint`), in which case it produces a report which describes which specific variants a given issue applies to. You can configure lint by adding a lintOptions section like following.

翻译

You typically only specify a few of these, see the DSL reference for all available options.

```
android {
    lintOptions {
        // turn off checking the given issue id's
        disable
'TypographyFractions','TypographyQuotes'

        // turn on the given issue id's
        enable 'RtlHardcoded','RtlCompat', 'RtlEnabled'

        // check *only* the given issue id's
        check 'NewApi', 'InlinedApi'
    }
}
```

Build Variants

One goal of the new build system is to enable creating different versions of the same application.

There are two main use cases:

1. Different versions of the same application
   For instance, a free/demo version vs the “pro” paid application.
2. Same application packaged differently for multi-apk in Google Play Store.
   See http://developer.android.com/google/play/publishing/multiple-apks.html for more information.
3. A combination of 1. and 2.

The goal was to be able to generate these different APKs from the same project, as opposed to using a single Library Projects and 2+ Application Projects.

Product flavors

A product flavor defines a customized version of the application build by the project. A single project can have different flavors which change the generated application.

This new concept is designed to help when the differences are very minimum. If the answer to “Is this the same application?” is yes, then this is probably the way to go over Library Projects.

Product flavors are declared using a `productFlavors` DSL container:

```
android {
    ....

    productFlavors {
        flavor1 {
            ...
        }
```

翻译

```
        flavor2 {
            ...
        }
    }
}
```

This creates two flavors, called **`flavor1`** and **`flavor2`**.

Note: The name of the flavors cannot collide with existing *Build Type* names, or with the **`androidTest`** and **`test`** source sets.

Build Type + Product Flavor = Build Variant

As we have seen before, each *Build Type* generates a new APK. *Product Flavors* do the same: the output of the project becomes all possible combinations of *Build Types* and, if applicable, *Product Flavors*. Each (*Build Type*, *Product Flavor*) combination is called a *Build Variant*. For instance, with the default **`debug`** and **`release`** *Build Types*, the above example generates four *Build Variants*:

- Flavor1 - debug
- Flavor1 - release
- Flavor2 - debug
- Flavor2 - release

Projects with no flavors still have *Build Variants*, but the single **`default`** flavor/config is used, nameless, making the list of variants similar to the list of *Build Types*.

Product Flavor Configuration

Each flavors is configured with a closure:

```
android {
    ...

    defaultConfig {
        minSdkVersion 8
        versionCode 10
    }

    productFlavors {
        flavor1 {
            applicationId "com.example.flavor1"
            versionCode 20
         }

         flavor2 {
             applicationId "com.example.flavor2"
             minSdkVersion 14
         }
    }
}
```

翻译

Note that the `android.productFlavors.*` objects are of type *ProductFlavor* which is the same type as the `android.defaultConfig` object. This means they share the same properties.

`defaultConfig` provides the base configuration for all flavors and each flavor can override any value. In the example above, the configurations end up being:

- `flavor1`
  - `applicationId`: com.example.flavor1
  - `minSdkVersion`: 8
  - `versionCode`: 20
- `flavor2`
  - `applicationId`: com.example.flavor2
  - `minSdkVersion`: 14
  - `versionCode`: 10

Usually, the *Build Type* configuration is an overlay over the other configuration. For instance, the *Build Type*'s `applicationIdSuffix` is appended to the *Product Flavor*'s `applicationId`. There are cases where a setting is settable on both the *Build Type* and the *Product Flavor*. In this case, it's is on a case by case basis. For instance, `signingConfig` is one of these properties. This enables either having all release packages share the same *SigningConfig*, by setting `android.buildTypes.release.signingConfig`, or have each release package use their own *SigningConfig* by setting each `android.productFlavors.*.signingConfig` objects separately.

