



# 实战 Gradle

[美] Benjamin Muschko 著  
Hans Dockter 作序  
李建 杨柳 朱本威 译

## Gradle IN ACTION



中国工信出版集团



电子工业出版社  
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY  
http://www.phei.com.cn

# 实战 Gradle

[美] Benjamin Muschko 著  
Hans Dockter 作序  
李建 杨柳 朱本威 译

## Gradle IN ACTION

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



## 内 容 简 介

Gradle 是 Java 软件开发中的自动化构建工具, 类似于传统工具如 Ant 和 Maven。Gradle 吸收或沿用了 Maven 中比较成功的一些实践, 但相对于 Maven 或 Ant 又有极强的扩展。

《实战 Gradle》(Gradle in Action) 全方位地讲解了 Gradle 工具相关的知识, 包括 Gradle 的基本知识、自动化构建的基本概念和最佳实践, 以实际案例的方式解释如何使用 Gradle 进行软件项目构建。

除了基础知识外, 由于软件开发绝对不仅仅是实现业务逻辑代码, 书中还介绍了一些解决软件开发中常见问题的实践, 如多语言、多项目构建, Gradle 在持续集成和持续交付中的应用, Gradle 构建 JVM 其他语言, 以及 Gradle 集成 JavaScript 构建等。

云计算和 DevOps 的兴起, 给软件行业带来了翻天覆地的变化, 书中对于云计算平台、开源社区中的一些工具与 Gradle 的结合使用也做了相关的介绍。

因为 Groovy 用于编写 Gradle 构建配置的 DSL, 所以为了帮助读者更好地理解 Gradle, 本书还讲解了 Groovy 的基本知识, 虽然不是全方位地讲解 Groovy, 但是理解 Gradle 足矣。

Original English language edition published by Manning Publications, USA. Copyright © 2014 by Manning Publications. Simplified Chinese-language edition copyright © 2015 by Publishing House of Electronics Industry. All rights reserved.

本书简体中文版专有版权由 Manning Publications 授予电子工业出版社。未经许可, 不得以任何方式复制或抄袭本书的任何部分。专有版权受法律保护。

版权贸易合同登记号 图字: 01-2014-8531

### 图书在版编目(CIP)数据

实战 Gradle / (美) 马斯可 (Muschko, B.) 著; 李建, 杨柳, 朱本威译. —北京: 电子工业出版社, 2015.9

书名原文: Gradle in Action

ISBN 978-7-121-26925-7

I. ①实… II. ①马… ②李… ③杨… ④朱… III. ①JAVA 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字(2015)第 187314 号

策划编辑: 张春雨

责任编辑: 葛 娜

印 刷: 三河市双峰印刷装订有限公司

装 订: 三河市双峰印刷装订有限公司

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: 30.75

字数: 637 千字

版 次: 2015 年 9 月第 1 版

印 次: 2015 年 9 月第 1 次印刷

定 价: 89.00 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。

质量投诉请发邮件至 [zltts@phei.com.cn](mailto:zltts@phei.com.cn), 盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

服务热线: (010) 88258888。

# 译者序

---

提到自动化构建，你能想到的是什么？是经典的 `make` 脚本语言？是曾经统治了 Java 构建世界的 `Ant`？或者是颠覆了 `Ant` 统治地位的 `Maven`？抑或是即将也正在颠覆 `Maven` 的 `Gradle`？不管你想到的是哪一个，如果你还在使用 Java 语言开发软件，那么 `Gradle` 将是你工具箱中必不可少的一个利器。而 *Gradle in Action* 将可能是你不可或缺的教材，或者参考书。

第一次接触到 `Gradle` 是加入 ThoughtWorks 的第一个项目，从此以后，我几乎认为 `Gradle` 应该是一个 Java 软件项目的标配，是每一个 Java 程序员都应该了如指掌的东西。本书结合简单实用的案例，由浅入深地讲解 `Gradle`，正是我在学习 `Gradle` 时所希望拥有的一本教材，所以我便欣然接受并参与此书的翻译工作。希望本书能帮助所有将软件质量和开发效率视为软件项目中不可或缺的一环的程序员，把软件开发带到另一个高度。

虽然本书是一本 `Gradle` 实用指南，但却不仅仅局限于此。软件构建在现代软件开发过程中并不是独立工作的，开发人员通常面临的问题还包括持续集成、持续交付，以及云平台下的基础设施搭建、多语言项目构建等。本书还对如何结合 `Gradle` 解决此类问题，以及一些常用的工具有一定的说明，如用于持续集成和交付的 `Jenkins`、虚拟化技术工具 `Vagrant` 等。`Groovy` 语言用于实现 `Gradle DSL`，因此理解基础的 `Groovy` 语言对理解 `Gradle` 也有极大的帮助，因此本书对 `Gradle` 语言的基本功能也做了讲解。

与另外两位同事合作翻译此书的过程，并非一帆风顺，由于物理位置的不同，使得沟通变得困难，我们不得不采用通读对方译文的形式使各自的行文风格尽量保

持一致。但这同时也是一个充实有趣的过程，想要把一个单词或一句话翻译得精准到位从而反复推敲，有些技术语言如果直译又会生硬难懂，我们得寻求适合中文的准确表达方式。这些过程无疑是翻译中最大的趣味所在。在这里我想感谢朱本威和杨柳的智慧结晶，感谢出版社同事的辛勤劳作，感谢滕云的严格审校，更感谢 Benjamin 为 Gradle 社区带来如此优秀的读物。

# 序

---

当你在创建一套类似于 Gradle 这样的新技术时，开发过程中最关键的部分往往不是写代码。一旦项目的初期版本被数以千计的开发者的使用，一个社区开始围绕该项目组建，挑战就变成了与更多关注者进行交流。他们会使用你的项目，会对项目的优缺点进行评价，而社区的大小也会成十倍或千倍的增长。Gradle 已经拥有了一大批关注者，而且在过去的两年里，我们已经看到人数有巨大的增长，而我们也准备迎接更多的用户。

因此，一点也不夸张地说，拥有一本好书是多么的重要。掌握多种技术和能力的开发者都希望有一本通俗易懂的书，它能够讲解语法和工具背后的哲理。因为只有这样，开发者才有信心通过这本唯一的 Gradle 权威参考书让社区成长。《实战 Gradle》就是这样一本书。不仅如此，这本书还让 Gradle 的新用户有机会去了解，Gradle 是如何适应到持续交付这样更广阔的领域中的。

