Broadview
www.broadview.com.cn

[PACKT] open source*
PUBLISHING
community experience distilled

Gradle for Android

Gradle for Android 中文版

通过Gradle为你的Android项目构建过程自动化

[美] Kevin Pelgrims 著
余小乐 译

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Gradle for Android

Gradle for Android 中文版

[美] Kevin Pelgrims 著
余小乐 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

Gradle是Android开发小组于2013年推出的Android 应用首选构建系统。Gradle可以很容易地扩展构建，并插入现有的构建过程中。它提供了一套类Groovy的DSL语言，用于申明构建和创建任务，让依赖管理变得更加简单。此外，它还是完全免费和开源的。

本书共9章，依次介绍了Gradle的基础知识、基本自定义构建、依赖管理、创建构建Variants、管理多模块构建、运行测试、创建任务和插件、设置持续集成，以及高级自定义构建。

本书是为那些希望成为构建能手的Android开发者编写的。

版权贸易合同登记号 图字：01-2016-3998

图书在版编目（CIP）数据

Gradle for Android中文版 /（美）凯文·贝利格里姆斯（Kevin Pelgrims）著；余小乐译. —北京：电子工业出版社，2016.10
书名原文：Gradle for Android
ISBN 978-7-121-30015-8

Ⅰ. ①G… Ⅱ. ①凯… ②余… Ⅲ. ①移动终端－应用程序－程序设计 Ⅳ. ①TN929.53

中国版本图书馆CIP数据核字（2016）第238765号

责任编辑：安 娜
印 刷：三河市鑫金马印装有限公司
装 订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
北京市海淀区万寿路173信箱 邮编：100036
开 本：787×980 1/16 印张：9.25 字数：180千字
版 次：2016年10月第1版
印 次：2016年10月第1次印刷
定 价：49.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888
质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。
本书咨询联系方式：010-51260888-819 faq@phei.com.cn。

关于作者

Kevin Pelgrims 是比利时的一名 .NET 开发者。在 Windows 和 Web 开发企业客户端工作几年后，他搬到了哥本哈根，成为了创办社区的一分子。在那里，他开始在移动平台上工作，并且在一年内成为了几款 Android 和 Windows Phone 应用的开发主管。Kevin 也是 Android 开发小组在哥本哈根的演讲常客。业余时间，他不仅维护了数个 Android 应用，还喜欢试用不同的服务器端技术。当他不忙于写代码时，他最喜欢的事是给他的妻子和他们的猫弹吉他。

如果想了解 Kevin 的最新进展，可以看看他的博客：`http://www.kevinpelgrims.com`，或者在 Twitter 上关注 `@kevinpelgrims`。

若是没有我漂亮的妻子——路易斯的支持，我不可能完成这本书的写作。她不仅忍受了我在电脑上花费的每个夜晚和周末，还校对了整本书，提高了本书语言和语法的准确度。

我要感谢 Packt 出版社能够给我写这本书的机会，特别是 Nikhil Karkal 和 Prachi Bisht 在整个写作过程中对我的信任和指导。感谢我的朋友 Koe Metsu 的帮助，他的评论使得本书的解释和例子更加容易理解。感谢 Anders Bo Pedersen 和 Emanuele Zattin 对本书提出的改进建议。

我还要感谢审稿人：Peter Friese、Jonathan H. Kau、Takuya Miyamoto、和 Marco A. Rodriguez-Suarez、Felix Schulze、Hugo Visser。他们的加入使得本书得到了显著提升。最后，但并非最不重要的，特别感谢我们的猫 Jigsaw，它坐在我的腿上，让我无法起身，使我得以保持专注并坚持下去。

关于审稿人

Peter Friese 是英国伦敦谷歌开发关系团队的一员，负责开发推广，他是一个专职的开源贡献者、博客作者和公众演讲家。他在 Twitter 上的名称是 `@peterfriese`，在 Google+ 上的名称是 `+PeterFriese`。你可以在 `http://www.peterfriese.de` 上找到他的博文。

Jonathan H. Kau 是一个经验丰富的 Android 开发者，在 Google Paly 上已发布过多款独立应用，并凭借 Android 技术参与过众多黑客马拉松项目。

之前，他供职于 Yelp 的移动应用团队，并且是 Propeller 实验室的一名合同工。现在，Jonathan 在 Shyp 的 Android 应用和相应的后端 API 工程团队工作。

> 非常感谢 Packt 出版社给我这个机会来审阅本书，对于广大开发人员来说，本书会让 Gradle 的用法简单化。

Takuya Miyamoto 是一个全栈工程师，在设计、实现和维护电商和 SNS 等多个版本的 Web 服务和 API 方面有 6 年经验。在 Android 开发方面，他也有一定的经验。他曾在一家企业担任过高级工程师，在一家初创公司担任过首席工程师，而现在他是一名自由开发者。

Marco A. Rodriguez-Suarez 是 Snapwire 手机端的领导人，他负责协调平台和市场，致力于连接世界各地的品牌摄影师和企业。在此之前，他在移动端的各个项目中从事咨询工作，内容包括从视频流到游戏仿真。从 2008 年的第一个 Android 发行版开始，他就开始致力于 Android 开发了，并热枕于移动开发。他热衷于构建系统，在 Gradle、Maven、Ant 方面有着广泛的经验。Marco 还取得了来自美国加州大学圣巴巴拉分校的电气工程硕士学位。

Felix Schulze 是 AutoScout24 移动软件开发部门的领导人，负责 Android 和 iOS 方面的开发工作。关于应用开发中的持续集成部分他同样有大量的演讲，并且对开源工具有所贡献。他的 Twitter ID 是 `@x2on`，你也可以查看他的个人网站 `www.felixschulze.de`。

Hugo Visser 是一个有着多年开发经验的软件开发工程师，范围从服务器端到桌面，从 Web 端到移动端。自 Android 开源以来，他就一直密切关注着 Android 平台的发展，在 2009 年，他开发了第一个应用 Rainy Days，该应用在全球已被下载超过 100 万次。

他经营着自己的公司 Little Robots，该公司致力于应用开发和其他一些在 Android 平台上的巧妙用途。他被 Google 公司评为 Android 方面的开发专家，同时也是荷兰 Android 用户组社区（Dutch Android User Group）的组织者。这个社区每月举办一次会议，以便荷兰的 Android 专业人士能够会面并彼此分享知识。

前言

Android 应用的构建过程是一个非常复杂的过程，涉及很多工具。首先，所有的资源文件都会被编译，并且在一个 R 文件中引用。然后 Java 代码被编译，之后通过 dex 工具转换成 dalvik 字节码。最后这些文件会被打包成一个 APK 文件，这个应用被最终安装在设备中之前，APK 文件会被一个 debug 或者 release 的 key 文件签名。

这些步骤如果由人工去完成，不仅烦琐，而且费时。幸运的是，Android 开发小组致力于提供关注这一打包过程的开发者工具，2013 年他们推出了 Gradle，作为 Android 应用新的首选构建系统。Gradle 设计的方式使得它可以很容易地扩展构建和插入到现有的构建过程中。它提供了一套类 Groovy 的 DSL 语言，用于申明构建和创建任务，让依赖管理变得更加简单。此外，它还是完全免费和开源的。

现在，许多 Android 开发者已经切换到 Gradle，但是仍有大部分人不知道如何很好地利用它，不清楚为什么几行代码就可以实现。本书旨在帮助那些开发人员，将他们变成 Gradle 的使用者。本书从 Gradle 基础知识开始，然后介绍依赖、构建 variants、测试、创建任务等。

本书涵盖内容

第 1 章，Gradle 和 Android Studio 入门，解释为什么 Gradle 很有用、如何利用 Android Studio 开始工作，以及 Gradle Wrapper 是什么。

第 2 章，基本自定义构建，深入到 Gradle 构建文件和任务，展示如何做简单的自定义构建过程。

第 3 章，依赖管理，展示如何使用依赖，包括本地和远程依赖，并解释了依赖相关的概念。

第 4 章，*创建构建 Variants*，介绍构建类型和 product flavors，解释其不同之处，并展示如何使用签名参数。

第 5 章，*管理多模块构建*，解释如何管理应用、依赖库、测试模块，以及如何集成它们。

第 6 章，*运行测试*，介绍了一些用于单元测试和功能测试的测试框架，以及如何进行自动化测试并获得测试覆盖率报告。

第 7 章，*创建任务和插件*，解释 Groovy 的基础知识，并展示了如何创建自定义任务，以及如何将其 hook 到 Android 构建进程。本章还介绍了如何创建一个可重用的插件。

第 8 章，*设置持续集成*，使用最常用的 CI 系统，提供自动化构建指导。

第 9 章，*高级自定义构建*，展示一些提示和技巧来缩小 APK 的大小，加快构建进程，基于密集度或平台来分割 APK。

你还需要什么

为了演示所有的例子，你需要一台装有 Windows、Mac OS X 或者 Linux 的电脑。你还需要安装 Java 开发组件，建议你安装 Android Studio，因为其在大部分章节都有提及。

这本书是为谁而写的

这本书是为了那些想更好地理解构建系统、成为构建进程能手的 Android 开发者而编写的。我们将从 Gradle 的基础知识讲起，然后是创建自定义任务和插件，再到自动生成构建进程。我们假设你熟悉 Android 平台的开发。

约定

在本书中，你会发现很多文本样式用于区分不同的信息。下面是这些样式的一些例子和它们的含义。

文本中的代码词汇、数据库表名、文件夹名、文件名、文件扩展、路径名称、假设的 URL 地址、用户输入以及 Twitter 用户定位如下所示。（每个 `build.gradle` 文件代表着一个项目。）

代码块的样式如下：

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

每个命令行输入或输出命令如下所示：

```
$ gradlew tasks
```

新的术语和重要的单词将会是黑体。例如，你在屏幕上看到的单词，例如菜单或弹窗，将会这样显示："在 Android Studio 中，你可以通过在屏幕上点击 **Start a new Android Studio Project**" 开始一个新的项目。

警告或重要注释将会像这样在一个框中显示。

提示和技巧会像这样显示。

用户反馈

非常欢迎读者的反馈。这可以让我们了解你对本书的看法——哪些你喜欢，哪些你不喜欢。读者的反馈可以帮助我们调整内容，从而使读者能够获得更多收益。

准备发送给我们的一般的反馈，只需发送电子邮件到 `feedback@packtpub.com`，并在邮件内容中指出书名即可。

如果你在某个方面有所专长，并且你有兴趣写书或者为一本书贡献内容，请阅读我们的作者向导：`www.packtpub.com/authors`。

客户支持

现在，你可以自豪地说你是一本 Packt 发行的书的主人了，并且，我们还有很多东西可以

帮助你充分利用它。

下载源码

你可以在 http://www.packtpub.com 站点上使用你的账号下载所有你购买过的书籍的示例代码。如果你是在其他地方购买的本书，那么你可以访问 http://www.packtpub.com/support，注册后，我们会把代码发送给你。

勘误

尽管我们已经尽力确保我们内容的准确性，但错误在所难免。如果你发现我们书中的错误——可能是文字错误或者代码错误——如果你将它报告给我们，我们将不胜感激。这样做，不仅可以让其他读者免于受挫，还能帮助我们改进本书的后续版本。如果你发现任何错误，请访问 http://www.packtpub.com/submit-errata 进行报告，在以上链接的网站中，选择书名，然后单击 **Errata Submission** 表单链接，填写你所发现的错误细节。一旦你发现的错误被确认，你提交的内容将会被接受，勘误信息将被上传到我们的网站或者添加到对应标题相关的勘误小节列表中。

如果想查询以前提交的勘误信息，请访问 https://www.packtpub.com/books/content/support，并在搜索区域填写书名。需要的信息将会出现在勘误列表中。

侵权

互联网上的盗版问题是所有媒体一直存在的问题。对于 Packt，我们非常重视保护我们的版权。如果你从互联网上遇到任何我们产品的非法拷贝，请立即为我们提供网址或者网站名称，以便我们寻求补救。

请通过电子邮件 copyright@packtpub.com 联系我们并附带盗版资料的链接。

非常感谢你帮助保护我们的作者，而我们将为你带来有价值的内容。

问题

如果你对本书有任何方面的疑问，都可以通过电子邮件 questions@packtpub.com 联系我们，我们将尽最大的努力解决这个问题。

目录

1

Gradle和Android Studio入门

当谷歌推广 Gradle 和 Android Studio 的时候，他们立下了一些目标。他们想让代码复用、构建 variant、配置和定制构建过程变得更加简单。最重要的是，他们想整合一个好用的 IDE，让构建系统不再依赖于 IDE，并使得通过命令行或者持续集成服务器运行 Gradle 生成的结果，和在 Android Studio 中执行构建生成的结果一致。

整本书我们都会谈到 Android Studio，因为其常常提供一些构建项目的简单方式，以及构建 variant 等。如果你还没有安装 Android Studio，那么你可以通过 Android 开发者网站下载：`http://developer.android.com/sdk/index.html`。

本章我们将讲解以下内容：

- 了解 Android Studio
- 理解 Gradle 基础
- 创建新的项目
- 开始使用 Gradle wrapper
- 从 Eclipse 迁移到 Android Studio

1.1 Android Studio

2013 年 5 月，谷歌发布了 Android Studio 开发工具（作为前期可访问的预览版），并对 Gradle 进行了支持。Android Studio 基于 JetBrains 的 IntelliJ IDEA，但是其针对 Android 开发进

行了相关的特定设计。其不仅适用于 Linux、Mac OS X 和 Microsoft Windows 操作系统，而且还是免费的。

与 Eclipse 相比，Android Studio 拥有更好的用户交互界面、更好的内存监视器、更好的字符串翻译编辑器，对可能出现的 Android 的特定问题给出警告，并提供了一系列针对 Android 开发者的特性。除了常用的 Project 视图和 Packages 视图外，Android Studio 还针对 Android 项目提供了 Android 视图。该视图对 Gradle 脚本、drawables 和其他资源进行了分组。随着 Android Studio 稳定版 1.0 的发布，谷歌不再针对 Eclipse 的 ADT 进行更新，并建议所有开发者都将开发工具转移到 Android Studio 上。这意味着谷歌不再为 Eclipse 提供新的功能。目前，所有 IDE 相关工具的开发都已基于 Android Studio，如果你还在使用 Eclipse，那么是时候改变了，只有这样才不会被时代抛在身后。

一个简单的 Android 应用项目在 Android Studio 中的样子如图 1-1 所示。

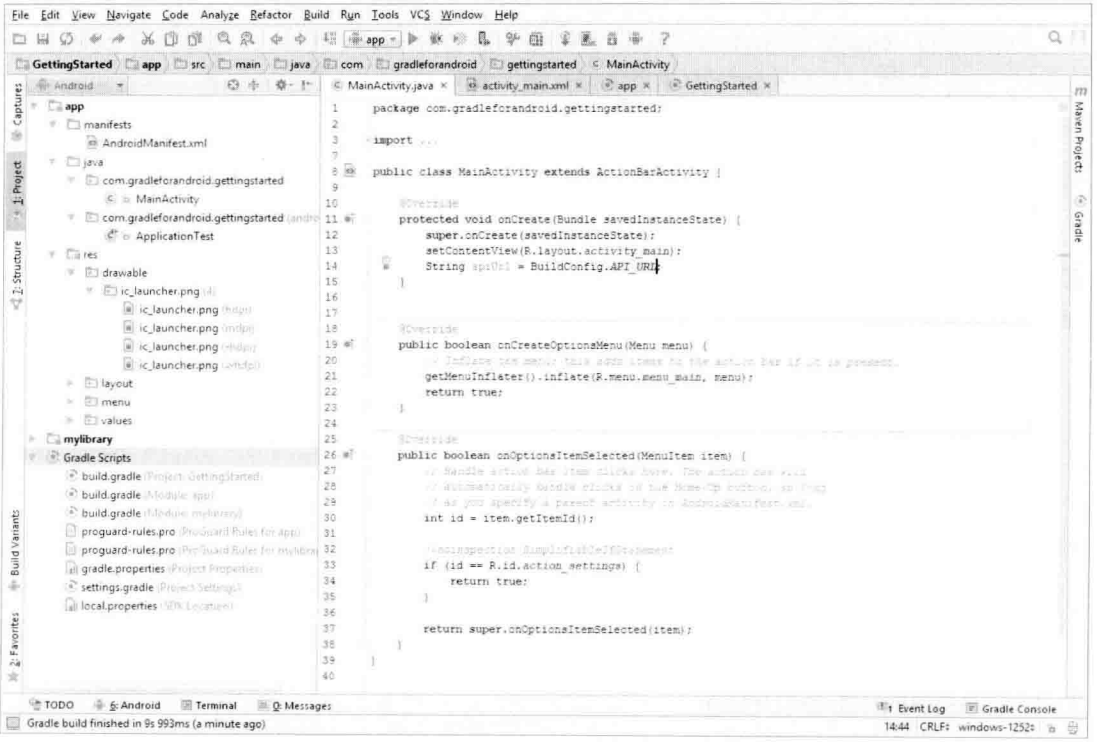

图 1-1　Android Studio

保持版本更新

Android Studio 有四个不同的更新途径：

- 金丝雀版为最新版，但是可能包含 bug。
- 开发版每个月或多或少都需要更新。
- 测试版有完整的更新，但是仍然可能包含 bug。
- Android Studio 默认为稳定版，作为功能全面测试的版本，它应该没有 bug。

默认情况下，Android Studio 会在每次启动的时候，检查是否有版本更新，并且及时通知你。

当你第一次启动 Android Studio 时，它会打开向导来帮助你设置环境变量，确保你有最新的 Android SDK 以及必要的 Google 依赖库。它也会让你选择是否创建虚拟机（Android Virtual Device，AVD），创建虚拟机后，你就可以在虚拟设备上运行 App 了。

1.2 理解Gradle基础

如果你想用 Gradle 构建你的 Android 项目，那么你需要创建一个构建脚本，这个脚本通常被叫作 `build.gradle`。注意，当我们学完 Gradle 基础后你会发现，Gradle 有约定优于配置的原则，即为设置和属性提供默认值。这使得相比 Ant 或者 Maven，它更容易上手使用，而 Ant 在 Android 项目的构建上存活了很长时间。你大可不必完全遵循这些约定，通常情况下，它们都是可覆盖的。

Gradle 构建脚本的书写没有基于传统的 XML 文件，而是基于 Groovy 的领域专用语言（DSL）。Groovy 是一种基于 Java 虚拟机的动态语言。Gradle 团队认为，基于动态语言的 DSL 语言与 Ant 或者任何基于 XML 的构建系统相比，优势都十分显著。

这并不意味着在我们开始学习构建脚本之前，需要学习 Groovy。Groovy 代码非常易读，如果你学习过 Java，那么学习 Groovy 的曲线不会陡峭。如果你想创建自己的任务和插件（这些我们会在稍后的章节谈到），那么对 Groovy 有更深层次的理解就显得尤为重要。然而，因为其是基于 Java 虚拟机的，所以你完全可以用 Java 或者其他基于 Java 虚拟机的语言来编写你的自定义插件。

1.2.1 项目和任务

在 Gradle 中，最重要的两个概念是项目和任务。每一次构建都包括至少一个项目，每一个项目又包括一个或多个任务。每个 `build.gradle` 文件都代表着一个项目，任务定义在构

建脚本里。当初始化构建过程时，Gradle 会基于 build 文件组装项目和任务对象。一个任务对象包含一系列动作对象，这些动作对象之后会按顺序执行。一个单独的动作对象就是一个待执行的代码块，它和 Java 中的方法类似。

1.2.2 构建生命周期

执行一个 Gradle 构建的最简单的形式是，只执行任务中的动作，而这些任务又依赖于其他任务。为了简化构建过程，构建工具会新建一个动态的模型流，叫作 **Directed Acyclic Graph（DAG）**。这意味着所有的任务都会被一个接一个地执行，循环是不可能的。一旦一个任务被执行，其就不会被再次执行。那些没有依赖的任务通常会被优先执行。在构建的配置阶段会生成依赖关系图。一个 Gradle 的构建通常有如下三个阶段。

- 初始化：项目实例会在该阶段被创建。如果一个项目有多个模块，并且每一个模块都有其对应的 `build.gradle` 文件，那么就会创建多个项目实例。
- 配置：在该阶段，构建脚本会被执行，并为每个项目实例创建和配置任务。
- 执行：在该阶段，Gradle 将决定哪个任务会被执行。哪些任务被执行取决于开始该次构建的参数配置和该 Gradle 文件的当前目录。

1.2.3 构建配置文件

每一个基于 Gradle 构建的项目，都应该至少有一个 `build.gradle` 文件。Android 的构建文件中，有一些元素是必需的：

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

这就是实际构建配置的地方。在 repositories 代码块中，JCenter 库被配置为整个构建过程的依赖仓库。JCenter 是一个预配置的 Maven 仓库，不需要额外的配置。当然在 Gradle 中有多个仓库可供选择，你可以很容易地添加自己的本地或远程仓库。

构建脚本代码块在 Android 构建工具上定义了一个依赖，就像 Maven 的 artifact。这就是 Android 插件的来源，Android 插件提供了构建和测试应用所需要的一切。每一个 Android 项目

都应该申请该插件：

```
apply plugin: 'com.android.application'
```

插件用于扩展 Gradle 构建脚本的能力。在一个项目中应用一个插件，该项目就可以使用该插件预定义的一些属性和任务。

如果你正在构建一个依赖库，那么你需要声明 'com.android.library'，而不是 'com.android.application'。你不能在一个模块中同时使用它们，因为这会导致构建错误。一个模块可以是一个 Android 应用模块，或者是一个 Android 依赖模块，但不能二者都是。

当你使用 Android 插件时，不仅可以配置针对 Android 的特殊约定，还可以生成只应用于 Android 的任务。下面的 Android 代码片段是由插件来定义的，可以配置在每个项目中：

```
android{
    compileSdkVersion 22
    buildToolsVersion "22.0.1"
}
```

这是配置 Android 特殊约定参数的一部分。Android 插件为 Android 的开发需求提供了一整套 DSL。唯一需要的参数属性是编译目标和构建工具。编译目标由 compileSdkVersion 规定，compileSdkVersion 是用来编译应用的 SDK 版本。使用最新的 Android API 版本作为编译目标。

在 build.gradle 文件中还有很多自定义特性。我们将在第 2 章中讲解最重要的特性，其他特性将在本书的其余部分介绍。

1.2.4 项目结构

和 Eclipse 项目相比，Android 项目的文件夹结构的改变相当大。正如前面所提到的，Gradle 的优点是约定优于配置，其同样适用于文件夹结构。

一个简单的应用的文件夹结构如图 1-2 所示。

Gradle 项目通常会在根目录下创建一个 gradle 脚本文件，这使得其在后续阶段新增模块变得更加简单。所有的应用源代码都在 app 文件夹下。该文件夹名称默认情况下也是模块的名称，但是其不必被命名为 app。例如，当你使用 Android Studio 创建一个手机应用和一个 Android 智能可穿戴手表应用项目时，默认情况下模块会被叫作 application 和 wearable。

```
MyApp
    ├── build.gradle
    ├── settings.gradle
    └── app
        ├── build.gradle
        ├── build
        ├── libs
        └── src
            └── main
                ├── java
                │   └── com.package.myapp
                └── res
                    ├── drawable
                    ├── layout
                    └── etc.
```

图 1-2　文件夹结构

Gradle 使用一个叫作源集（source set）的概念。Gradle 的官方文档是这样解释的：一个源集就是一组源文件，它们会被一起执行和编译。对于一个 Android 项目而言，main 就是一个源集，它包含了所有的源代码和资源，是应用程序默认版本的源集。当你开始为 Android 应用编写测试代码时，你可以把所有测试相关的源代码都放在一个独立的源集中，该源集被叫作 androidTest，只包含测试代码。

表 1-1 是一个 Android 应用中主要的文件夹的简短概述。

表 1-1　Android 应用中主要的文件夹的简短描述

目录	内容
/src/main/java	应用的源代码
/src/main/res	应用相关的资源（drawables、layouts、strings 等）
/libs	外部库（.jar 或者 .aar）
/build	构建过程的输出

1.3　创建新项目

在 Android Studio 中，你可以通过在启动界面单击 **Start a new Android Studio project** 或导航到 **File|New Project...** 来创建新的项目，如图 1-3 所示。

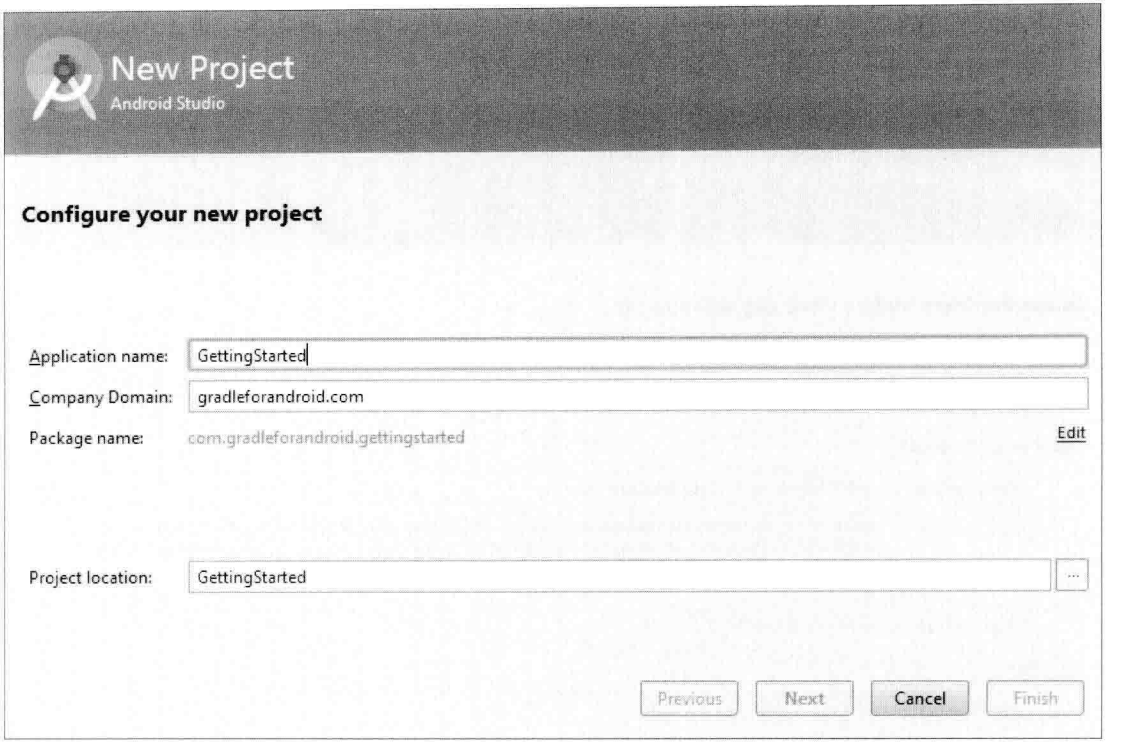

图 1-3 创建新项目

在 Android Studio 中创建新项目时会启动一个向导，它会帮助你设置好一切。向导的第一屏用于设置应用名称和公司域名。应用程序的名称就是在应用安装时，被用作应用的名称和 toolbar 的默认名称。公司域名结合应用程序名称，可以确定包的名称，包的名称是所有 Android 应用的唯一标识符。如果你想要一个不同的包的名称，那么你可以单击 **Edit** 去更改它。你也可以更改项目在你的硬盘上的位置。

在通过向导执行下面所有步骤之前，不会生成任何文件，因为下面几个步骤将会定义哪些文件需要被创建。

Android 不仅可以运行在手机和平板上，还支持很多广泛的电子设备，例如 TV、手表和眼镜等。图 1-4 所示的向导页可以帮助你在项目中设置你的所有目标电子设备。你可以选择开发包含的依赖包和构建插件。在这里，你可以决定是否只制作一个手机和平板应用，还是想包含一个 Android TV 模块、一个 Android Wear 模块或者一个 Google Glass 模块。在这之后，你仍然可以添加这些模块，但是向导页可以让添加所有必需文件和配置变得更加简单。在这里，

你需要选择你想支持的 Android 版本号。如果你选择了 API 21 以下的版本，那么 Android 支持库（包括 appcompat 依赖库）将会自动被添加为依赖。

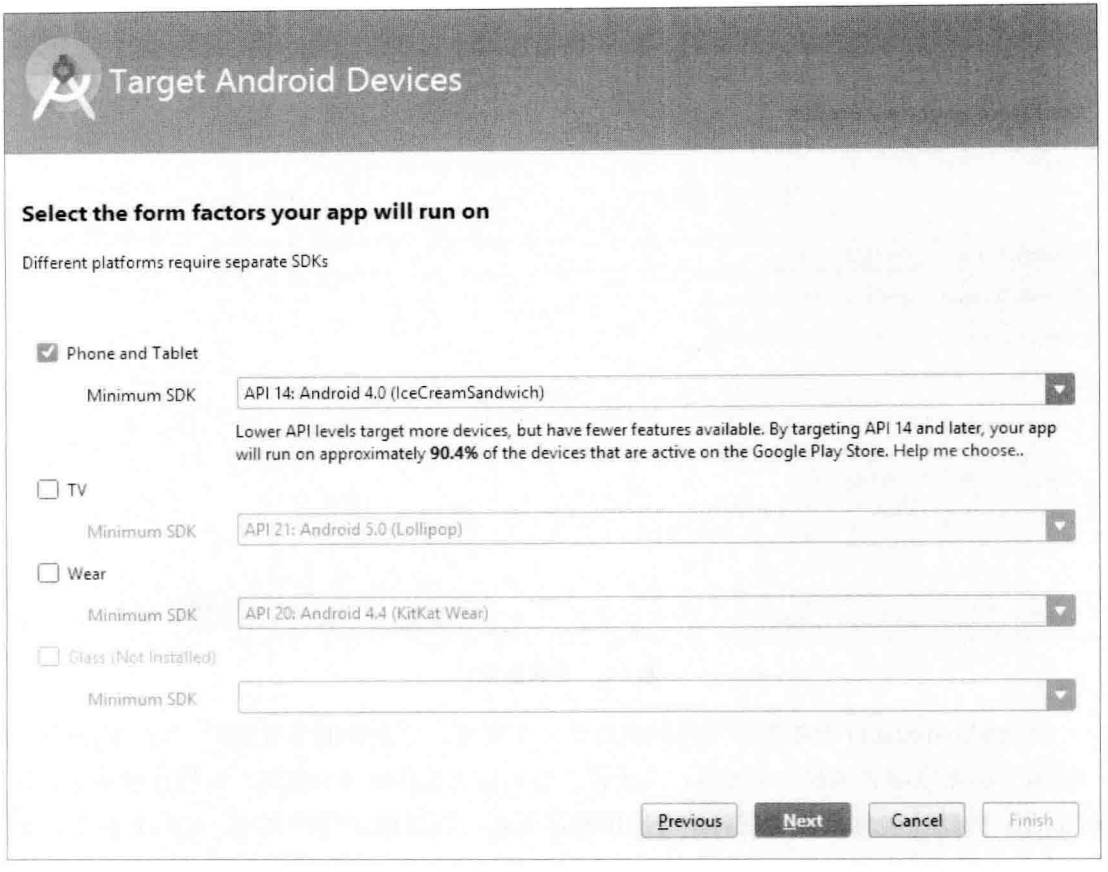

图 1-4 向导

下载示例代码

你可以从 http://www.packtpub.com 网站（通过你的账户）下载所有你购买过的 Packt 出版的书籍的示例代码文件。如果你在其他地方购买了这本书，那么你可以访问 http://www.packtpub.com/support，并注册，我们会直接通过邮件的方式将示例代码文件发送给你。

