Android Studio 功能 用户指南 预览

← 用户指南
探索 Android Studio
Workflow Basics
管理项目
编写您的应用
构建和运行您的应用
配置构建
设置应用 ID
Add Build Dependencies
Optimize Your Build Speed
Use Build Cache
配置构建变体
Build Multiple APKs
合并多个清单
Inject Build Variables into the Manifest
Shrink Your Code and Resources
配置方法数超过 64K 的应用
Analyze Your Build with APK Analyzer
Gradle Tips and Recipes
调试您的应用
测试应用

配置构建变体

本文内容
- 配置构建类型
- 配置产品风味
  - 组合多个产品风味
  - 过滤变体
- 创建用于构建变体的源集
  - 更改默认源集配置
  - 使用源集构建
- 声明依赖项
  - 配置依赖项
- 配置签署设置
  - 签署 Android Wear 应用

另请参阅
- 配置构建概览
- 合并清单

此页面以配置构建概览为基础，向您介绍如何配置构建变体，以便从同一个项目中创建应用的不同版本，以及如何正确地管理依赖项并签署配置。

每个*构建变体*都代表您可以为应用构建的一个不同版本。例如，您可能希望构建应用的免费版本（只提供有限的内容）和付费版本（提供更多内容）。您还可以针对不同的设备、根据 API 级别或其他设备变体构建应用的不同版本。然而，如果您希望根据设备 ABI 或屏幕密度构建不同的版本，则请改用 APK 拆分。

构建变体是 Gradle 按照特定规则集合并在构建类型和产品风味中配置的设置、代码和资源所生成的结果。您并不直接配置构建变体，而是配置组成变体的构建类型和产品风味。

例如，一个“演示”*产品风味*可以指定不同的功能和设备要求，例如自定义源代码、资源和最低 API 级别，而“调试”*构建类型*则应用不同的构建和打包设置，例如调试选项和签署密钥。最终生成的构建变体是应用的“演示调试”版本，其既包含“演示”产品风味中包含的各种配置和资源，又包含“调试”构建类型和 `main/` 源集。

配置构建类型

您可以在模块级 `build.gradle` 文件的 `android {}` 代码块内部创建和配置构建类型。当您创建新模块时，Android Studio 会自动为您创建调试和发布这两种构建类型。尽管调试构建类型不会出现在构建配置文件中，Android Studio 会为其配置 `debuggable true`。这样，您可以在安全的 Android 设备上调试应用并使用通用调试密钥库配置 APK 签署。

如果您希望添加或更改特定设置，您可以将调试构建类型添加到您的配置中。以下示例为调试构建类型指定了 `applicationIdSuffix`，并配置了一个使用调试构建类型中的设置进行初始化的“jnidebug”构建类型。

```
android {
    ...
    defaultConfig {...}
    buildTypes {
        release {
            minifyEnabled true
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }

        debug {
            applicationIdSuffix ".debug"
        }

        /**
         * The 'initWith' property allows you to copy configurations from other build types,
         * so you don't have to configure one from the beginning. You can then configure
         * just the settings you want to change. The following line initializes
         * 'jnidebug' using the debug build type, and changes only the
         * applicationIdSuffix and versionNameSuffix settings.
         */

        jnidebug {

            // This copies the debuggable attribute and debug signing configurations.
            initWith debug

            applicationIdSuffix ".jnidebug"
            jniDebuggable true
        }
    }
}
```

> **注**：当您对构建配置文件进行更改时，Android Studio 会要求您为项目同步新配置。要同步项目，您可以点击在做出更改后立即出现在通知栏中的 **Sync Now**，也可以点击工具栏中的 **Sync Project** 。如果 Android Studio 通知配置出现错误，会显示 **Messages** 窗口，具体描述该问题。

如需详细了解对于构建类型可以配置的所有属性，请阅读构建类型 DSL 参考。

配置产品风味

创建产品风味与创建构建类型类似：只需将它们添加到 `productFlavors {}` 代码块并配置您想要的设置。产品风味支持与 `defaultConfig` 相同的属性，这是因为 `defaultConfig` 实际上属于 `ProductFlavor` 类。这意味着，您可以在 `defaultConfig {}` 代码块中提供所有风味的基本配置，每种风味均可更改任何这些默认值，例如 `applicationId`。要详细了解应用 ID，请阅读设置应用 ID。