Sourcesets and Dependencies

Similar to *Build Types*, *Product Flavors* also contribute code and resources through their own *sourceSets*. The above example creates four *sourceSets*:

- `android.sourceSets.flavor1`
  Location `src/flavor1/`
- `android.sourceSets.flavor2`
  Location `src/flavor2/`
- `android.sourceSets.androidTestFlavor1`
  Location `src/androidTestFlavor1/`
- `android.sourceSets.androidTestFlavor2`
  Location `src/androidTestFlavor2/`

Those *sourceSets* are used to build the APK, alongside `android.sourceSets.main` and the *Build Type sourceSet*. The following rules are used when dealing with all the sourcesets used to build a single APK:

- All source code (`src/*/java`) are used together as multiple folders generating a single output.

翻译

- Manifests are all merged together into a single manifest. This allows *Product Flavors* to have different components and/or permissions, similarly to *Build Types*.
- All resources (Android res and assets) are used using overlay priority where the *Build Type* overrides the *Product Flavor*, which overrides the `main` *sourceSet*.
- Each *Build Variant* generates its own R class (or other generated source code) from the resources. Nothing is shared between variants.

Finally, like *Build Types*, *Product Flavors* can have their own dependencies. For instance, if the flavors are used to generate a ads-based app and a paid app, one of the flavors could have a dependency on an Ads SDK, while the other does not.

```
dependencies {
    flavor1Compile "..."
}
```

In this particular case, the file `src/flavor1/AndroidManifest.xml` would probably need to include the internet permission.

Additional sourcesets are also created for each variants:

- `android.sourceSets.flavor1Debug`
  Location `src/flavor1Debug/`
- `android.sourceSets.flavor1Release`
  Location `src/flavor1Release/`
- `android.sourceSets.flavor2Debug`
  Location `src/flavor2Debug/`
- `android.sourceSets.flavor2Release`
  Location `src/flavor2Release/`

These have higher priority than the build type sourcesets, and allow customization at the variant level.

Building and Tasks

We previously saw that each *Build Type* creates its own `assemble<name>` task, but that *Build Variants* are a combination of *Build Type* and *Product Flavor*.

When *Product Flavors* are used, more assemble-type tasks are created. These are:

1. `assemble<Variant Name>`
2. `assemble<Build Type Name>`
3. `assemble<Product Flavor Name>`

#1 allows directly building a single variant. For instance `assembleFlavor1Debug`.

翻译

#2 allows building all APKs for a given Build Type. For instance **`assembleDebug`** will build both `Flavor1Debug` and `Flavor2Debug` variants.

#3 allows building all APKs for a given flavor. For instance **`assembleFlavor1`** will build both `Flavor1Debug` and `Flavor1Release` variants.

The task **`assemble`** will build all possible variants.

Multi-flavor variants

In some case, one may want to create several versions of the same apps based on more than one criteria.
For instance, multi-apk support in Google Play supports 4 different filters. Creating different APKs split on each filter requires being able to use more than one dimension of Product Flavors.

Consider the example of a game that has a demo and a paid version and wants to use the ABI filter in the multi-apk support. With 3 ABIs and two versions of the application, 6 APKs needs to be generated (not counting the variants introduced by the different Build Types).
However, the code of the paid version is the same for all three ABIs, so creating simply 6 flavors is not the way to go.
Instead, there are two dimensions of flavors, and variants should automatically build all possible combinations.

This feature is implemented using Flavor Dimensions. Flavors are assigned to a specific dimension:

```
android {
    ...

    flavorDimensions "abi", "version"

    productFlavors {
        freeapp {
            dimension "version"
            ...
        }

        paidapp {
            dimension "version"
            ...
        }

        arm {
            dimension "abi"
            ...
        }

        mips {
            dimension "abi"
            ...
        }
```

翻译

```
            x86 {
                dimension "abi"
                ...
            }
        }
    }
```

The **android.flavorDimensions** array defines the possible dimensions, as well as the order. Each defined *Product Flavor* is assigned to a dimension.