图 1-5 所示的向导页提示你添加一个 activity 活动页面，并提供了很多选择，所有这些都会生成相关代码，使得它更加容易上手。如果你选择让 Android Studio 为你生成一个 activity，

那么下一步就会输入该 activity 的类名、layout 文件名和菜单资源，以及这个 activity 的标题。

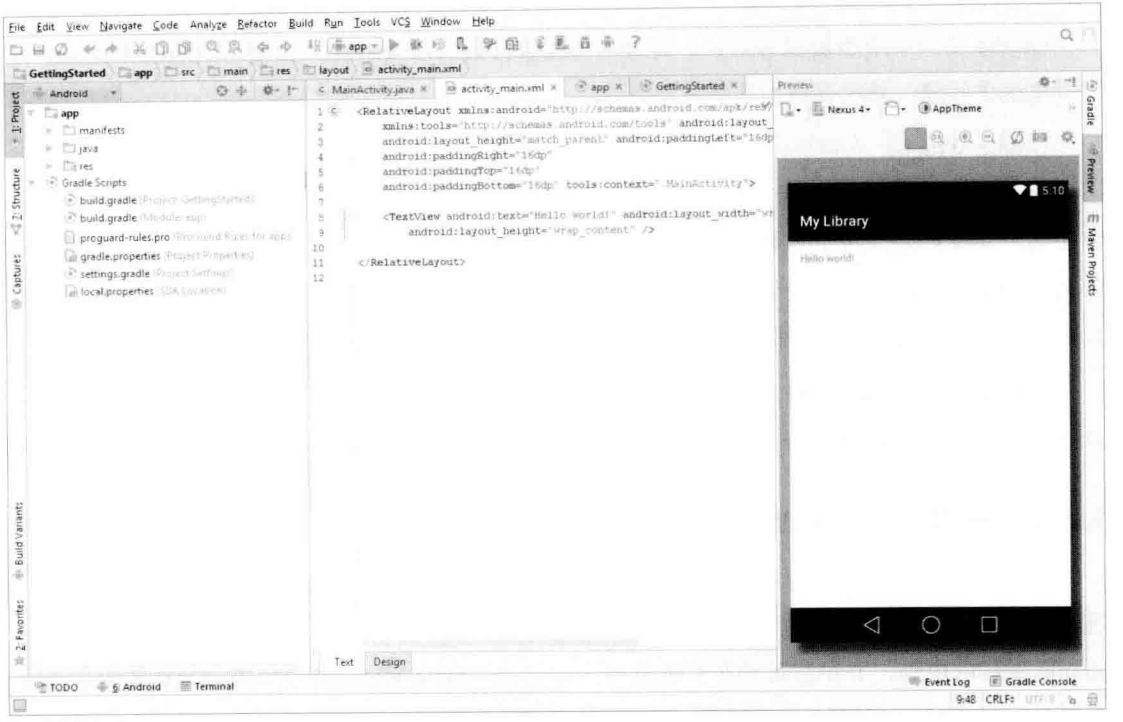

图 1-5　生成一个 activity

在你走完整个向导流程之后，Android Studio 会根据你在向导期间的选择，生成一个 activity 和一个 fragment 的源代码。为了构建该项目，Android Studio 还会生成基本的 Gradle 文件。你可以在整个项目的根目录下找到一个叫作 `settings.gradle` 的文件和一个叫作 `build.gradle` 的文件。在应用模块文件夹内，也有一个 `build.gradle` 文件。我们会在第 2 章中详细地介绍这些内容和创建这些文件的用途。

触发 Android Studio 构建任务的方式有以下几种：

- 在 **Build** 菜单里，你可以单击 **Make Project**，或者使用快捷键，在 PC 上是 Ctrl + F9，在 Mac OS X 上是 Cmd + F9 。
- 单击工具栏上的 **Make Project** 按钮。
- Gradle 工具窗列出了所有可用的 Gradle 任务。

你可以试着执行 Gradle 工具窗中的 `assembleDebug` 来构建项目，或者执行 `installDebug`，在设备或模拟器上安装应用。我们将在 1.4 节中讲解这些任务。

1.4 Gradle Wrapper入门

Gradle 是一个不断发展的工具，新版本可能会打破向后兼容性，而使用 Gradle Wrapper 可以避免这个问题，并能确保构建是可重复的。

Gradle Wrapper 为微软的 Windows 操作系统提供了一个 batch 文件，为其他操作系统提供了一个 shell 脚本。当你运行这段脚本时，需要的 Gradle 版本会被自动下载（如果它还不存在）和使用。其原理是，每个需要构建应用的开发者或自构建系统可以仅仅运行 Wrapper，然后由 Wrapper 搞定剩余部分。因此，我们不需在开发者机器或构建服务器上手动安装正确的 Gradle 版本，此外，Gradle 还建议把 Wrapper 文件添加到你的版本控制系统中。

运行 Gradle Wrapper 和直接运行 Gradle 没什么不同，你只需在 Linux 和 Mac OS X 上执行 `gradlew`，在 Microsoft Windows 中运行 `gradlew.bat` 来代替常规 `gradle` 命令即可。

1.4.1 获取 Gradle Wrapper

为了方便起见，每个新的 Android 项目都会包含 Gradle Wrapper，所以当你创建一个新项目时，你不必做任何事情就可以获得 Gradle Wrapper 的必要文件。当然，你也可以手动安装 Gradle 到你的电脑，并把它运用在你的项目上，但是 Gradle Wrapper 可以做同样的事情，并能确保 Gradle 版本的正确性。在使用 Gradle 构建和脱离 Android Studio 开发时，最好的方式就是使用 Wrapper。

你可以通过导航来到项目根目录，在 terminal 上运行 `./gradlew -v` 或在命令行上运行 `gradlew.bat -v` 来检查你的项目中的 Gradle Wrapper 是否可用。运行该命令将会显示 Gradle 的版本号和一些你额外设置的参数信息。如果你转换到一个 Eclipse 项目，那么默认情况下 Wrapper 将不会存在。这种情况下，你可以使用 Gradle 来生成 Wrapper，但是你首先需要安装 Gradle，然后才能得到 Wrapper。

Gradle 下载页面（`http://gradle.org/downloads`）有二进制文件和源代码文件的链接，如果你使用 Mac OS X 系统，则可以使用一个包管理器，例如 Homebrew。安装页面上有全部的安装说明（`http://gradle.org/installation`）。

在下载和安装完 Gradle 之后，把它添加到你的 `PATH` 下，创建一个包含下面 3 行的

build.gradle 文件：

```
task wrapper(type: Wrapper) {
       gradleVersion = '2.4'
}
```

之后，运行 gradle wrapper 来生成 Wrapper 文件。

在最近几个版本的 Gradle 中，你也可以不修改 build.gradle 文件来运行 Wrapper 任务，因为 Gradle 默认包含了这个任务。这种情况下，你可以将版本号作为 --gradle-version 的参数，就像这样：

```
$ gradle wrapper --gradle-version 2.4
```

如果你没有指定特定版本号，那么 Wrapper 版本会被配置为使用这个任务执行的 Gradle 对应的版本。

这些都是通过 wrapper task 生成的文件：

```
myapp/
  ├── gradlew
  ├── gradlew.bat
  └── gradle/wrapper/
      ├── gradle-wrapper.jar
      └── gradle-wrapper.properties
```

可以看到，Gradle Wrapper 有三部分：

- Microsoft Windows 上的 batch 文件，Linux 和 Mac OS X 上的 shell 脚本。
- batch 文件和 shell 脚本需要用到的 JAR 文件。
- 一个 properties 文件。

gradle-wrapper.properties 文件包含了参数配置，并能决定使用哪一个 Gradle 版本：

```
#Sat May 30 17:41:49 CEST 2015
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
distributionUrl=https\://services.gradle.org/distributions/
                          gradle-2.4-all.zip
```

如果你想使用内部分发的自定义 Gradle 版本，那么可以修改该 URL 路径。这也意味着，任何你使用的应用或者依赖库都可以有一个不同的 Gradle URL，所以在运行 wrapper 之前，一

定要检查它是否是可信任的属性。

当项目中使用的不是最新的 Gradle 版本时，Android Studio 将会显示一个提醒，提示并建议你自动更新 Gradle 版本。其原理是，Android Studio 更改了 `gradle-wrapper.properties` 的配置并触发了一次构建，这样最新版本的 Gradle 就会被下载。

Android Studio 根据 properties 文件中的信息来决定应该使用的 Gradle 版本，它会在你项目的 Gradle Wrapper 文件夹下运行 Wrapper。但是，它不能使用 shell 或 bash 脚本，所以你无法自定义它们。

1.4.2 运行基本构建任务

使用 terminal 或命令提示符，可以导航到项目根目录，运行带有 `tasks` 的 Gradle Wrapper 命令：

```
$ gradlew tasks
```

这将打印出所有可用的任务列表。如果你添加了 `--all` 参数，那么你将获得每个任务对应依赖的详细介绍。

在微软的 Windows 操作系统上，你需要运行 `gradlew.bat`；而在 Linux 和 Mac OS X 操作系统上，完整的命令是 `./gradlew`。为简便起见，我们在整本书中将只写 `gradlew`。

当你在开发阶段构建项目时，可以运行带有 debug 参数的 assemble 任务：

```
$ gradlew assembleDebug
```

这个任务会为这个应用创建一个 debug 版本的 APK。默认情况下，Gradle 的 Android 插件会把 APK 保存在 `MyApp/app/build/outputs/apk` 目录下。

任务名称简写

为了避免大量的终端输入，Gradle 提供了一种快捷方式，即驼峰式缩写任务名称。例如，你可以通过运行 `gradlew assDeb` 来执行 `assembleDebug`，或直接在命令行界面输入 `gradlew aD`。

有一点需要注意，其只在驼峰式缩写独一无二的情况下有效。如果另一个任务具有相同的缩写，那么此快捷方式就不再生效。

除了 assemble 外，还有其他三个基本任务。

- Check：运行所有的检查，这通常意味着在一个连接的设备或模拟器上运行测试。
- Build：触发 assemble 和 check。
- Clean：清除项目的输出。

我们将在第 2 章中详细讨论这些任务。

1.5 迁移出Eclipse

把一个 Eclipse 项目迁移至 Gradle 的方式有两种：

- 使用 Android Studio 导入向导，让其自动处理迁移。
- 在 Eclipse 项目中添加 Gradle 脚本，并手动设置一切。

对大部分项目来说，使用导入向导迁移非常简单，它能够把一切东西都自动转换。如果有哪些文件是向导不能转换的，那么它也会给出提示，告诉你哪里必须修改后才能继续迁移。

但是有些项目可能非常复杂，需要手动迁移。如果你有一个庞大的项目，那么你可能更倾向于一步一步迁移，而不是一次全部迁移。Gradle 支持执行 Ant 任务，甚至整个 Ant 构建。这样做，你可以平稳地过渡并且缓慢地迁移所有组件。

1.5.1 导入向导

在使用导入向导之前，你需要打开 Android Studio，单击 **File** 菜单，然后单击 **Import Project...**，或者在 Android Studio 的开始窗口，单击 **Import Non-Android Studio project**。

如果你迁移一个带有 JAR 文件或者依赖库源码的项目，那么导入向导会建议你把它们替换为 Gradle 的依赖包。这些依赖包可以来自于本地的 Google 仓库（例如 Android Support Library）或者知名的远程仓库中心。如果在 Google 或在线仓库中没有找到与之匹配的依赖包，那么 JAR 文件会和之前的 Eclipse 一样被使用。导入向导会为你的应用创建至少一个模块。如果你的项目中带有源代码的依赖库，那么它们也会被转换为相应的模块。

图 1-6 为导入向导示意图。

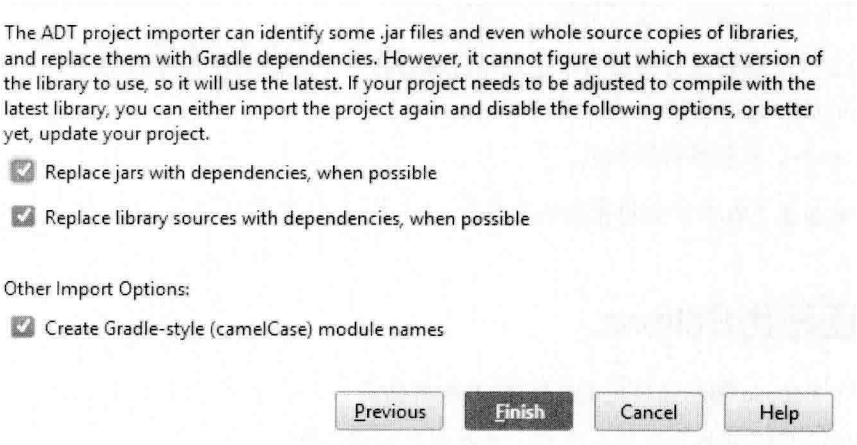

图 1-6　导入向导

在迁移项目时，Android Studio 会创建一个新文件夹以确保你不会丢失任何文件，你可以轻松地比较原有工程和导入后的结果。当迁移完成后，Android Studio 会打开项目并展示一个导入摘要。

该摘要列出了导入向导决定忽略且没有复制到新项目中的所有文件，如果你想包含这些文件，那么你只能把它们手动复制到新项目中。在被忽略文件的正下方，摘要还展示了导入向导能够将 JAR 替换成 Gradle 依赖包的所有 JAR 文件。Android Studio 会在 JCenter 上尝试找到这些依赖包。如果你使用了支持库，那么包含 Google 仓库的方式是使用 SDK manager 下载到你的电脑，而不是使用 JAR 文件。最后，摘要列出了导入向导迁移的所有文件，展示了它们的原始位置和最新位置。

导入向导也会添加三个 Gradle 文件：`settings.gradle`、根目录的 `build.gradle`，以及模块中的 `build.gradle`。

如果你有包含源代码的依赖库，那么导入向导会把它们全部迁移为 Gradle 工程，并且根据需要链接在一起。

现在该项目构建应该没有任何问题了，但是请记住，你可能需要连接互联网去下载一些必要的依赖包。

复杂的项目可能需要我们做一些额外的工作，所以下面就一起来看看如何手动迁移。

Eclipse 导出向导

在 Eclipse 中也有一个导出向导，但是其已经完全被遗弃，因为谷歌的 Android 工具小组已经停止了对 Eclipse 的 Android 开发工具的开发。所以，作为替代，建议使用 Android Studio 中的导入向导。

1.5.2 手动迁移

手动迁移到基于 Gradle 的 Android 项目的方式有很多种，不需要更改为新的目录结构，甚至可以在你的 Gradle 脚本上运行 Ant 脚本。这使得迁移过程变得更加灵活，并且让较大的项目更容易过渡。我们将在第 9 章中介绍如何在 Gradle 上运行 Ant 任务。

1. 保留旧的项目结构

如果你不想移动文件位置，那么可以在你的项目中保留 Eclipse 文件目录结构。要想做到这一点，你需要更改一下源集（source set）的配置。在我们讨论项目结构的时候提到过源集。通常情况下，Gradle 和 Android 插件是有默认值的，但是其有可能被覆盖。

你需要做的第一件事就是在项目的根目录下创建一个 `build.gradle` 文件。该文件应申请 Android 插件，并设置 Gradle 和 Android 插件的属性。最简单的形式如下所示：

```
buildscript {
    repositories {
        jcenter()
     }
    dependencies {
        classpath 'com.android.tools.build:gradle:1.2.3'
     }
}
apply plugin: 'com.android.application'
android {
    compileSdkVersion 22
    buildToolsVersion "22.0.1"
}
```

然后你就可以开始修改源集了。通常来讲，按照 Eclipse 的目录结构来覆写 main 源集的示例如下：

```
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')
    }
}
```

在 Eclipse 的文件夹结构中，所有的源文件都被放在同一个文件夹下，所以你需要告诉 Gradle，所有的组件都应放在 src 文件夹下。你只需将项目中的组件包含在内即可，但是将它们都放进去也没什么不妥。

只要你有 JAR 文件依赖，你就需要告诉 Gradle 依赖包的文件路径。假设 JAR 文件在一个叫作 libs 的文件夹下，那么配置应该是这样的：

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

这一行将 libs 目录下所有后缀名为 .jar 的文件视为一个依赖。

2. 转换到新的项目结构

如果你决定手动转换到新的项目结构，那么你还需要创建一些文件夹并移动一些文件。表 1-2 描述了最重要的文件和文件夹，以及转移到新项目结构时，你需要将它们移动到哪儿。

表 1-2　转到新的项目结构

旧的位置	新的位置
src/	app/src/main/java/
res/	app/src/main/res/
assets/	app/src/main/assets/
AndroidManifest.xml	app/src/main/AndroidManifest.xml

如果你有单元测试，那么你需要将源代码迁移到 app/src/test/java/ 下，以便让 Gradle 自动识别它们。功能测试代码则应放到 app/src/androidTest/java/ 文件夹下。

接下来就是在项目的根目录下创建 settings.gradle 文件。该文件只需一行代码，它

的目的是告诉 Gradle 将 app 模块包含到构建中去：

```
include: ':app'
```

当这些都准备好后，你需要两个 build.gradle 文件来进行一次成功的 Gradle 构建。第一个文件位于项目根目录（和 settings.gradle 同级）下，用来定义项目级别的参数设置：

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

上述代码为项目中的所有模块都设置了一些属性。第二个 build.gradle 文件在 app 目录下，其包含了一些特定模块的参数设置：

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 22
    buildToolsVersion "22.0.1"
}
```

这些都是最基础的知识。如果你有一个不依赖于第三方代码的简单的 Android 应用，那就足够了。如果你有任何依赖，那么你就需要将其迁移到 Gradle。

3. 迁移依赖库

如果在你的项目中有包含 Android 代码的依赖库，那么它们也需要使用 Gradle，以便更好地与 app 模块交互。同样需要申请插件，只是你需要使用 Android 依赖库插件来代替 Android 应用插件，具体细节我们在第 5 章中讨论。

1.6 总结

在本章的开篇，我们分析了 Gradle 的优势，以及它比现阶段其他正在使用的构建系统更加好用的原因，随后简单分析了 Android Studio 以及其是如何通过生成构建文件来帮助我们的。

简单介绍之后，我们分析了让维护和分享项目变得更加容易的 Gradle Wrapper。我们通过

Android Studio 创建了一个新项目，并且介绍了如何自动和手动地将一个 Eclipse 项目迁移到 Android Studio 和 Gradle 上。最后我们还介绍了如何利用 Android Studio 构建项目，或是直接通过命令行来构建项目。

在接下来的几章中，我们将着眼于如何自定义构建，即进一步自动化构建的过程，这可使得维护项目更加容易。下一章，我们将审视所有的标准 Gradle 文件，探索基本的构建任务，以及自定义部分构建。

2

基本自定义构建

本章我们将从分析 Gradle 的用法开始，然后讲解创建和转换 Android 项目。通过本章我们将更好地理解这些构建文件，本章还会分析一些有用的任务，并探索 Gradle 插件和 Android 插件的可能性。

本章我们将介绍以下主题：

- 理解 Gradle 文件
- 开始构建任务
- 自定义构建

2.1 理解Gradle文件

当用 Android Studio 创建一个新项目时，会默认生成三个 Gradle 文件。其中的两个文件——`settings.gradle` 和 `build.gradle` 位于项目的根目录。另外一个 `build.gradle` 文件则在 Android app 模块内被创建。图 2-1 展示了在项目中 Gradle 文件的位置。

```
MyApp
    ├── build.gradle
    ├── settings.gradle
    └── app
        └── build.gradle
```

图 2-1 项目中 Gradle 文件的位置

这三个文件每一个都有自己的用途，我们会在接下来的章节中进一步研究。

2.1.1 settings 文件

对于一个只包含一个 Android 应用的新项目来说，settings.gradle 应该是这样的：

```
Include ':app'
```

settings 文件在初始化阶段被执行，并且定义了哪些模块应该包含在构建内。在本例中，app 模块被包含在内。单模块项目并不一定需要 setting 文件，但是多模块项目必须要有 setting 文件，否则，Gradle 不知道哪个模块应包含在构建内。

在这背后，Gradle 会为每个 settings 文件创建一个 Settings 对象，并调用该对象的相关方法。你不必知道 Settings 类的相关细节，但意识到这一点是非常好的。

完整的 Setting 类解释超出了本书范围。如果你想了解更多信息，可以到 Gradle 文档中寻找（https://gradle.org/docs/current/dsl/org. gradle.api.initialization.Settings.html）。

2.1.2 顶层构建文件

在项目中，所有模块的配置参数都应在顶层 build.gradle 文件中配置。默认情况下其包含如下两个代码块：

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

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

实际构建配置在 buildscript 代码块内，在第 1 章中我们曾简要介绍过。repositories 代码块将 JCenter 配置成一个仓库，在这种情况下，一个仓库意味着一系列的依赖包，或者说，在我们应用和依赖项目中可使用的一系列可下载的函数库。JCenter 是一个很有名的 Maven 库。

dependencies 代码块用于配置构建过程中的依赖包。这也意味着你不能将你的应用或依赖项目所需要的依赖包包含在顶层构建文件中。默认情况下，唯一被定义的依赖包是 Gradle 的 Android 插件。每个 Android 模块都需要有 Android 插件，因为该插件可使其执行 Android 相关的任务。

allprojects 代码块可用来声明那些需要被用于所有模块的属性。你甚至可以在 allprojects 中创建任务，这些任务最终会被运用到所有模块。

只要你使用了 allprojects，模块就会被耦合到项目。这意味着，其很可能在没有主项目构建文件的情况下，无法独立构建模块。最初，这看起来可能不是一个问题，但是如果你后面想要分离一个内部依赖库到自己的项目，那么你将需要重构你的构建文件。

2.1.3 模块的构建文件

模块层的 build.gradle 文件的属性只能应用在 Android app 模块，它可以覆盖顶层 build.gradle 文件的任何属性。该模块的构建文件示例如下：

```
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'
}
```

我们将对这三个主要代码块进行详细研究。

1. 插件

第一行用到了 Android 应用插件，该插件在顶层构建文件中被配置成了依赖，这些我们在前面讨论过。谷歌的 Android 工具团队负责 Android 插件的编写和维护，并提供构建、测试和打包 Android 应用以及依赖项目的所有任务。

2. Android

在构建文件中，占比最大的是 `android` 代码块。该代码块包含了全部的 Android 特有配置，这些特有配置之所以可被使用，是因为之前我们使用了 Android 插件。

必须有的属性是 `compileSdkVersion` 和 `buildToolsVersion`：

- 第一个，`compileSdkVersion`，是要用来编译应用的 Android API 版本。
- 第二个，`buildToolsVersion`，是构建工具和编译器使用的版本号。

构建工具包含命令行应用，如 aapt、zipalign、dx 和 renderscript，这些都被用来在打包应用时生成各种中间产物。你可以通过 SDK Manager 下载构建工具。`defaultConfig` 代码块用于配置应用的核心属性。此代码块中的属性可覆盖 `AndroidManifest.xml` 文件中对应的条目：

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

`applicationId` 是这段代码中的第一个属性。该属性覆盖了 manifest 文件中的 package name，但 `applicationId` 和 package name 有一些不同。在 Gradle 被用作默认的 Android 构

建系统之前，AndroidManifest.xml 中的 package name 有两个用途：作为一个应用的唯一标志，以及在R资源类中被用作包名。使用构建 variants，Gradle 可更容易地创建不同版本的应用。例如，构建 variants 可以很容易地构建一个免费版和一个付费版应用。这两个版本需要有独立的标识符，这样它们才能在 Google Play Store 中以不同的应用出现，并且可以同时被安装。然而，资源代码和生成的 R 类，必须在任何时候都保持相同的包名，否则，你的所有源文件都要随着你正在构建的版本而改变。这就是 Android 工具团队解耦了 package name 两种不同用法的原因。定义在 manifest 文件中的 package，继续用在你的源代码和 R 类中，而之前被用作设备和 Google Play 唯一标识的 package name，现在则被称之为 application id。在我们开始尝试不同的构建类型时，application id 将会变得更加有趣。

在 defaultConfig 中，接下来的两个属性是 minSdkVersion 和 targetSdkVersion。这两个属性看起来很熟悉，是因为它们通常作为 <uses-sdk> 标签的一部分，在 manifest 中被定义。minSdkVersion 被用来配置运行应用的最小 API 级别。targetSdkVersion 用于通知系统，该应用已经在某特定 Android 版本通过测试，从而操作系统不必启用任何向前兼容的行为。这和我们之前看到的 compileSdkVersion 没有任何关系。

versionCode 和 versionName 在 manifest 文件中的作用相同，即为你的应用定义一个版本号和一个友好的版本名称。构建文件中的属性会全面覆盖 manifest 文件中的属性。因此，如果你在 build.gradle 中定义了它们，就没有必要在 manifest 文件中再去定义。如果构建文件不包含某个属性，那么 manifest 中的该属性就会被用作后备。

buildTypes 代码块可用来定义如何构建和打包不同构建类型的应用，我们将在第 4 章中详细介绍。

3. 依赖包

依赖代码块是标准 Gradle 配置的一部分（这就是其放在 android 代码块之外的原因），其定义了一个应用或依赖项目的所有依赖包。默认情况下，一个新的 Android 应用，在 libs 文件夹下对所有 JAR 文件构成依赖。根据你在新建项目向导中的选择，其也可能依赖于 AppCompat。我们将在第 3 章中详细讨论依赖。

2.2 任务入门

要想知道在一个项目中有哪些任务可被使用，则可以运行 gradlew tasks 任务，该命令会打印出所有可用的任务。在新建的 Android 项目中，其会包括 Android tasks、build tasks、

build setup tasks、help tasks、install tasks、verification tasks 和其他 tasks。如果你不仅想看这些任务，还想看到它们之间的依赖，那么你可以运行 `gradlew tasks --all`。该命令会试运行这些任务，在运行某一特定的任务时，打印出所有被执行的步骤。试运行实际上不会执行上述任何步骤，所以当运行某一特定的任务时，其会是一种安全的方式，让你能够预测将会发生什么。你可以通过添加参数 -m 或 --dry-run 来做一次试运行。

2.2.1 基础任务

Gradle 的 Android 插件使用了 Java 基础插件，而 Java 基础插件又使用了基础插件。基础插件添加了任务的标准生命周期和一些共同约定的属性。基础插件定义了 `assemnle` 和 `clean` 任务，Java 插件定义了 `check` 和 `buildtasks`。在基础插件中，这些任务既不被实现，也不执行任何操作。它们被用来定义插件之间的约定，即添加实际工作的任务。

这些任务的约定如下：

- `assemble`：集合项目的输出。
- `clean`：清理项目的输出。
- `check`：运行所有检查，通常是单元测试和集成测试。
- `build`：同时运行 `assemble` 和 `check`。

Java 基础插件添加了源集的概念。Android 插件构建于这些约定，这样经验丰富的 Gradle 用户就可以看到这些暴露的任务。在这些基本的任务之上，Android 插件也添加了很多 Android 特有的任务。

2.2.2 Android 任务

Android 插件扩展了基本任务，并实现了它们的行为。下面是在 Android 环境下任务所做的事情：

- `assemble`：为每个构建版本创建一个 APK。
- `clean`：删除所有的构建内容，例如 APK 文件。
- `check`：运行 Lint 检查，如果 Lint 发现一个问题，则可终止构建。
- `build`：同时运行 `assemble` 和 `check`。

`assemble` 任务默认依赖于 `assembleDebug` 和 `assembleRelease`，如果你添加了更多的构建类型，那么就会有更多的任务。这意味着，运行 `assemble` 将会触发每一个你所拥有的构建类型，并进行一次构建操作。

除了扩展这些任务之外，Android 插件还添加了一些新的任务。下面是一些值得注意的新任务：

- `connectedCheck`：在连接设备或模拟器上运行测试。
- `deviceCheck`：一个占位任务，专为其他插件在远端设备上运行测试。
- `installDebug` 和 `installRelease`：在连接的设备或模拟器上安装特定版本。
- 所有的 `installtasks` 都会有相关的 `uninstall` 任务。

构建任务依赖于 `check`，而不是 `connectedCheck` 或 `deviceCheck`。这是因为构建任务只需确保正常的 check，而无须一个连接的设备或模拟器。运行 check 任务会生成一份 Lint 报告，Lint 报告会包含所有的警告和错误，以及一份详细的说明和一个相关文档的链接。该报告在 `app/build/outputs` 目录下，名称为 `lint-results.html`，如图 2-2 所示。

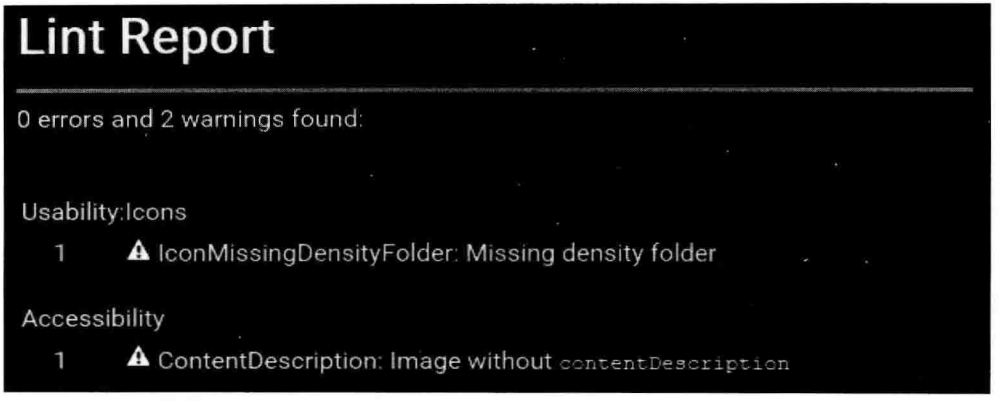

图 2-2 Lint 报告

当你集合一份生产版本时，Lint 将会检查那些导致应用崩溃的致命问题。一旦发现有问题，Lint 就会终止构建，并在命令行界面打印错误信息。Lint 也会在 `app/build/outputs` 文件夹下生成一份叫作 `lint-results-release-fatal.html` 的报告。当你有多个问题时，浏览 HTML 报告将比在命令行界面来回滚动更令人愉悦。其所提供的链接也非常有用，因为它们会指引你到问题的详细描述。

2.2.3 Android Studio

你不必经常在命令行界面运行 Gradle 任务。Android Studio 有一个包含了所有可用任务的工具窗。该工具窗被称之为 **Gradle**，如图 2-3 所示。

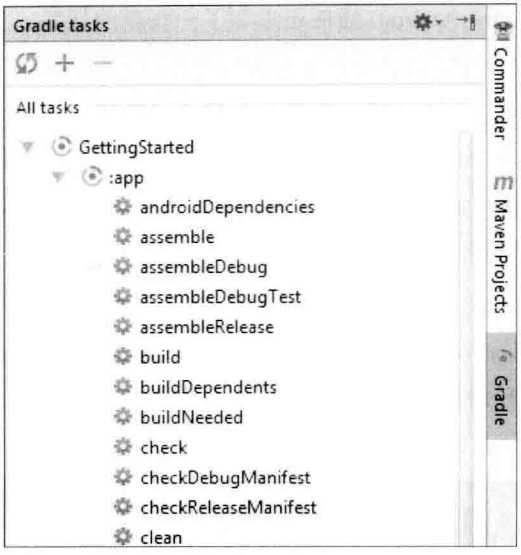

图 2-3　Gradle 工具窗

在该窗口中，你可以通过简单的双击任务的名称来运行某个任务。你可以在 **Gradle Console** 工具窗中跟进任何正在运行的任务的进度。如果找不到这些工具窗，那么你可以在 **Tool Window** 下的 **View** 菜单中打开它们。图 2-4 是 Gradle Console 工具窗的示意图。

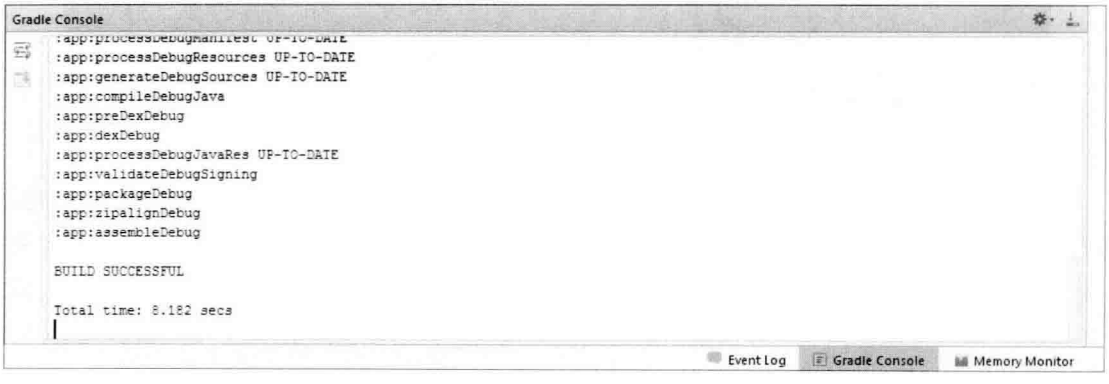

图 2-4　Gradle Console 工具窗

你也可以在 Android Studio 内通过命令行运行任务，所以只要你喜欢，你就可以在 IDE 内做所有与应用相关的工作。在运行命令之前，你需要打开 **Terminal** 工具窗。这是一个成熟的终端，所以你可以在上面运行任何命令。为了能够使用 Gradle Wrapper，你可能需要先导航到项目的根目录。

改变 Android Studio 终端

在 Android Studio 中，可以通过配置终端来使用不同的 shell。例如，在微软的 Windows 操作系统中，默认的终端是命令提示符。但如果你喜欢使用 Git Bash（或任何其他 shell），则打开 Android Studio 的设置页面（在 `File` 和 `Settings` 下面），找到 Terminal。你可以改变 shell 的路径。在微软的 Windows 操作系统中，Git Bash 路径看起来应该是这样：`C:\Program Files(x86)\Git\bin\sh.exe -login -i`。

2.3 自定义构建

自定义构建过程的方式有很多种，当你在 Android Studio 中编写构建文件时，无论你在构建文件中自定义了什么，都应该始终同步该项目。当你开始添加依赖或 `BuildConfig` 变量时，同步将变得尤其重要，我们后面会讨论这些问题。

在你编写 `settings.gradle` 或 `build.gradle` 时，Android Studio 将会在编辑器展示一条消息，通过导航到 **Tools | Android | Sync Project with Gradle Files**，或单击工具栏上对应的按钮都可以触发同步，如图 2-5 所示。

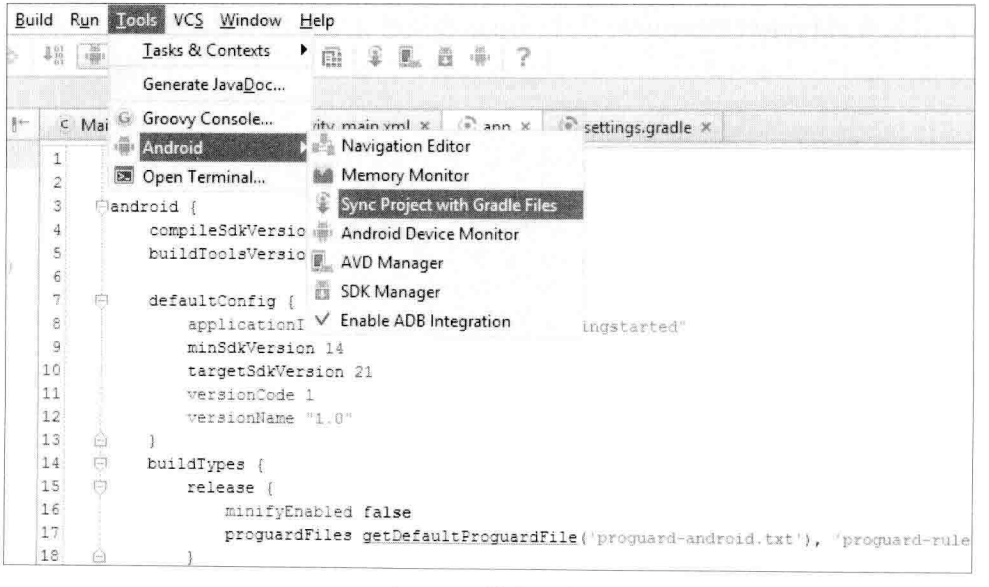

图 2-5 触发同步

在底层，Android Studio 同步实际上运行了 `generateDebugSources` 任务来生成所有必需的类。

2.3.1 操控 manifest 条目

我们可以直接通过构建文件而不是 manifest 文件来配置 `applicationId`、`minSdkVersion`、`targetSdkVersion`、`versionCode` 和 `versionName`。另外，下面一些属性也是我们可以操控的：

- `testApplicationId`：针对 instrument 测试 APK 的 applicationId。
- `testInstrumentationRunner`：JUnit 测试运行器的名称，被用来运行测试（请看第 6 章）。
- `signingConfig`（请看第 4 章）。
- `proguardFile` 和 `proguardFiles`（请看第 9 章）。

在 Android Studio 内部

不必手动修改构建文件，在 Android Studio 的 Project Structure 对话框中可以修改基本的设置。你可以通过 File 菜单打开该对话框，对话框允许你修改项目范围内的设置和每个模块的设置。对于每一个 Android 模块，你都可以修改 Android 插件的属性和所有 manifest 属性。如图 2-6 所示，你可以看到 **Project Structure** 对话框中 app 模块的 release 版本对应的属性。

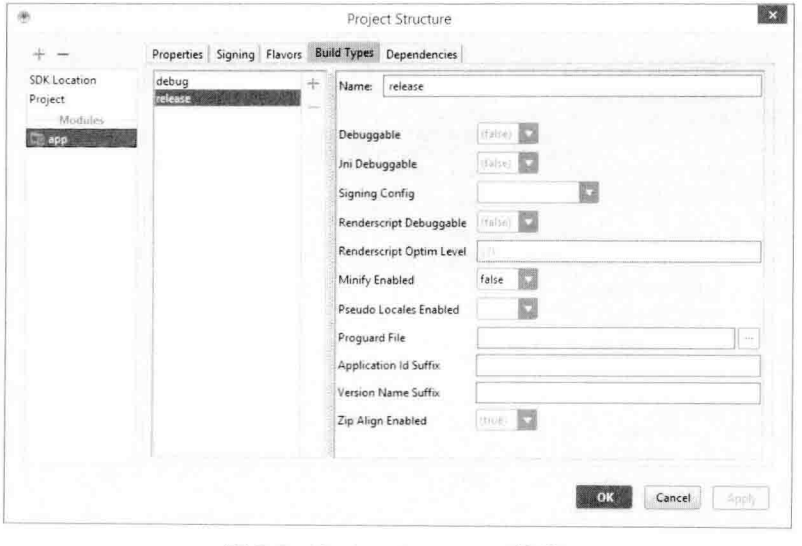

图 2-6　Project Structure 对话框

注意，Android Studio 会把在 **Project Structure** 对话框中所做的任何更改都写入 Gradle 构建配置文件中。

2.3.2 BuildConfig 和资源

自 SDK 工具版本升级到 17 之后，构建工具都会生成一个叫作 BuildConfig 的类，该类包含一个按照构建类型设置值的 DEBUG 常量。如果有一部分代码你只想在 debugging 时期运行，比如 logging，那么 DEBUG 会非常有用。你可以通过 Gradle 来扩展该文件，这样在 debug 和 release 时，就可以拥有不同的常量。

这些常量可用于切换功能或设置服务器 URL，例如：