本书的作者 Benjamin，是你希望能够从开源社区中闪现出来的一类专家。他是 Gradle 的长期贡献者，也是多个 Gradle 流行插件的作者。他拥有传播者和开发者两种角色。Benjamin 拥有一种很少见的力量，能够深入到一个特定开发挑战的核心细节中，我们并向用户解释这些细节。他在最近加入了 Gradleware，并成为 Gradle 开发团队的一员，我们感到非常高兴。

我真心希望你能够喜欢这本书，也享受在工作中使用 Gradle。祝你的软件交付过程变得有趣而且有效。

Hans Dockter,  
Gradle and Gradleware 创始人

# 前言

---

当我以软件开发开始我的职业生涯时，我完全没有意识到对项目自动化的需求。我选择 IDE 作为开发工具，它可以让我运行所有需要的任务，并完成自动化软件开发环节。2003 年，Rainer Sawitzki<sup>1</sup>，我所在项目的一个外部咨询师，他向我介绍了 Ant。我觉得它简直太神奇了，Ant 能够通过已有的功能帮助我描述大部分自动化逻辑，并按照预定义的顺序执行。尽管定义语言是 XML（曾经有那么一段时间 XML 还是很流行的），通过给不同的目标平台构建产品，给 Web 容器编写部署逻辑和配置持续集成服务器，我的野心开始变得越来越大。

从那时起，自动化的需求发生了巨大的变化。项目变得越来越大，也越来越复杂。部署和交付模式已经远比以前麻烦。这些年为了满足自动化需求，我探索了许多其他类型的构建工具，但我总是会发现有不合逻辑的地方。许多开发者接受和维持这种现状，因此给他们留下了一些痛苦的经历。很少有话题去真正地讨论这个问题<sup>2</sup>，而不仅仅是赞成或者反对构建工具，以及为什么如此地不喜欢构建工具。本书的目的并不是要说服你从当前的构建工具切换到 Gradle。如果你很满意你的配置方式（不管你用的什么），无论怎样，要坚持住。而我会更多地谈论 Gradle 带来的巨大创新，以及与现有解决方案的对比。我诚心地请求你做出自己的判断。

---

1 再次感谢你在工作中交给我面向对象的思想。你是我的一位伟大导师。

2 这个话题就像对比 Windows 和 Linux 或者 Web 应用框架。

在我的头脑里，我带着一种特殊的目的开始写这本书：教给你 Gradle 的核心概念，但不止于此。在一个拥抱持续集成和持续交付实践的软件开发世界，你必须考虑一个问题，那就是将构建系统集成到工具的生态系统中。但愿，我在这本书里找到了平衡点。如果你有任何问题、评论或者想法，我都非常愿意去倾听。你的反馈也许会促使我去写第二版书或者附加新的内容。请随时向我发送电子邮件或者在 Manning 的论坛上和我联系。

就和所有的书一样，纸张的数量是有限的。为了控制本书的讨论范围，我不得不去掉一部分我原本打算写进去的内容（这是我的第一本书，带着初为作者的天真，我以为我可以将所有的内容都写进去）。关于本书的源代码，你可以在 <https://github.com/bmuschko/gradle-in-action-source> 中找到，里面扩展了书中的一些内容，也列出了其他代码样例和资料的参考文献。我希望你可以和我享受写这本书一样，享受阅读这本书。

# 致谢

---

当考虑写一本书时，你完全不知道有多少工作需要去做。可以肯定地说，它绝对会长时间掌控着你的生活。一段时间之后，写作的部分会变得简单一些。困难的是要坚持每天写。在没有支持、鼓励和别人帮助的情况下，是完全不可能的。

2010 年，我开始给前雇主评估 Gradle 能否作为一个 Maven 项目的替代品。如果没有 Jonathan Bodner 发起的这次尝试，我很有可能无法完成这次评估。Jonathan Bodner 是我的一个老朋友，我非常钦佩他在技术方面的洞察力，是他激起了我对 Gradle 的热情，让我深深地融入到这个社区，并开始实现我自己的插件。

在开始写书之前，我已经担任多年 Manning 出版社所出版书籍的技术审查员。这一切都开始于我与 Dan Allen 在 No Fluff Just Stuff 会议上的一次偶遇，他是 *Seam in Action* (Manning, 2008) 的作者。在那次会议上，仅仅是跟他聊的这一会儿，就很快激起了我对他这份事业的热情，于是我开始帮助他审阅他的书籍。这次给别人审阅书的机会，让我第一次了解到写一本书的意义。而我其实一直都想要写一本书，但从来就没有找到合适的时间或者主题。Gradle 给了我机会。感谢 Dan，你的热情鼓励着我继续传递这只火炬，也鼓励着我点燃自己的火炬。

在开始写书之前，你首先要做的事情之一就是完成大纲和目录。David James 是第一个阅读本书初稿的人，他是华盛顿地区 Groovy 用户组的组织者。感谢你对本书组织结构提供的独特观点，也感谢你对于细节的一丝不苟，以及你的鼓励让这本书



成为现实。

商业化图书的出版离不开背后做出贡献的人，这包括 Manning 出版社所有参与到本书出版中的每个人。Michael Stephens 是第一个和我交谈的人，对于本书的思想，他很买账，也最终信任我能够做好这份工作。我的感激也要送给 Cynthia Kane，她帮助我找到我自己的写作风格。我也想要感谢 Jennifer Stout，我的发展编辑，他总是设法让我做到最好，让我以一种不同的方式去考虑整个文本段落，并忍受我的不耐烦，这对我的帮助非常大。还要感谢 Manning 的产品和市场团队一直的指导，它促使这本书成为现在这个样子。我知道你们做了大量的工作。

我还想要感谢 Gradleware 团队的成员以及 Gradle 社区创建了 Gradle，并不断推进构建自动化的发展。你们持续不断的努力和对高质量产品的信仰，提高了全球正对构建恼火的大师的生活质量。特别要感谢 René Gröschke 和 Luke Daley 在技术方面的洞察力，以及他们对本书三分之一内容的审阅。我也非常感激 Gradle 的创始人，Hans Dockter，他帮助本书作序，以及对这本书给予早期认可，并通过 Gradleware 持续提供帮助。