> **注**：您仍需在 `main/` 清单文件中使用 `package` 属性指定程序包名称。您还必须在源代码中使用此程序包名称引用 R 类或者解析任何相关的 Activity 或服务注册。这样，您可以使用 `applicationId` 为每个产品风味分配一个唯一的 ID，以用于打包和分发，而不必更改您的源代码。

以下代码示例创建了一个“演示”和“完整”产品风味，以赋予其自己的 `applicationIdSuffix` 和 `versionNameSuffix`：

```
android {
    ...
    defaultConfig {...}
    buildTypes {...}
    productFlavors {
        demo {
            applicationIdSuffix ".demo"
            versionNameSuffix "-demo"
        }
        full {
            applicationIdSuffix ".full"
            versionNameSuffix "-full"
        }
    }
}
```

> **注**：要利用 Google Play 中的多 APK 支持分发您的应用，请为所有变体分配相同的 `applicationId` 值并为每个变体分配一个不同的 `versionCode`。要在 Google Play 中以独立应用的形式分发应用的不同变体，您需要为每个变体分配一个不同的 `applicationId`。

在创建和配置您的产品风味之后，在通知栏中点击 **Sync Now**。在同步完成后，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 `<product-flavor><Build-Type>` 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。然而，要开始自定义每个构建变体及其功能和资源，您需要了解如何创建和管理源集。

组合多个产品风味

某些情况下，您可能希望组合多个产品风味中的配置。例如，您可能希望基于 API 级别为“完整”和“演示”产品风味创建不同的配置。为此，您可以通过适用于 Gradle 的 Android 插件创建产品风味组，称为风味维度。构建您的应用时，Gradle 会将您定义的每个风味维度中的产品风味配置与构建类型配置组合来创建最终构建变体。Gradle 不会组合属于相同风味维度的产品风味。

> **提示**：要根据 ABI 和屏幕密度创建不同版本的应用，您应配置 APK 拆分，而不是使用产品风味。

下面的代码示例使用 `flavorDimensions` 属性创建一个“模式”风味维度以组织“完整”和“演示”产品风味，以及一个“api”风味维度以基于 API 级别组织产品风味配置：

```
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`

例如，

> 构建变体：`minApi24DemoDebug`
>
> 对应 APK：`app-minApi24-demo-debug.apk`

除了可以为各个产品风味和构建变体创建源集目录外，您也可以为每个产品风味*组合*创建源集目录。例如，您可以创建 Java 源并将其添加到 `src/demoMinApi24/java/` 目录中，Gradle 仅会在构建组合了这两种产品风味的变体时使用这些源。与属于各个产品风味的源集相比，您为产品风味组合创建的源集拥有更高的优先级。要详细了解源集和 Gradle 如何合并源，请阅读有关如何创建源集的部分。

过滤变体

Gradle 会为您配置的产品风味与构建类型的每个可能的组合创建构建变体。不过，某些特定的构建变体在您的项目环境中并不必要，也可能没有意义。您可以在模块级 `build.gradle` 文件中创建一个变体过滤器，以移除某些构建变体配置。

以上一部分中的构建配置为例，假设您计划为演示版本的应用仅支持 API 级别 23 和更高级别。您可以使用 `variantFilter {}` 代码块过滤出组合了“minApi21”和“演示”产品风味的所有构建变体配置：

```
android {
  ...
  buildTypes {...}

  flavorDimensions "api", "mode"
  productFlavors {
    demo {...}
    full {...}
    minApi24 {...}
    minApi23 {...}
    minApi21 {...}
  }

  variantFilter { variant ->
      def names = variant.flavors*.name
      // To check for a certain build type, use variant.buildType.name == "<buildType>"
      if (names.contains("minApi21") && names.contains("demo")) {
          // Gradle ignores any variants that satisfy the conditions above.
          setIgnore(true)
      }
  }
}
...
```

在您向构建配置添加变体过滤器并点击通知栏中的 **Sync Now** 后，Gradle 将忽略满足您指定的条件的任何构建变体，在您点击菜单栏中的 **Build > Select Build Variant**（或工具窗口栏中的 **Build Variants**）时，这些构建变体将不会再 显示在下拉菜单中。

创建源集

默认情况下，Android Studio 会创建 `main/` 源集和目录，用于存储您要在所有构建变体之间共享的一切资源。然而，您可以创建新的源集来控制 Gradle 要为特定的构建类型、产品风味（以及使用风味维度时的产品风味组合）和构建变体编译和打包的确切文件。例如，您可以在 `main/` 源集中定义基本的功能，使用产品风味源集针对不同的客户更改应用的品牌，或者仅针对使用调试构建类型的构建变体包含特殊的权限和日志记录功能。

Gradle 要求您按照与 `main/` 源集类似的特定方式组织源集文件和目录。例如，Gradle 要求您的“调试”构建类型所特定的 Java 类文件位于 `src/debug/java/` 目录中。

适用于 Gradle 的 Android 插件提供了一项有用的 Gradle 任务，可向您展示如何针对每种构建类型、产品风味和构建变体组织您的文件。例如，报告的以下部分描述了 Gradle 要求在何处能找到“调试”构建类型的特定文件：