From the following dimensioned *Product Flavors* [freeapp, paidapp] and [x86, arm, mips] and the [debug, release] *Build Types*, the following build variants will be created:

- `x86-freeapp-debug`
- `x86-freeapp-release`
- `arm-freeapp-debug`
- `arm-freeapp-release`
- `mips-freeapp-debug`
- `mips-freeapp-release`
- `x86-paidapp-debug`
- `x86-paidapp-release`
- `arm-paidapp-debug`
- `arm-paidapp-release`
- `mips-paidapp-debug`
- `mips-paidapp-release`

The order of the dimension as defined by **android.flavorDimensions** is very important.

Each variant is configured by several *Product Flavor* objects:

- **android.defaultConfig**
- One from the abi dimension
- One from the version dimension

The order of the dimension drives which flavor override the other, which is important for resources when a value in a flavor replaces a value defined in a lower priority flavor.
The flavor dimension is defined with higher priority first. So in this case:

```
    abi > version > defaultConfig
```

Multi-flavors projects also have additional sourcesets, similar to the variant sourcesets but without the build type:

- **android.sourceSets.x86Freeapp**
  Location `src/x86Freeapp/`
- **android.sourceSets.armPaidapp**
  Location `src/armPaidapp/`
- etc...

翻译

These allow customization at the flavor-combination level. They have higher priority than the basic flavor sourcesets, but lower priority than the build type sourcesets.

Testing

Testing multi-flavors project is very similar to simpler projects.

The `androidTest` sourceset is used for common tests across all flavors, while each flavor can also have its own tests.

As mentioned above, *sourceSets* to test each flavor are created:

- `android.sourceSets.androidTestFlavor1`
  Location `src/androidTestFlavor1/`
- `android.sourceSets.androidTestFlavor2`
  Location `src/androidTestFlavor2/`

Similarly, those can have their own dependencies:

```
dependencies {
    androidTestFlavor1Compile "..."
}
```

Running the tests can be done through the main `deviceCheck` anchor task, or the main `androidTest` tasks which acts as an anchor task when flavors are used.

Each flavor has its own task to run tests: `androidTest<VariantName>`. For instance:

- `androidTestFlavor1Debug`
- `androidTestFlavor2Debug`

Similarly, test APK building tasks and install/uninstall tasks are per variant:

- `assembleFlavor1Test`
- `installFlavor1Debug`
- `installFlavor1Test`
- `uninstallFlavor1Debug`
- `...`

Finally the HTML report generation supports aggregation by flavor.
The location of the test results and reports is as follows, first for the per flavor version, and then for the aggregated one:

- `build/androidTest-results/flavors/<FlavorName>`
- `build/androidTest-results/all/`
- `build/reports/androidTests/flavors<FlavorName>`
- `build/reports/androidTests/all/`

翻译

Customizing either path, will only change the root folder and still create sub folders per-flavor and aggregated results/reports.

BuildConfig

At compilation time, Android Studio generates a class called `BuildConfig` that contains constant values used when building a particular variant. You can inspect the values of these constants to change behavior in different variants, e.g.:

```
private void javaCode() {
    if (BuildConfig.FLAVOR.equals("paidapp")) {
        doIt();
    else {
        showOnlyInPaidAppDialog();
    }
}
```

Here are the values that BuildConfig contains:

- `boolean DEBUG` – if the build is debuggable.
- `int VERSION_CODE`
- `String VERSION_NAME`
- `String APPLICATION_ID`
- `String BUILD_TYPE` – name of the build type, e.g. "release"
- `String FLAVOR` – name of the flavor, e.g. "paidapp"

If the project uses flavor dimensions, additional values are generated. Using the example above, here's an example BuildConfig:

- `String FLAVOR = "armFreeapp"`
- `String FLAVOR_abi = "arm"`
- `String FLAVOR_version = "freeapp"`

Filtering Variants

When you add dimensions and flavors, you can end up with variants that don't make sense. For example you may define a flavor that uses your Web API and a flavor that uses hard-coded fake data, for faster testing. The second flavor is only useful for development, but not in release builds. You can remove this variant using the `variantFilter` closure, like this:

```
android {
    productFlavors {
        realData
        fakeData
    }

    variantFilter { variant ->
        def names = variant.flavors*.name

        if (names.contains("fakeData") &&
variant.buildType.name == "release") {
            variant.ignore = true
        }
    }
}
```

翻译

With the configuration above, your project will have only three variants:

- `realDataDebug`
- `realDataRelease`
- `fakeDataDebug`

See the DSL reference for all properties of `variant` that you can check.