感谢以下所有的手稿审阅人，他们向我提供了无价的反馈和对书中内容独特的观点：Andy Keffalas, BJ Peter DeLaCruz, Chris Grijalva, Chris Nauroth, Dominik Helleberg, Eric Wendelin, Iain Starks, John Moses, Jon Bodner, Marcin Nowina-Krowicki, Mayur S. Patil, Mehrdad Karjoo, Mohd Suhaizal Md Kamari, Nacho Ormeño, Nick Watts, Pawel Dolega, Rob Bugh, Robin Percy, Samuel Brown, Scott Bennett-McLeish, Steve Dickson, Tarin Gamberini, Wellington R. Pinheiro 和 Zekai Otles。也要感谢 Barry Kern 在手稿正式进入印刷之前，仔细地进行技术方面的校对。

特别要感谢 Spencer Allain, Jonathan Keam 和 Robert Wenner 通读了本书的每个章节，并在写书不同阶段向我提供了逐字逐句的修改建议和评论。要感谢 Michael McGarr 和 Samuel Brown 反复审阅书中关于持续交付和 DevOps 的内容，以及要感谢来自于 JFrog 的 Baruch Sadogursky 帮助我从技术的角度审阅了第 14 章，并在本书出版之前，就帮助推销这本书。我也想要感谢无情的 Author Online 论坛的用户不断地将书中的内容推向更高的层次。

写一本书需要做出牺牲，而且加剧了我人际关系的紧张程度。我想要感谢我的家人和朋友，他们在我不断朝着这个宏伟目标前进的时候，给予我支持、鼓励和理解。还有就是，以后出去玩的时候，我不会再去思考书中的内容了。

我深深地感激我的妻子 Sarah 对我无止境的支持以及她乐观的心态。她让我相信自己，让我在写作时学会休息，也忍受了大部分的日子里，我在晚上 9:00 点之前就睡了。没有你，写作带来的疲惫会远比现在多。

# 关于本书

---

## 路线图

本书分为三部分。第 1 部分介绍了 Gradle 的概念和思想，解释了它和其他构建工具的不同以及如何编写脚本来自动化简单的任务。第 2 部分探索了工具的构建模块以及更深层次的核心技术。你应该能够使用这些知识去实现复杂的和可扩展的企业级构建。第 3 部分描述了如何在持续交付中使用 Gradle，主要集中在多语言构建、代码质量、工件组装和部署等话题上。

第 1 部分，Gradle 介绍：

- 1 项目自动化介绍——本章简单介绍了为什么要实现项目自动化，以及构建工具是如何帮助做到这一点的。
- 2 下一代构建工具：Gradle——Gradle 和其他的基于 JVM 语言的构建工具有什么不同？本章覆盖了 Gradle 丰富的特性集，以及它是在持续交付的部署管道中帮助自动化软件交付过程的。作为初次体验，你会写一个简单的构建脚本，并在命令行中运行它。
- 3 通过范例学习构建 Gradle 项目——本章以一个基于 Java 的 Web 应用为例说明 Gradle 的一些核心特性。我们会针对标准的和非约定的用例，探索 Java 插件的使用，验证这种高效率的工具是如何彻底改变开发速度的。

第2部分,掌握基本原理,集中将第1部分介绍的 Gradle 重要概念应用到案例中:

- 4 构建脚本概要——哪些是一个 Gradle 项目的主要构建块?本章讨论了重要领域对象的使用,即项目和任务。我们会去了解这些对象如何映射到 Gradle API 中相关的类、Gradle 的构建生命周期、增量构建特性,以及注册生命周期钩子的机制。
- 5 依赖管理——没有哪个企业级项目能够不重用外部库提供的功能。本章将探索 Gradle 对声明式依赖管理的支持、版本冲突的解决策略以及 Gradle 缓存的内部工作方式。
- 6 多项目构建——你的项目是否由多个模块化组件组成?本章覆盖了在一个多项目设置中组织构建逻辑的方式、如何声明项目依赖,以及如何部分构建以提高执行效率。
- 7 Gradle 测试——测试功能代码是软件开发生命周期中的一个重要环节。在这一章的最后,你会用 JUnit、TestNG 和 Spock 去写测试,并将它们作为构建生命周期的一部分去执行。你也会学习如何配置测试的执行、注册监听器去响应测试生命周期事件,并在 source sets 的帮助下组织不同类型的测试。
- 8 扩展 Gradle——Gradle 提供了可扩展的领域对象模型。如果你想要添加一个全新的功能到项目中或者扩展已有的领域模型,那么这一章很适合你。你会学习如何编写属于自己的插件,来将样例应用程序部署到云端。
- 9 集成与迁移——在本章中,我们会看到 Gradle 如何与 Ant 和 Maven 集成。我们也会探索迁移策略,以防你决定长期使用 Gradle。

第3部分,从构建到部署,剖析如何在构建管道的帮助下,使用 Gradle 将样例应用程序从开发者机器部署到产品环境:

- 10 IDE 支持和工具——IDE 是提高开发人员生产力的关键因素。本章将介绍 Gradle 生成如 Eclipse、IntelliJ 和 NetBeans 这些流行的 IDE 工程文件的能力。我们也将讨论在这些 IDE 中如何使用和管理基于 Gradle 的项目。
- 11 构建多语言项目——在本章中,我们将通过案例学习的方式讨论 Gradle 是如何面对组织和构建多语言项目的挑战的。集成的语言包括 JavaScript、Groovy 和 Scala。
- 12 代码质量管理和监测——本章我们会集中在这样一些工具上,它们能够检测代码质量,并可视化结果来帮助你准确地找到代码中的问题。读完这一章之后,你就会知道如何将代码质量工具集成到构建中。
- 13 持续集成——持续集成(CI)是一种软件开发实践,源代码会被频繁地集成在一起,最好是一天集成多次。本章将讨论在 Jenkins(一种开源的持续集成服务器)上运行 Gradle 的安装和配置过程。
- 14 打包和发布——一个构建要么是二进制工件的消费者,要么是生产者。本

章探索了工件的组装过程，以及要发布工件到二进制仓库所需要的配置，包括它们的元数据。

- 15 基础环境准备和部署**——一个可配置的目标环境是任何软件开发的先决条件。在这一章中，我们会讨论在以自动化的方式，建设和配置环境以及服务中“基础设施即代码”的重要性。之后，你会使用 Gradle 实现一个经典的部署过程。

两个附录涵盖了额外的话题：

- A 驾驭命令行**——该附录解释了如何从命令行操作 Gradle。我们会探索所有 Gradle 构建中可用的任务，包括命令行选项和它们的用例。
- B Gradle 用户所需要了解的 Groovy**——如果你是第一次接触 Groovy，该附录会简单地介绍 Groovy 中最重要和使用最广泛的语言特性。