```
android {
    buildTypes {
        debug {
            buildConfigField "String", "API_URL",
            "\"http://test.example.com/api\""
            buildConfigField "boolean", "LOG_HTTP_CALLS", "true"
        }
        release {
            buildConfigField "String", "API_URL",
            "\"http://example.com/api\""
            buildConfigField "boolean", "LOG_HTTP_CALLS", "false"
        }
    }
}
```

字符串值必须用转义双引号括起来，这样才会生成实际意义上的字符串。在添加 buildConfigField 行后，你可以在你的 Java 代码中使用 BuildConfig.API_URL 和 BuildConfig.LOG_HTTP。

最近，Android 工具开发小组还增加了类似的方式来配置资源值：

```
android {
     buildTypes {
          debug {
              resValue "string", "app_name", "Example DEBUG"
          }
          release {
              resValue "string", "app_name", "Example"
          }
```

```
    }
}
```

这里可以不加转义双引号，因为资源值通常默认被包装成 value=" "。

2.3.3 项目范围的设置

如果在一个项目中，你有多个 Android 模块，并且没有人为修改每个模块的构建文件，而是将它们设置应用到所有的模块，那么将变得非常有用。我们已经知道在生成的顶层构建文件中，allprojects 代码块是如何定义依赖仓库的，你也可以使用相同的策略来运用 Android 特定的设置：

```
allprojects {
    apply plugin: 'com.android.application'
    android {
        compileSdkVersion 22
        buildToolsVersion "22.0.1"
    }
}
```

该段代码只有在你的所有模块都是 Android app 项目的时候才有效，因为你需要运用 Android 插件来获取 Android 特有的设置。实现这种行为的更好方式是在顶层构建文件中定义值，然后将它们应用到模块中。Gradle 允许在 Project 对象上添加额外属性。这意味着任何 build.gradle 文件都能定义额外的属性，添加额外属性需要通过 ext 代码块。

你可以给顶层构建文件添加一个含有自定义属性的 ext 代码块：

```
ext {
    compileSdkVersion = 22
    buildToolsVersion = "22.0.1"
}
```

该段代码使得模块层的构建文件可以使用 rootProject 来获取属性：

```
android {
    compileSdkVersion rootProject.ext.compileSdkVersion
    buildToolsVersion rootProject.ext.buildToolsVersion
}
```

2.3.4 项目属性

前面例子的 ext 代码块是定义额外属性的一种方式。你可以使用属性来动态定制构建过程。

在第 7 章我们将开始利用它们来编写自定义任务。定义属性的方式有很多种，这里我们只介绍三种常用的：

- ext 代码块
- gradle.properties 文件
- -P 命令行参数

下面的 build.gradle 文件就包含了这三种添加额外属性的方式：

```
ext {
     local = 'Hello from build.gradle'
}

  task printProperties << {
     println local          // Local extra property
     println propertiesFile        // Property from file
     if (project.hasProperty('cmd')) {
       println cmd          // Command line property
     }
}
```

gradle.properties 文件的实现（相同文件夹下）：

```
propertiesFile = Hello from gradle.properties
```

在这个例子中，我们创建了一个新的任务。我们将在第 7 章研究任务并解释它的语法。

如果通过命令行参数运行 printProperties 任务，那么输出如下：

```
$ gradlew printProperties -Pcmd='Hello from the command line'
:printProperties
Hello from build.gradle
Hello from gradle.properties
Hello from the command line
```

自定义属性使得更改构建配置就像更改单一的属性或添加一个命令行参数一样简单。

我们可以同时在顶层构建文件和模块构建文件中定义属性。如果一个模块定义了一个在顶层文件中早已存在的属性时，那么新属性将会直接覆盖原来的属性。

2.3.5 默认的任务

如果没有指定任何任务而运行 Gradle 的话，其会运行 help 任务，它会打印一些如何使用 Gradle 工作的信息。help 任务被设置为默认的任务，在每次运行没有明确指定任务的 Gradle 时，可以覆写默认的任务，添加一个常用的任务，甚至是多个任务。

在顶层 build.gradle 文件中加入一行，可用来指定默认的任务：

```
defaultTasks 'clean', 'assembleDebug'
```

现在，当你运行没有任何参数的 Gradle Wrapper 时，它会运行 clean 和 assembleDebug。通过运行 tasks 任务和格式化输出，可以清晰地看到哪些任务被设置为默认任务。

```
$ gradlew tasks | grep "Default tasks"
Default tasks: clean, assembleDebug
```

2.4 总结

本章我们详细研究了 Android Studio 自动生成的不同的 Gradle 文件。现在，你也可以自己创建构建文件，添加所有必需字段和配置属性值了。

本章我们学习了基本的构建任务，并学习了如何用 Android 插件基于 base 插件构建，以及如何通过新建 Android 特定任务来扩展它。我们也了解到如何通过使用命令行或 Android Studio 来运行构建任务。

本章的最后，我们研究了影响构建输出的几种方式，以及如何配置构建过程。

最近几年，Android 的开发者生态已大大增强，有很多有趣的依赖包可供所有人使用。下一章，我们将研究在一个项目中添加依赖的几种方式，以便利用这些丰富的资源。

3 依赖管理

依赖管理是 Gradle 最耀眼的特点之一。最佳情况下，你需要做的仅仅是在构建文件中添加一行代码，Gradle 将会从远程仓库下载依赖，确保你的项目能够使用依赖中的类。Gradle 甚至可以做得更多。如果你的项目中有一个依赖，并且其有自己的依赖，那么 Gradle 将会处理并解决这些问题。这些依赖中的依赖，被称之为**传递依赖**。

本章介绍依赖管理的概念，并介绍添加依赖到 Android 项目的多种方式。我们将讨论以下内容：

- 依赖仓库
- 本地依赖
- 依赖的概念

3.1 依赖仓库

当我们在讨论依赖时，通常指的是外部依赖，例如其他开发者提供的依赖库。手动管理依赖会是一个大麻烦。你必须找到该依赖，下载 JAR 文件，将其拷贝到项目，引用它。通常这些 JAR 文件在它们的名称中没有版本号，所以你需要添加 JAR 的版本，以便知道什么时候更新。你还需要确保依赖库中存储在了源代码管理系统，以便团队成员在没有手动下载这些依赖时，也可以使用基于依赖的代码。

使用依赖仓库可以解决这些问题。一个依赖仓库可以被看作是文件的集合。Gradle 默认情况下没有为你的项目定义任何依赖仓库，所以你需要在 `repositories` 代码块中添加它们。如果使用 Android Studio，那么它会为你自动完成。在第 2 章，我们简要地提及过 `repositories` 代码块，如下所示：

```
repositories {
        jcenter()
}
```

Gradle 支持三种不同的依赖仓库：Maven、Ivy 和静态文件或文件夹。在构建的执行阶段，依赖从依赖仓库中被获取出来。Gradle 也有本地缓存，所以一个特定版本的依赖只会在你的机器上下载一次。

一个依赖通常是由三种元素定义的：group、name 和 version。group 指定了创建该依赖库的组织，通常是反向域名。name 是依赖库的唯一标识。version 指定了需要使用依赖库的版本号。使用这三个元素，就可以在 dependencies 代码块中声明一个依赖了：

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

下面是完整的 Groovy 映射标识：

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

对于一个依赖来说，其唯一需要的字段是 name。group 和 version 都是可选的元素。尽管如此，为了表述清楚，建议添加 group。而 version 可确保依赖库不会自动更新，因为这可能会导致构建失败。

3.1.1 预定义依赖仓库

为了方便，Gradle 预定义了三个 Maven 仓库：JCenter、Maven Central 和本地 Maven 仓库。为了在构建脚本中包含它们，你需要使用下面几行代码：

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

Maven Central 和 JCenter 是两个有名的远程仓库，一般不同时使用它们，通常推荐使用

JCenter，其也是 Android Studio 创建 Android 项目时的默认依赖仓库。JCenter 是 Maven Central 的超集，所以当你切换仓库时，可以不必修改已经定义的依赖。更重要的是，JCenter 不同于 Maven Central，它还支持 HTTPS。

本地 Maven 仓库是你已经使用了的所有依赖的本地缓存，你也可以自己添加依赖。默认情况下，依赖仓库可以在一个名为 .m2 目录文件夹下的主目录中找到。在 Linux 或 Mac OS X 上，该路径是 ~/.m2，在 Microsoft Windows 上，路径为 `%UserProfile%.m2`。

除了这些预定义的依赖仓库外，你还可以添加其他的公有甚至私有仓库。

3.1.2 远程仓库

一些组织会创建有趣的插件或依赖库，并且更喜欢将它们放在自有的 Maven 或 Ivy 服务器上，而不是将它们发布到 Mvean Central 或 JCenter。为了在构建中添加这些依赖，你需要在 `maven` 代码块中添加 URL：

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

与 Ivy 仓库类似，Apache Ivy 是一个依赖管理器，在 Ant 中十分流行。在 Gradle 中，这些仓库的格式和 Maven 仓库相同。在 `ivy` 代码块中添加仓库 URL 的方式如下：

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

如果你的团队正在使用自己的仓库，那么你需要有资格访问它们。下面是为一个仓库添加凭证的方法：

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

```
                password 'secretpassword'
            }
        }
}
```

Maven 和 Ivy 的方式相同。你可以在你的 Ivy 仓库中添加相同格式的 `credentials` 代码块。

存储凭证

不建议在构建配置文件中存储你的凭证。因为构建配置文件是纯文本，其会被迁入源码控制系统。更好的方法是使用一个单独的 Gradle 属性文件，就像我们在第 2 章中看到的那样。

3.1.3 本地仓库

我们可以在自己的硬盘驱动器或网络驱动器上运行 Maven 和 Ivy 仓库。要想在构建中添加本地仓库，你只需配置一个相对或绝对路径的 URL 即可：

```
repositories {
    maven {
        url "../repo"
    }
}
```

新的 Android 项目默认情况下有一个 Android Support Library 的依赖。当使用 SDK manager 来安装 Google 仓库时，在硬件驱动器上，会创建两个 Maven 仓库，分别是 `ANDROID_SDK/extras/google/m2repository` 和 `ANDROID_SDK/extras/android/m2repository`。Gradle 通过它们获取谷歌提供的依赖包，例如 Android Support Library 和 Google Play Services。

当然，你也可以通过使用 `flatDirs` 来添加一个常用文件夹作为仓库。下面的代码让你能够在 `dependency` 代码块中添加文件夹中的文件：

```
repositories {
   flatDir {
      dirs 'aars'
   }
}
```

在本章的后面，我们将通过一个示例研究如何使用依赖项目。

3.2 本地依赖

在某些情况下，你可能仍然需要手动下载 JAR 文件或原生库。你可能想创建自己的依赖库，这样你就可以在没有将其发布到公有或私有仓库时在多个项目中复用。在这种情况下，你不能使用任何在线资源，而是必须通过其他方式来添加依赖。下面将介绍如何使用文件依赖、如何引入原生依赖，以及在项目中如何引入依赖项目。

3.2.1 文件依赖

你可以使用 Gradle 提供的 files 方法来添加 JAR 文件作为一个依赖，如下所示：

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

当你有很多 JAR 文件时，这种方式会变得异常烦琐，一次添加一个完整的文件夹可能会更容易些：

```
dependencies {
    compile fileTree('libs')
}
```

默认情况下，新建的 Android 项目会有一个 libs 文件夹，其会被声明为依赖使用的文件夹。一个过滤器可以保证只有 JAR 文件会被依赖，而不是简单地依赖文件夹中的所有文件：

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

这意味着所有由 Android Studio 创建的 Android 项目，你都可以将 JAR 文件放置在 libs 文件夹中，其会自动包含在构建路径和最终的 APK 中。

3.2.2 原生依赖库

用 C 或 C++ 编写的依赖库可以被编译为特定平台的原生代码。这些依赖库通常包含几个 .so 文件，可用于所有平台。Android 插件默认支持原生依赖库，你所需要做的就是在模块层创建一个 jniLibs 文件夹，然后为每个平台创建子文件夹，将 .so 文件放在适当的文件夹中。

文件夹结构如图 3-1 所示。

```
app
    ├── AndroidManifest.xml
    └── jniLibs
        ├── armeabi
        │   └── nativelib.so
        ├── armeabi-v7a
        │   └── nativelib.so
        ├── mips
        │   └── nativelib.so
        └── x86
            └── nativelib.so
```

图 3-1　文件夹结构

如果此约定不生效，那么你可以在构建文件中设置相关位置：

```
android {
      sourceSets.main {
          jniLibs.srcDir 'src/main/libs'
      }
}
```

3.2.3　依赖项目

如果想分享一个使用 Android APIs 或 Android 资源的依赖库，那么你需要创建一个依赖项目。依赖项目通常和应用项目类似。你可以使用相同的任务来构建和测试依赖项目，并且它们可以有不同的构建 variants。不同之处在于输出。应用项目将生成一个可被安装和运行在 Android 设备上的 APK，依赖项目则生成一个 .aar 文件。该文件可被 Android 应用项目用作依赖库。

1. 创建和使用依赖项目模块

不同于应用 Android 应用插件，构建脚本需要应用 Android 依赖库插件：

```
apply plugin: 'com.android.library'
```

在应用中包含依赖项目的方式有两种。一种是在项目中当作一个模块，另一种是创建一个可在多个应用中复用的 .aar 文件。

如果在项目中创建了一个模块作为依赖项目，那么你需要在 settings.gradle 中添加该模块，在应用模块中将它作为依赖：

```
Include ':app', ':library'
```

在这种情况下，依赖模块被称之为依赖库，并且文件夹的名称与此相同。为了在 Android 模块中使用依赖库，你需要在 Android 模块的 build.gradle 文件中添加一个依赖库：

```
dependencies {
        compile project(':library')
}
```

其会在应用模块的类路径中包含依赖库的输出。我们将在第 5 章中详细介绍该方法。

2. 使用 .aar 文件

如果你创建了一个依赖库，并且想在不同的 Android 应用中复用，那么你可以创建一个 .aar 文件，然后将其作为一个依赖添加到你的项目中。在构建依赖库时，模块目录下的 build/output/aar/ 文件夹将会生成 .aar 文件。为了添加一个 .aar 文件作为依赖，你需要在应用模块中创建一个文件夹，复制 .aar 文件至该文件夹，并添加该文件夹作为依赖仓库：

```
repositories {
        flatDir {
            dirs 'aars'
        }
}
```

这使得我们可以填加任何文件到该目录下，并将其作为依赖，具体操作方法如下：

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

其告知 Gradle 查找具有特定名称且扩展名为 .aar 的依赖库。

3.3 依赖概念

这里有几个依赖相关的概念理解起来非常有意思（即便现在用不到它们），其中之一就是配置的概念，其解释了贯穿整章用作依赖的 compile 关键字。

3.3.1 配置

有时可能你不得不和一个只在特定设备上工作的 SDK 打交道，比如特定厂商的蓝牙 SDK。为了能够编译该代码，你需要将 SDK 添加至编译类路径。你并不需要添加 SDK 到你的 APK 中，因为其早已存在于设备中。这就是所谓的依赖配置。

Gradle 将多个依赖添加至配置，并将其命名为集文件。下面是一个 Android 应用或依赖库的标准配置：

- compile
- apk
- provided
- testCompile
- androidTestCompile

compile 是默认的配置，在编译主应用时包含所有的依赖。该配置不仅会将依赖添加至类路径，还会生成对应的 APK。

如果依赖使用 apk 配置，则该依赖只会被打包到 APK，而不会添加到编译类路径。provided 配置则完全相反，其依赖不会被打包进 APK。这两个配置只适用于 JAR 依赖。如果试图在依赖项目中添加它们，那么将会导致错误。

最后，testCompile 和 androidTestCompile 配置会添加用于测试的额外依赖库。在运行测试相关的任务时，这些配置会被使用，并且在添加如 JUnit 或 Espresso 测试框架时，特别有用。如果你只希望在测试 APK 时使用这些框架，那么就不会生产 APK。

除了这些标准配置外，Android 插件还针对每个构建 variant 都生成了一份配置，使其添加依赖配置化成为可能，例如 debugCompile、releaseProvided 等。当你只想在 debug 构建中添加 logging 框架时，这会非常有用。你可以在第 4 章中看到更多关于这方面的信息。

3.3.2 语义化版本

版本化是依赖管理的重要部分。将依赖添加到 JCenter 等依赖仓库时，约定遵循了一套版本化规则，我们称之为语义化版本。在语义化版本中，版本数字的格式一般为 major.minor.patch，数字则按照下列规则依次增加：

- 当做不兼容的 API 变化时，major 版本增加。
- 当以向后兼容的方式添加功能时，minor 版本增加。
- 当修复一些 bug 时，patch 版本增加。

3.3.3 动态化版本

在某些情况下，你可能希望在每次构建你的应用或依赖库时，都能够获取到最新的依赖。

要想做到这一点，最好的实现方式是使用动态化版本。动态化版本的使用方式有很多种，例如：

```
dependencies {
        compile 'com.android.support:support-v4:22.2.+'
        compile 'com.android.support:appcompat-v7:22.2+'
        compile 'com.android.support:recyclerview-v7:+'
}
```

第一行，我们告知 Gradle 获取最新的 patch 版本。第二行，我们希望能获取每一个最新的 minor 版本，且 minor 版本至少是 2。最后一行，我们告知 Gradle 获取依赖库的最新版本。

在使用动态化版本时，需要格外小心。如果你允许 Gradle 获取最新版本，则很可能 Gradle 获取的依赖版本并不稳定，它会导致构建中断。更糟糕的是，其会导致在构建服务器上和你自己的机器上运行着不同版本的依赖，从而导致应用程序的行为不一致。

当你试图在构建文件中使用动态化版本时，Android Studio 将会警告你可能存在的问题，如图 3-2 所示。

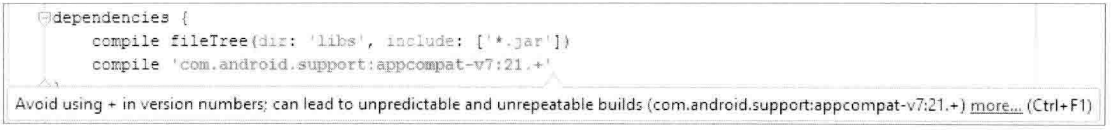

图 3-2 警告

3.4 Android Studio

添加新依赖的最简单的方式是使用 Android Studio 的 Project Structure 对话框。从 File 菜单栏打开对话框，导航到 Dependencies，获取当前依赖概要，如图 3-3 所示。

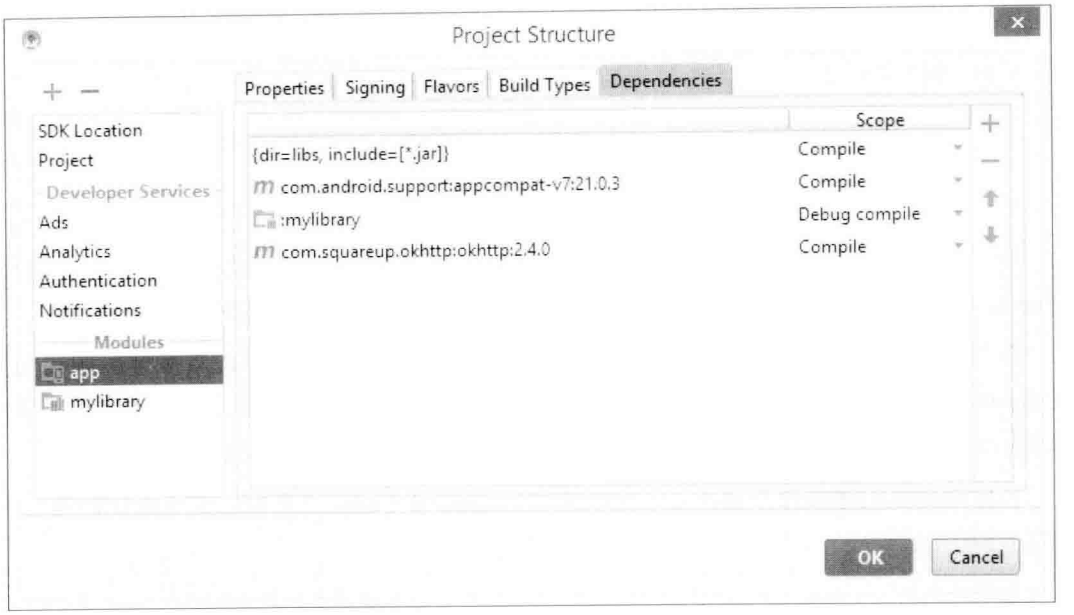

图 3-3　获取当前依赖概要

在该对话框中，你可以通过单击绿色加号图标来添加新的依赖。你可以添加其他模块和文件，甚至可以通过 JCenter 搜索相关依赖，如图 3-4 所示。

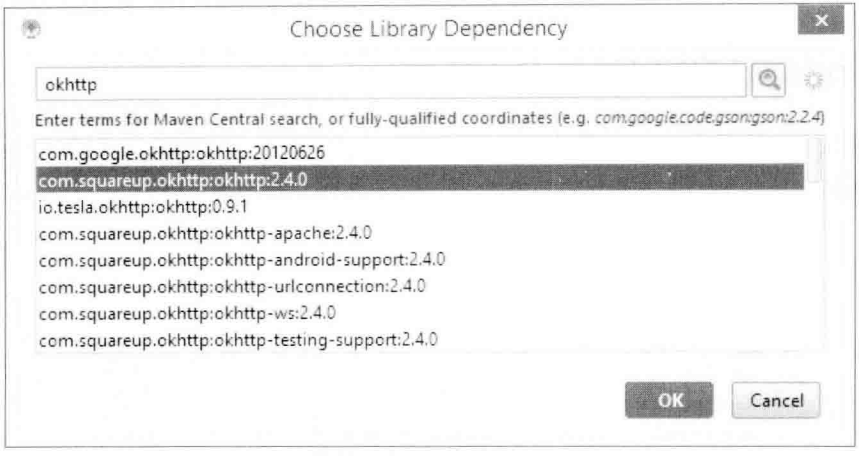

图 3-4　搜索相关依赖

在项目中使用 Android Studio 对话框，不仅能够轻松获取依赖概要，添加新的依赖库，而且无须在 `build.gradle` 文件中手动添加代码，即可直接通过 IDE 搜索 JCenter。

3.5 总结

本章我们研究了添加依赖到 Android 项目的多种方式，学习了所有形式的依赖仓库，以及在未使用依赖仓库时，如何依赖文件。

本章还介绍了关于依赖的一些重要概念：配置名称、语义化版本、动态化版本。

前面我们多次提到过构建 variants，在下一章，我们将解释什么是构建 variants，为什么它们非常重要。构建 variants 可以使开发、测试和分发应用变得更加容易。了解 variants 工作原理可以显著提高开发和分发过程的速度。

4

创建构建Variant

当开发一个应用时，通常会有几个不同版本。最常见的情况是，你有一个手动测试用于保证质量的测试版本和一个生产版本。这些版本通常有不同的配置。举个例子，测试 API 和生产 API 的 URL 可以不同。除此之外，你的应用可能还有一个免费版和一个有额外功能的付费版。在这种情况下，你就需要处理四种不同的版本：免费测试版、付费测试版、免费生产版、付费生产版。不同版本的不同配置让项目变得十分复杂。

在 Gradle 中，有一些便捷和可扩展的概念可用来定位这些常见问题。前面提到过，每个由 Android Studio 创建的新项目都会生成 `debug` 和 `release` 构建类型。另外一个概念是 product flavor（不同定制的产品），其让管理多个应用或依赖库版本成为可能。构建类型和 product flavor 经常结合在一起使用，可以很容易地处理测试和生产应用的免费和付费版本。构建类型和 product flavor 的结合结果被称之为构建 variant。

本章中，我们将首先研究构建类型和构建类型的用途，以及如何使用它们。然后，我们将讨论构建类型与 product flavor 的不同之处，以及如何同时使用它们。我们也将研究对于发布应用的必要签名配置，以及如何针对每个构建 variant 设置不同的签名配置。

本章我们将讲解以下内容：

- 构建类型
- product flavor
- 构建 variant
- 签名配置

4.1 构建类型

在 Gradle 的 Android 插件中，构建类型通常被用来定义如何构建一个应用或依赖库。每个构建类型都能特殊化，不管 debug 标示是否被包含，applicationID 是什么，无用的资源是否需要被移除等。你可以在 buildTypes 代码块中定义构建类型。下面是 Android Studio 创建的构建文件的标准 buildTypes 代码块：

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

新模块的默认 build.gradle 文件配置了一个构建类型，叫作 release。该构建类型用于禁用清除无用的资源（通过设置 minifyEnabled 为 false）和定义默认 ProGuard 配置文件的位置。让开发人员可以更直观地在他们的生产构建上使用 ProGuard。

在你的项目中，release 构建类型不是被创建的唯一构建类型。默认情况下，每个模块都有一个 debug 构建类型。其被设置为默认构建类型，但是你可以通过将其包含至 buildTypes 代码块，然后覆写任何你想改变的属性来修改其配置。

debug 构建类型的默认设置可以让调试更加容易。当你创建你自己的构建类型时，可以应用不同的默认值。例如，debuggable 属性在 debug 构建类型中被设置为 true，但是在你创建的其他构建类型中，其均被设置为 false。

4.1.1 创建构建类型

当默认的设置不够用时，我们可以很容易地创建自定义构建类型。新的构建类型只需在 buildTypes 代码块中新增一个对象即可。例如，下面的 staging 自定义构建类型：