Advanced Build Customization

Running ProGuard

The ProGuard plugin is applied automatically by the Android plugin, and the tasks are created automatically if the *Build Type* is configured to run ProGuard through the *minifyEnabled* property.

```
android {
    buildTypes {
        release {
            minifyEnabled true
            proguardFile
getDefaultProguardFile('proguard-android.txt')
        }
    }

    productFlavors {
        flavor1 {
        }
        flavor2 {
            proguardFile 'some-other-rules.txt'
        }
    }
}
```

Variants use all the rules files declared in their build type, and product flavors.

There are 2 default rules files

- proguard-android.txt
- proguard-android-optimize.txt

They are located in the SDK. Using *getDefaultProguardFile()* will return the full path to the files. They are identical except for enabling optimizations.

Shrinking Resources

You can also remove unused resources, automatically, at build time. For more information, see the Resource Shrinking document.

翻译

Manipulating tasks

Basic Java projects have a finite set of tasks that all work together to create an output.
The **`classes`** task is the one that compile the Java source code.
It's easy to access from *build.gradle* by simply using **`classes`** in a script. This is a shortcut for **`project.tasks.classes`**.

In Android projects, this is a bit more complicated because there could be a large number of the same task and their name is generated based on the *Build Types* and *Product Flavors*.

In order to fix this, the **`android`** object has two properties:

- **`applicationVariants`** (only for the app plugin)
- **`libraryVariants`** (only for the library plugin)
- **`testVariants`** (for both plugins)

All three return a DomainObjectCollection of `ApplicationVariant`, `LibraryVariant`, and `TestVariant` objects respectively.

Note that accessing any of these collections will trigger the creations of all the tasks. This means no (re)configuration should take place after accessing the collections.

The `DomainObjectCollection` gives access to all the objects directly, or through filters which can be convenient.

```
android.applicationVariants.all { variant ->
    ....
}
```

All three variant classes share the following properties:

Property Name	Property Type	Description
name	String	The name of the variant. Guaranteed to be unique.
description	String	Human readable description of the variant.
dirName	String	subfolder name for the variant. Guaranteed to be unique. Maybe more than one folder, ie "debug/flavor1"

翻译

baseName	String	Base name of the output(s) of the variant. Guaranteed to be unique.
outputFile	File	The output of the variant. This is a read/write property
processManifest	ProcessManifest	The task that processes the manifest.
aidlCompile	AidlCompile	The task that compiles the AIDL files.
renderscriptCompile	RenderscriptCompile	The task that compiles the Renderscript files.
mergeResources	MergeResources	The task that merges the resources.
mergeAssets	MergeAssets	The task that merges the assets.
processResources	ProcessAndroidResources	The task that processes and compile the Resources.
generateBuildConfig	GenerateBuildConfig	The task that generates the BuildConfig class.
javaCompile	JavaCompile	The task that compiles the Java code.
processJavaResources	Copy	The task that process the Java resources.
assemble	DefaultTask	The assemble anchor task for this variant.

The `ApplicationVariant` class adds the following:

Property Name	Property Type	Description
buildType	BuildType	The BuildType of the variant.
productFlavors	List<ProductFlavor>	The ProductFlavors of the variant. Always non Null but could be empty.
mergedFlavor	ProductFlavor	The merging of android.defaultConfig and variant.productFlavors
signingConfig	SigningConfig	The SigningConfig object used by this variant
isSigningReady	boolean	true if the variant has all the information needed to be signed.
testVariant	BuildVariant	The TestVariant that will tes

翻译

		this variant.
dex	Dex	The task that dex the code. Can be null if the variant is a library.
packageApplication	PackageApplication	The task that makes the final APK. Can be null if the variant is a library.
zipAlign	ZipAlign	The task that zipaligns the apk. Can be null if the variant is a library or if the APK cannot be signed.
install	DefaultTask	The installation task. Can be null.
uninstall	DefaultTask	The uninstallation task.