## 谁应该阅读本书

本书主要提供给那些想要实现一种易读和可扩展的重复构建的开发者和自动构建工程师。我假设你已经对面向对象编程语言有了基本了解。如果你拥有 Java 语言的开发知识，你应该能够理解书中的大部分内容。

在本书中，你会大量使用 Groovy。然而，我不会假设你已经拥有使用该语言的经验。如果想要了解 Groovy，请看附录 B——Gradle 用户所需要了解的 Groovy。如果你想要深入学习关于这种语言的高级部分，该附录也提供了一些额外的参考资料。

贯穿本书的全部章节，我们会接触一些与自动化构建打交道时不可绕开的话题。如果你拥有以下这些知识，会对你阅读本书有所帮助：Ant、Ivy 和 Maven 工具；持续集成和持续交付的实践；依赖管理的概念。但是如果这些不是你的技术背景，你也不用担心。每一章节都会非常详细地解释“为什么”。

## 代码约定和下载

代码清单和文本中的源代码都会以等宽字体显示，就像这样 `fixed-width font like this`，以便将它与普通文本区分开。许多代码清单中都有代码注解，并且强调了重要的概念。

所有的源代码都可以在出版社的网站 [www.manning.com/GradleInAction](http://www.manning.com/GradleInAction) 和 GitHub 的仓库 <https://github.com/bmuschko/gradle-in-action-source> 中找到。在该仓库里，你也会发现一些额外的参考资料，这些资料要么继续深入了书中的例子，要么说明了书中没有涉及的关于使用 Gradle 的内容。

# 关于封面

---

本书封面上的图片为“来自伊斯特里亚的女人”。伊斯特里亚是亚得里亚海上一个很大的半岛，面向克罗地亚。该图片来自克罗地亚斯普利特民族博物馆 2008 年出版的 Balthasar Hacquet 的《图说西南及东汪达尔人、伊利里亚人和斯拉夫人》(*Images and Descriptions of Southwestern and Eastern Wenda, Illyrians, and Slavs*) 的最新重印版本。Hacquet (1739—1815) 是一名奥地利内科医生及科学家，他花费数年时间去研究各地的植物、地质和人种，这些地方包括奥匈帝国的多个地区，以及伊利里亚部落过去居住的（罗马帝国的）威尼托地区、尤里安阿尔卑斯山脉及西巴尔干等地区。Hacquet 发表的很多论文和书籍中都有手绘插图。

Hacquet 出版物中丰富多样的插图生动地描绘了 200 年前西阿尔卑斯和巴尔干西北地区的独特性和个体性。那时候相距几英里的两个村庄村民的衣着都迥然不同，当有社交活动或交易时，不同地区的人们很容易通过着装来辨别。此后着装的要求发生了改变，不同地区的多样性也逐渐消亡。现在很难说出不同大陆的居民有多大区别，比如，现在很难区分斯洛文尼亚的阿尔卑斯山地区或巴尔干沿海那些美丽小镇或村庄里的居民与欧洲其他地区或美国的居民。

Manning 出版社利用两个世纪之前的服装来设计书籍封面，以此来赞颂计算机产业所具有的创造性、主动性和趣味性。正如本书封面上的图片一样，这些图片也把我们带回到过去的生活中去。

## 关于作者

Benjamin Muschko 是一名拥有超过 10 年开发和交付商业软件工作经验的软件开发工程师。他是 Gradleware 工程团队的成员，也是多个 Gradle 流行插件的作者。

# 目录

---

第 1 部分 Gradle 介绍 .....	1
1 项目自动化介绍 .....	3
1.1 没有项目自动化的生活 .....	4
1.2 项目自动化的好处 .....	5
1.2.1 防止手动介入 .....	5
1.2.2 创建可重复的构建 .....	5
1.2.3 让构建便携 .....	5
1.3 项目自动化的类型 .....	6
1.3.1 按需构建 .....	6
1.3.2 触发构建 .....	7
1.3.3 预定构建 .....	7
1.4 构建工具 .....	8
1.4.1 什么是构建工具 .....	9
1.4.2 构建工具的剖析 .....	10
1.5 Java 构建工具 .....	12



1.5.1 Apache Ant .....	12
1.5.2 Apache Maven .....	16
1.5.3 对下一代构建工具的需求 .....	19
1.6 总结 .....	20
<b>2 下一代构建工具：Gradle .....</b>	<b>23</b>
2.1 为什么要用 Gradle，为什么是现在 .....	24
2.1.1 Java 构建工具的演变 .....	25
2.1.2 为什么应该选择 Gradle .....	27
2.2 Gradle 引人注目的特性集 .....	29
2.2.1 可表达性的构建语言和底层的 API .....	29
2.2.2 Gradle 就是 Groovy .....	31
2.2.3 灵活的约定 .....	32
2.2.4 鲁棒和强大的依赖管理 .....	33
2.2.5 可扩展的构建 .....	34
2.2.6 轻松的可扩展性 .....	34
2.2.7 和其他构建工具集成 .....	35
2.2.8 社区和公司的推动 .....	36
2.2.9 锦上添花：额外的特性 .....	36
2.3 更大的场景：持续交付 .....	36
2.3.1 从构建到部署自动化项目 .....	37
2.4 安装 Gradle .....	38
2.5 开始使用 Gradle .....	40
2.6 使用 Gradle 的命令行 .....	42
2.6.1 列出项目中所有可用的 task .....	43
2.6.2 任务执行 .....	44
2.6.3 命令行选项 .....	46
2.6.4 Gradle 守护进程 .....	47
2.7 总结 .....	48
<b>3 通过范例学习构建 Gradle 项目 .....</b>	<b>49</b>
3.1 介绍学习案例 .....	50
3.1.1 To Do 应用程序 .....	50

3.1.2 任务管理用例 .....	50
3.1.3 检查组件交互 .....	51
3.1.4 构建应用功能 .....	52
3.2 构建 Java 项目 .....	55
3.2.1 使用 Java 插件 .....	55
3.2.2 定制你的项目 .....	59
3.2.3 配置和使用外部依赖 .....	60
3.3 用 Gradle 做 Web 开发 .....	62
3.3.1 添加 Web 组件 .....	62
3.3.2 使用 War 和 Jetty 插件 .....	64
3.4 Gradle 包装器 .....	69
3.4.1 配置包装器 .....	70
3.4.2 使用包装器 .....	71
3.4.3 定制包装器 .....	73
3.5 总结 .....	73