```
android {
    buildTypes {
        staging {
```

```
            applicationIdSuffix ".staging"
            versionNameSuffix "-staging"
            buildConfigField "String", "API_URL",
                "\"http://staging.example.com/api\""
        }
    }
}
```

staging 构建类型针对 applicationID 定义了一个新的后缀，使其和 debug 以及 release 版本的 applicationID 不一样。假设你的构建类型是默认的，并且添加了 staging 构建类型，那么不同构建类型的 applicationID 应该像下面这样：

- Debug：com.package
- Release：com.package
- Staging：com.package.staging

这意味着你将能够在相同色设备上同时安装 staging 版本和 release 版本，而不发生任何冲突。staging 构建类型也有版本名后缀，其在相同设备上区分多个应用版本时非常重要。buildConfigField 属性使用一个构建配置变量，为 API 定义了一个自定义 URL，就像我们在第 2 章中看到的那样。

在创建一个新的构建类型时，还可以用另一个构建类型的属性来初始化该构建类型：

```
android {
    buildTypes {
        staging.initWith(buildTypes.debug)
        staging {
            applicationIdSuffix ".staging"
            versionNameSuffix "-staging"
            debuggable = false
        }
    }
}
```

initWith() 方法创建了一个新的构建类型，并且复制了一个已经存在的构建类型的所有属性到新的构建类型中。我们可以通过在新的构建类型对象中简单地定义它们来覆写属性或定义额外的属性。

4.1.2 源集

当创建一个新的构建类型时，Gradle 也会创建一个新的源集。源集目录名称默认和构建类型相同。该目录不是在定义新的构建类型时自动创建的，因而你需要在构建类型使用自定义的源码和资源之前，手动创建该源集目录。

下面是标准的 debug 和 release，另加一个额外 staging 构建类型的目录结构，如图 4-1 所示。

```
app
└── src
    ├── debug
    │   ├── java
    │   │   └── com.package
    │   │       └── Constants.java
    │   ├── res
    │   │   └── layout
    │   │       └── activity_main.xml
    │   └── AndroidManifest.xml
    ├── main
    │   ├── java
    │   │   └── com.package
    │   │       └── MainActivity.java
    │   ├── res
    │   │   ├── drawable
    │   │   └── layout
    │   │       └── activity_main.xml
    │   └── AndroidManifest.xml
    ├── staging
    │   ├── java
    │   │   └── com.package
    │   │       └── Constants.java
    │   ├── res
    │   │   └── layout
    │   │       └── activity_main.xml
    │   └── AndroidManifest.xml
    └── release
        ├── java
        │   └── com.package
        │       └── Constants.java
        └── AndroidManifest.xml
```

图 4-1 标准的 debug 和 release 目录结构

这些源集让一切皆为可能。例如，你可以针对特定的构建类型覆写某些属性，为某些构建类型添加自定义代码，以及为不同的构建类型添加自定义布局或字符串。

当添加 Java 类到构建类型时，请记住这一过程是相互排斥的。这就意味着，如果你添加了 `CustomLogic.java` 类到 staging 源集，你将能够添加相同的类到 debug 和 release 源集，但不能添加到 main 源集。那样该类会被定义两次，且当你试图构建的时候会出现一个异常。

当使用不同的源集时，资源会被一种特殊的方式处理。Drawables 和 layout 文件将完全覆盖在 main 源集中有相同名称的资源，但是，`values` 文件夹中的文件则不会被覆盖（例如 `strings.xml`）。Gradle 将合并构建类型中的资源到 main 资源中。

例如，如果在 main 源集中有一个类似下面的 `strings.xml` 文件：

```
<resources>
    <string name="app_name">TypesAndFlavors</string>
    <string name="hello_world">Hello world!</string>
</resources>
```

并且在 `staging` 构建类型源集中有一个 `strings.xml` 文件：

```
<resources>
    <string name="app_name">TypesAndFlavors STAGING</string>
</resources>
```

那么，合并后的 `strings.xml` 文件将会是这样：

```
<resources>
    <string name="app_name">TypesAndFlavors STAGING</string>
    <string name="hello_world">Hello world!</string>
</resources>
```

当你构建了一个非 staging 的构建类型时，最终的 `strings.xml` 文件将只会是 main 源集中的 `strings.xml` 文件。

manifest 文件与之类同。如果你为一个构建类型创建了一个 manifest 文件，那么你不必从 main 源集中拷贝整个 manifest 文件，只需添加需要的标签即可。Android 插件将会合并 manifest。

在本章的后面我们会讨论合并的具体细节。

4.1.3 依赖

每个构建类型都可以有自己的依赖。Gradle 自动为每个构建类型创建新的依赖配置。如果你只想在 debug 构建中添加一个 logging 框架。那么，你可以这么做：

```
dependencies {
        compile fileTree(dir: 'libs', include: ['*.jar'])
        compile 'com.android.support:appcompat-v7:22.2.0'
        debugCompile 'de.mindpipe.android:android-logging-log4j:1.0.3'
}
```

你可以以这种方式结合任何构建配置的构建类型，来让依赖变得更加具体。

4.2 product flavor

与被用来配置相同 App 或 library 的不同构建类型相反，product flavor 被用来创建不同的版本。典型的例子是一个应用有免费版和付费版。另一个常见的情况是，一个机构为多个客户端构建相同功能而品牌不同的应用。这在出租车企业或银行应用中十分常见，一个公司创建一个可以在同一类别所有客户端重复使用的应用。唯一改变的是颜色、图标和后台 URL。Product flavors 极大简化了基于相同代码构建多个版本的应用的进程。

如果你不确定是否需要一个新的构建类型或新的 product flavor，那么你应该问自己，是否想要创建一个供内部使用的应用，或一个会发布到 Google Play 的新 APK。如果你需要一个全新的 App，独立于你已有的应用发布，那么 product flavor 就是你需要的，否则，你应该坚持使用构建类型。

4.2.1 创建 product flavor

创建 Product flavor 和创建构建类型类似。你可以通过在 `productFlavor` 代码块中添加新的 `Product flavor` 来创建：

```
android {
        productFlavors {
            red {
                applicationId 'com.gradleforandroid.red'
                versionCode 3
            }
            blue {
                applicationId 'com.gradleforandroid.blue'
```

```
            minSdkVersion 14
            versionCode 4
        }
    }
}
```

product flavor 和构建类型相比有不同的属性。这是因为 product flavor 是 ProductFlavor 类中的对象，就像存在于所有构建脚本的 defaultConfig 对象。这就意味着 defaultConfig 和你的所有 product flavors 享有相同的属性。

4.2.2 源集

和构建类型类似，product flavor 也可以拥有它们自己的源集目录。为一个特殊的 flavor 创建一个文件夹就和创建一个有 flavor 名称的文件夹一样简单。你甚至可以为一个特定构建类型和 flavor 的结合体创建一个文件夹。该文件夹的名称将是 flavor 名称＋构建类型的名称。例如，如果你想让 blue flavor 的 release 版本有一个不同的应用图标，那么文件夹将会被叫作 blueRelease。合并文件夹的组成将比构建类型文件夹和 product flavor 文件夹拥有更高优先级。

4.2.3 多种定制的版本

在某些情况下，你可能想要更进一步，创建 product flavor 的结合体。例如，客户 A 和客户 B 在他们的 App 中都想要免费版和付费版，并且是基于相同的代码、不同的品牌。创建四种不同的 flavor 意味着需要像这样设置多个拷贝，所以这不是最佳做法。而使用 flavor 维度是结合 flavor 的有效方式，如下所示：

```
android {
        flavorDimensions "color", "price"

        productFlavors {
            red {
                flavorDimension "color"
            }
            blue {
                flavorDimension "color"
            }
            free {
                flavorDimension "price"
            }

            paid {
```

```
            flavorDimension "price"
        }
    }
}
```

当你为 flavor 添加了维度后，Gradle 会希望你能够为每个 flavor 都添加维度。如果你忘了，那么你会收到一个带有错误接受的构建错误。`flavorDimensions` 数组定义了维度，维度的顺序非常重要。当结合两个 flavor 时，它们可能定义了相同的属性或资源。在这种情况下，flavor 维度数组的顺序就决定了哪个 flavor 配置将被覆盖。在上一个例子中，color 维度覆盖了 price 维度。该顺序也决定了构建 variant 的名称。

假设默认的构建配置是 debug 和 release 构建类型，那么前面例子中定义的 flavor 将会生成下面这些构建 variant：

- `blueFreeDebug` and `blueFreeRelease`
- `bluePaidDebug` and `bluePaidRelease`
- `redFreeDebug` and `redFreeRelease`
- `redPaidDebug` and `redPaidRelease`

4.3 构建variant

构建 variant 是构建类型和 product flavor 结合的结果。不论什么时候创建一个构建类型或 product flavor，新的 variant 都会被创建。举个例子，如果你有标准的 `debug` 和 `release` 构建类型，并且你创建了一个 red 和 blue 的 product flavor，那么将会生成如图 4-2 所示的构建 variant。

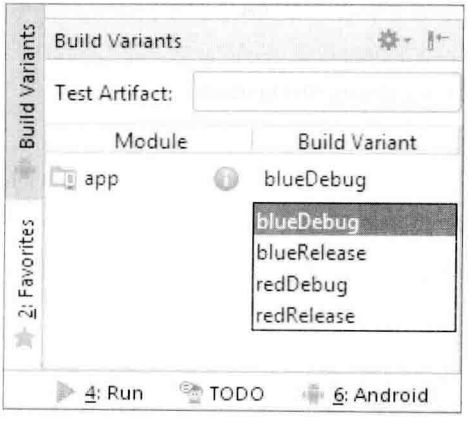

图 4-2 生成的构建 variant

这是 Android Studio 中的 **Build Variants** 工具窗的截图。你可以在编辑器的左下角找到该工具窗，或者通过 **View | Tool Windows | Build Variants** 打开。该工具窗列举了所有的构建 variant，并允许它们之间切换。修改选中的构建 variant，将会影响当 **Run** 按钮被点击时，运行哪个 variant。

如果没有 product flavor，variant 将只会包含构建类型。没有任何构建类型是不可能的，即便你自己不定义任何构建类型，Gradle 的 Android 插件也会为你的 App 或 library 创建一个 debug 构建类型。

4.3.1 任务

Gradle 的 Android 插件将会为你配置的每一个构建 variant 创建任务。一个新的 Android 应用默认有 `debug` 和 `release` 两种构建类型，所以你可以用 `assembleDebug` 和 `assembleRelease` 来分别构建两个 APKs，即用单个命令 `assemble` 来创建两个 APKs。当添加一个新的构建类型时，新的任务也将被创建。一旦你开始添加 flavor，那么整个全新的任务系列将会被创建，因为每个构建类型的任务会和每个 product flavor 相结合。这意味着，仅用一个构建类型和一个 flavor 做一个简单设置，你就会有三个任务用于构建全部 variant：

- `assembleBlue`：使用 blue flavor 配置和组装 `BlueRelease` 及 `BlueDebug`。
- `assembleDebug`：使用 debug 构建配置类型，并为每个 product flavor 组装一个 debug 版本。
- `assembleBlueDebug`：用构建配置类型结合 flavor 配置，并且 flavor 设置将会覆盖构建类型设置。

新的 tasks 是为每个构建类型、每个 product flavor、每个构建类型和 product flavor 结合体而创建。

4.3.2 源集

构建 variant，是一个构建类型和一个或多个 product flavor 的结合体，其可以有自己的源集文件夹。例如，由 `debug` 构建类型 blue flavor 和 free flavor 创建的 variant 可以有其自己的源集 `src/blueFreeDebug/java/`。其可以通过使用 sourceSets 代码块来覆盖文件夹的位置，这在第 1 章中曾介绍过。

4.3.3 源集合并资源和 manifest

源集的引入额外增加了构建进程的复杂性。Gradle 的 Android 插件在打包应用之前将 main

源集和构建类型源集合并在一起。此外，library 项目也可以提供额外的资源，这些也需要被合并。这同样适用于 manifest 文件。例如，在你应用的 debug variant 中可能需要额外的 Android 权限来存储 log 文件。你并不想在 main 源集中申明该权限，因为这会吓跑潜在客户。相反，你可以在 debug 构建类型的源集中额外添加一个 manifest 文件来申明额外的权限。

资源和 manifest 的优先顺序如图 4-3 所示。

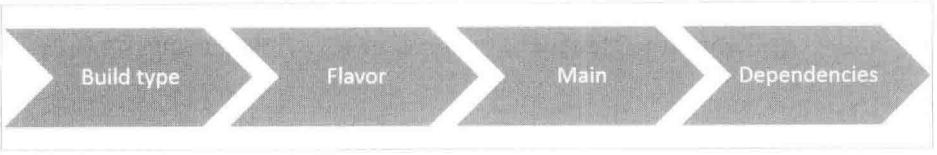

图 4-3 资源和 manifest 的优先顺序

如果资源在 flavor 和 main 源集中都有申明，那么 flavor 中的资源将被赋予更高的优先级。在这种情况下，flavor 源集中的资源将会被打包。在 library 项目中申明的资源通常具有最低优先级。

关于资源和 manifest 合并还有很多需要学习的地方。这是一个令人难以置信的复杂问题，如果详细地解释它们，那么将需要一个完整的章节。如果你想了解更多，则阅读该主题的官方文档 http://tools.android.com/tech-docs/new-build- system/user-guide/manifest-merger 是一个不错的主意。

4.3.4 创建构建 variant

Gradle 让处理复杂的构建 variant 变得简单。即便在创建和配置两个构建类型和两个 product flavor 时，构建文件依然很简洁：

```
android {
        buildTypes {
            debug {
                buildConfigField "String", "API_URL",
                "\"http://test.example.com/api\""
            }

            staging.initWith(android.buildTypes.debug)
            staging {
                buildConfigField "String", "API_URL",
```

```
                "\"http://staging.example.com/api\""
            applicationIdSuffix ".staging"
        }
    }

    productFlavors {
        red {
            applicationId "com.gradleforandroid.red"
            resValue "color", "flavor_color", "#ff0000"
        }

        blue {
            applicationId "com.gradleforandroid.blue"
            resValue "color", "flavor_color", "#0000ff"
        }
    }
}
```

在本例中，我们创建了四个不同的构建 variant：blueDebug、blueStaging、redDebug 和 redStaging。每个 variant 都有自己的 API URL 组合以及 flavor 颜色。图 4-4 是 blueDebug 在手机上的样子。

图 4-4　blueDebug 在手机上的样子

相同应用的 redStaging variant 则如图 4-5 所示。

图 4-5　redStaging 在手机上的样子

图 4-4 展示了 blueDebug variant，其使用了 debug 构建类型中定义的 URL，基于 blue product flavor 中定义的 flavor_color 会使得文本变蓝。图 4-5 显示了配有 staging URL 和红色文本的 redStaging。red staging 版本也有一个不同的应用图标，因为 staging 构建类型在源集 drawable 文件夹下，有其自己的应用图标。

4.3.5 variant 过滤器

在你的构建中，可以完全忽略某些 variant。通过这种方式，你就可以通过 assemble 命令来加快构建所有 variant 的进程，并且你的任务列表不会被任何无须执行的任务污染。variant 过滤器也可确保在 Android Studio 的构建 variant 切换器中不会出现过滤的构建 variant。

你可以在 App 或 library 根目录下的 build.gradle 文件使用以下代码来过滤 variants：

```
android.variantFilter { variant ->
        if(variant.buildType.name.equals('release')) {
            variant.getFlavors().each() { flavor ->
                    if (flavor.name.equals('blue')) {
                    variant.setIgnore(true);
                }
            }
        }
}
```

在本例中，我们首先检查了 variant 的构建类型中是否含有 release，然后我们去除了所有含有该名称的 Product flavor。当不使用 flavor 维度时，在 flavor 数组中将只含有一个 Product flavor。一旦你开始使用 flavor 维度，那么 flavor 数组就将会持有和维度一样多的 flavor。在示例脚本中，我们检查了 blue product flavor, 并且告诉构建脚本忽略这一 variant。

图 4-6 是 Android Studio 的构建 variant 转换器在 variant 过滤后的结果。

你可以看到两个 blue release variant（blueFreeRelease 和 bluePaidRelease）被过滤出构建 variants 列表。如果现在运行 gradlew tasks，那么你会发现所有和这些 variant 相关的任务都已不再存在。

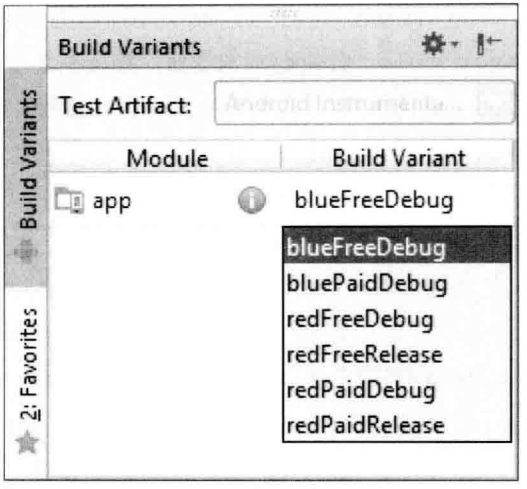

图 4-6　过滤后的结果

4.4　签名配置

在你发布应用到 Google Play 或任何其他应用商店之前，都需要用私钥给它签名。如果你有一个付费版和免费版或针对不同用户的不同应用，那么你需要为每个 flavor 使用不同的私钥签名。这就是签名配置出现的原因。

签名配置如下所示：

```
android {
      signingConfigs {
          staging.initWith(signingConfigs.debug)

          release {
              storeFile file("release.keystore")
              storePassword"secretpassword"
              keyAlias "gradleforandroid"
              keyPassword "secretpassword"
          }
      }
}
```

在本例中，我们创建了两个不同的签名配置。

Android 插件使用了一个通用 keystore 和一个已知密码自动创建了 debug 配置，所以就没

有必要为该构建类型再创建一个签名配置了。

本例中的 staging 配置使用了 `initWith()`，其会从另外一个签名配置中复制所有属性。这意味着 staging 构建是通过 debug 密钥签名的，而不是自己定义。

release 配置使用 `storeFile` 来指定 keystore 文件的路径，之后定义了密钥别名和两个密码。

正如前面提到的，在构建配置文件中存储凭证不是一个好主意。更好的方式是使用 Gradle 配置文件。在本书第 7 章中，完整地介绍了如何为一个大 ask 存储签名配置密码。

在定义签名配置之后，你需要将它们应用到你的构建类型或 flavor 中。构建类型和 flavor 都有一个叫作 `signingConfig` 的属性，如下所示：

```
android {
        buildTypes {
            release {
                signingConfig signingConfigs.release
            }
        }
}
```

该例使用了构建类型，但如果你想为每个你所创建的 flavor 使用不同的凭证，那么你就需要创建不同的签名配置，不过你可以以相同的方式来定义它们：

```
android {
        productFlavors {
            blue {
                signingConfig signingConfigs.release
            }
        }
}
```

使用签名配置这种方式会造成很多问题。当为 flavor 分配一个配置时，实际上它是覆盖了构建类型的签名配置。如果你不想这样，那么在使用 flavor 时，就应该为每个构建类型的每个 flavor 分配不同的密钥：

```
android {
        buildTypes {
            release {
```

```
            productFlavors.red.signingConfig signingConfigs.red
            productFlavors.blue.signingConfig signingConfigs.blue
        }
    }
}
```

该例展示了在不影响 debug 和 staging 构建类型的情况下，如何在 release 构建类型中为 red flavor 和 blue flavor 使用不同的签名配置。

4.5 总结

本章我们讨论了构建类型、product flavor 以及所有可能的组合体。这些是在任何应用中都可以使用的强力工具。从简单的设置不同 URL 和 keys 开始，到分享相同代码和资源但有不同的品牌和多个版本的复杂应用，构建类型和 product flavor 可以让其变得更加简单。

我们不仅讨论了签名配置以及如何运用它们，还谈到了在签名 product flavor 时的一个常见错误。

下一章，将介绍多模块构建。当你想提取代码到一个依赖或一个依赖项目时，或者当你想将 Android Wear 模块包含至你的应用时，多模块构建非常有用。

5 管理多模块构建

Android Studio 不仅可以为应用和依赖库创建模块，还可以为 Android Wear、Android TV 和 Google App Engine 等创建模块。所有这些模块都可以在一个单一的项目中使用。例如，你可能想创建一个使用 Google Cloud Endpoints 作为后台，融合 Android Wear 的应用。在这种情况下，你的项目就需要包含三个不同模块：一个针对 App，一个针对后端，还有一个针对 Android Wear 的整合。了解多模块项目的结构和构建可以显著加快开发周期。

Gradle 文档和 Gradle Android 插件都使用了 multiproject 构建术语。然而在 Android Studio 中，一个模块和一个项目之间是有区别的。举个例子，一个模块可以是一个 Android 应用或一个 Google App Engine 的后端，而一个项目是模块的集合。在本书中，我们将以和 IDE 相同的方式使用模块和项目，来避免混乱。当你浏览整个文档时，请务必记住这一点。

本章，我们将介绍多模块构建的理论，然后展示一些实际项目的例子：

- 解剖多模块构建
- 为一个项目添加模块
- 提示和最佳实践

5.1 解剖多模块构建

通常，一个多模块项目有一个根目录，在其子文件夹中包含所有的模块。为了告知 Gradle，项目的结构以及哪个文件夹包含哪些模块，你需要在项目的根目录提供一个 `settings.gradle` 文件。每个模块都可以提供自己的 `build.gradle` 文件。在第 2 章中，我们已经学

习了 settings.gradle 和 build.gradle 是如何工作的，所以在这里，我们将只关注如何在多模块项目中使用它们。

图 5-1 是多模块项目的结构图。

```
project
  ├── setting.gradle
  ├── build.gradle
  ├── app
  │     └── build.gradle
  └── library
        └── build.gradle
```

图 5-1　多模块项目的结构图

这是建立多模块项目最简单、最直接的方式。在项目的 settings.gradle 文件中声明了所有的模块，如下所示：

```
include ':app', ':library'
```

该代码确保了 app 和 library 模块包含在构建配置中。你需要做的就是添加模块的目录名。

如果想给 app 模块添加 library 模块作为一个依赖，那么你需要将它添加到 app 模块的 build.gradle 文件中：

```
dependencies {
       compile project(':library')
}
```

为了在一个模块中添加一个依赖，你需要使用带有模块路径作为参数的 project() 方法。

如果你想使用子文件夹来管理你的模块，那么 Gradle 也已通过配置满足了你的要求。例如，你可以有一个如图 5-2 所示的目录结构。

```
project
├── setting.gradle
├── build.gradle
├── app
│     └── build.gradle
└── libraries
      ├── library1
      │     └── build.gradle
      └── library2
            └── build.gradle
```

图 5-2　目录结构

app 模块依然像之前一样位于根目录，但现在该项目有两个不同的 library，并且这些 library 模块不在项目的根目录下，而是位于一个特定的 library 目录。通过这个目录结构，你可以在 settings.gradle 中声明 app 和 library 模块：

```
include ':app', ':libraries:library1', ':libraries:library2'
```

注意，在一个子目录中声明模块十分简单。所有的路径都是相对于根目录的（settings.gradle 文件的位置）。冒号被用来替代路径中的斜线。

当在一个子目录中添加一个模块作为另一个模块的依赖时，你应该总是从根目录引用它。这意味着如果前面例子的 app 模块依赖 library1，那么 app 模块的 build.gradle 文件应该像这样：

```
dependencies {
        compile project(':libraries:library1')
}
```

如果你在子目录中声明了依赖，那么所有路径都会相对于根目录。这样做的原因是，Gradle 是从项目的根目录开始创建项目依赖模型的。

5.1.1 重访构建生命周期

了解了构建进程模型是如何构造的之后，会让理解多模块项目的组成变得更加容易。我们在第 1 章中曾讨论过构建生命周期，里面曾介绍过一些基本知识，但针对多模块构建的一些细节也很重要。

在第一阶段，即初始化阶段，Gradle 搜寻 settings.gradle 文件。如果该文件不存在，Gradle 则假设你只有一个单独的模块构建。如果你有多个模块，那么你可以在 settings 文件中定义子目录来包含各个模块。如果这些子目录有其自己的 build.gradle 文件，则 Gradle 会处理，并把它们合并到构建进程模型。这就解释了为什么你应该用相对于根目录的路径来声明模块上的依赖。因为 Gradle 始终尝试从根目录找到依赖。

一旦你理解了构建过程模型是如何工作的，就会知道配置多模块项目构建策略的方式有以下几种。它们可以在根目录的 build.gradle 文件中配置所有模块的设置。这让概览一个项目的整个构建配置变得容易，但是其也会变得非常凌乱，特别是当你的模块需要不同的插件，而它们的插件又有其自己的 DSL 时。另一种方式是每个模块都有一个独立的 build.gradle 文件。该策略可确保模块不紧密耦合，也使得跟踪构建变化变得容易，因为你不需要找到哪些改变应用了哪些模块。

最后一种策略是混合的做法。你可以在项目的根目录中设置一个构建文件，用来定义所有模块的通用属性，然后每个模块的构建文件用于配置只针对该模块的设置。Android Studio 遵循了这一策略。其会在根目录创建一个 `build.gradle` 文件，然后在每个模块内创建 `build.gradle` 文件。

5.1.2 模块任务

只要你的项目中有多个模块，那么在你运行任务之前，就需要三思而行。当你在项目根目录下的命令行界面运行一个任务时，Gradle 会找出哪个模块有这一名称的任务，然后为每个模块执行该任务。举个例子，如果你有一个手机 app 模块和一个 Android Wear 模块，那么运行 `gradlew assembleDebug` 将会构建手机 app 模块和 Android Wear 模块的 debug 版本。当你切换到其中一个模块的目录时，Gradle 将只针对该模块运行任务，即便你在项目的根目录中还在使用 Gradle wrapper。例如，在 Android Wear 模块目录下运行 `../gradlew assembleDebug`，将只会构建 Android Wear 模块。

切换目录来运行模块特定的任务时会让人十分恼火。幸运的是，还有另一种途径。你可以在一个任务名称之前加上模块名，用来在特定模块上运行该 task。例如，为了只构建 Android Wear 模块，你可以使用 `gradlew :wear:assembleDebug` 命令。

5.2 将模块添加到项目

在 Android Studio 中添加一个新模块就像通过一个向导添加新模块一样简单。该向导也创建了基本的构建文件。在某些情况下，添加一个模块甚至会导致 Android Studio 去编辑你的 app 模块的构建文件。例如，当添加一个 Android Wear 模块时，IDE 假定你想在 Android app 中使用它，并在构建文件中添加一行引用 Android Wear 模块的代码。

Android Studio 中 **New Module** 对话框如图 5-3 所示。

在下面的章节中，我们将展示如何通过 Android Studio 将不同模块添加到 Android 项目中，并解释它们的自定义属性，指出它们是如何改变构造过程的。

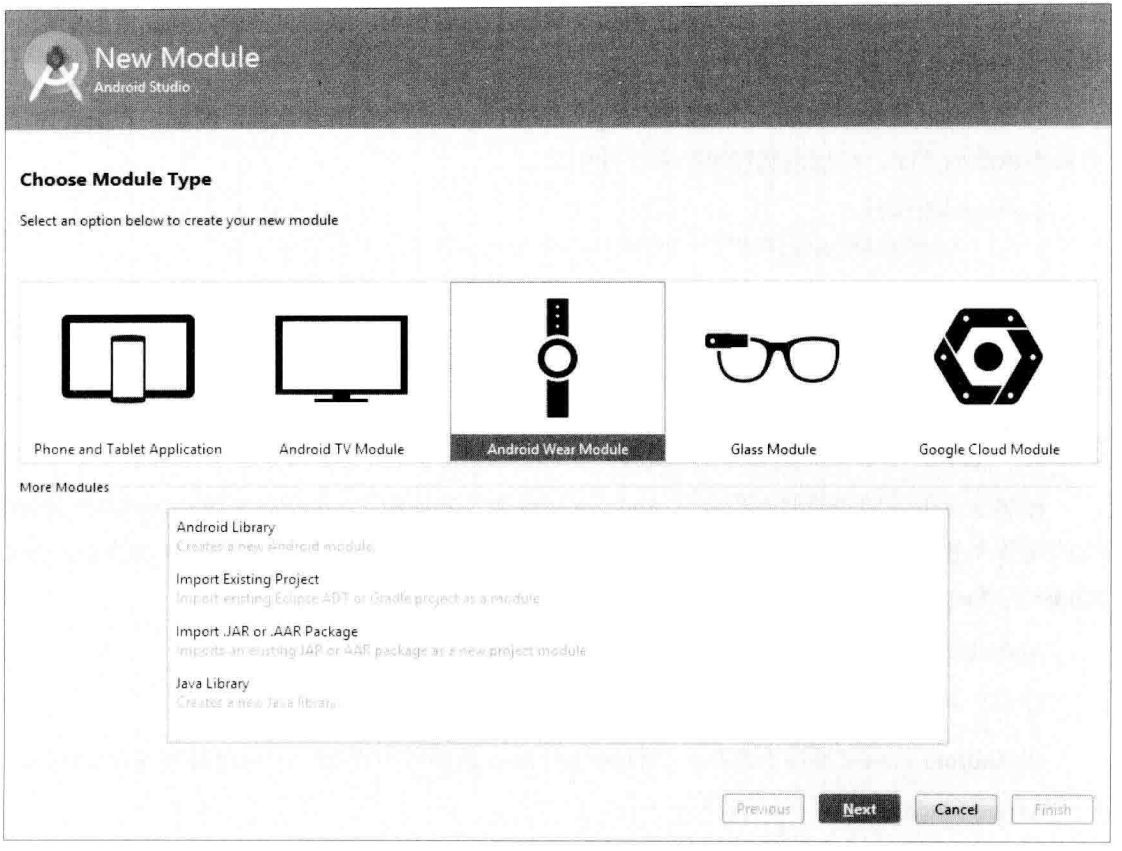

图 5-3 New Module 对话框

5.2.1 添加一个 Java 依赖库

当你添加一个新的 Java 依赖模块后，Android Studio 生成的 `build.gradle` 将如下所示：

```
apply plugin: 'java'

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

Java 依赖库模块使用的是 Java 插件，而不是我们之前看到的的 Android 插件。这意味着，许多 Android 特定属性和任务都不能使用，但是你不需要在一个 Java 依赖库中拥有这些。