```
------------------------------------------------------------
Project :app
------------------------------------------------------------

...

debug
----
Compile configuration: compile
build.gradle name: android.sourceSets.debug
Java sources: [app/src/debug/java]
Manifest file: app/src/debug/AndroidManifest.xml
Android resources: [app/src/debug/res]
Assets: [app/src/debug/assets]
AIDL sources: [app/src/debug/aidl]
RenderScript sources: [app/src/debug/rs]
JNI sources: [app/src/debug/jni]
JNI libraries: [app/src/debug/jniLibs]
Java-style resources: [app/src/debug/resources]
```

要为您的构建配置生成和查看此报告，请继续如下操作：

1. 点击 IDE 窗口右侧的 **Gradle** Gradle 。
2. 导航至 **MyApplication > Tasks > android** 并双击 **sourceSets**。
3. 要查看报告，请点击 IDE 窗口底部的 **Gradle Console** Gradle Console 。

> **注**：此报告还向您展示了如何为您希望用来运行应用测试的文件组织源集，例如 `test/` 和 `androidTest/` 测试源集。

当您创建新的构建变体时，Android Studio 不会为您创建源集目录，但会为您提供几个选项，帮助您创建。例如，要为“调试”构建类型只创建 `java/` 目录，请执行以下操作：

1. 打开 **Project** 窗格并从窗格顶端的下拉菜单中选择 **Project** 视图。
2. 导航至 `MyProject/app/src/`。
3. 右键点击 `src` 目录并选择 **New** > **Folder** > **Java Folder**。
4. 从 **Target Source Set** 旁边的下拉菜单中，选择 **debug**。
5. 点击 **Finish**。

Android Studio 将会为您的调试构建类型创建源集目录，然后在该目录内部创建 `java/` 目录。或者，在针对特定的构建变体向您的项目中添加新文件时，您也可以让 Android Studio 为您创建目录。例如，要为“调试”构建类型创建 XML 值文件：

1. 在相同的 **Project** 窗格中，右键点击 `src` 目录并选择 **New** > **XML** > **Values XML File**。
2. 为 XML 文件输入名称或保留默认名称。
3. 从 **Target Source Set** 旁边的下拉菜单中，选择 **debug**。
4. 点击 **Finish**。

由于“调试”构建类型被指定为目标源集，Android Studio 会在创建 XML 文件时自动创建必要的目录。最终的目录结构看上去应该类似于图 2。

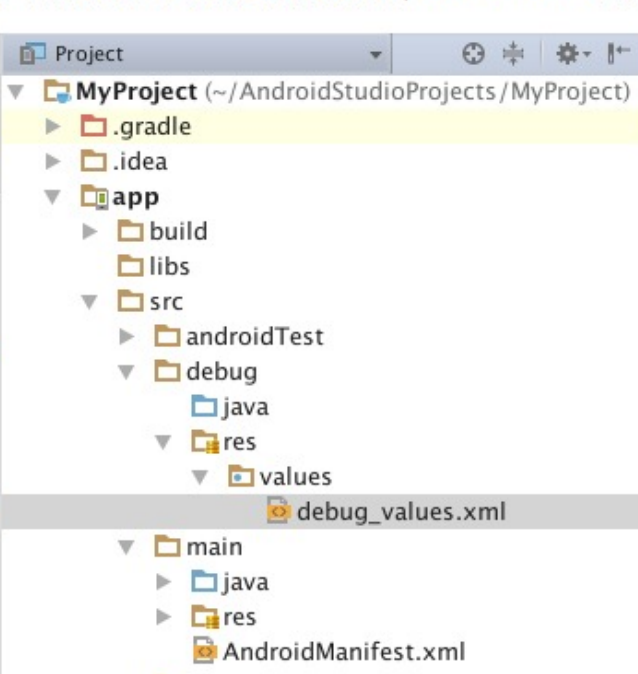

图 2. 调试构建类型的新源集目录。

按照同样的方法，您还可以为产品风味创建源集目录（例如 `src/demo/`），为构建变体创建源集目录（例如 `src/demoDebug/`）。此外，您还可以创建针对特定构建变体的测试源集，例如 `src/androidTestDemoDebug/`。如需了解更多信息，请转至测试源集。

更改默认源集配置

如果您的源未组织到 Gradle 期望的默认源集文件结构中（如上面的创建源集部分中所述），您可以使用 `sourceSets {}` 代码块更改 Gradle 希望为源集的每个组件收集文件的位置。您不需要重新定位文件；只需要为 Gradle 提供相对于模块级 `build.gradle` 文件的路径，Gradle 应当可以在此路径下为每个源集组件找到文件。要了解您可以配置哪些组件，以及是否可以将其映射到多个文件或目录，请阅读适用于 Gradle 的 Android 插件参考。

下面的代码示例可以将 `app/other/` 目录中的源映射到 `main` 源集的某些组件，并更改 `androidTest` 源集的根目录。