## 第 2 部分 掌握基本原理 ..... 75

## 4 构建脚本概要 ..... 77

4.1 构建块 .....	78
4.1.1 项目 .....	78
4.1.2 任务 .....	80
4.1.3 属性 .....	80
4.2 使用 task .....	82
4.2.1 项目版本管理 .....	82
4.2.2 声明 task 动作 .....	83
4.2.3 访问 DefaultTask 属性 .....	84
4.2.4 定义 task 依赖 .....	85
4.2.5 终结器 task .....	86
4.2.6 添加任意代码 .....	87
4.2.7 理解 task 配置 .....	87
4.2.8 声明 task 的 inputs 和 outputs .....	90
4.2.9 编写和使用自定义 task .....	92

4.2.10 Gradle 的内置 task 类型 .....	95
4.2.11 task 规则 .....	97
4.2.12 在 buildSrc 目录下构建代码 .....	100
4.3 挂接到构建生命周期 .....	101
4.3.1 挂接到 task 执行图 .....	103
4.3.2 实现 task 执行图监听器 .....	103
4.3.3 初始化构建环境 .....	105
4.4 总结 .....	106
<b>5 依赖管理 .....</b>	<b>107</b>
5.1 依赖管理概述 .....	108
5.1.1 不完善的依赖管理技术 .....	108
5.1.2 自动化依赖管理的重要性 .....	108
5.1.3 使用自动化依赖管理 .....	110
5.1.4 自动化依赖管理的挑战 .....	110
5.2 通过例子学习依赖管理 .....	112
5.3 依赖配置 .....	113
5.3.1 理解配置 API 表示 .....	113
5.3.2 自定义配置 .....	114
5.3.3 访问配置 .....	115
5.4 声明依赖 .....	115
5.4.1 理解依赖 API 表示 .....	116
5.4.2 外部模块依赖 .....	117
5.4.3 文件依赖 .....	121
5.5 使用和配置仓库 .....	122
5.5.1 理解仓库 API 表示 .....	123
5.5.2 Maven 仓库 .....	124
5.5.3 Ivy 仓库 .....	126
5.5.4 扁平的目录仓库 .....	126
5.6 理解本地依赖缓存 .....	127
5.6.1 分析缓存结构 .....	127
5.6.2 显著的缓存特性 .....	129

5.7 解决依赖问题 .....	130
5.7.1 应对版本冲突 .....	130
5.7.2 强制指定一个版本 .....	131
5.7.3 使用依赖观察报告 .....	131
5.7.4 刷新缓存 .....	132
5.8 总结 .....	133
<b>6 多项目构建 .....</b>	<b>135</b>
6.1 模块化项目 .....	136
6.1.1 耦合与内聚 .....	136
6.1.2 模块划分 .....	137
6.1.3 模块化重构 .....	138
6.2 组装多项目构建 .....	139
6.2.1 settings 文件介绍 .....	140
6.2.2 理解 settings API 表示 .....	141
6.2.3 settings 执行 .....	142
6.2.4 获取 settings 文件 .....	142
6.2.5 分层布局与扁平布局 .....	143
6.3 配置子项目 .....	144
6.3.1 理解 Project API 表示 .....	145
6.3.2 定义特定的行为 .....	146
6.3.3 声明项目依赖 .....	147
6.3.4 多项目部分构建 .....	149
6.3.5 声明跨项目的 task 依赖 .....	151
6.3.6 定义公共行为 .....	153
6.4 独立的项目文件 .....	154
6.4.1 为每个项目创建构建文件 .....	155
6.4.2 定义根项目的构建代码 .....	155
6.4.3 定义子项目的构建代码 .....	155
6.5 自定义项目 .....	156
6.6 总结 .....	157
<b>7 Gradle 测试 .....</b>	<b>159</b>
7.1 自动化测试 .....	160

7.1.1 自动化测试类型 .....	160
7.1.2 自动化测试金字塔 .....	160
7.2 测试 Java 应用程序 .....	161
7.2.1 项目布局 .....	162
7.2.2 测试配置 .....	162
7.2.3 测试 task .....	163
7.2.4 自动化测试检测 .....	164
7.3 单元测试 .....	164
7.3.1 使用 JUnit .....	164
7.3.2 使用其他的单元测试框架 .....	168
7.3.3 结合使用多个单元测试框架 .....	170
7.4 配置测试执行 .....	172
7.4.1 命令行选项 .....	173
7.4.2 理解测试 API 表示 .....	174
7.4.3 控制运行时行为 .....	175
7.4.4 控制测试日志 .....	176
7.4.5 并行执行测试 .....	178
7.4.6 响应测试生命周期事件 .....	179
7.4.7 实现测试监听器 .....	180
7.5 集成测试 .....	181
7.5.1 引入用例研究 .....	181
7.5.2 编写测试类 .....	181
7.5.3 在构建中支持集成测试 .....	182
7.5.4 为集成测试建立约定 .....	184
7.5.5 引导测试环境 .....	186
7.6 功能测试 .....	187
7.6.1 引入用例研究 .....	187
7.6.2 在构建中支持功能测试 .....	188
7.7 总结 .....	192
<b>8 扩展 Gradle .....</b>	<b>195</b>
8.1 通过案例学习介绍插件 .....	196
8.1.1 在云中使用 Grade 管理应用 .....	196
8.1.2 设置云环境 .....	197

8.2 从零起步构建插件 .....	200
8.3 写一个脚本插件 .....	201
8.3.1 添加 CloudBees 的 API 类库 .....	201
8.3.2 在 task 中使用 CloudBees 的 API .....	202
8.4 编写定制的 task 类 .....	206
8.4.1 定制 task 的实现选项 .....	206
8.4.2 在 buildSrc 下定义定制任务 .....	207
8.5 使用和构建对象插件 .....	214
8.5.1 使用对象插件 .....	215
8.5.2 解析对象插件 .....	217
8.5.3 编写对象插件 .....	218
8.5.4 插件扩展机制 .....	219
8.5.5 给插件一个有意义的名字 .....	222
8.5.6 测试对象插件 .....	222
8.5.7 开发和使用独立的对象插件 .....	224
8.6 总结 .....	226
<b>9 集成与迁移 .....</b>	<b>229</b>
9.1 Ant 与 Gradle .....	230
9.1.1 在 Gradle 中使用 Ant 脚本功能 .....	231
9.1.2 在 Gradle 中使用标准的 Ant 任务 .....	237
9.1.3 迁移策略 .....	239
9.2 Maven 和 Gradle .....	242
9.2.1 异同之处 .....	243
9.2.2 迁移策略 .....	246
9.3 比较构建 .....	249
9.4 总结 .....	251
<b>第 3 部分 从构建到部署 .....</b>	<b>253</b>
<b>10 IDE 支持和工具 .....</b>	<b>255</b>
10.1 使用 IDE 插件生成项目文件 .....	256
10.1.1 使用 Eclipse 插件 .....	257
10.1.2 使用 IDEA 插件 .....	265