构建文件也有基本的依赖管理设置，所以你可以在不需要任何特定配置的情况下，向你的

libs 文件夹添加 JAR 文件。你可以使用第 3 章中学到的知识，添加更多的依赖。依赖配置不依赖于 Android 插件。

在你的 app 模块中添加一个模块名为 javalib 的 Java 依赖库作为依赖，例如，在 app 模块中简单地添加下面一行代码到构建配置文件中：

```
dependencies {
        compile project(':javalib')
}
```

该行代码告诉 Gradle 在构建的时候引入一个名为 javalib 的模块。如果在你的 app 模块中添加了该依赖，那 javalib 模块将始终在构建 app 模块之前构建。

5.2.2 添加一个 Android 依赖库

在第 3 章中我们曾简要地提到了 Android 依赖库，我们称之为依赖库项目。两个名词都在文档和各个教程中使用。在本节，我们将使用 Androidlibrary 这一名称，因为在 Android Studio 的 **New Module** 对话框中使用了该名称。

Android 依赖库的默认 build.gradle 文件开始于下面这行代码：

```
apply plugin: 'com.android.library'
```

在 Android 依赖库模块中添加一个依赖与在 Java 依赖库中添加一个依赖的方式完全相同：

```
dependencies {
        compile project(':androidlib')
}
```

一个 Android 依赖库不仅包含依赖库中的 Java 代码，还包含所有的 Android 资源，如 manifest、strings 和 layouts。一旦在你的 app 中引用了一个 Android 依赖库，那么你就可以在该 app 中使用依赖库的所有类和资源了。

5.2.3 融合 Android Wear

如果你正在寻找深度融合 Android Wear 到你的应用，那么你就需要添加一个 Android Wear 模块。需要注意的是，Android Wear 模块也使用了 Android 应用插件。这意味着所有的构建属性和 tasks 都可用。

与一个常规的 Android app 模块相比，build.gradle 文件中唯一的不同是依赖配置：

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

```
        compile 'com.google.android.support:wearable:1.1.0'
        compile 'com.google.android.gms:play-services-wearable:6.5.87'
}
```

每个 Android Wear 应用都依赖于 Google 提供的 Android Wear 特有依赖库。为了在 Android 应用中使用 Android Wear 应用，你需要和 App 一起打包。你可以通过在 Android 应用中添加依赖做到这一点：

```
dependencies {
        wearApp project(':wear')
}
```

wearApp 配置确保 Wear 模块能被添加到 Android 应用的最终 APK 中，并会为你做一些必要的配置。

5.2.4 使用 Google App Engine

Google App Engine 是一个云计算平台，你可以使用托管的 Web 应用，而无须建立自己的服务器。在一定的使用级别，它是免费的。这使得它具有良好的实验环境。Google App Engine 还提供了一个名为 Cloud Endpoints 的服务，其用于创建 RESTful 服务。使用 Google App Engine 的 Cloud Endpoints，可以很容易地为你的应用创建后端。App Engine 的 Gradle 插件使得在 Android 应用中生成一个客户端依赖库变得非常简单，其意味着你不需要编写任何和 API 相关的代码。这使得 Google App Engine 成为 App 后端的一个选择，所以在接下来的部分，我们将研究 App Engine 的 Gradle 插件是如何工作的，以及如何使用 Cloud Endpoints。

单击 **File | New Module...**，打开 **New Module** 对话框，然后选择 **Google Cloud Module**，来创建一个带有 Cloud Endpoints 的 Google App Engine 模块。在创建模块时，你可以修改 Cloud Endpoints 的类型，然后选择将要使用这个后端的客户端模块，如图 5-4 所示。

Google App Engine 和 Cloud Endpoints 的详尽解释超出了本书范围。我们将只关注在 App Engine 模块和客户端应用模块的 Gradle 整合。

图 5-4　New Module 对话框

1. 分析构建文件

此模块的 `build.gradle` 文件变得相当大，所以我们将只研究最有趣的部分，从新的构建脚本依赖开始：

```
buildscript {
    dependencies {
        classpath 'com.google.appengine:gradle-appengine-
          plugin:1.9.18'
    }
}
```

我们需要在 `classpath` 构建脚本中定义 App Engine 插件。在前面添加 Android 插件时，我们已经看到过 classpath 了。当一切完毕后，其可以和其他两个插件一起应用 App Engine 插件：

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

```
apply plugin: 'war'
apply plugin: 'appengine'
```

Java 插件主要用来为 Cloud Endpoints 生成 JAR 文件。WAR 插件是运行和分发整个后端的必备品。WAR 插件生成一个用于分发 Java Web 应用的 WAR 文件。最终，App Engine 插件添加一些任务用于构建、运行和部署整个后端。

下面的代码块十分重要，它定义了 App Engine 模块的依赖：

```
dependencies {
        appengineSdk 'com.google.appengine:appengine-java-sdk:1.9.18'
        compile 'com.google.appengine:appengine-endpoints:1.9.18'
        compile 'com.google.appengine:appengine-endpoints-deps:1.9.18'
        compile 'javax.servlet:servlet-api:2.5'
}
```

第一个依赖使用 appengineSdk 来指定在这个模块中应该使用哪一个 SDK。endpoints 依赖是 Cloud Endpoints 能够工作的必需依赖。如果你选择在你的模块中使用 Cloud Endpoints，那么这些依赖就会被添加。任一 Google App Engine 模块都需要 servlet 依赖。

在 appengine 代码块中配置任一 App Engine 特有的设置：

```
appengine {
        downloadSdk = true
        appcfg {
            oauth2 = true
        }
        endpoints {
            getClientLibsOnBuild = true
            getDiscoveryDocsOnBuild = true
        }
}
```

设置 downloadSdk 属性为 true 时可以很容易地运行一个本地开发服务器，如果它不存在，就会自动下载 SDK。如果在你的设备上已经安装了 Google App Engine SDK，那么你就可以将 downloadSdk 属性设置为 false。

appcfg 代码块被用来配置 App Engine SDK。在一个典型的 Google App Engine 安装过程中，你可以通过使用 appcfg 命令行工具手动设置一些配置。使用 appcig 代码块，而不是命令行工具，可以让配置更加便捷，因为只要有一人构建过模块，其他人就不需要运行任何额外命令，即可拥有相同的配置。

endpoints 代码块包含了一些 Cloud Endpoints 特有的设置。

Google App Engine 和 Cloud Endpoints 的详细解释超出了本书范围。如果你想了解更多，可以研究以下文档 https://cloud.google.com/appengine/docs。

2. 在 app 中使用后端

当创建 App Engine 模块后，Android Studio 会自动在 Android app 模块的构建文件中添加一个依赖，如下所示：

```
dependencies {
        compile project(path: ':backend', configuration: 'android-
            endpoints')
}
```

我们之前看到过这样的语法（引用 Java 和 Android 依赖库的时候），使用带有两个参数而不是一个参数的 Project 去定义依赖。path 参数是默认的。我们前面曾用到过，但并没有指定它的名称。一个 Google App Engine 模块可以有不同类型的输出。通过 configuration 参数，你可以指定你想要的输出。我们需要用 App Engine 模块来生成 Cloud Endpoints，所以我们使用 android-endpoints 配置。实际上，该配置运行 _appengineEndpointsAndroidArtifact task。该任务生成一个 JAR 文件，其中包含了在 Android app 模块中使用到的类。此 JAR 文件不仅包含了使用到的 Cloud Endpoints 模块，还包含了 API 方法。这种融合，使得多模块项目能够更好地工作，因为其加快了开发时间。App Engine 模块中的 Gradle 任务使得运行和部署后端变得更加容易。

3. 自定义 tasks

App Engine 插件添加了许多任务，其中使用最多的是 appengineRun 和 appengineUpdate。

appengineRun 任务用来开启一个本地开发服务器，这样就可以在上传到 Google App Engine 之前，本地测试整个后端。第一次运行该任务时，构建可能需要一段时间，因为 Gradle 需要下载 App Engine SDK。我们早已通过 downloadSdk = true 来设置下载行为。要想停止服务器，你可以使用 appengineStop。

一旦准备部署后端程序到 Google App Engine，并在生产中使用它时，你就可以使用 appengineUpdate 了。该任务处理了所有的部署细节。如果你在 appengine 配置代码块

中设置了 `oauth2 = true`，那么该任务将打开一个浏览器窗口，以便可以登录到你的谷歌账户并获取一个验证 token。如果不想每次在部署的时候都这么做，那么你可以在 Android Studio 中登录你的谷歌账户，然后使用 IDE 来部署后端。Android Studio 运行相同的 Gradle 任务，但是保存的是你的认证信息。

5.3 提示和最佳实践

这里有几种更简单的处理多模块项目的方式，在和多模块打交道时，有几件事情需要牢记，这样不仅可以节约你的时间，还会减少挫败感。

5.3.1 在 Android Studio 中运行模块任务

就像在第 2 章中看到的那样，你可以在 Android Studio 中直接运行 Gradle 任务。当有多个模块时，Android Studio 可以识别它们，并显示所有可用任务的分组情况，如图 5-5 所示。

图 5-5 task 的分组情况

Gradle 工具窗让运行模块特有的任务变得更加容易。在同一时间针对所有模块运行一个任务的选项是不存在的，如果你想这么做，建议使用命令行界面，那样会更快些。

5.3.2 加速多模块构建

当构建一个多模块项目时，Gradle 会按顺序处理所有的模块。随着计算机提供越来越多的内核，我们可以通过并行运行所有模块来使得构建过程更快。此功能已存在于 Gradle 中，但默认不开启。

如果你想并行构建适用于你的项目，那么你需要在项目根目录的 `gradle.properties` 文件中配置 `parallel` 属性：

```
org.gradle.parallel=true
```

Gradle 会基于可用的 CPU 内核，来选择正确的线程数量。为了防止出现同一模块同时执行两个任务的问题，每个线程只拥有一个完整的模块。

并行构建执行是一个还在孵化的功能。这意味着它正在积极开发，其很可能在任何时候实现。现在，该功能已经成为 Gradle 的一部分，并被广泛使用。因此，可以放心地假设实现不会消失或急剧变化。

每个人的构建时间可能都会有所不同，但通过简单启动并行构建，你可以从你的构建中省下大量的时间。需要注意的是，为了使并行工作有效，你需要确保你的模块之间是分离的。

5.3.3 模块耦合

就像在第 2 章中看到的那样，通过在 `build.gradle` 文件中使用 `allprojects`，我们可以在一个项目中定义所有模块的属性。当你的项目有多个模块时，你可以在任何模块中使用 `allprojects` 来应用属性到项目的所有模块中。Gradle 甚至允许一个模块引用另一个模块的属性。这些强大的功能，使得维护多模块构建更加方便。不足之处是你的模块耦合在一起了。

只要两个模块互相访问了对方的任务或属性，就认为这两个模块间是耦合的。耦合会造成几个后果。例如，你放弃了可移植性。一旦你决定从项目中提取一个 library，那么在复制所有项目范围的设置之前，你将不能运行该 library。模块耦合对并行构建也有所影响。在任一模块中使用 `allprojects`，都将使得并行构建变得无用。当你给所有模块添加项目范围的属性时，需要注意这一点。

你可以通过不直接从其他模块访问任务或属性来避免耦合。如果你需要这么做，那么可以使用根模块作为中介，这样模块就只与根模块耦合，而不是和其他模块耦合。

5.4 总结

本章，我们首先研究了多模块构建的结构。然后，研究了在单一项目中如何创建多模块。我们也看到了添加新的模块会如何影响构建任务的执行。

之后我们研究了一些新模块的实际例子，以及它们是如何整合到一个项目的。最后，我们介绍了一些如何在一个项目中让多模块工作变得更加简单的提示和技巧。

在下一章中，我们将创建各种测试，并运行这些测试，来看看如何更加容易地使用Gradle。我们不仅直接在Java虚拟机上运行单元测试，还会在真实设备或模拟器上运行测试。

6
运行测试

为了确保应用或依赖库的质量，使用自动化测试是非常重要的。在很长一段时间内，Android 开发工具都缺乏对自动化测试的支持。但最近，谷歌已付出巨大的努力，使得开发人员能够更加容易地上手测试。一些旧的框架已经更新，新的框架也已经添加，以确保我们可以彻底地测试应用和依赖库。我们不仅可以在 Android Studio 中运行它们，还可以直接使用 Gradle 在命令行界面中运行它们。

本章，我们将探讨以不同的方式来测试 Android 的应用和依赖库。我们将研究 Gradle 是如何帮助自动化测试过程的。

在本章，我们将介绍讲解主题：

- 单元测试
- 功能测试
- 测试覆盖率

6.1 单元测试

在项目中拥有好的单元测试，不仅可以确保项目质量，还可以很容易地检测新代码是否会中断一些功能。Android Studio 和 Gralde 的 Android 插件原生支持单元测试，但在使用它们之前需要做一些配置。

6.1.1 JUnit

JUnit 是一个非常受欢迎的单元测试依赖库，从诞生到现在已超过十年。它可以很容易地编写测试，同时确保它们的可读性。注意，这些特定的单元测试用例只对业务逻辑有效，对和

Android SDK 相关的代码无效。

在为你的 Android 项目编写 JUnit 之前，你需要为测试用例创建一个文件夹。按照惯例，这个文件夹被命名为 test，并且应该和你的 main 目录在同一级别。test 目录结构如图 6-1 所示。

```
app
└── src
    ├── main
    │   ├── java
    │   │   └── com.example.app
    │   └──res
    └── test
        └── java
            └── com.example.app
```

图 6-1 test 目录结构

现在，你可以在 src/test/java/com.example.app 中创建测试类了。

为了利用 JUnit 中的最新功能，建议使用 JUnit 4。你可以通过为测试构建添加依赖来确保使用的是 JUnit 4：

```
dependencies {
        testCompile 'junit:junit:4.12'
}
```

注意，在这里，我们使用的是 testCompile，而不是 compile。使用该配置可确保依赖只在运行测试时构建，而不会在分发的 app 中打包。在常规的 assemble 任务中，通过 testCompile 添加的依赖将永远不会被包含在 releases 的 APK 中。

如果在你的构建类型或 product flavor 中有任何特定的条件，那么你都可以针对该特殊构建添加只用于测试的依赖。举个例子，如果你只想添加 JUnit 测试到 paid flavor 中，那么你可以像下面这样操作：

```
dependencies {
        testPaidCompile 'junit:junit:4.12'
}
```

当一切设置完毕后，就可以开始编写一些测试用例了。下面是测试两数相加方法的简单例子：

```
import org.junit.Test;
```

```
import static org.junit.Assert.assertEquals;
public class LogicTest {
        @Test
        public void addingNegativeNumberShouldSubtract() {
            Logic logic = new Logic();
            assertEquals("6 + -2 must be 4", 4, logic.add(6, -2));
            assertEquals("2 + -5 must be -3", -3, logic.add(2, -5));
        }
}
```

只需执行 gradlew test，就可以通过 Gralde 运行所有的测试了。如果你只想在某个构建 variant 上运行测试，那么只需简单添加该 variant 的名称即可。举个例子，如果你只想在 debug variant 上运行所有测试，那么只需运行 gradlewtestDebug 即可。如果一个测试用例失败了，则 Gradle 会在命令行界面上打印错误消息。如果所有的测试都运行成功，则 Gradle 会显示 **BUILD SUCCESSFUL** 信息。

一个失败的测试用例会导致整个 test 任务失败，并立即停止整个进程。这意味着，在失败的情况下，不是所有的测试用例都会被执行。如果你想确保所有的构建 variant 都执行整套测试，请使用 continue：

```
$ gradlew test -continue
```

你也可以通过在相应的目录下存储测试类，来编写针对某一构建 variant 的特定测试用例。例如，如果你想在付费版本的 App 中测试特定的行为，那么可以将测试类放在 src/testPaid/java/ com.example.app 文件夹中。

如果你不想运行整套测试用例，而是只针对一个特定的测试类，那么可以像下面这样使用测试标识：

```
$ gradlew testDebug --tests="*.LogicTest"
```

执行测试 task 不仅会运行所有的测试用例，而且还会在 app/build/reports/tests/debug/index.html 中创建一份测试报告。当出现故障时，这份报告可以很容易地找到问题，这在测试用例被自动化执行时非常有用。Gradle 会为每个你运行测试用例的构建 variant 都生成一份报告。

如果所有的测试用例均运行成功，那么你的单元测试报告将如图 6-2 所示。

图 6-2 单元测试报告

你也可以在 Android Studio 中运行测试用例。这样你就可以在 IDE 中立即得到反馈，并且可以点击失败的测试用例导航到相应的代码，如图 6-3 所示。

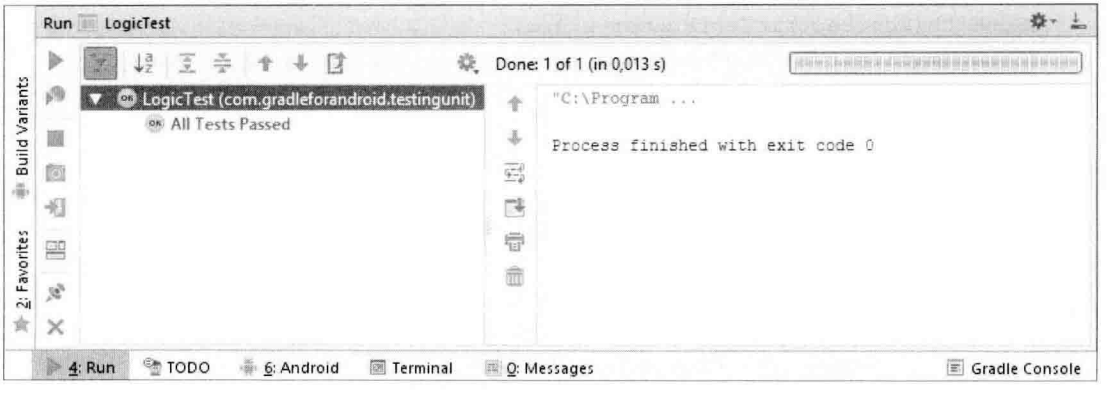

图 6-3 导航到相应的代码

如果你想测试包含引用 Android 特有的类或资源的部分代码，则常规的单元测试将无法满足需求。你可能已经尝试过，并得到 `java.lang.RuntimeException: Stub!` 错误。为了解决这个问题，你需要在你的 Android SDK 中实现所有的方法，或使用一个模拟框架。幸运的是，有几个依赖库考虑到了 Android SDK。这些依赖库中最有名的是 Robolectric，它提供了一种简单的方法来测试 Android 的功能，而不需要设备或模拟器。

6.1.2 Robolectric

使用 Robolectric，你可以利用 Android SDK 和资源来编写测试用例，并将所有的测试用例运行在 Java 虚拟机内。这意味着你不需要一个运行设备或模拟器来在你的测试用例中使用 Android 资源，从而加快测试 App 或 library 的 UI 组件行为的速度。

为了使用 Robolectric，你需要添加一些测试依赖。除了 Robolectric 本身，你还需要包含 JUnit，并且如果你使用了 Android support library，那么你还需要 Robolectric shadow：

```
apply plugin: 'org.robolectric'
   dependencies {
       compile fileTree(dir: 'libs', include: ['*.jar'])
       compile 'com.android.support:appcompat-v7:22.2.0'
       testCompile 'junit:junit:4.12'
       testCompile'org.robolectric:robolectric:3.0'
       testCompile'org.robolectric:shadows-support:3.0'
}
```

Robolectric 测试类像常规的单元测试类一样，被创建在 src/test/java/com.example.app 目录。所不同的是，你现在可以编写包含 Android 类和资源的测试用例。例如，该测试验证在点击某个特定按钮之后，某个 TextView 的文本变化：

```
@RunWith(RobolectricTestRunner.class)
@Config(manifest = "app/src/main/AndroidManifest.xml", sdk = 18)
public class MainActivityTest {
       @Test
       public void clickingButtonShouldChangeText() {
           AppCompatActivity activity = Robolectric.buildActivity
             (MainActivity.class).create().get();
           Button button = (Button)
             activity.findViewById(R.id.button);
           TextView textView = (TextView)
             activity.findViewById(R.id.label);
           button.performClick();
           assertThat(textView.getText().toString(), equalTo
             (activity.getString(R.string.hello_robolectric)));
       }
}
```

Robolectric 在 Android Lollipop 和兼容库中有一些已知的问题。如果你遇到关于依赖库缺失资源的错误，则可以像下面这样来修复它。

你需要在模块中添加一个名为 `project.properties` 的文件，然后添加数行：

```
android.library.reference.1=../../build/intermediates/
exploded-aar/com.android.support/appcompat-v7/22.2.0
android.library.reference.2=../../build/intermediates/
exploded-aar/com.android.support/support-v4/22.2.0
```

它们会帮助 Roboectric 找到兼容库资源。

6.2 功能测试

功能测试用于测试应用中的几个组件是否可以一起正常工作。例如，你可以创建一个功能测试用例来确认某个按钮是否打开了一个新的 `Activity`。Android 中有几个功能测试框架，但最简单最基础的测试框架是 Espresso 框架。

Espresso

Google 创造了 Espresso 框架，使其更容易为开发人员编写功能测试。该依赖库由 Android support repository 提供，所以你可以使用 SDK Manager 来安装它。

为了在设备上运行测试用例，你需要定义一个测试执行器，通过测试支持依赖库。Google 提供了 `AndroidJUnitRunner` 测试执行器，其可以让你在 Android 设备上运行 JUnit 测试类。测试执行器将加载应用 APK 和测试 APK 到一个设备上，并运行所有的测试用例，然后根据测试结果生成测试报告。

在下载了测试支持依赖库之后，即可设置测试执行器：

```
defaultConfig {
      testInstrumentationRunner
        "android.support.test.runner.AndroidJUnitRunner"
}
```

在可以开始使用 Espresso 之前，你还需要设置一些依赖：

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

```
    compile 'com.android.support:appcompat-v7:22.2.0'

    androidTestCompile 'com.android.support.test:runner:0.3'
    androidTestCompile 'com.android.support.test:rules:0.3'
    androidTestCompile
      'com.android.support.test.espresso:espresso-core:2.2'
    androidTestCompile
      'com.android.support.test.espresso:espresso-contrib:2.2'
}
```

在开始使用 Espresso 之前，你还需要引用测试支持依赖库和 espresso-core。最后一个依赖——espresso-contrib，是 Espresso 的补充依赖库，但不属于核心依赖库。

注意，这些依赖使用的是 androidTestCompile 配置，而不是之前我们使用的 testCompile 配置。其可用来区分单元测试和功能测试。

如果你试图现在就运行测试，那么你会遇到这样的错误：

```
Error: duplicate files during packaging of APK app-androidTest.apk
     Path in archive: LICENSE.txt
     Origin 1: ...\hamcrest-library-1.1.jar
     Origin 2: ...\junit-dep-4.10.jar
```

错误描述得非常详细。因为一个重复的文件，Gradle 不能完成整个构建。幸运的是，该文件只是许可证描述，所以我们可以将其从构建中剔除。错误本身也包含了应如何修改的信息：

```
You can ignore those files in your build.gradle:
   android {
   packagingOptions {
        exclude 'LICENSE.txt'
     }
}
```

一旦构建文件设置完毕，你就可以开始添加测试用例了。相较于常规的单元测试，功能测试被放置在不同的目录。就像依赖配置，你需要使用 androidTest，而不是仅 test 本身，所以功能测试的正确目录是 src/androidTest/java/com.example.app。下面是一个测试类，用来检查 MainActivity 中 TextView 的文本是否正确：

```
@RunWith(AndroidJUnit4.class)
@SmallTest
public class TestingEspressoMainActivityTest {
       @Rule
       public ActivityTestRule<MainActivity> mActivityRule = new
```

```
        ActivityTestRule<>(MainActivity.class);
    @Test
    public void testHelloWorldIsShown() {
        onView(withText("Hello world!")).check
          (matches(isDisplayed()));
    }
}
```

在运行 Espresso 测试之前，你需要确保你有一个设备或模拟器。如果忘记连接设备，则在执行测试任务时将会抛出异常：

```
Execution failed for task ':app:connectedAndroidTest'.
>com.android.builder.testing.api.DeviceException:
java.lang.RuntimeException: No connected devices!
```

一旦连接了一个设备或模拟器，你就可以使用 gradlewconnectedCheck 来运行你的 Espresso 测试。该任务将会执行 connectedAndroidTest 和 createDebugCoverageReport。connectedAndroidTest 用来在所有连接设备的 debug 构建上运行全部测试用例。而 createDebugCoverageReport 则用来创建一份测试报告。

你可以在应用目录下的 build/outputs/reports/androidTests/connected 文件夹下找到生成的测试报告。打开 index.html，可以看到报告如图 6-4 所示。

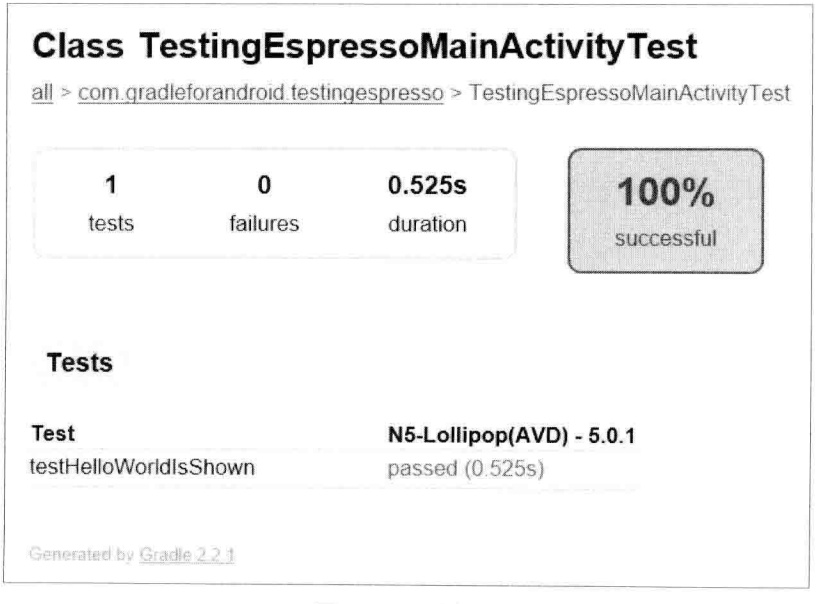

图 6-4 测试报告

功能测试报告展示了测试用例运行在哪个设备和哪个 Android 版本。你可以在多设备上同时运行这些测试用例，这样会更快地找到设备或版本相关的 bugs。

如果你想在 Android Studio 中获得测试用例的反馈，则可以在 IDE 中，设置 run/debug 配置来直接运行这些测试用例。Android Studio 工具栏上有一个配置选择器，你可以选择你想使用的 run/debug 配置，如图 6-5 所示。

图 6-5　配置选择器

设置一个新的配置的方法如下：点击 **Edit Configurations...**，打开配置编辑框，然后创建一个新的 Android 测试配置。选择模块并指定 `AndroidJUnitRunner` 为 instrumentation 执行器，如图 6-6 所示。

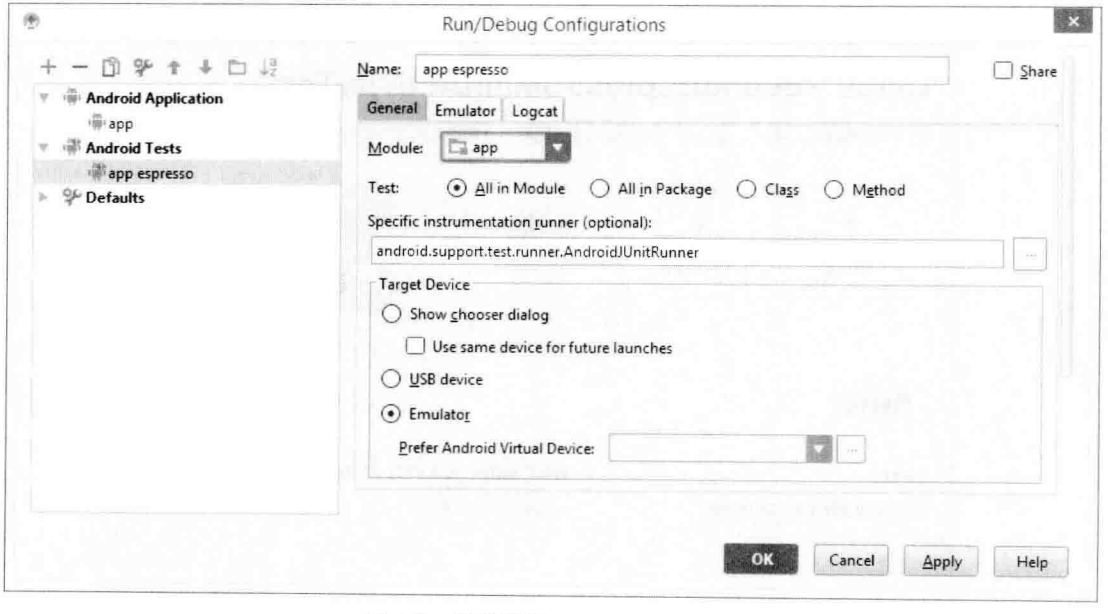

图 6-6　创建新的 Android 测试配置

一旦保存了这一新配置，你就可以在配置选择器中选择它，并点击 **Run** 按钮来运行所有的测试用例。

在 Android Studio 中运行 Espresso 测试时有一点需要注意：不会生成测试报告。因为 Android Studio 会执行 `connectedAndroidTest` 任务而不是 `connectedCheck` 任务，`connectedCheck` 任务是需要生成测试报告的。

6.3 测试覆盖率

在开始为你的 Android 项目编写测试用例之后，最好知道你的代码被测试用例覆盖了多少。针对 Java 的测试覆盖工具有很多，其中最受欢迎的是 Jacoco。Jacoco 也是默认被包含的依赖包，这也让 Jacoco 入门变得更加容易。

Jacoco

激活覆盖率报告的方法非常简单，你仅需在你测试的构建类型上设置 `testCoverageEnabled = true` 即可。为 debug 的构建类型激活测试覆盖率，就像下面这样：

```
buildTypes {
     debug {
       testCoverageEnabled = true
     }
}
```

当激活了测试覆盖率后，在执行 `gradlew connectedCheck` 时，会自动创建覆盖率报告。创建报告自身的任务是 `createDebugCoverageReport`。即便其没有记录，甚至在你运行 `gradlew` 任务时，其没有出现在任务列表中，你也可以直接运行它。然而，由于 `createCoverageReport` 依赖于 `connectedCheck`，所以你不能分开执行它们。依赖于 `connectedCheck`，也意味着你需要一个连接的设备或模拟器来生成一份测试覆盖率报告。

在该任务被执行之后，你可以在 `app/build/ outputs/reports/coverage/debug/index.html` 文件夹下找到覆盖率报告。由于每个 variant 都可以有不同的测试用例，因此每一个构建 variant 都有其自己的目录用于存放报告。测试覆盖报告如图 6-7 所示。

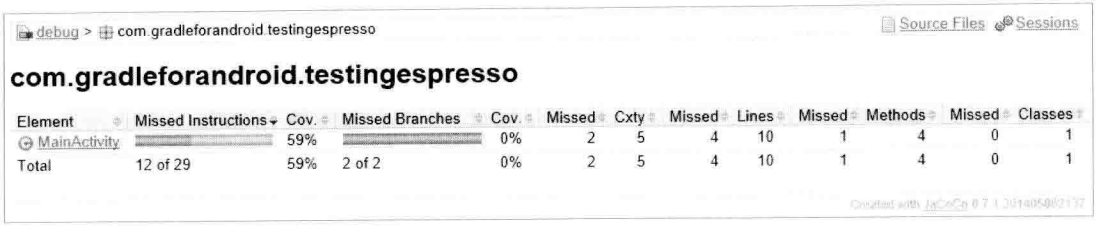

图 6-7　测试覆盖报告

这份报告在类层次上展示了很好的覆盖概述，你可以通过点击类名获得更多的信息。在大部分详细视图中，通过有用的颜色代码文件视图，你可以看到哪些文件被测试过，哪些没有。

如果你想指定 Jacoco 版本，则只需在构建类型中添加 Jacoco 配置代码块，定义其版本号即可：

```
jacoco {
    toolVersion = "0.7.1.201405082137"
}
```

注意，完全没有必要准确地定义为某一版本，因为 Jacoco 会无视它。

6.4　总结

本章，我们研究了测试 Android 应用和依赖库的多种选择。从简单的单元测试开始，随后研究了如何通过 Robolectric 测试 Android 特性。之后，我们学习了功能测试，了解了 Espresso。最后，介绍了如何激活测试覆盖率报告，并通过其查看整套测试需要改进的地方。现在你已经知道如何通过 Gradle 和 Android Studio 来运行整套测试，并生成覆盖率报告，你可以自己写测试用例了。在第 8 章中，我们将介绍通过持续集成工具来进行自动化测试的多种方式。

下一章，也是覆盖自定义构建进程最重要的部分之一：创建自定义任务和插件。该章也会简要介绍 Groovy。学习 Groovy 不仅对创建任务和插件大有帮助，还可以让你能够更容易地理解 Gradle 是如何工作的。

7

创建任务和插件

至此，我们已为 Gradle 构建了各种操纵属性，并学习了如何运行任务。在本章中，我们将深入理解这些属性，并开始创建自己的任务。一旦学会了如何编写自己的 tasks，我们将可以更进一步，研究如何制作可在多个项目中重复使用的插件。

在研究如何自定义任务之前，我们需要学习一些重要的 Groovy 概念。了解 Groovy 的工作原理可以使自定义任务和插件变得更加容易。了解 Groovy 还可以帮助我们理解 Gradle 是如何工作的，以及为什么构建配置文件可以按照它们的方式工作。

在本章，我们将介绍以下主题：

- 理解 Groovy
- 任务入门
- Hooking Android 插件
- 编写自定义插件

7.1 理解Groovy

由于大部分 Android 开发人员都精通 Java 开发，所以研究 Groovy 相比 Java 是如何工作的肯定很有趣。如果你是名 Java 开发者，那么可以很容易地理解 Groovy。不过在没有介绍之前，编写自己的 Groovy 代码还将是件困难的任务。

使用 Groovy 的一个好方法是使用 Groovy 控制台。该应用带有 Groovy SDK，可以很容易地使用 Groovy 语句，同时得到即时响应。Groovy 控制台也可以处理纯 Java 代码，这也让比较 Java 和 Groovy 代码变得更加容易。你可以从 Groovy 网站 `http://groovy-lang.org/download.html` 下载包含 Groovy 控制台的 Groovy SDK。

7.1.1 简介

Groovy 是从 Java 衍生出来的，运行在 Java 虚拟机上的敏捷语言。其目标是，不管是作为脚本语言，还是编程语言，都可简单、直接使用。在本节中，我们将比较 Groovy 和 Java，以便更容易地掌握 Groovy 是如何工作的，使读者清楚地看到两种语言之间的差异。

在 Java 中，打印字符串到屏幕上，代码如下：

```
System.out.println("Hello, world!");
```

在 Groovy 中，则通过下面代码来实现：

```
println 'Hello, world!'
```

你会注意到以下几个主要不同之处：

- 没有 `System.out` 命名空间
- 方法周围没有括号
- 一行末尾没有分号

这个例子在字符串周围使用了单引号。对于字符串你既可以使用单引号，也可以使用双引号，但它们有不同的用途。双引号字符串可以插入表达式。插值是评估包含占位符的字符串，并通过它们的值替换占位符的过程。这些占位符表达式可以是变量或方法。包含一个方法或多个变量的占位符表达式，需要有 $ 前缀并被花括号包裹。包含一个单独变量的占位符表达式可以只含有 $ 前缀。下面是 Groovy 字符串插值的一些例子：

```
def name = 'Andy'
def greeting = "Hello, $name!"
def name_size "Your name is ${name.size()} characters long."
```

`greeting` 变量包含字符串“Hello, Andy”，`name_size` 变量包含“Your name is 4 characters long.”