The `LibraryVariant` class adds the following:

Property Name	Property Type	Description
buildType	BuildType	The BuildType of the variant.
mergedFlavor	ProductFlavor	The defaultConfig values
testVariant	BuildVariant	The Build Variant that will test this variant.
packageLibrary	Zip	The task that packages the Library AAR archive. Null if not a library.

The `TestVariant` class adds the following:

Property Name	Property Type	Description
buildType	BuildType	The BuildType of the variant.
productFlavors	List<ProductFlavor>	The ProductFlavors of the variant. Always non Null but could be empty.
mergedFlavor	ProductFlavor	The merging of android.defaultConfig and variant.productFlavors
signingConfig	SigningConfig	The SigningConfig object used by this variant
isSigningReady	boolean	true if the variant has all the information needed to be signed.
testedVariant	BaseVariant	The BaseVariant that is tested by this TestVariant.
dex	Dex	The task that dex the code. Can be null if the variant is a library.
packageApplication	PackageApplication	The task that makes the final

翻译

		APK. Can be null if the variant is a library.
zipAlign	ZipAlign	The task that zipaligns the apk. Can be null if the variant is a library or if the APK cannot be signed.
install	DefaultTask	The installation task. Can be null.
uninstall	DefaultTask	The uninstallation task.
connectedAndroidTest	DefaultTask	The task that runs the android tests on connected devices.
providerAndroidTest	DefaultTask	The task that runs the android tests using the extension API.

API for Android specific task types.

- `ProcessManifest`
  - `File manifestOutputFile`
- `AidlCompile`
  - `File sourceOutputDir`
- `RenderscriptCompile`
  - `File sourceOutputDir`
  - `File resOutputDir`
- `MergeResources`
  - `File outputDir`
- `MergeAssets`
  - `File outputDir`
- `ProcessAndroidResources`
  - `File manifestFile`
  - `File resDir`
  - `File assetsDir`
  - `File sourceOutputDir`
  - `File textSymbolOutputDir`
  - `File packageOutputFile`
  - `File proguardOutputFile`
- `GenerateBuildConfig`
  - `File sourceOutputDir`
- `Dex`
  - `File outputFolder`
- `PackageApplication`
  - `File resourceFile`
  - `File dexFile`
  - `File javaResourceDir`
  - `File jniDir`

翻译

        - `File outputFile`
            - To change the final output file use "outputFile" on the variant object directly.
    - `ZipAlign`
        - `File inputFile`
        - `File outputFile`
            - To change the final output file use "outputFile" on the variant object directly.

The API for each task type is limited due to both how Gradle works and how the Android plugin sets them up.
First, Gradle is meant to have the tasks be only configured for input/output location and possible optional flags. So here, the tasks only define (some of) the inputs/outputs.

Second, the input for most of those tasks is non-trivial, often coming from mixing values from the *sourceSets*, the *Build Types*, and the *Product Flavors*. To keep build files simple to read and understand, the goal is to let developers modify the build by tweak these objects through the DSL, rather than diving deep in the inputs and options of the tasks and changing them.

Also note, that except for the ZipAlign task type, all other types require setting up private data to make them work. This means it’s not possible to manually create new tasks of these types.

This API is subject to change. In general the current API is around giving access to the outputs and inputs (when possible) of the tasks to add extra processing when required). Feedback is appreciated, especially around needs that may not have been foreseen.

For Gradle tasks (DefaultTask, JavaCompile, Copy, Zip), refer to the Gradle documentation.

Setting language level

You can use the `compileOptions` block to choose the language level used by the compiler. The default is chosen based on the `compileSdkVersion` value.

```
android {
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_6
        targetCompatibility JavaVersion.VERSION_1_6
    }
}
```

翻译