10.1.3 使用 Sublime Text 插件 .....	270
10.2 在流行的 IDE 中管理 Gradle 项目 .....	273
10.2.1 SpringSource STS 对 Gradle 的支持 .....	274
10.2.2 IntelliJ IDEA 对 Gradle 的支持 .....	278
10.2.3 NetBeans 对 Gradle 的支持 .....	280
10.3 使用工具 API 集成 Gradle .....	283
10.4 总结 .....	287
<b>11 构建多语言项目 .....</b>	<b>289</b>
11.1 使用 Gradle 管理 JavaScript .....	290
11.1.1 处理 JavaScript 的典型 task .....	290
11.1.2 在 To Do 应用程序中使用 JavaScript .....	291
11.1.3 对 JavaScript 库依赖管理 .....	292
11.1.4 利用第三方 Ant task 合并和压缩 JavaScript .....	294
11.1.5 将 JavaScript 优化作为开发工作流的一部分 .....	296
11.1.6 使用外部的 Java 库分析 JavaScript 代码 .....	297
11.1.7 使用第三方 Gradle JavaScript 插件 .....	299
11.1.8 在 Gradle 中使用 Grunt .....	300
11.2 构建基于 JVM 的多语言项目 .....	303
11.2.1 JVM 语言插件的基本功能 .....	303
11.2.2 构建 Groovy 项目 .....	305
11.2.3 构建 Scala 项目 .....	310
11.3 其他语言 .....	314
11.4 总结 .....	315
<b>12 代码质量管理和监测 .....</b>	<b>317</b>
12.1 将代码分析集成到构建中 .....	318
12.2 衡量代码覆盖率 .....	319
12.2.1 探索代码覆盖率工具 .....	320
12.2.2 使用 JaCoCo 插件 .....	322
12.2.3 使用 Cobertura 插件 .....	324
12.3 执行静态代码分析 .....	326
12.3.1 探讨静态代码分析工具 .....	327
12.3.2 使用 Checkstyle 插件 .....	329



12.3.3 使用 PMD 插件 .....	331
12.3.4 使用 FindBugs 插件 .....	333
12.3.5 使用 JDepend 插件 .....	334
12.4 集成 Sonar .....	335
12.4.1 安装并运行 Sonar .....	337
12.4.2 使用 Sonnar Runner 分析项目 .....	338
12.4.3 将代码覆盖率报告发布到 Sonar .....	340
12.5 总结 .....	343
<b>13 持续集成 .....</b>	<b>345</b>
13.1 持续集成的好处 .....	346
13.2 安装 Git .....	348
13.2.1 创建 GitHub 账号 .....	348
13.2.2 forking GitHub 仓库 .....	349
13.2.3 安装和配置 Git .....	349
13.3 使用 Jenkins 构建项目 .....	350
13.3.1 开始使用 Jenkins .....	350
13.3.2 安装 Git 和 Gradle 插件 .....	350
13.3.3 定义 build job .....	352
13.3.4 执行 build job .....	354
13.3.5 添加测试报告 .....	356
13.4 探索基于云的解决方案 .....	359
13.5 使用 Jenkins 创建构建管道 .....	360
13.5.1 创建构建管道的挑战 .....	360
13.5.2 探索基本的 Jenkins 插件 .....	361
13.5.3 配置构建管道 .....	364
13.6 总结 .....	366
<b>14 打包和发布 .....</b>	<b>367</b>
14.1 打包和分发 .....	368
14.1.1 定义附加包 .....	369
14.1.2 创建分发包 .....	371
14.2 发布 .....	374
14.2.1 发布到 Maven 仓库中 .....	375

14.2.2 老的和新的发布机制 .....	376
14.2.3 声明软件组件为 Maven 发布包 .....	376
14.2.4 发布软件组件到本地 Maven 缓存中 .....	377
14.2.5 声明自定义的发布包 .....	379
14.2.6 修改所生成的 POM 文件 .....	381
14.2.7 发布到本地 Maven 仓库中 .....	383
14.2.8 发布到远程的 Maven 仓库中 .....	385
14.3 发布到公共的二进制仓库 .....	388
14.3.1 发布到 JFrog Bintray 中 .....	388
14.3.2 发布到 Maven Central .....	392
14.4 打包和发布作为构建管道的一部分 .....	393
14.4.1 构建一次 .....	393
14.4.2 发布一次并重用 .....	394
14.4.3 选择一个合适的版本管理方案 .....	395
14.4.4 在可部署包中加入构建信息 .....	398
14.4.5 发布 To Do 应用程序 WAR 文件 .....	399
14.4.6 扩展构建管道 .....	400
14.5 总结 .....	401
<b>15 基础环境准备和部署 .....</b>	<b>403</b>
15.1 准备基础环境 .....	404
15.1.1 基础设施即代码 .....	404
15.1.2 使用 Vagrant 和 Puppet 创建虚拟机 .....	405
15.1.3 从 Gradle 执行 Vagrant 命令 .....	407
15.2 针对部署环境 .....	409
15.2.1 在 Groovy 脚本中定义配置 .....	409
15.2.2 使用 Groovy 的 ConfigSlurper 读取配置 .....	411
15.2.3 在构建中使用配置 .....	412
15.3 自动部署 .....	413
15.3.1 从二进制仓库中获取包 .....	413
15.3.2 确定必需的部署步骤 .....	415
15.3.3 通过 SSH 命令部署 .....	415
15.4 部署测试 .....	420
15.4.1 使用冒烟测试验证部署成功 .....	420

15.4.2 使用验收测试验证应用程序功能 .....	423
15.5 将部署集成到构建管道中 .....	424
15.5.1 自动部署到测试环境 .....	425
15.5.2 部署测试 .....	425
15.5.3 按需部署到 UAT 和产品环境 .....	426
15.6 总结 .....	427
<b>A 驾驭命令行 .....</b>	<b>429</b>
<b>B Gradle 用户所需要了解的 Groovy .....</b>	<b>435</b>
<b>索引 .....</b>	<b>447</b>