字符串插值还允许你动态执行代码。下面这个例子是打印当前日期：

```
def method = 'toString'
new Date()."$method"()
```

在 Java 中，这种用法看起来会很奇怪，但在动态编程语言中，这是正常的语法。

7.1.2 类和成员变量

在 Groovy 中创建一个类和在 Java 中创建一个类相似。下面是一个仅包含一个成员变量的类：

```
class MyGroovyClass {
        String greeting

        String getGreeting() {
            return 'Hello!'
        }
}
```

请注意，无论是类，还是成员变量，都没有明确的访问修饰符。Groovy 中的默认访问修饰符与 Java 不同。类和方法一样，是公有的，而类成员却是私有的。

要想使用 MyGroovyClass，则需要为它创建一个实例：

```
def instance = new MyGroovyClass()
instance.setGreeting 'Hello, Groovy!'
instance.getGreeting()
```

你可以使用关键字 def 来创建新的变量。一旦有一个类的新的实例，你就可以操作它的成员变量了。在 Groovy 中，访问修饰符是被自动添加的。你可以像我们在定义 MyGroovy-Class 中的 getGreeting() 那样来覆盖它。如果没有指定，那么你仍然可以在类中使用每个成员变量的 getter 和 setter 方法。

当你试图直接调用一个成员变量时，实际上，你调用的是 getter 方法。这就意味着你不必输入 instance.getGreeting()，可以用更简短的 instance.greeting 来代替：

```
println instance.getGreeting()
println instance.greeting
```

这两句代码将打印的内容完全相同。

7.1.3 方法

就像使用变量一样，你无须为你的方法定义一个特定的返回类型。但指定返回类型对你来

说没有任何影响，哪怕仅仅是为了更加清晰。Java 和 Groovy 的另一个不同是，在 Groovy 中，方法的最后一行通常默认返回，即使没有使用 return 关键字。

为了演示 Java 和 Groovy 之间的差异，我们用下面这个 Java 例子，返回了一个数字的平方：

```
public int square(int num) {
        return num * num;
}
square(2);
```

你需要指明该方法是公开访问的，返回类型是什么，参数类型是什么。在方法的结尾，你还需指定返回值的类型。

同样的方法在 Groovy 中是这样定义的：

```
def square(def num) {
        num * num
}
square 4
```

无论是返回类型，还是参数类型都没有明确的定义。这里使用了 def 关键字，而不是一个明确的类型，在没有使用 return 关键字的情况下，方法隐晦返回了一个值。然而，为了结构清晰，依然建议使用 return 关键字。当调用该方法时，不需要括号或分号。

还有另外一种利用 Groovy 来定义新方法的更简短的方式。同样的 square 方法也可以这样定义：

```
def square = { num ->
        num * num
}
square 8
```

这不是一个常规的方法，而是一个 closure。closure 的概念并没有在 Java 中以相同的方式存在，但它在 Groovy 和 Gradle 中发挥了显著作用。

7.1.4 Closures

Closures 是匿名代码块，可以接受参数和返回值。它们可以被视为变量，被当作参数传递给方法。

就像上一个例子一样，你可以在花括号之间添加一段代码块，来简单定义一个 clousure。如果想更详细点，则还可以添加定义类型，就像下面这样：

```
Closure square = {
```

```
        it * it
}
square 16
```

添加 Closure 类型可以让每个使用该代码的人都能清楚地知道，一个 closure 被定义了。前面的例子也引入了内含的未分类参数 it 的概念。如果你没有明确给 closure 指定一个参数，则 Groovy 会自动添加一个。这个参数通常被称为 it，你可以在所有的 closures 中使用它。如果调用者没有指定任何参数，则 it 为空。it 可以让你的代码更加简洁，其前提是，closure 中只有一个参数。

在 Gradle 中，我们时时和 closures 打交道。本书中，我们曾提到过 closures 作为代码块。举个例子，这意味着 android 和 dependencies 代码块都是 closures。

7.1.5 集合

当在 Gradle 中使用 Groovy 时，有两个重要的集合类型：lists 和 maps。

在 Groovy 中创建一个新的 list 非常容易，无须初始化，像下面这样，即可简单地创建一个 list：

```
List list = [1, 2, 3, 4, 5]
```

迭代一个 list 也非常的简单，你可以使用 each 方法来迭代 list 中的所有元素：

```
list.each() { element ->
        println element
}
```

each 方法使你能够访问 list 中的每个元素。你可以使用前面提到的 it 变量，来让代码更加简洁：

```
list.each() {
        println it
}
```

另外一个集合类型 Map.Maps，在 Gradle 中非常重要，它可在多个 Gralde 设置和方法中使用。一个 Map 就是一个包含所有 key-value 键值对的列表。你可以这样定义一个 map：

```
Map pizzaPrices = [margherita:10, pepperoni:12]
```

可使用 get 方法或方括号来访问 map 中的特定项目：

```
pizzaPrices.get('pepperoni')
pizzaPrices['pepperoni']
```

Groovy 也有一个针对该功能的简写。你可以使用点符号来获取 map 元素，使用 key 来检索 value：

```
pizzaPrices.pepperoni
```

7.1.6 Gradle 中的 Groovy

学习了 Groovy 基础之后，回头再看 Gradle 的构建文件，你会发现很有趣。注意，这使理解为什么这样配置语法变得更加容易。例如，下面一行代码，Android 插件被应用到构建：

```
apply plugin: 'com.android.application'
```

这段代码完全是 Groovy 的简写。如果在没有任何简写的情况下，编写它，则会是这样：

```
project.apply([plugin: 'com.android.application'])
```

在没有 Groovy 简写的情况下重写该行，可以清楚地看到 `apply()` 是 `Project` 类的一个方法，Project 类是每个 Gradle 构建的基础构建代码块。`apply()` 需要一个参数，其是一个 key 为 `plugin`，value 为 `com.android.application` 的 Map。

另外一个例子是 `dependencies` 代码块。之前，我们定义依赖如下：

```
dependencies {
        compile 'com.google.code.gson:gson:2.3'
 }
```

现在我们知道该代码块是一个 closure，将 `dependencies()` 方法传递给 `Project` 对象。该 closure 被传递给一个包含 `add()` 方法的 `DependencyHandler`。该方法接收三个参数：一个定义配置的字符串，一个定义依赖标志的对象，一个针对依赖特定属性的 closure。当你将其完整写出时，应如下所示：

```
project.dependencies({
        add('compile', 'com.google.code.gson:gson:2.3', {
            // Configuration statements
    })
})
```

至此，我们对研究的构建配置文件开始有更多的理解，现在你应该知道其背后的含义了吧！

如果你想了解更多关于 Gradle 底层使用 Groovy 的方式，则可以将官方文档的 project 部分作为出发点。你可以在 `http://gradle.org/docs/current/javadoc/org/gradle/api/Project.html` 中找到它。

7.2 任务入门

自定义 Gradle tasks 可以显著提高一个开发者的日常生活。任务可以操作存在的构建进程，添加新的构建步骤，或影响构建输出。你可以运行简单的任务，例如通过 hooking 到 Gradle 的 Android 插件，给一个生成的 APK 重命名。任务也能运行更加复杂的代码，举个例子，你可以在应用打包之前生成几张不同密度的图片。一旦知道如何创建自己的任务，你就可以在构建过程的任何细节上进行修改。当你学习了如何 hook 到 Android 插件之后，更是如此。

7.2.1 定义任务

任务属于一个 `Project` 对象，并且每个任务都可以执行 `task` 接口。定义一个新任务的最简单的方式是，执行将任务名称作为其参数的任务方法：

```
task hello
```

其创建了任务，但当你执行时，它不会做任何事情。为了创建一个有用的任务，你需要添加一些动作。初学者通常会犯的一个错误是像下面这样创建任务：

```
task hello {
    println 'Hello, world!'
}
```

当你执行该任务时，会看到输出如下：

```
$ gradlew hello
Hello, world!
:hello
```

从输出来看，你可能觉得该任务运行了，但实际上，“Hello, world!”在执行该任务之前就已被打印出来了。为了理解发生了什么，我们需要回到最初。在第 1 章中，我们讨论了 Gradle 构建的生产周期。在任一 Gradle 构建中，都有三个阶段：初始化阶段、配置阶段和执行阶段。当像上个例子那样以相同方式添加代码到一个任务时，你实际上是设置了任务的配置。即使你执行了不同的任务，“Hello, world!”也依然会出现。

如果你想在执行阶段给一个任务添加动作，则可以使用下面的表示法：

```
task hello << {
    println 'Hello, world!'
}
```

唯一的不同就是 closure 之前的 <<，其告知 Gradle，代码在执行阶段执行，而不是在配置

阶段。

为了演示不同之处，请看以下构建文件：

```
task hello << {
  println 'Execution'
}

hello {
  println 'Configuration'
}
```

我们定义了任务 hello，它会在执行时打印到屏幕。我们还在配置阶段定义了任务 hello，其会打印 Configuration 到屏幕。即使配置代码块在实际任务定义代码之后定义，其依然会先执行。下面是上一个例子的输出：

```
$ gradlew hello
Configuration
:hello
Execution
```

错误使用配置阶段导致任务失败是一个常见的错误。当你开始创建自己的任务时，请记住这一点。

Groovy 有很多简写，在 Gradle 中定义任务的常用方式有以下几种：

```
task(hello) << {
     println 'Hello, world!'
}

task('hello') << {
     println 'Hello, world!'
}

tasks.create(name: 'hello') << {
     println 'Hello, world!'
}
```

前两个代码块只是以两种不同的方式通过 Groovy 来实现相同的事情。你可以使用括号，但你不需要这么做。你也不需要给参数加上单引号。在这两个代码块中，我们可以调用 task() 方法，其需要两个参数：一个是名为任务的字符串，另一个是 closure。task() 方法

是 Gradle Project 类的一部分。

最后一个代码块没有使用 task() 方法。相反，它利用了一个名叫 tasks 的对象。tasks 对象是 TaskContainer 的实例，存在于每个 Project 对象中。该类提供了一个 create() 方法，需要一个 Map 和一个 closure 作为参数，最终返回一个 Task。

为了方便起见，大部分在线例子和教程将使用简写。然而在学习阶段，编写完整形式的代码更加有利于学习。通过这种方式，Gradle 将显得少了些魔法，但也更加容易理解发生了什么。

7.2.2 任务剖析

Task 接口是所有任务的基础，其定义了一系列属性和方法。所有这些都是由一个叫作 DefaultTask 的类实现的。这是标准的任务实现方式，你创建的每一个新的任务，都是基于 DefaultTask 的。

从技术上来讲，DefaultTask 并不是真正实现 Task 接口所有方法的类。Gradle 有一个叫作 AbstractTask 的内部类，其包含了所有方法的实现。因为 AbstractTask 是内部类，所以我们不能重写它。因此，我们专注于 DefaultTask，因为其源自 AbstractTask，且可被重写。

每个任务都包含一个 Action 对象的集合。当一个任务被执行时，所有这些动作会以连续的顺序被执行。你可以使用 doFirst() 和 doLast() 方法来为一个任务添加动作。这些方法都是以一个 closure 作为参数，然后被包装到一个 Action 对象中的。

如果你想在执行阶段执行你的代码，那么你需要使用 doFirst() 或 doLast() 来为一个 task 添加代码。我们之前用来定义 tasks 的左位移运算符(<<)，就是 doFirst() 方法的简写。

下面是使用 doFirst() 和 doList() 的例子：

```
task hello {
     println 'Configuration'

     doLast {
       println 'Goodbye'
     }

     doFirst {
       println 'Hello'
     }
}
```

当你执行 hello task 时，输出如下：

```
$ gradlew hello
Configuration
:hello
Hello
Goodbye
```

即使打印“Goodbye”的代码在打印“Hello”的代码之前定义，但当执行 task 时，它们仍会按正确的顺序执行。你甚至可以多次使用 doFirst() 和 doLast()，如下所示：

```
task mindTheOrder {
    doFirst {
      println 'Not really first.'
    }

    doFirst {
      println 'First!'
    }

    doLast {
      println 'Not really last.'
    }

    doLast {
      println 'Last!'
    }
}
```

执行这个 task 的输出如下：

```
$ gradlew mindTheOrder
:mindTheOrder
First!
Not really first.
Not really last.
Last!
```

注意，doFirst() 总是添加一个动作到 task 的最前面，而 doLast() 总是添加一个动作到最后面。这意味着，当你使用这些方法时，需要小心，特别是当顺序十分重要的时候。

当涉及到给 tasks 排序时，你可以使用 mustRunAfter() 方法。该方法将影响 Gradle 如

何构建依赖关系图。当使用 mustRunAfter() 时，你需要指定，如果两个任务都被执行，那么必须有一个任务始终先执行。

```
task task1 << {
    println 'task1'
}
task task2 << {
    println 'task2'
}
task2.mustRunAfter task1
```

同时运行 task1 和 task2，不管你指定了什么样的顺序，task1 总是在 task2 之前执行。

```
$ gradlew task2 task1
:task1
task1
:task2
task2
```

在这两个任务之间，mustRunAfter() 方法不会添加任何依赖，因而我们可以只执行 task2 而不执行 task1。如果你需要一个任务依赖于另一个，那么可使用 dependsOn() 方法。下面用一个例子来说明 mustRunAfter() 和 dependsOn() 之间的差别：

```
task task1 << {
       println 'task1'
}
task task2 << {
       println 'task2'
}
task2.dependsOn task1
```

当你试图只执行 task2 而不执行 task1 时，结果如下：

```
$ gradlew task2
:task1
task1
:task2
task2
```

使用 mustRunAfter()，当同时执行 task1 和 task2 时，task1 总是在 task2 之前执行，两者也可以独立执行。使用 dependsOn()，即使在未明确提及的情况下，tasks2 的执行也总是会触发 task1。这是一个重要的区别。

7.2.3 使用任务来简化 release 过程

在发布一个 Android 应用到 Google Play 商店之前，你需要使用证书对其签名。要想做到这一点，你需要创建自己的 keystore，其中包含一对私钥。当你有了自己的 keystore 和私钥后，你就可以在 Gradle 中按如下方式定义配置了：

```
android {
    signingConfigs {
        release {
            storeFile file("release.keystore")
            storePassword "password"
            keyAlias "ReleaseKey"
            keyPassword "password"
        }
    }

    buildTypes {
        release {
            signingConfig signingConfigs.release
        }
    }
}
```

这种方法的缺点是，你的 keystore 密码是以明文的形式存放在依赖仓库中的。如果你正在为一个开源项目工作，那么这是明确禁止的。这会使得任一同时访问 keystore 文件和密码的人，都能使用你的身份发布应用程序。为了防止这种情况发生，你可以创建一个任务，在你每次装配发布包的时候，都询问发布密码。这样做有点麻烦，而且使得你不可能构建服务器来自动生成发布版本。有一个好的解决方案是创建一个配置文件，用来存放 keystore 密码，而不是将其包含到依赖仓库中。

在项目的根目录中创建一个名为 `private.properties` 的文件，然后添加一行代码：

```
release.password = thepassword
```

我们假设 keystore 和密码本身相同。如果你有两个不同的密码，那么可以添加第二属性，这很容易。

一旦设置完毕，你就可以定义一个名为 `getReleasePassword` 的新任务了：

```
task getReleasePassword << {
      def password = ''
```

```
        if (rootProject.file('private.properties').exists()) {
            Properties properties = new Properties();
            properties.load( rootProject.file
    ('private.properties').newDataInputStream())
            password = properties.getProperty('release.password')
        }
}
```

该任务会在项目的根目录搜寻一个叫作 private.properties 的文件。如果该文件存在，则任务将加载其内容的所有属性。properties.load() 方法搜寻 key-value 键值对，就像我们在属性文件中定义的 release.password 那样。

为了确保任何人在没有私有属性文件的情况下仍可运行脚本。或当属性文件存在但密码属性不存在的情况，可添加一个密码属性作为备用，以便运行脚本。如果密码依然为空，则可以在控制台询问密码：

```
if (!password?.trim()) {
     password = new String(System.console().readPassword
             ("\nWhat's the secret password? "))
}
```

在 Groovy 中，检查字符串是否为 null 或空的过程很简单。该问题使用 password?.trim() 来做 null 检查，如果 password 为空，则其会阻止 trim() 方法被调用。因为在一个 if 判断句中，无论是 null 还是空字符串都等于 false。

使用 new String() 是很有必要的，因为 System.readPassword() 返回的是字符数组，其需要被显示转换为字符串。

一旦有了 keystore 密码，我们就可以在 release 构建中配置签名属性了：

```
android.signingConfigs.release.storePassword = password
android.signingConfigs.release.keyPassword = password
```

现在，我们的任务已经完成了，我们需要确保当执行 release 构建时，该任务被执行。要想做到这一点，需要在 build.gradle 文件中添加如下代码：

```
tasks.whenTaskAdded { theTask ->
       if (theTask.name.equals("packageRelease")) {
           theTask.dependsOn "getReleasePassword"
       }
 }
```

这段代码通过添加一个 closure 来链接（hooks into）到 Gradle 和 Android 插件，其会在任务被添加到依赖关系图时运行。在 packageRelease 任务执行之前，不需要密码，所以我们可以确保 packageRelease 依赖于我们的 getReleasePassword 任务。我们不能只使用 packageRelease.dependsOn() 的原因是，Gradle 的 Android 插件是基于构建 variants 动态生成的 packaging 任务。这意味着在 Android 插件发现所有构建 variant 之前，packageRelease 任务都不会存在，即发现过程是在每个单独构建之前。

在添加任务和构建钩子之后，执行 gradlew assembleRelease 的结果如图 7-1 所示。

```
:GradleForAndroid:compileReleaseAidl UP-TO-DATE
:GradleForAndroid:compileReleaseRenderscript UP-TO-DATE
:GradleForAndroid:generateReleaseBuildConfig UP-TO-DATE
:GradleForAndroid:generateReleaseAssets UP-TO-DATE
:GradleForAndroid:mergeReleaseAssets UP-TO-DATE
:GradleForAndroid:generateReleaseResValues UP-TO-DATE
:GradleForAndroid:generateReleaseResources UP-TO-DATE
:GradleForAndroid:mergeReleaseResources UP-TO-DATE
:GradleForAndroid:processReleaseManifest UP-TO-DATE
:GradleForAndroid:crashlyticsGenerateResourcesRelease
:GradleForAndroid:processReleaseResources
:GradleForAndroid:generateReleaseSources
:GradleForAndroid:compileReleaseJava UP-TO-DATE
:GradleForAndroid:proguardRelease UP-TO-DATE
:GradleForAndroid:dexRelease UP-TO-DATE
:GradleForAndroid:crashlyticsStoreDeobsRelease
:GradleForAndroid:crashlyticsUploadDeobsRelease
:GradleForAndroid:lintVitalRelease
:GradleForAndroid:compileReleaseNdk UP-TO-DATE
:GradleForAndroid:getReleasePassword
If you want to make it easier on yourself, create a file called private.properties in the root of the
roject and add "release.password = [password]" to it. Then the script will read that property instead
f asking for the password every time.
> Building 87% > :GradleForAndroid:getReleasePasswordWhat's the secret password?
:GradleForAndroid:processReleaseJavaRes UP-TO-DATE
:GradleForAndroid:validateReleaseSigning
:GradleForAndroid:packageRelease
:GradleForAndroid:zipalignRelease
:GradleForAndroid:assembleRelease

BUILD SUCCESSFUL

Total time: 22.926 secs
```

图 7-1　执行 gradlew assembleReleaser 的结果

如图 7-1 所示，private.properties 文件不可用，所以任务会在控制台询问密码。在这种情况下，我们添加了一个消息，用来说明如何创建属性文件以及添加密码属性，以使将来的构建更加容易。一旦我们的任务获得了 keystore 密码，Gradle 就能够打包我们的应用并完成构建。

为了使这个任务能够正常工作，hook 到 Gradle 和 Android 插件是很有必要的。这是一个强大的概念，我们会在下一节中探讨。

7.3 Hook到Android插件

在开发 Android 时，我们希望大部分 tasks 都能涉及到 Android 插件。通过 hook 到构建进程来增加任务的行为是可行的。在上一个例子中，我们已经看到如何为自定义任务添加一个依赖，以在常规的构建过程中包含新的任务。在本节，我们将研究一些 Android 特定构建 hooks 的可行性。

Hook 到 Android 插件的方式之一是操控构建 variants。这种做法相当简单，你只需用下面的代码片段遍历应用的所有构建 variant 即可：

```
android.applicationVariants.all { variant ->
     // Do something
}
```

你可以使用 `applicationVariants` 对象来得到构建 variant 的集合。一旦你有一个构建 variant 的引用，你就可以访问和操作它的属性，比如名称、说明等。如果你想对 Android 依赖库使用相同的逻辑，请使用 `libraryVariants` 来代替 `applicationVariants`。

注意，我们通过 `all()` 来遍历构建 variant，而不是之前提到的 `each()`。这是因为 `each()` 会在构建 variant 被 Android 插件创建之前的评测阶段被触发。另外，`all()` 方法会在每次添加新项目到集合时被触发。

该 Hook 可以用来修改保存之前的 APK 的名称，并为文件名添加版本号。这使得维护 APK 的归档文件变得更加容易，因为无须手动编辑文件名。在下一节，我们将看到如何实现这一目标。

7.3.1 自动重命名 APK

一个常用案例是操纵构建过程来重命名 APKs，在它们被打包之后，添加版本号。你可以通过遍历应用的构建 variant，来改变它们的输出属性 `outputFile`，代码片段如下：

```
android.applicationVariants.all { variant ->
     variant.outputs.each { output ->
        def file = output.outputFile
        output.outputFile = new File(file.parent,
            file.name.replace(".apk", "-${variant.versionName}.apk"))
     }
}
```

每个构建 variant 都有输出集合。Android 应用的唯一输出就是一个 APK。每个输出对象都有一个名为 outputFile 类型的属性。一旦你知道其输出路径，就可以操纵它。在本例中，我们添加了 variant 的版本号到文件名中。APK 的最终命名将会是 app-debug-1.0.apk，而不是 app-debug.apk。

结合 Android 插件构建钩子的威力，以及 Gradle 任务的简易性，开辟了无限的可能。在下一节，我们将看到如何为一个应用的所有构建 variant 创建一个任务。

7.3.2 动态创建新的任务

由于 Gradle 的工作方式和任务的构造方式，我们可以轻松地基于 Android 构建 variant，在配置阶段创建我们自己的任务。为了证明这一强大的概念，我们将创建一个不是用于 install 的任务，并且使其能在 Android 应用的任何构建 variant 上运行。install 任务是 Android 插件的一部分，但如果通过命令行界面使用 installDebug 任务来安装 App，那么在安装完成时，你仍然需要手动启动它。我们将在本节创建新的任务，来省掉最后一步。

从我们之前使用的 hook 到 applicationVariants 属性开始：

```
android.applicationVariants.all { variant ->
    if (variant.install) {
      tasks.create(name: "run${variant.name.capitalize()}",
        dependsOn: variant.install) {
          description "Installs the ${variant.description} and runs
            the main launcher activity."
        }
    }
}
```

对于每一个 variant，我们都会检查其是否是一个有效的 install 任务。该步不能省略，因为我们正在创建的新的任务依赖于 install 任务。一旦验证了 install task 的存在，我们就可以创建新的任务，并基于 variant 的名称为其命名。我们也让新的任务依赖于 variant.install，其将在我们的任务执行之前触发 install 任务。首先在 tasks.create() closure 内添加一份说明，此说明将在你执行 gradlew 任务时显示。

除添加说明外，我们还需要添加实际的任务动作。在本例中，我们希望启动应用。你可以通过 **Android Debug Tool（ADB）**，在一个连接的设备或模拟器上启动一个应用：

```
$ adb shell am start -n com.package.name/com.package.name.Activity
```

Gradle 有一个叫作 exec() 的方法，其可以执行一个命令行进程。为了使 exec() 能够工作，我们需要提供一个存在的可执行的 PATH 环境变量。我们还需要通过 args 属性传递所有参数，因为其需要一系列字符串，下面是代码块：

```
doFirst {
        exec {
            executable = 'adb'
            args = ['shell', 'am', 'start', '-n',
              "${variant.applicationId}/.MainActivity"]
        }
}
```

使用构建 variant 的 application ID，可获得完整的包名，包括后缀如果有的话。在这种情况下，后缀依然有一个问题。即便我们添加了后缀，activity 的类路径依然相同。例如下面的配置：

```
android {
        defaultConfig {
            applicationId 'com.gradleforandroid'
        }

        buildTypes {
            debug {
                applicationIdSuffix '.debug'
            }
        }
}
```

包名为 com.gradleforandroid.debug，但是 activity 的路径依然是 com.gradleforandroid.Activity。为了确保能够得到正确的 activity，我们需要从 applicationID 中剥离后缀：

```
doFirst {
        def classpath = variant.applicationId
        if(variant.buildType.applicationIdSuffix) {
            classpath -= "${variant.buildType.applicationIdSuffix}"
}
        def launchClass =
            "${variant.applicationId}/${classpath}.MainActivity"
        exec {
            executable = 'adb'
            args = ['shell', 'am', 'start', '-n', launchClass]
        }
}
```

首先，我们创建了一个基于 application ID 的变量，名为 classpath。然后我们通过 buildType.applicationIdSuffix 属性找到了后缀。在 Groovy 中，使用减运算符可以从一个字符串中减去另一个字符串。这些变化确保了，在安装应用之后，即便后缀被使用时，运行应用也不会失败。

7.4 创建自己的插件

如果你想在多个项目中复用一系列 Gradle tasks，那么提取这些 tasks 到一个自定义插件中将非常有用。这样就可以重用自己的构建逻辑，并与他人分享。

插件既可以用 Groovy 编写，也可以使用其他 JVM 语言编写，例如 Java 和 Scala。实际上，Gradle 的 Android 插件大部分都是由 Java 结合 Groovy 编写的。

7.4.1 创建一个简单的插件

你可以在 build.gradle 文件内创建一个插件，来提取已存储在你构建配置文件中的构建逻辑。这也是入门自定义插件的最简单的方式。

要想创建一个插件，首先需要创建一个实现 Plugin 接口的类。我们将利用在本章编写的代码，动态创建 run 任务。我们的插件类将如下所示：

```
class RunPlugin implements Plugin<Project> {
    void apply(Project project) {
        project.android.applicationVariants.all { variant ->
            if (variant.install) {
                project.tasks.create(name:
                    "run${variant.name.capitalize()}",
                    dependsOn: variant.install) {
                        // Task definition}
                }
            }
        }
    }
```

Plugin 接口定义了一个 apply() 方法。Gradle 在插件被构建文件使用时，调用此方法。Project 将作为参数被传递，这样插件就可以配置项目或使用它的方法和属性了。在前面的例子中，我们无法从 Android 插件中直接调用属性，而是需要先访问 project 对象。注意，在我们的

自定义插件被应用之前，Android 插件需要被应用到项目中来，否则，project.android 将导致异常。

这个任务的代码和之前的相同，只有一个方法调用不同：不是调用 exec()，这里需要调用 project.exec()。

为了确保该插件会应用到我们的构建配置，需要在 build.gradle 中加入这一行：

```
apply plugin: RunPlugin
```

7.4.2 分发插件

为了分发插件，并与他人分享，你需要把它移一个独立的模块（或项目中）。一个独立的插件有其自己的构建文件来配置依赖关系和分发方式。这个模块会产生一个包含插件类和属性的 JAR 文件。你可以在多个模块和项目中使用此 JAR 文件，并且可以与他人分享。