```
android {
  ...
  sourceSets {
    // Encapsulates configurations for the main source set.
    main {
      // Changes the directory for Java sources. The default directory is
      // 'src/main/java'.
      java.srcDirs = ['other/java']

      // If you list multiple directories, Gradle uses all of them to collect
      // sources. Because Gradle gives these directories equal priority, if
      // you define the same resource in more than one directory, you get an
      // error when merging resources. The default directory is 'src/main/res'.
      res.srcDirs = ['other/res1', 'other/res2']

      // Note: You should avoid specifying a directory which is a parent to one
      // or more other directories you specify. For example, avoid the following:
      // res.srcDirs = ['other/res1', 'other/res1/layouts', 'other/res1/strings']
      // You should specify either only the root 'other/res1' directory, or only the
      // nested 'other/res1/layouts' and 'other/res1/strings' directories.

      // For each source set, you can specify only one Android manifest.
      // By default, Android Studio creates a manifest for your main source
      // set in the src/main/ directory.
      manifest.srcFile 'other/AndroidManifest.xml'
      ...
    }

    // Create additional blocks to configure other source sets.
    androidTest {

      // If all the files for a source set are located under a single root
      // directory, you can specify that directory using the setRoot property.
      // When gathering sources for the source set, Gradle looks only in locations
      // relative to the root directory you specify. For example, after applying the
      // configuration below for the androidTest source set, Gradle looks for Java
      // sources only in the src/tests/java/ directory.
      setRoot 'src/tests'
      ...
    }
  }
}
...
```

使用源集构建

您可以使用源集目录包含您希望仅针对某些配置打包的代码和资源。例如，如果您要构建“演示调试”构建变体（它是“演示”产品风味和“调试”构建类型的合体），Gradle 会查看这些目录并赋予以下优先级顺序：

1. `src/demoDebug/`（*构建变体源集*）
2. `src/debug/`（*构建类型源集*）
3. `src/demo/`（*产品风味源集*）
4. `src/main/`（*主源集*）

> **注**：如果您组合多个产品风味，产品风味之间的优先级将由它们所属的风味维度决定。在列示具有 `android.flavorDimensions` 属性的风味维度时，所列示的第一个风味维度中的产品风味比第二个维度中的产品风味拥有更高的优先级，以此类推。此外，与属于各个产品风味的源集相比，您为产品风味组合创建的源集拥有更高的优先级。

上面列出的顺序决定了在 Gradle 合并代码和资源时哪个源集具有较高的优先级。由于 `demoDebug/` 源集目录很可能包含特定于该构建变体的文件，如果 `demoDebug/` 包含在 `debug/` 中也有定义的文件，Gradle 将使用 `demoDebug/` 源集中的文件。同样，Gradle 会为构建类型和产品风味源集中的文件赋予比 `main/` 中相同文件更高的优先级。Gradle 在应用以下构建规则时会考虑此优先级顺序：

- 一起编译 `java/` 目录中的所有源代码以生成单一的输出。

  > **注**：对于给定的构建变体，如果找到两个或两个以上定义同一 Java 类的源集目录，Gradle 就会引发一个构建错误。例如，在构建调试 APK 时，您不能同时定义 `src/debug/Utility.java` 和 `src/main/Utility.java`。这是因为 Gradle 会在构建过程中检查这两个目录并引发“duplicate class”错误。如果针对不同的构建类型需要不同版本的 `Utility.java`，您可以让每个构建类型定义其自己的文件版本，而不将其包含在 `main/` 源集中。

- 所有清单合并为单个清单。将按照上述列表中的相同顺序指定优先级。也就是说，某个构建类型的清单设置会替换某个产品风味的清单设置，依此类推。如需了解更多信息，请阅读合并清单。
- 同样，`values/` 目录中的文件也会合并在一起。如果两个文件同名，例如存在两个 `strings.xml` 文件，将按照上述列表中的相同顺序指定优先级。也就是说，在构建类型源集中的文件中定义的值将会替换产品风味中同一文件中定义的值，依此类推。
- `res/` 和 `asset/` 目录中的资源将打包到一起。如果两个或两个以上的源集中定义有同名资源，将按照上述列表中的相同顺序指定优先级。
- 最后，在构建 APK 时，Gradle 会为随库模块依赖项包含的资源和清单分配最低的优先级。

声明依赖项

下面的示例可以在 `app/` 模块的 `build.gradle` 文件中声明三种不同类型的直接依赖项：