# 第1部分

## Gradle介绍

有效的项目自动化是促使软件成功交付给终端用户的关键因素。构建工具的选择不应该成为软件交付中的障碍；相反，它应该提供一种灵活且可持续的方式去塑造你的自动化需求。Gradle 的核心能力就是提供一种易于理解且强大的工具，从端到端自动化你的项目。

在第 1 章中，我们会讨论项目自动化的好处，以及它对于以可重复、可靠和便捷的方式开发和交付软件的影响。你将会学习到该构建工具的基本概念和组件，以及它们是如何与 Ant 和 Maven 一起工作的。通过对比它们的优缺点，你将会看到对下一代构建工具的需求。

Gradle 从已有构建工具中吸取经验，取其精华，并进一步提升。第 2 章将会介绍 Gradle 中那些引人注目的特性集。你将了解如何安装 Gradle，学习如何写一个简单的构建脚本，并在命令行中运行它。

简单的构建脚本到第 2 章就结束了。第 3 章会介绍一个真实世界中基于 Java 的 Web 应用程序。你将会学习到从编译、单元测试、打包到运行该样例程序所需要的全部配置。在第 1 部分结束时，你会对 Gradle 的表达力和灵活性有一定的感受。



# 项目自动化介绍

## 本章涵盖

- 理解项目自动化的好处
- 了解不同类型的项目自动化
- 考察构建工具的特性和架构
- 探索构建工具实现的优点和缺点

Tom 和 Joe 以软件开发人员的身份在 Acme Enterprises 公司工作，这是一家刚刚成立的公司，该公司提供一种免费的在线服务，帮助你在某个具体的领域中寻找最划算的交易。这家公司最近收到投资者的资金投入，目前正疯狂地朝着第一次发布前进。Tom 和 Joe 正处于一个危急时刻。到下个月底，他们需要向投资者展现产品的第一个版本。两个开发人员都是有紧迫感的人，他们每天都能开发出新的特性。到目前为止，软件开发控制在规定的的时间和预算之内，这让他们感到满意。首席技术官（CTO）时不时会过来拍拍他们的后背，生活非常的美好。然而，手动构建和错误构建以及整个交付的过程都严重影响他们的速度。结果是，整个团队不得不忍受着零散的编译问题、构建不一致的软件工件和失败的部署。这就是引入构建工具的原因。

本章会给出一个简单的介绍，告诉你为什么要自动化项目，以及如何通过自动化工具做到这一点。我们会聊聊充分的项目自动化所带来的好处、项目自动化的类型和特性，以及那些能够让你实现项目自动化的工具。

传统的面向 Java 项目的构建工具有两个：Ant 和 Maven。我们会重温它们的主要特性，阅读一些构建代码，谈谈它们的缺点。最后，我们会一起讨论一个满足当代项目自动化需求的构建工具应该提供什么。

## 1.1 没有项目自动化的生活

回到 Tom 和 Joe 的困境中，让我们看看为什么项目自动化是一件无须用脑的事情。不管你相信与否，许多开发人员都面临下面这些情况。原因各有不同，但是能听起来很熟悉。

- IDE 给我们做了这些事情。在 Acme 公司，开发人员在 IDE 中进行编码，从浏览源代码、实现新的特性、编译、重构，到运行单元测试和集成测试。当有新的代码编写出来时，他们就按一下“编译”按钮。如果 IDE 告诉他们没有编译错误且测试通过，他们就会把代码提交到版本控制中，这样就可以共享代码了。IDE 是一个强大的工具，但是每个开发人员第一次使用时都需要安装一个标准化的版本来执行所有的任务。这是 Joe 在使用新特性时学到的一课，因为他发现只能使用最新版本的编译器编译。
- 在我的机器上可以跑。眼睛盯着滴答作响的时钟，Joe 从版本控制系统中更新代码，却发现根本不能编译。似乎源代码中有一个类丢失了。他问 Tom，Tom 也奇怪为什么在 Joe 的机器上不能编译代码。在经过讨论之后，Tom 意识到他可能忘记提交某个类文件了，结果导致编译失败。团队的其他人员都由于这个原因而不能继续工作，直到 Tom 把丢失的文件提交上去。
- 代码的集成完全就是个灾难。Acme 公司有两个不同的开发组，其中一个组专注于构建基于 Web 的用户界面，另一个组工作于服务器后台代码。两个组的人员一起坐在 Tom 的机器前，运行编译整个应用程序，要构建出一个可交付的软件，然后部署到测试环境中的一个 Web 服务器上。第一次的欢呼声很快就结束了，因为团队发现某些功能并没有像预期的那样正常工作。某些 URL 根本不能被解析或者导致错误。虽然团队写了一些功能测试，但却不能定期地在 IDE 里工作。
- 测试过程慢得像在爬。质量保证（QA）团队渴望立刻拿到应用程序的第一个版本。你能想象到，他们可不想测试低质量的软件。开发团队的每一次修复，他们都不得不手动地走完同样的流程。团队停止向版本控制系统中提交新的代码，一个新的软件版本从 IDE 中构建出来，并且将可交付软件拷贝到测试



服务器上。每一次，一个开发人员都要全身心地投入到这项工作中，而不能给公司带来其他价值。在经过几周测试之后，一个成功的样例程序交付到投资者手中，QA 团队说应用程序已经准备好在黄金时间上演一出好戏。

- 部署过程变成了马拉松。从经验上看，团队知道部署一个应用的结果是不可预知的，因为存在不可预知的问题。基础设施和运行时环境必须配置好，数据库需要的种子数据必须要准备好，实际的部署需要被执行，并且需要执行初期的健康监控。当然，团队是有计划的，但是每一步都需要手动执行。

软件发布非常的成功。接下来的一周，CTO 在开发人员的桌子前游荡；他已经有了一个新的想法来提高用户体验。一个好朋友告诉他敏捷开发，一种时间控制的迭代式实现和发布软件的方法。他向团队建议两周一次的发布周期。Tom 和 Joe 互相看着对方，两个人都已经被预知的手动和重复工作吓到了。于是，他们俩想到了一起，决定自动化实现和交付软件的每一步，以此来减少构建失败、后期集成和痛苦部署的风险。

## 1.2 项目自动化的好处

这个故事很清楚地说明了项目自动化对于团队的成功是多么的重要。如今，发布时间对于市场变得比以前更重要了。能够以一种可重复、可持续的方式构建和交付软件是关键。让我们看一看项目自动化所带来的好处。