与任一 Gradle 项目相同，我们可以创建一个 build.gradle 文件来配置构建：

```
apply plugin: 'groovy'

dependencies {
        compile gradleApi()
        compile localGroovy()
}
```

由于是在 Groovy 中编写插件，因此我们需要应用这个 Groovy 插件。Groovy 插件继承自 Java 插件，其允许你构建和打包 Groovy 类。Groovy 和普通的 Java 都支持它，所以如果你喜欢，你可以任意混合。你甚至可以走得更远，例如使用 Groovy 继承一个 Java 类，或其他方式。这使其易于上手，即便你对全部使用 Groovy 没有信心。

我们的构建配置文件包含两个依赖：gradleApi() 和 localGroovy()。从自定义插件中访问 Gradle 命名空间需要用 Gradle API，localGroovy() 是 Groovy SDK 的一个分发包，来自于 Gradle 安装包。Gradle 默认情况下提供了这些依赖。如果 Gradle 没有提供这些现成的依赖，那么我们必须手动下载和引用它们。

如果你打算公开发布你的插件，那么请确保在构建配置文件中，指定 group 和 version 信息，就像这样：

```
group = 'com.gradleforandroid'
version = '1.0'
```

在开始使用独立模块的代码之前，我们首先需要确保使用了正确的目录结构，如图 7-2 所示。

```
plugin
└── src
    └── main
        ├── groovy
        │   └── com
        │       └── package
        │           └── name
        └── resources
            └── META-INF
                └── gradle-plugins
```

图 7-2　目录结构

和其他 Gradle 模块一样，我们需要提供一个 src/main 文件夹。由于这是一个 Groovy 项目，因此 main 的子文件夹被称之为 groovy，而不是 java。main 还有另外一个叫作 resources 的子文件夹，我们将用其来指定插件的属性。

在包目录中，我们创建了一个叫作 RunPlugin.groovy 的文件，在这里我们定义了插件的类：

```
package com.gradleforandroid

import org.gradle.api.Project
import org.gradle.api.Plugin

class RunPlugin implements Plugin<Project> {
       void apply(Project project) {
           project.android.applicationVariants.all { variant ->
               // Task code
           }
       }
}
```

为了使 Gradle 能够找到该插件，我们需要提供一个属性文件。添加该属性文件到 src/main/resources/META-INF/gradle-plugins/ 目录。该文件的名称需要和我们插件的 ID 相匹配。对于 RunPlugin 而言，文件名为 com.gradleforandroid.run.properties，内容为：

```
implementation-class=com.gradleforandroid.RunPlugin
```

该属性文件只包含实现 Plugin 接口的包和类名。

当插件和属性文件都准备好之后，我们可以通过使用 gradlew assemble 命令来构建插件。该命令会在构建输出文件夹中创建一个 JAR 文件。如果想将插件推送到 Maven 仓库，那么你首先需要应用 Maven 插件：

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

然后，你需要像下面这样配置 uploadArchives task：

```
uploadArchives {
       repositories {
           mavenDeployer {
             repository(url: uri('repository_url'))
}
       }
}
```

uploadArchives 任务是预定的任务。一旦在该任务上配置了一个仓库，就可以执行它来发布你的插件了。在本书中，我们不会讨论如何安装一个 Maven 仓库。

如果你想让你的插件公开，则可以考虑发布到 Gradleware 的插件门户（https://plugins.gradle.org）。在插件门户中有大量的 Gradle 插件，可以用来扩展 Gradle 的默认行为。你可以在 https://plugins. gradle.org/docs/submit 中找到如何发布一个有文档的插件的信息。

为自定义插件编写测试不在本书范围内。但如果你打算公开发布你的插件，那么强烈建议你编写测试。你可以在 Gradle 用户指南的 https://gradle.org/docs/ current/userguide/custom_plugins.html#N16CE1 中，找到更多关于为插件编写测试的信息。

7.4.3 使用自定义插件

为了使用插件，我们需要在 buildscript 代码块中添加其作为依赖。首先，我们需要配置一个新的仓库。仓库的配置取决于插件的分发方式。其次，我们需要在 dependencies 代码块中配置插件的 classpath。

如果你想包含上一个例子中创建的 JAR 文件，则可以定义一个 flatDir 仓库：

```
buildscript {
      repositories {
          flatDir { dirs 'build_libs' }
```

```
        }
        dependencies {
            classpath 'com.gradleforandroid:plugin'
        }
}
```

如果我们上传插件至 Maven 或 Ivy 仓库，将会有一点不同。我们在本书第 3 章中曾介绍过依赖管理，所以这里不再赘述。

在设置依赖之后，我们需要应用该插件：

```
apply plugin: com.gradleforandroid.RunPlugin
```

当使用 apply() 方法时，Gradle 会创建一个插件类的实例，执行插件本身的 apply() 方法。

7.5 总结

在本章中，我们介绍了 Groovy 和 Java 的不同之处，以及在 Gradle 中如何使用 Groovy。我们介绍了如何创建自己的任务，以及如何 hook 到 Android 插件，这给我们提供了很多操作构建过程或动态添加自己任务的方法。

在本章的最后一部分，我们研究了创建插件，并通过创建一个独立的插件来确保在多个项目中复用它。关于插件需要学习的东西还有很多，但不幸的是，本书不能覆盖一切。幸运的是，在 Gradle 用户指南中，地址为 https://gradle.org/docs/current/userguide/custom_plugins.html，有所有可能性的深入描述。

在下一章，我们将探讨持续集成（CI）的重要性。一个良好的 CI 系统，使得我们可以构建、测试、并一键部署应用和依赖库。一般而言，持续集成是自动化构建的重要组成部分。

8

设置持续集成

持续集成（CI）是一个开发实践，需要一个团队的开发者，每天多次地整合他们的工作。每一次推送到主仓库都需要一次自动构建验证。这种做法有助于尽早发现问题，从而加速开发，并提高代码的质量。伟大的 Martin Fowler 写了一篇以诠释概念和介绍最佳实践为主题的文章（`http://martinfowler.com/articles/continuousIntegration.html`）。

为 Android 设置 CI 的方式有许多种，使用最为广泛的是 Jenkins、TeamCity 和 Travis CI。Jenkins 有最大的生态系统，有大约一千个可用的插件。这也是很多开源贡献者共同努力的结果。TeamCity 是 JetBrains 的产品，JetBrains 是一家创造了 IntelliJ IDEA 的公司。Travis CI 相对而言更加年轻，主要集中在开源项目。

我们将研究这些 CI 系统，并介绍如何让 Gradle 在它们上面工作。在本章的最后，针对所有 CI 系统，我们将介绍一些让 CI 更加容易的 Gradle 技巧。

本章我们将介绍以下主题：

- Jenkins
- TeamCity
- Travis CI
- 自动化更进一步

8.1 Jenkins

Jenkins 最初是由 Sun Microsystems 公司在 2005 年作为 Hudson 发布的。多年来，它已经成长为 Java 社区最流行的 CI 系统。在 Sun 公司被 Oracle 收购后不久，Oracle 和 Java 社区就

有一次关于 Hudson 的冲突。当该问题不能得以解决时，社区只能继续为改名为 Jenkins 的项目服务，因为 Oracle 是 Hudson 名称的所有者。

Jenkins 的强大之处在于它的插件系统。谁在构建系统中有新的功能需求，就可以创建一个新的插件来扩展 Jenkins 的能力。这也是为什么为 Android 应用或依赖库设置自动化构建超级简单的原因。

8.1.1 设置 Jenkins

如果你尚未在你的构建机器上安装和运行 Jenkins 的话，则可以从网站（`https://jenkins-ci.org`）下载，然后按照步骤操作。

在实际开始设置 Jenkins 之前，你需要确保你拥有所有构建 Android 应用和 libraries 所需要的依赖库。为了能在 Java 中构建，你需要首先下载并安装 JDK，可以从 Java 网站下载(`http://www.oracle.com/technetwork/java/ javase/downloads/index.html`)。

你还需要确保你已经安装了 Android SDK 和构建工具，但没有必要在你的构建服务器上安装一个 IDE，除非你计划在构建机器上打开项目。如果只是想安装 SDK 工具，则可以从 Android 开发者网站下载（`https://developer. android.com/sdk/index.html#Other`）。一旦下载并安装了软件包，你就需要在 SDK 目录下运行 Android 可执行文件，以便安装所需要的 APIs 和构建工具。

Java 和 Android SDK 安装完成后，你需要将它们配置到 Jenkins。从打开你的浏览器开始，导航在构建服务器上 Jenkins 的主页。进入 **Manage Jenkins | Configure System**，然后滑动到 Global properties。添加两个环境变量：`ANDROID_HOME` 和 `JAVA_HOME`，然后设置它们的值到正确的目录，如图 8-1 所示。

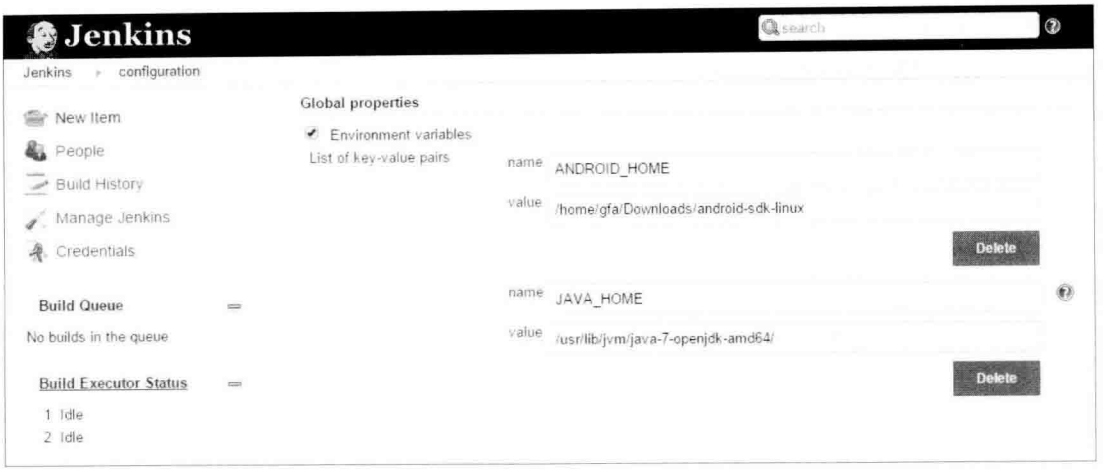

图 8-1 配制到 Jenkins

另外，我们还需要安装 Gradle 插件。打开 **Manage Jenkins | Manage Plugins**，导航到 **Available** 标签，然后搜索 `Gradle`。找到 Gradle 插件后，选中复选框，然后点击 **Download now and install after restart**，就可以在 Gradle 中创建构建步骤了。

8.1.2 配置构建

待安装完所有需要的东西后，你就可以在 Jenkins 中创建一个 CI 系统了。首先是设置 VCS 仓库，以便 Jenkins 知道从哪里可以获取你的项目源码。你可以基于仓库活动，使用构建触发器，自动构建你的 Jenkins 应用或依赖库，也可以选择手动构建。为了实际执行构建，你需要添加一个构建步骤用来调用 Gradle 脚本。你可以使用在 Android 项目中默认存在的 Gradle Wrapper 来配置 Jenkins。使用 Gradle Wrapper 不仅免去了在构建服务器上手动安装 Gradle 的需求，其还能让任何 Gradle 升级都可以自动处理。另外，其还能检查 **Make gradlew executable** 工具箱。当在 Microsoft Windows 机器上创建项目时，执行 Gradle Wrapper 可以解决权限问题。

你可以给构建步骤输入一个好的描述说明，还可以选择性地添加 `info` 和 `stacktrace` 这两个开关。`info` 开关被用来在构建过程中打印更多的信息，当出现问题时，这些信息会非常的有用。如果构建导致异常，则 `stacktrace` 开关会打印出该异常的堆栈轨迹。有时可能需要更多详细的信息，在这种情况下，你就可以使用 `full-stacktrace` 开关了。

要想完成配置，你需要指定你想执行的 Gradle 任务。首先，执行 `clean` 任务，以确保没有上一次构建遗留下的任何输出残留。其次，执行 `build` 任务，来触发一个构建的所有

variants。Jenkins 配置应如图 8-2 所示。

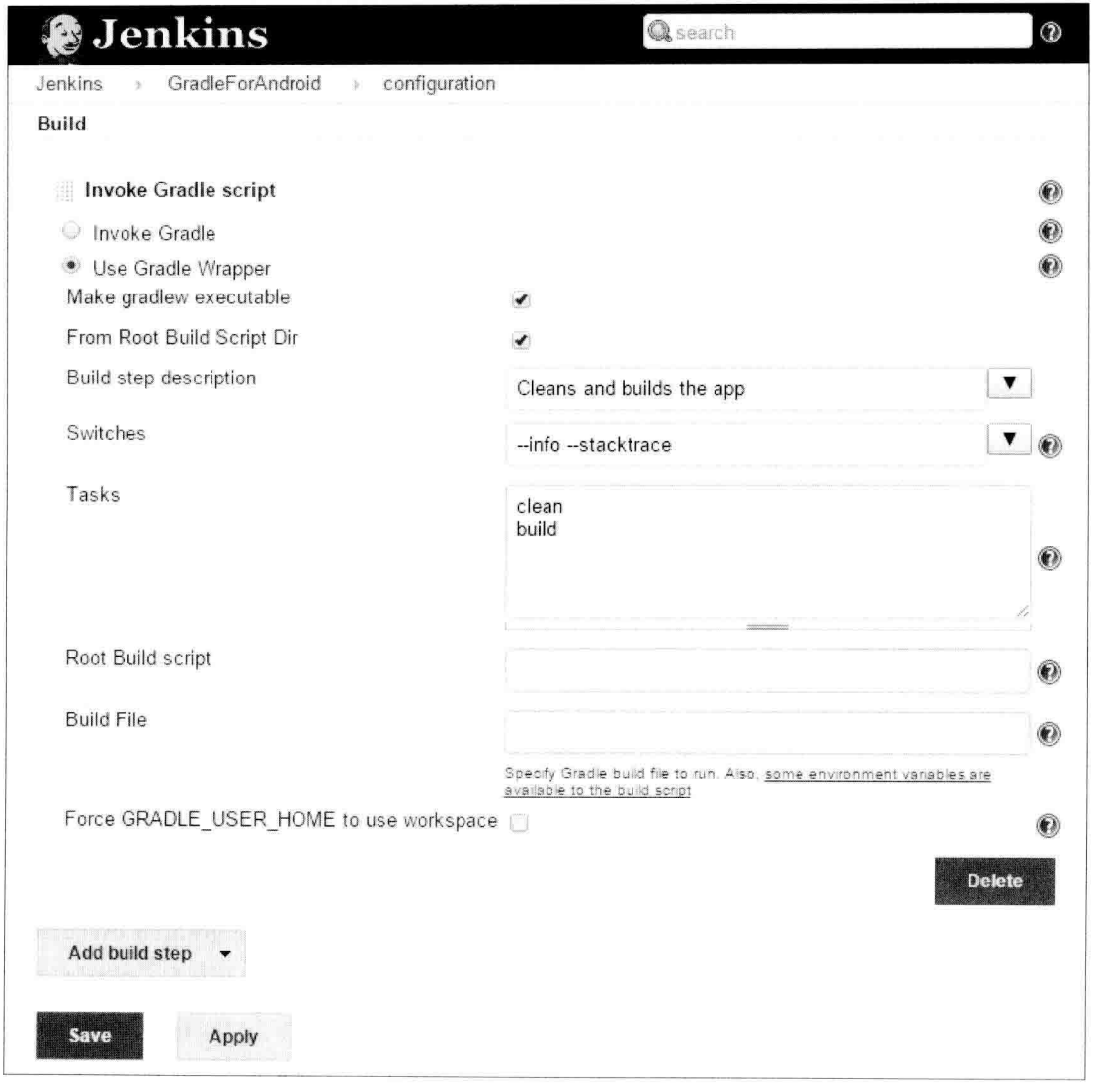

图 8-2 Jenkins 配置

保存项目配置后，你就可以运行构建了。

如果你的构建服务器被安装在一个 64 位的 Linux 机器上，则你可能会遇到这个异常 `java.io.IOException: Cannot run program "aapt": error=2, No such file or directory`。这是因为 AAPT 是 32 位的应用，在 64 位机器上运行时，需要一些额外的依赖库。你可以使用下面这个命令，来安装必要的依赖库。

```
$ sudo apt-get install lib32stdc++6 lib32z1
```

如果构建没有任何问题，其完成后，会为你所有的构建 variants 创建 APKs。你可以使用指定的 Gradle tasks 来分发这些 APKs。我们会在本章的最后部分介绍自动分发，因为其不针对任何构建系统，如图 8-3 所示。

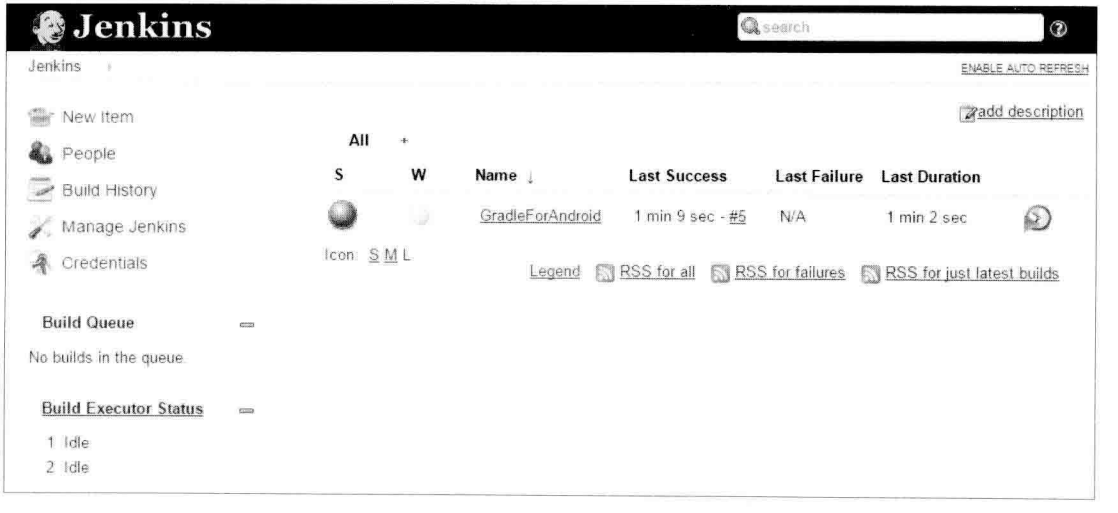

图 8-3 构建完成

8.2 TeamCity

和 Jenkins 不同，TeamCity 是一个拥有所有权的产品，只针对开源项目免费。TeamCity 是由 JetBrains 创建和管理的。JetBrains 公司的产品还有 IntelliJ IDEA，Android Studio 是基于此 IDE 开发的。TeamCity 支持在 Gradle 的外部进行 Android 构建。

8.2.1 设置 TeamCity

如果你尚未安装 TeamCity，则可以从 JetBrains 网站（https://www.jetbrains.com/teamcity）下载，然后按照步骤操作。

在使用 TeamCity 构建 Android 应用和依赖库之前，你需要确保在你的构建服务器上已经安装了 JDK、Android SDK 和 Android 构建工具。具体操作方法可参看 8.1 节内容。你还需要添加 ANDROID_HOME 到机器的环境变量，并指向你的 Android SDK 目录。

和 Jenkins 不同，TeamCity 不需要任何插件来触发 Gradle 构建，因为 TeamCity 内置了对运行 Gradle 的支持。

8.2.2 配置构建

要想设置 Android 构建，则需要从创建一个新的项目开始，并且你只需提供一个名称。一旦该项目被创建，就可以开始配置它了。首先，需要添加一个 VCS 根目录，以便 TeamCity 能够找到你的项目的源代码。然后你需要创建一个新的构建配置。你还需要连接到 VCS 根目录来构建配置。当设置完毕后，你可以添加一个新的构建步骤。如果点击了 **Auto-detect build steps** 按钮，则 TeamCity 将通过项目内容来自动确定必要的构建步骤。以基于 Gradle 基础的 Android 项目为例，其结果如图 8-4 所示。

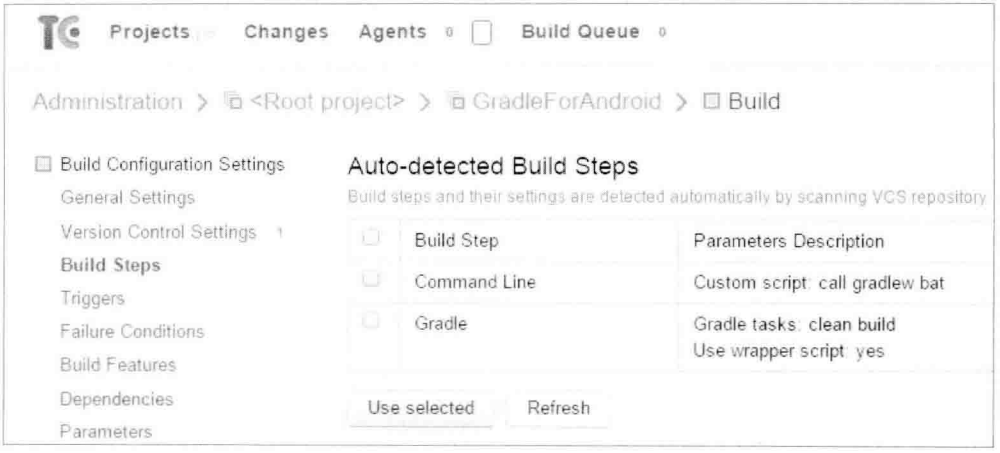

图 8-4 自动确定构建步骤

TeamCity 会检测项目是否使用了 Gradle，甚至会检测 Gradle wrapper 是否存在。你可以只选择 Gradle 的构建步骤，然后将其添加到构建配置中。如果不需要任何高级功能，那么这就

足够构建你的 Android 应用了。你可以打开项目概览，然后单击 Android 项目的 **Run...** 按钮来测试配置。

8.3 Travis CI

如果你的项目仓库是在 GitHub 上托管的，则可以使用 Travis CI 来自动构建。Travis CI (`https://travis-ci.org`) 是一个开源持续集成系统，可以免费使用公有仓库。Travis CI 的私有仓库是付费的，本书只介绍其免费版。

当一份新的提交被推送到仓库时，Trawis 会对其检测并自动开始一个新的构建。默认情况下，Travis 会构建所有分支，而不仅仅是 master 分支。其还会自动构建 pull requests，一个在开源项目中有用的特性。

由于 Travis 是内部工作，因此你无法自己配置构建服务器。相反，你需要创建一个包含所有信息的配置文件，因为 Travis 需要用它来构建你的应用或依赖库。

配置构建

如果你想在项目中使用 Travis 构建，那么首先你需要登录到 Travis CI，并连接到你的 GitHub 项目。完成之后，你就可以设置你想构建的项目了。

为了配置构建过程，Travis 要求你创建一个包含所有设置的名为 `.travis.yml` 的文件。为了配置 Android 项目，你需要确定语言，并添加一些 Android 的特有属性：

```
language: android
   android:
     components:
       # The build tools version used by your project
       - build-tools-22.0.1

       # The SDK version used to compile your project
       - android-22

       # Additional components
       - extra-android-m2repository
```

设置语言指定了你想运行哪一种构建过程。在本例中，你会构建一个 Android 应用。Android 特有属性包括你需要使用的构建工具的版本和 Android SDK 版本。Travis 在运行 `build` 任务之前会下载它们。如果使用的是支持依赖库或 Google Play Services，则需要明确指

定，因为 Travis 需要为这些依赖下载对应的仓库。

Travis CI 并没有强制你配置构建工具和 SDK 版本号，但是如果你指定的版本和`build.gradle`文件的版本一致，那么你将会遇到更少的问题。

如果是在 Microsoft Windows 上创建了一个 Android 项目，则要注意 Gradle Wrapper 文件的权限问题。因此，在运行实际构建脚本之前，应先确认权限。你可以像下面这样预构建步骤：

```
before_script:
     # Change Gradle wrapper permissions
     - chmod +x gradlew
```

将下面这行代码添加到 Travis 配置文件，以启动构建：

```
# Let's build
script: ./gradlew clean build
```

该命令会运行 Gradle Wrapper，就像你在开发机器上执行`clean`和`build`任务一样。

当完成了 Travis 构建配置后，你就可以提交和推送文件到项目的 GitHub 仓库了。如果全部设置正确，则 Travis 将按照你在 Travis 网设置的步骤启动构建过程。项目构建成功后如图 8-5 所示。

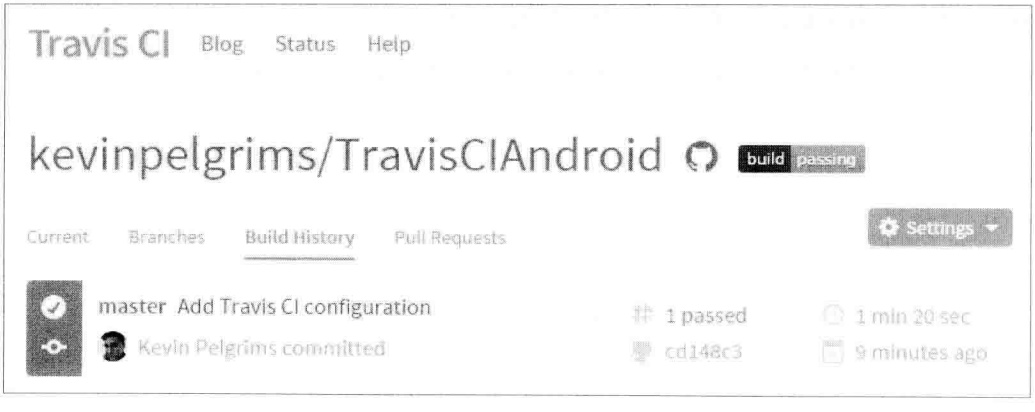

图 8-5　项目构建成功

在每次构建后，Travis 都会发送一份 E-mail 报告。如果你是一个定期获取 pull requests 的开源依赖库的维护者，那么这份报告将非常有用。当构建成功后，Travis 的 E-mail 报告如图 8-6 所示。

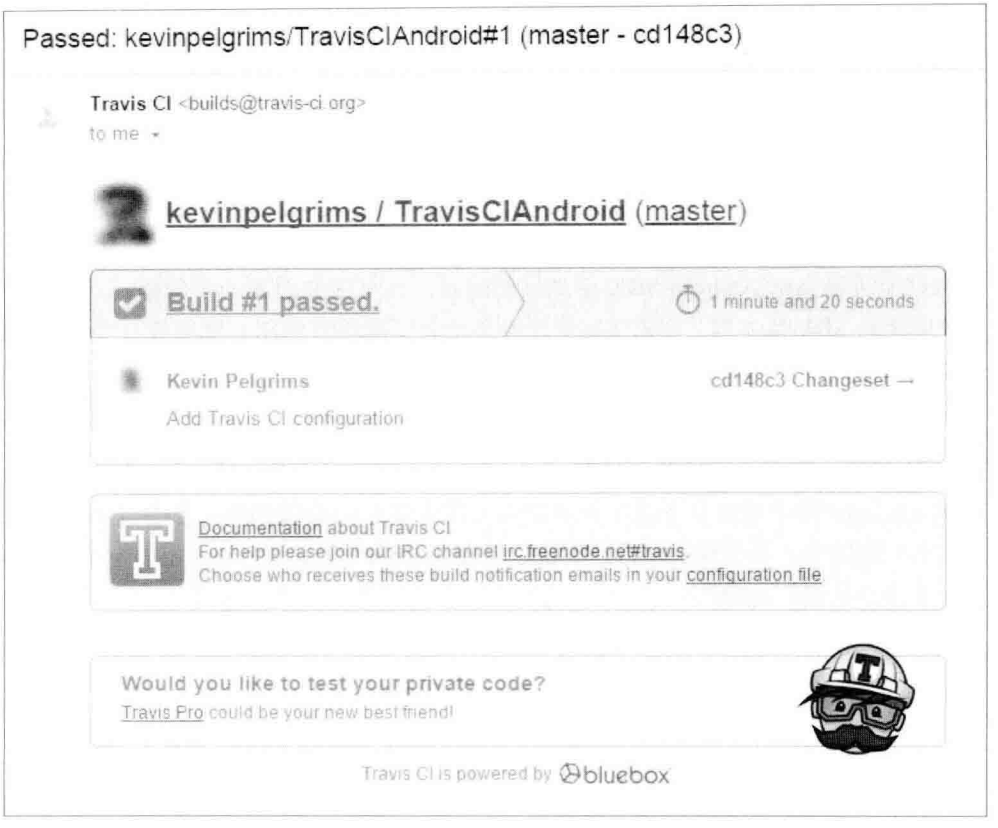

图 8-6 E-mail 报告

Travis 有一个很大的缺点，那就是速度。Travis 不会专门为你提供一台机器，它只是在你每次触发构建时开启一个虚拟机。这就意味着，Travis 在开始构建应用或依赖库之前，每次新的构建，其都需要下载和安装 Android SDK 和构建工具。

从另一方面来看，Travis 是免费和公开的，其让开源项目更加完美。当有人给你的代码提交了一个补丁时，Travis 会自动构建 pull requests，这会让你更加安心。

8.4 自动化进阶

无论是默认的还是通过插件，大多数现代的持续集成系统都支持 Gradle。这意味着你不仅可以构建应用或依赖库，还可以创建各种各样的 Gradle 任务来进一步自动化构建。通过 Gradle 任务定义额外构建步骤（而不是在 CI 系统内部）的优点是额外构建步骤更加便捷。在你的开

发机器上运行一个自定义 Gradle 任务非常的容易。另一方面，一个自定义 Jenkins 构建步骤不可能在未安装 Jenkins 的情况下运行。当某个 CI 系统有额外的构建步骤时，如果将其切换到其他 CI 系统，将非常困难。Gradle 任务可以很容易地移植到其他项目。本节我们将介绍进一步自动化构建，以及使用 Gradle 任务和插件来部署应用和依赖库的几种方式。

8.4.1 SDK mananger 插件

当构建服务器上的 Android SDK 不是最新的时，你可能会遇到一个问题。当你为你的应用或依赖库更新 SDK 版本时，你需要在构建服务器上安装新的 SDK。如果你有多个构建代理，这将会非常麻烦。

幸好社区提供了一个 Gradle 插件，专门用来检测构建依赖的 Android SDK 版本是否存在。如果 SDK 不存在，则插件会自动下载它。

SDK manager 插件不仅会下载在构建配置文件中指定的编译 SDK，还会下载构建工具和平台工具的正确版本。如果你的项目在支持依赖库或 Google Play Services 上有一个依赖，则该插件也会下载这些指定的版本。

SDK manager 插件是一个开源插件，你可以在 GitHub 上找到其源代码（`https://github.com/JakeWharton/sdk-manager-plugin`）。

8.4.2 运行测试

如果你想在构建服务器的构建过程期间，运行单元测试（JUnit or Robolectric），则只需添加相应的任务到 Gradle 中执行即可。如果想运行任意功能测试，则需要一个安装应用的模拟器，这样你就可以使用 `gradlew connectedAndroidTest` 来运行测试了。

运行一个模拟器最简单的方式是在构建服务器上启动一个模拟器，然后一直打开它。但这不是最佳解决方案，因为 Android 模拟器很容易随机崩溃，尤其是当它们持续开启数天时。

如果你正在使用 Jenkins，那么有一个叫作 Android Emulator Plugin 的插件，可以用来为你的应用或依赖库的每次构建启动一个模拟器（`https://wiki.jenkins-ci.org/display/JENKINS/Android+Emulator+Plugin`）。TeamCity 也有一个活跃的插件生态系统，并且有一个叫作 Android Emulator 的插件，它可以像 Jenkins 插件那样设置一个模拟器。你可以在官方的 TeamCity 插件页面（`https://confluence.jetbrains.com/display/TW/TeamCity+Plugins`）找到它。