```
android {...}
...
dependencies {
    // The 'compile' configuration tells Gradle to add the dependency to the
    // compilation classpath and include it in the final package.

    // Dependency on the "mylibrary" module from this project
    compile project(":mylibrary")

    // Remote binary dependency
    compile 'com.android.support:appcompat-v7:25.3.1'

    // Local binary dependency
    compile fileTree(dir: 'libs', include: ['*.jar'])
}
```

下面逐一介绍了每种直接依赖项。

模块依赖项

> `compile project(':mylibrary')` 行声明了一个名为“mylibrary”的本地 Android 库模块作为依赖项，并要求构建系统在构建应用时编译并包含该本地模块。

远程二进制依赖项

> `compile 'com.android.support:appcompat-v7:25.3.1'` 行会通过指定其 JCenter 坐标，针对 Android 支持库的 25.3.1 版本声明一个依赖项。默认情况下，Android Studio 会将项目配置为使用顶级构建文件中的 JCenter 存储区。当您将项目与构建配置文件同步时，Gradle 会自动从 JCenter 中抽取依赖项。或者，您也可以通过使用 SDK 管理器下载和安装特定的依赖项。

本地二进制依赖项

> `compile fileTree(dir: 'libs', include: ['*.jar'])` 行告诉构建系统在编译类路径和最终的应用软件包中包含 `app/libs/` 目录内的任何 JAR 文件。如果您有模块需要本地二进制依赖项，请将这些依赖项的 JAR 文件复制到项目内部的 `<moduleName>/libs` 中。

该模块的某些直接依赖项可能会有其自己的依赖项，这称为该模块的*传递依赖项*。Gradle 将会自动为您收集并添加这些传递依赖项，而不必手动逐一加以声明。适用于 Gradle 的 Android 插件提供了一项有用的 Gradle 任务，可为每个构建变体和测试源集生成依赖项树，因此，您可以轻松地可视化模块的直接和传递依赖项。要生成此报告，请继续如下操作：

1. 点击 IDE 窗口右侧的 **Gradle** Gradle 。
2. 导航至 **MyApplication > Tasks > android** 并双击 **androidDependencies**。
3. 要查看报告，请点击 IDE 窗口底部的 **Gradle Console** Gradle Console 。

以下示例报告显示了调试构建变体的依赖项树，并包含前一示例中的本地模块依赖项和远程依赖项。

```
Executing tasks: [androidDependencies]
:app:androidDependencies
debug
/**
 * Both the library module dependency and remote binary dependency are listed
 * with their transitive dependencies.
 */
+--- MyApp:mylibrary:unspecified
|    \--- com.android.support:appcompat-v7:25.3.1
|         +--- com.android.support:animated-vector-drawable:25.3.1
|         |    \--- com.android.support:support-vector-drawable:25.3.1
|         |         \--- com.android.support:support-v4:25.3.1
|         |              \--- LOCAL: internal_impl-25.3.1.jar
|         +--- com.android.support:support-v4:25.3.1
|         |    \--- LOCAL: internal_impl-25.3.1.jar
|         \--- com.android.support:support-vector-drawable:25.3.1
|              \--- com.android.support:support-v4:25.3.1
|                   \--- LOCAL: internal_impl-25.3.1.jar
\--- com.android.support:appcompat-v7:25.3.1
     +--- com.android.support:animated-vector-drawable:25.3.1
     |    \--- com.android.support:support-vector-drawable:25.3.1
     |         \--- com.android.support:support-v4:25.3.1
     |              \--- LOCAL: internal_impl-25.3.1.jar
     +--- com.android.support:support-v4:25.3.1
     |    \--- LOCAL: internal_impl-25.3.1.jar
     \--- com.android.support:support-vector-drawable:25.3.1
          \--- com.android.support:support-v4:25.3.1
               \--- LOCAL: internal_impl-25.3.1.jar
...
```

如需了解在 Gradle 中管理依赖项的详细信息，请参阅《Gradle 用户指南》中的依赖项管理基础知识。

配置依赖项

您可以使用特定的配置关键字告诉 Gradle 如何以及何时使用某个依赖项，例如前述示例中的 `compile` 关键字。下面介绍了您可以用来配置依赖项的一些关键字：

`compile`

> 指定编译时依赖项。Gradle 将此配置的依赖项添加到类路径和应用的 APK。这是默认配置。

`apk`

> 指定 Gradle 需要将其与应用的 APK 一起打包的仅运行时依赖项。您可以将此配置与 JAR 二进制依赖项一起使用，而不能与其他库模块依赖项或 AAR 二进制依赖项一起使用。

`provided`

> 指定 Gradle *不*与应用的 APK 一起打包的编译时依赖项。如果运行时无需此依赖项，这将有助于缩减 APK 的大小。您可以将此配置与 JAR 二进制依赖项一起使用，而不能与其他库模块依赖项或 AAR 二进制依赖项一起使用。

此外，您可以通过将构建变体或测试源集的名称应用于配置关键字，为特定的构建变体或测试源集配置依赖项，如下例所示。

```
dependencies {
    ...
    // Adds specific library module dependencies as compile time dependencies
    // to the fullRelease and fullDebug build variants.
    fullReleaseCompile project(path: ':library', configuration: 'release')
    fullDebugCompile project(path: ':library', configuration: 'debug')

    // Adds a compile time dependency for local tests.
    testCompile 'junit:junit:4.12'

    // Adds a compile time dependency for the test APK.
    androidTestCompile 'com.android.support.test.espresso:espresso-core:2.2.2'
}
```

配置签署设置

除非您为发布构建显式定义签署配置，否则，Gradle 不会签署发布构建的 APK。您可以轻松创建发布密钥并使用 Android Studio 签署发布构建类型。

要使用 Gradle 构建配置为您的发布构建类型手动配置签署配置：

1. *创建密钥库*。**密钥库**是一个二进制文件，它包含一组私钥。您必须将密钥库存放在安全可靠的地方。
2. *创建私钥*。**私钥**代表将通过应用识别的实体，如某个人或某家公司。
3. 将签署配置添加到模块级 `build.gradle` 文件中：

   ```
   ...
   android {
       ...
       defaultConfig {...}
       signingConfigs {
           release {
               storeFile file("myreleasekey.keystore")
               storePassword "password"
               keyAlias "MyReleaseKey"
               keyPassword "password"
           }
       }
       buildTypes {
           release {
               ...
               signingConfig signingConfigs.release
           }
       }
   }
   ```

要生成签署的 APK，请从菜单栏中选择 **Build > Generate Signed APK**。现在，`app/build/apk/app-release.apk` 中的软件包已使用您的发布密钥进行签署。

> **注**：将发布密钥和密钥库的密码放在构建文件中并不安全。作为替代方案，您可以将此构建文件配置为通过环境变量获取这些密码，或让构建流程提示您输入这些密码。

要通过环境变量获取这些密码：

```
storePassword System.getenv("KSTOREPWD")
keyPassword System.getenv("KEYPWD")
```

要让构建流程在您要从命令行调用此构建时提示您输入这些密码：

```
storePassword System.console().readLine("\nKeystore password: ")
keyPassword System.console().readLine("\nKey password: ")
```

在完成此流程后，您可以分发您的应用并在 Google Play 上发布它。

> **警告**：将密钥库和私钥存放在安全可靠的地方，并确保您为其创建了安全的备份。如果您将应用发布到 Google Play，随后丢失了您用于签署应用的密钥，那么，您将无法向您的应用发布任何更新，因为您必须始终使用相同的密钥签署应用的所有版本。

签署 Android Wear 应用

发布 Android Wear 应用时，请将穿戴式设备应用打包到手持类应用中，因为用户无法直接在穿戴式设备上浏览和安装应用。这两个应用均必须进行签署。如需有关打包和签署 Android Wear 应用的更多信息，请参阅打包穿戴式设备应用。

Except as noted, this content is licensed under Creative Commons Attribution 2.5. For details and restrictions, see the Content License.