Travis CI 可以启动一个模拟器，但目前还处于实验阶段。如果你想试试，则可以添加下面片段到 `.travis.yml` 配置文件，在 Travis 构建期间，启动一个 Android 模拟器：

```
# Emulator Management: Create, Start and Wait
before_script:
    - echo no | android create avd --force -n test -t android-22 --
      abi armeabi-v7a
    - emulator -avd test -no-skin -no-audio -no-window &
    - android-wait-for-emulator
    - adb shell input keyevent 82 &
```

android-wait-for-emulator 命令告知 Travis 等待模拟器启动。当模拟器启动后，adb shell input keyevent 82 & 被用来执行屏幕解锁。在此之后，你就可以告知 Gradle 运行测试了。

8.4.3 持续部署

为了帮助开发者自动部署 Android 应用，Google 发布了通过编程方式推送 APKs 到 Google Play 的开发者 API（https://developers.google.com/android-publisher）。该 API 不需要你打开浏览器，登录到 Google Play，再使用网页界面来上传 APKs，即你不需要基于 Google Play deueloper API 来创建你自己的发布脚本。你可以在一次成功构建之后，使用众多插件中的一个，直接从构建系统中推送 APKs 到 Google Play。

有一个叫作 **Google Play Android Publisher**（https://wiki. jenkins-ci.org/display/JENKINS/Google+Play+Android+Publisher+Plugin）的 Jenkins 插件可以为你处理这些。更好的选择是使用 Gradle 插件，这样你就可以在任何设备、任何持续集成系统上执行 publishing 任务了。Android 社区的一些人基于 Google Play Developer API 创建了一个 Gradle 插件，允许你配置完整的 publishing 流程。你可以在 GitHub 上找到 Gradle Play Publisher Gradle 插件的源代码（https://github.com/Triple-T/gradle-play-publisher），也可以通过 Maven Central 或 JCenter 访问。

在开始使用该插件之前，你需要添加下面代码到你的 main build.gradle 文件中：

```
buildscript {
        repositories {
            jcenter()
        }

        dependencies {
            classpath 'com.github.triplet.gradle:play-publisher:1.0.4'
        }
}
```

然后在 Android 模块的 build.gradle 文件中应用该插件：

```
apply plugin: 'play'
```

当你将 Gradle Play Publisher 插件应用到你的构建中时，你将可以使用下面几个新的任务：

- publishApkRelease：上传 APK 和最近的变化。
- publishListingRelease：上传详细描述和图片。
- publishRelease：上传一切。

如果你有不同的构建 variants，那么你也可以执行这些任务的特定版本 variant，例如，publishApkFreeRelease 和 publishApkPaidRelease。

为了访问 Google Play Developer API，你需要创建一个服务账户。创建账户超出了本书的范围，因此不再介绍。注意，如果你想使用 Gradle Play Publisher 插件，那么必须创建账户。你可以按照 Google Play Developer API 文档中的步骤来创建 https://developers.google.com/android- publisher/getting_started。

创建一个服务账户后，你就可以在你的构建文件中输入验证了，如下所示：

```
play {
        serviceAccountEmail = 'serviceaccount'
        pk12File = file('key.p12')
}
```

play 代码块是针对 Gradle Play Publisher 插件的特定属性。除服务账户验证外，你还可以指定 APK 的推送 track：

```
play {
        track = 'production'
}
```

默认路径是 alpha，但你可以将其更改为 beta 或 production。

8.4.4 Beta 分发包

Android 应用的 beta 测试有很多选择，例如 Google Play 商店上的 beta track。另外一个选择是和 Gradle 整合的非常好的 **Crashlytics**（https://crashlytics. Com）。Crashlytics 团队创建了一个自定义插件，不仅创建新的 Gradle tasks，用来发布构建到对应的平台，而且还 hooks 到 Android 插件的 tasks 来处理 ProGuard 的映射。

我们可以跟随 Crashlytics 网站的步骤，来入门 Crashlytics。一旦设置了 Crashlytics，其就会开始 hook 到你的构建。Crashlytics 插件暴露了一个名为 `crashlyticsUploadDistributionInternal` 的新任务，可以被用来上传 APKs 到 Crashlytics。为了推送你的应用的一个新版本，首先你需要使用 build 或 assemble 任务来构建它。一旦生成 APK，你就可以使用 `crashlyticsUploadDistributionInternal` 任务上传至 Crashlytics。Crashlytics 插件为你项目中的每个构建 variant 都创建了一个 upload 任务。

自定义 Gradle 插件让开发者上手 Crashlytics 变得更加容易。上传测试构建至 Crashlytics 也是一件轻而易举的事，因为你只需在构建过程期间执行一个额外的任务即可。这就是一个好的 Gradle 插件可以让开发者的生活更加简单的最佳例子。

8.5 总结

本章我们引入了一些流行的持续集成系统，并解释了如何使用它们来自动化构建 Android 应用和依赖库。同时，还介绍了如何配置 CI 系统来构建 Android 项目。然后我们研究了几个可以帮助我们进一步实现自动化构建和部署过程的 Gradle 插件，并解释了如何在构建服务器上自动运行测试。

在下一章，我们将研究几个 Gradle 的高级特性和基于 Gradle 构建的优化方案。同时也会介绍，如何通过在 Gradle 中直接使用 Ant tasks 来迁移一个大型的 Ant 构建配置，并一步步移植到 Gradle 中。

9

高级自定义构建

至此，你已经学习了 Gradle 是如何工作，如何创建属于自己的 tasks 和插件，如何运行测试，以及如何设置持续集成，你几乎可以称自己为 Gradle 专家了。本章包含了一些在前面章节中并没有提及的提示和技巧。这些提示和技巧可以让使用 Gradle 来构建、开发、部署 Android 项目变得更加简单。

本章，我们将介绍以下内容：

- 减少 APK 文件大小
- 加速构建
- 忽略 Lint
- 在 Gradle 中使用 Ant
- 高级应用部署

首先，我们将研究如何减少构建输出大小，以及为什么要这样做。

9.1 减少APK文件大小

在过去几年中，APK 文件的大小曾急剧增长态势。一般来说，其原因如下：Android 开发者获取了更多的依赖库，添加了更多的密度，Apps 增加了更多的功能。

但实际上我们应该让 APKs 尽可能的小。不仅是因为在 Google Play 中，APK 文件有 50MB 的限制，而且更小的 APK 意味着用户可以更快地下载和安装应用，并使它占用更小的内存。

本节我们将研究如何设置 Gradle 构建配置文件中的几个属性，以缩小 APK 文件。

9.1.1 ProGuard

ProGuard 是一个 Java 工具，其不仅可以缩减 APK 文件大小，还可以在编译期优化、混淆和预校验你的代码。其通过你应用的所有代码路径，来找到未被使用的代码，并将其删除。ProGuard 还会重命名你的类和字段。这一过程将保留应用的踪迹，让反编译工程师更加难以读懂代码。

在 Gradle 的 Android 插件中，其构建类型下面有一个叫作 `minifyEnabled` 的布尔类型属性，你需要将它设置为 true 来激活 ProGuard：

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

当 `minifyEnabled` 被设置为 `true` 后，在构建过程中，`proguardRelease` task 会被执行，并调用 ProGuard。

在激活 ProGuard 之后，应重新测试整个应用，因为 ProGuard 可能会移除一些你仍需要使用的代码。这也是很多开发者不喜欢 ProGuard 的原因。为了解决这个问题，你可以定义 ProGuard 规则，排除那些被删除或混淆的类。我们可以使用 `proguardFiles` 属性来定义包含 ProGuard 规则的文件。例如，为了保留一个类，你可以像下面这样添加一条简单的规则：

```
-keep public class <MyClass>
```

`getDefaultProguardFile('proguard-android.txt')` 方法从 Android SDK 的 `tools/proguard` 文件夹下的 `proguard-android.txt` 文件中获取默认的 ProGuard 设置。在 Android Studio 中，`proguard-rules.pro` 文件被默认添加到新的 Android 模块，所以你可以在该文件中简单地添加一些针对该模块的规则。

你构建的每个应用或依赖库都有不同的 ProGuard 规则，所以在本书中，我们不会考虑更多的细节。如果你想了解更多关于 ProGuard 和 ProGuard 规则的信息，则可以通过 `http://developer.android.com/tools/help/proguard.html`. 来查阅 Android ProGuard 的官方文档。

除了缩减 Java 代码外，还可以缩减使用过的资源。

9.1.2 缩减资源

当给 App 打包时，Gradle 和 Gradle 的 Android 插件可以在构建期间删除所有未使用的资源。如果你有旧的资源忘记删除，那么这个功能可能非常有用。另外一个使用案例是当你导入一个拥有很多资源的依赖库，而你只使用了其中的一小部分时，你可以通过激活缩减资源来解决这个问题。缩减资源的方式有两种：自动和手动。

1. 自动缩减

最简单的方式是在你的构建中设置 shrinkResources 属性。如果设置该属性为 true，则 Android 构建工具将自动判定哪些资源没有被使用，并将它们排除在 APK 外。

使用此功能有一个要求，即必须同时启动 ProGuard。这是因为缩减资源的工作方式是，直到代码引用这些资源被删除之前，Android 构建工具不能指出哪些资源没有被用到。

下面的代码片段展示了在某个构建类型中，如何配置自动化资源缩减：

```
android {
       buildTypes {
       release {
               minifyEnabled = true
               shrinkResources = true
              }
        }
}
```

如果你想看看在激活了自动化资源缩减之后，APK 缩减了多少，则可以运行 shrinkReleaseResources 任务。该任务会打印出包的大小缩小了多少：

```
:app:shrinkReleaseResources
Removed unused resources: Binary resource data reduced from 433KB
to 354KB: Removed 18%
```

你可以通过在构建命令中添加 --info 标志，来获得 APK 缩减资源的概览：

```
$ gradlew clean assembleRelease -info
```

当你使用该标志时，Gradle 会打印出许多关于构建过程的额外信息，包括最终构建不会输出的每个资源。

自动资源缩减有一个问题，即它可能移除了过多的资源，特别是那些被动态使用的资源可

能会被意外删除。为了防止这种情况的发生，你可以在 res/raw/ 下的一个叫作 keep.xml 的文件中定义这些例外。一个简单的 keep.xml 文件如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<resources xmlns:tools="http://schemas.android.com/tools"
    tools:keep="@layout/keep_me,@layout/also_used_*"/>
```

keep.xml 文件自身也将从最终的结果中被剥离出来。

2. 手动缩减

去除某种语言文件或某个密度的图片，是删减资源的一种比较好的方式。一些依赖库，例如 Google Play Services，其包含了多种语言。如果你的应用只支持一两种语言，那么在最终的 APK 中，包含所有语言的文件就会浪费许多资源。这时你就可以使用 resConfigs 属性来配置你想保留的资源，将其余部分删除。

如果你只想保留英语、丹麦语和荷兰语的字符串，则可以这样使用 resConfigs：

```
android {
        defaultConfig {
            resConfigs "en", "da", "nl"
        }
}
```

你也可以这样处理密度集合：

```
android {
        defaultConfig {
            resConfigs "hdpi", "xhdpi", "xxhdpi", "xxxhdpi"
        }
}
```

你甚至可以结合语言和密度。实际上，使用此属性可以限制每一种类型的资源。

如果设置 ProGuard 让你感觉很困难，或者你仅仅想在应用中去除不支持的语言资源或密度，那么你可以使用 resConfigs 来缩减资源。

9.2 加速构建

很多 Android 开发者在开始使用 Gradle 的时候，都会抱怨编译时间过长。和 Ant 相比，用 Gradle 构建需要更长的时间，因为你每执行一个任务时，Gradle 都要构建生命周期中的三个阶段。这使得整个过程非常方便配置，但是也相当的缓慢。下面就来介绍几种加快 Gradle 构建的方式。

9.2.1 Gradle 参数

加速 Gradle 构建的一种方式是改变一些默认设置。我们曾在第 5 章中提到并行执行构建，但还有些一些设置你可以调整。

这里只是回顾一下，你可以通过在根目录下的 `gradle.properties` 文件中设置一个属性来启动并行构建，只需添加下面一行代码即可：

```
org.gradle.parallel=true
```

另外一种方式是启动 Gradle daemon, 其会在你第一次运行构建时，开启一个守护进程。任何后续构建都将复用该守护进程，从而减少启动成本。只要你使用 Gradle, 该进程就一直存活，并且在空闲三个小时后终止。当你在很短的时间内多次使用 Gradle 时，使用 daemon 会非常有用。在 `gradle.properties` 文件中启动 daemon 的代码如下：

```
org.gradle.daemon=true
```

在 Android Studio 中，Gradle daemon 是默认开启的。这意味着在 IDE 内部的第一次构建后，下一次构建会快一些。然而如果你是通过命令行来构建界面的，则 Gradle daemon 是关闭的，除非你在配置中启动它。

你可以调整 Java 虚拟机的参数来加速编译。Gradle 有一个叫作 `jvmargs` 的属性可以让你为 JVM 的内存分配池设置不同的值。对构建速度有直接影响的两个参数是 `Xms` 和 `Xmx`。`Xms` 参数用来设置初始内存大小，`Xmx` 用来设置最大内存。你可以在 `gradle.properties` 文件中，手动设置这些值：

```
org.gradle.jvmargs=-Xms256m -Xmx1024m
```

你需要设置千兆字节所需要的量和单位：k 代表千字节，m 表示兆字节，g 代表千兆字节。默认情况下，最大内存分配（Xmx）被设置为 256 MB，初始内存分配（Xms）则没有被设置。最佳设置取决于你计算机的性能。

最后一个可以用来影响构建速度的可配属性是 `org.gradle.configureondemand`。如果你的项目中有多个模块，那么该属性会非常有用，因为，它会忽略正在执行的 task 不需要的模块来限制在配置阶段的时间消耗。如果你设置该属性为 `true`，则 Gradle 将在运行配置阶段前，指出哪一个模块有配置改变，哪一个没有。当你的项目中只有一个 Android 应用和一个依赖库时，该特性没有任何用处。如果你的项目中有很多松散耦合的模块，那么该特性会为你节省大量的构建时间。

系统级别的 Gradle 属性

如果你想将这些系统级别的属性，应用到所有基于 Gradle 的项目，那么你可以在你的 home 目录下的 `.gradle` 文件夹下创建一个 `gradle.properties` 文件。在微软的 Windows 中，到该目录的完整路径是 `%UserProfile%\.gradle`，在 Linux 和 Mac OS X 系统中，完整路径是 `~/.gradle`。在 home 目录下，而不是在项目层面上设置这些属性，是一个很好的做法。这样做的原因是，你通常想在构建服务器上降低内存消耗，而构建时间并不十分重要。

9.2.2 Android Studio

用来加速编译过程的 Gradle 属性还可以通过 Android Studio 设置配置。打开 **Settings** 对话框，然后导航至 **Build, Execution, Deployment | Compiler**，就可以找到编译设置了。在屏幕上，你可以找到并行构建、JVM 参数、配置 demand 等设置。这些设置只在基于 Gralde 的 Android 模块中显示，如图 9-1 所示。

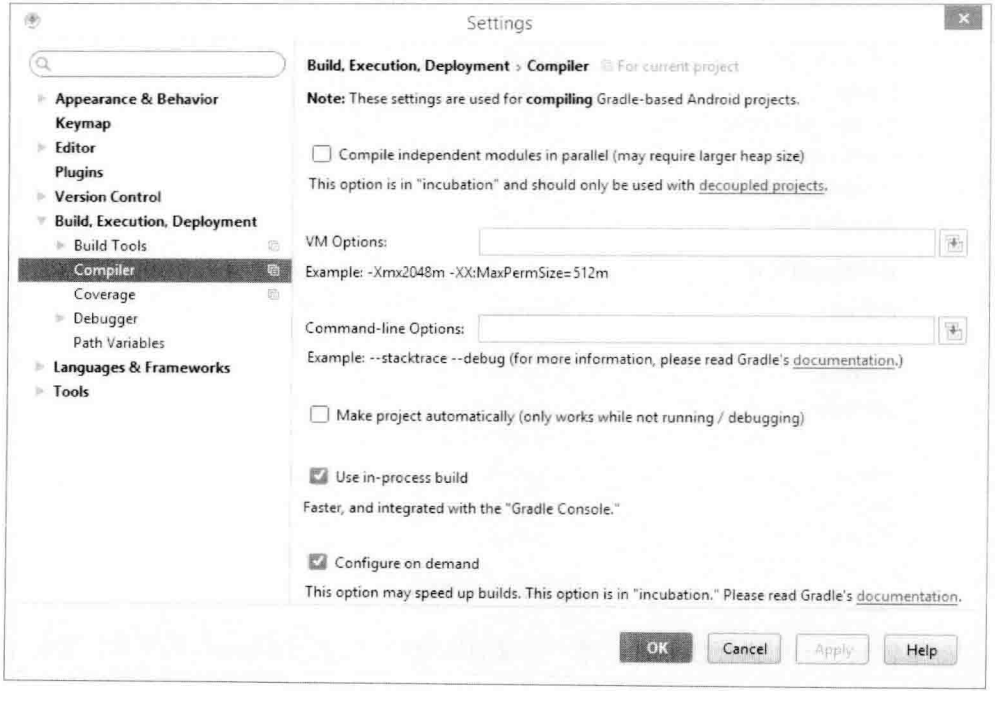

图 9-1 Settings

在 Android Studio 中配置这些设置，比在构建配置文件手动配置要它们容易得多，并且通过设置对话框来寻找那些影响构建过程的属性也十分简单。

9.2.3 Profiling

如果你想找出构建中让速度变慢的具体位置，则可以拆分整个构建过程。当执行一个 Gradle 任务时，你可以通过添加 `--profile` 标志来实现这一点。当你提供了该标志时，Gradle 会创建一份分拆报告，其会告诉你构建过程中究竟哪一部分最为耗时。一旦知道瓶颈在哪里，你就可以进行必要的修改。报告最终会被保存为一份 HTML 文件，存放在你模块中的 `build/reports/profile` 下。

图 9-2 是一个多模块项目执行构建任务后生成的报告。

Profile report

Profiled build: build

Summary | Configuration | Dependency Resolution | Task Execution

Summary

Description	Duration
Total Build Time	1m19.46s
Startup	0.867s
Settings and BuildSrc	0.323s
Loading Projects	0.094s
Configuring Projects	2.192s
Task Execution	3m22.12s

Configuration

Project	Duration
All projects	2.192s
:backend	0.879s
:androidlib	0.657s
:	0.486s
:mobile	0.088s
:wear	0.048s
:javalib	0.034s

图 9-2 分拆报告

分拆报告展示了在执行任务时，每一阶段的耗费时间。在 Summary 的下面，是每个模块花费在配置阶段的时间。报告中还有两个部分未被显示在截图中。**Dependency Resolution** 部

分展示了为每个模块处理所用掉的时间。最后的 **Task Execution** 部分则包含了非常详细的任务执行概述。概述中列出了每个独立任务的执行时间，并对执行时间从高到低进行了排序。

9.2.4 Jack 和 Jill

如果你愿意使用实验工具，那么你可以使用 Jack 和 Jill 来加快构建。**Jack（Java Android Compiler Kit）**是一个新的 Android 构建工具链，其可以直接编译 Java 源码为 Android Dalvik 的可执行格式。它有自己的 `.jack` 依赖库格式，也采用了打包和缩减。**Jill（Jack Intermediate Library Linker）**是一个可以将 `.aar` 和 `.jar` 文件转换成 `.jack` 依赖库的工具。这些工具还在实验阶段，可用来改善编译时间和简化 Android 构建过程。不建议在项目的生产版本中使用 Jack and Jill，但因为它们可以获取到，所以你可以试试。

为了能够使用 Jack 和 Jill，你需要使用版本为 21.1.1 以上的构建工具，版本为 1.0.0 以上的 Gradle Android 插件。启动 Jack 和 Jill 就像在 `defaultConfig` 代码块中设置一个属性那样简单：

```
android {
        buildToolsRevision '22.0.1'
        defaultConfig {
          useJack = true
        }
}
```

你也可以在某个构建类型或 product flavor 中启动 Jack 和 Jill。这样，你就可以一边使用常规的构建工具链，一边使用实验性构建了：

```
android {
        productFlavors {
            regular {
                useJack = false
            }

            experimental {
                useJack = true
            }
        }
}
```

只要你设置了 `useJack` 为 `true`，就再也不能通过 ProGuard 进行缩小和混淆了，但你仍然可以使用 ProGuard 的语法规则来指定某些规则和异常。例如，使用我们之前在 ProGuard 部

分提及的 proguardFiles 方法。

9.3 忽略Lint

当你通过 Gradle 执行一个 release 构建时，你的代码中就会执行一个 Lint 检查。Lint 是一种静态代码分析工具，它会在你的布局和 Java 代码中标记潜在 bug。在某些情况下，甚至会阻塞构建过程。如果你之前在项目中从未使用 Lint，而现在想将项目迁移至 Gradle，那么 Lint 可能出现会很多错误。为了让构建能够正常工作，你可以禁用 abortOnError，让 Gradle 忽略 Lint 错误，防止它们影响构建。这只是一个临时的解决方案，因为忽略 Lint 错误会导致一些问题，如缺失翻译，会直接导致应用崩溃。为了防止 Lint 阻塞构建过程，应像下面这样禁用 abortOnError：

```
android {
        lintOptions {
            abortOnError false
        }
}
```

临时禁用 Lint 异常中止，使得从一个已存在的 Ant 构建过程迁移到 Gradle 变得更加容易。另外一个让过渡更加平滑的方式是直接在 Gradle 中执行 Ant 任务。

9.4 在Gradle中使用Ant

如果你在设置 Ant 构建过程中投入了大量时间，那么切换到 Gradle 听起来可能很不可思议。在这种情况下，Gradle 不仅可以执行 Ant 任务，还可以扩展它们。这意味着，你可以一步步从 Ant 迁移到 Gradle，而不是在改造整个构建配置中花费数天时间。

Gradle 使用 Groovy 的 **AntBuilder** 来进行 Ant 整合。AntBuilder 允许你执行任何标准的 Ant 任务、自定义 Ant 任务和整个 Ant 构建。这也使得你可以在 Gradle 构建配置中定义 Ant 属性。

9.4.1 在 Gradle 中运行 Ant 任务

在 Gradle 中运行一个标准的 Ant 任务十分简单。你只需在任务名称的前面加上 ant. 即可。例如，你可以使用该任务来创建一个 archive：

```
task archive << {
        ant.echo 'Ant is archiving...'
```

```
    ant.zip(destfile: 'archive.zip') {
        fileset(dir: 'zipme')
    }
}
```

该任务在 Gradle 中定义，但是使用了两个 Ant 任务：echo 和 zip。

当然，你应该多研究一下对于标准 Ant 任务的 Gradle 替代品。要想创建一个类似前面例子中的 archive，你可以通过定义一个 Gradle 任务来实现：

```
task gradleArchive(type:Zip) << {
    from 'zipme/'
    archiveName 'grarchive.zip'
}
```

基于 Gradle 的 archive 任务更简洁，也更易于理解。因为其不需要使用 AntBuilder，也比使用 Ant 任务更快。

9.4.2 导入整个 Ant 脚本

如果你创建了一个 Ant 脚本来构建你的应用，那么你就可以使用 ant.importBuild 来导入整个构建配置。所有目标 Ant 会自动转化为 Gradle 任务，这样你就可以通过原有名称来访问它们了。

例如，下面的 Ant 构建文件：

```
<project>
    <target name="hello">
        <echo>Hello, Ant</echo>
    </target>
</project>
```

你可以将其导入到你的 Gradle 构建中：

```
ant.importBuild 'build.xml'
```

这样 hello 任务就暴露给你的 Gradle 构建了，所以你可以像常规的 Gradle 任务那样来执行它，它会打印出 Hello, Ant：

```
$ gradlew hello
:hello
[ant:echo] Hello, Ant
```

由于 Ant 任务被转化为了 Gradle 任务，因此你可以使用 doFirst 和 doLast 代码块来扩

展（或简写）它。例如，你可以打印另一行到控制台：

```
hello << {
        println 'Hello, Ant. It\'s me, Gradle'
}
```

如果现在只想执行 hello task，则这样操作：

```
$ gradlew hello
:hello
[ant:echo] Hello, Ant
Hello, Ant. It's me, Gradle
```

你也可以依赖于从 Ant 导入的任务，就像你通常做的那样。举个例子，如果你想创建一个依赖于 hello 任务的任务，那么你可以这样做：

```
 task hi(dependsOn: hello) << {
        println 'Hi!'
}
```

使用 dependsOn 保证了当执行 hi 任务时，hello 任务被触发：

```
$ gradlew intro
:hello
[ant:echo] Hello, Ant
Hello, Ant. It's me, Gradle
:hi
Hi!
```

如果需要，你甚至可以创建一个依赖于 Gradle 任务的 Ant 任务。为了实现它，你需要在 build.xml 文件中添加 depends 属性到任务，如下所示：

```
<target name="hi" depends="intro">
        <echo>Hi</echo>
</target>
```

如果你有一个大的 Ant 构建文件，并且想确定没有名为 overlap 的任务，那么你可以使用下面的代码片段来重命名所有导入的 Ant 任务：

```
ant.importBuild('build.xml') { antTargetName ->
        'ant-' + antTargetName
}
```

如果想对所有 Ant 任务进行重命名，请千万记得，如果有 Ant 任务依赖于 Gradle 任务，则该 Gradle 任务也需要被预处理。否则，Gradle 将无法找到它，并抛出 UnknownTaskException。

9.4.3 属性

Gradle和Ant不仅可以分享任务，还可以在Gralde中定义属性，以便在Ant构建文件中使用。注意这个Ant目标，其打印出的属性叫作version：

```
<target name="appVersion">
       <echo>${version}</echo>
</target>
```

你可以通过在属性名前添加ant.来定义Gradle构建配置的版本属性，就和任务一样。下面是定义一个Ant属性的最简单的方式：

```
ant.version = '1.0'
```

Groovy在这里隐藏了很多实现。如果写全该属性定义，则像下面这样：

```
ant.properties['version'] = '1.0'
```

执行version task，将与你期待的一样，即打印1.0到控制台：

```
$ gradlew appVersion
:appVersion
[ant:echo] 1.0
```

Gradle中的深入Ant整合，让基于Ant的构建迁移到Gradle变得更加容易，你可以轻松地完成它。

9.5 高级应用部署

在第4章中，我们研究了使用构建类型和product flavor来创建多版本应用的几种方式。然而，在某些情况下，使用更具体的技术会更容易些，例如APK splits。

分割APK

构建variant可以被看作是单独的实体，每一个variant都有其独立的代码、资源和manifest文件。另一方面，APK splits只影响应用的打包。编译、缩减、混淆等仍可共享。这种机制允许你根据密度或**application binary interface（ABI）**来分割APKs。

你可以通过在android配置代码块中定义一个splits代码块来配置分隔。为了配置密度分隔，你需要在splits代码块内部创建一个density代码块。如果你想设置ABI分隔，则可以使用abi代码块。

如果你启用了密度分隔，则 Gradle 会为每个密度创建一个独立的 APK。如果你不想要某些密度，则可以手动排除它们，来加速构建过程。下面这个例子展示了如何启动密度分隔并排除设备的低密度：

```
android {
        splits {
            density {
                enable true
                exclude 'ldpi', 'mdpi'
                compatibleScreens 'normal', 'large', 'xlarge'
            }
        }
}
```

如果只支持几种密度，则可以使用 include 来创建一个密度白名单。为了使用 include，你首先需要使用 reset() 属性，其会将包含密度的列表设置为空字符串。

在前面的代码片段中，compatibleScreens 属性是可选的，并在 manifest 文件中注入了一个 matching 节点。例子中的配置，使应用可支持正常屏幕和超大屏幕，排除了小屏幕设备。

基于 ABI 的分隔 APK 以相同的方式工作，除了 compatibleScreens 外，其和密度分隔拥有完全相同的属性，ABI 分隔无关于屏幕尺寸，所以没有 compatibleScreens 属性。

在配置密度分隔后，执行一次构建，其结果是，Gradle 会创建一个通用的 APK 和几个特定密度的 APKs。这意味着你将获得一系列的 APKs：

```
app-hdpi-release.apk
app-universal-release.apk
app-xhdpi-release.apk
app-xxhdpi-release.apk
app-xxxhdpi-release.apk
```

使用 APK splits 有一点需要注意。如果想推送多个 APKs 至 Google Play，则你需要确保每个 APK 都有一个不同的版本号。这意味着每个 split 都应该有一个特有的版本号。幸运的是，现在你可以通过设置 Gradle 中的 applicationVariants 属性来实现。

下面的代码片段摘自 Gradle Android 插件文档，其展示了如何为每个 APK 生成不同的版本号：

```
ext.versionCodes = ['armeabi-v7a':1, mips:2, x86:3]

import com.android.build.OutputFile
```

```
android.applicationVariants.all { variant ->
        // assign different version code for each output
        variant.outputs.each { output ->
            output.versionCodeOverride =  project.ext.versionCodes.get
              (output.getFilter(OutputFile.ABI)) * 1000000 +
              android.defaultConfig.versionCode
        }
}
```

这个小片段检查了在一个构建variant上到底使用了哪种ABI，然后给版本号应用一个乘子，以确保每个variant都有一个特有的版本号。

9.6 总结

读完本章后，你应该知道了如何减小构建输出的大小，如何通过配置Gradle和JVM来加速构建，并且大的迁移项目不再让你害怕。同时，你也学习了一些让开发和部署变得更加简单的技巧。

这是本书的结尾了。现在你已经知道了Gradle的可能性，你可以调整和自定义你的Android项目的构建过程，除了执行任务外，你不必再做任何手动的工作，你可以配置构建variant，管理依赖和配置多模块项目。在理解了Groovy语法后，Gradle DSL对你来说就很容易理解了，并且你可以很舒服地在Android插件中挂钩子。你甚至可以创建任务或插件，并分享它们，以帮助其他人自动化构建。现在你需要做的就是应用你的新技能！

Broadview
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

Gradle for Android 中文版

Gradle是一个开源的自动化构建系统，其引入了一种基于Groovy领域的特定语言（DSL）来配置项目。Gradle可以让Android开发者更容易地管理依赖，并建立整个构建过程。

本书首先介绍了Gradle的基础知识，然后讲解了其如何与Android Studio协同工作。此外，本书还讲解了如何添加本地依赖和远程依赖到你的项目。你将同构建变种版本打交道，例如调试和生产版本、付费和免费版本，甚至是这些版本的组合。本书也将通过不同的依赖，帮助你建立单元测试和集成测试，并展示了Gradle和Android Studio如何让运行测试更加容易。最后，你将在你的应用构建过程的高级自定义部分，看到一些提示和技巧。本书的结尾，你将能够自定义整个构建过程，并为你的Gradle构建创建属于自己的任务和插件。

本书的目标读者

如果你是有经验的Android开发者，并且想增进你的Gradle Android构建系统技能，那么本书适合你。作为先决条件，你需要知道一些Android应用开发的概念。

通过本书你能学到

- 使用Android Studio和Gradle来构建新的Android应用和依赖
- 将项目从Eclipse迁移到Android Studio和Gradle
- 为你的项目管理本地和远程依赖
- 创建多个构建变种版本
- 在单个项目中包含多个模块
- 将测试整合到构建过程
- 为Android项目创建自定义的任务和插件

2016

上架建议：程序设计

[PACKT] PUBLISHING open source* community experience distilled

ISBN 978-7-121-30015-8

9 787121 300158 >

定价：49.00元

责任编辑：安　娜
封面设计：李　玲