### 1.2.1 防止手动介入

不得不手动地执行每一步去实现和交付软件是耗时且易于犯错的。坦白地说，作为一个开发人员和系统管理员，比起编译过程和拷贝文件，还有更重要的事情要做。我们都是人，难免会犯错，而且手动介入还会占用你真正做实际事情的时间。软件开发过程中的任何一步都是能够且应该被自动化的。

### 1.2.2 创建可重复的构建

软件的构建通常都有预定义和有序的步骤。比如，你需要先编译源代码，然后运行测试，最后组装可交付软件。你将需要每天一遍又一遍地重复运行相同的步骤。这应该和按一下按钮一样简单。无论是谁在运行该构建，构建过程的结果都应该是可重复的。

### 1.2.3 让构建便携

你已经看到，能够在 IDE 中运行的构建是非常有限的。首先，你必须将特定的

产品安装在机器上。其次，IDE 也许只适用于某一种操作系统。一个自动化构建不应该依赖于特定的运行环境才能工作，无论是操作系统还是 IDE。最佳的方式应该是，自动化任务从命令行运行，它允许你在任何时间和任何一台想要运行构建的机器上运行。

## 1.3 项目自动化的类型

在本章的开始，你看到了用户可以请求一个构建运行。用户可以是任何想要触发该构建的利益相关者，比如开发人员、测试人员或者产品拥有者。例如，我们的好朋友 Tom，当他想要编译代码时，就按一下“编译”按钮。按需自动化只是项目自动化的一种类型。你也可以将构建安排在某一个预定义时间触发或者某个特殊事件发生时触发。

### 1.3.1 按需构建

按需构建的典型用例就是用户在自己的机器上触发构建，如图 1.1 所示。而使用版本控制系统（VCS）来管理构建定义的版本和源代码文件是一种很常见的做法。

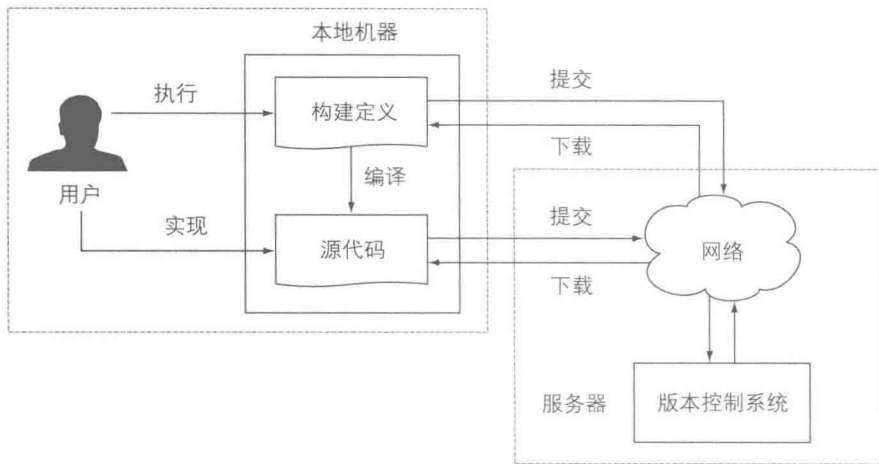


图 1.1 在版本控制系统支持下按需执行构建定义

在大多数情况下，用户在命令行执行一个脚本去运行预先定义的有序任务——比如，编译源代码，拷贝文件从 A 目录到 B 目录，或者组装一个可交付软件。通常，这种类型的构建一天要运行多次。

### 1.3.2 触发构建

如果你正在实践敏捷开发，那么你会对一件事情感兴趣，那就是快速得到项目健康程度的反馈。你会想要知道，是否源代码能够无错误地编译，是否有单元测试或集成测试失败，以反馈软件中存在的潜在缺陷。这种类型的自动化通常是在向版本控制系统中提交代码时触发，如图 1.2 所示。

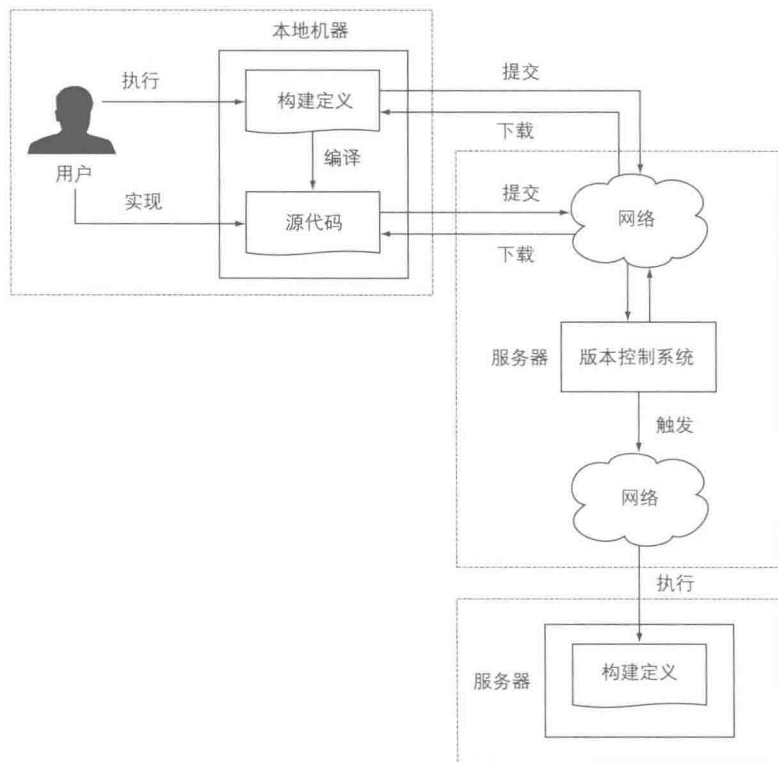


图 1.2 构建由一次代码提交触发

### 1.3.3 预定构建

预定构建是一种基于时间的程序调度方案（在 UNIX 操作系统背景下，也叫作定时作业）。它在特定的间隔时间或者某个具体时间运行——例如，每天的凌晨 1:00 点，或者每隔 15 分钟。和所有的定时作业一样，预定的自动化任务通常运行在一个专用的服务器上。图 1.3 显示的是一个预定的构建在每天凌晨 5:00 点运行。这种类型的构建特别适用于生成报告或者项目的文件操作。

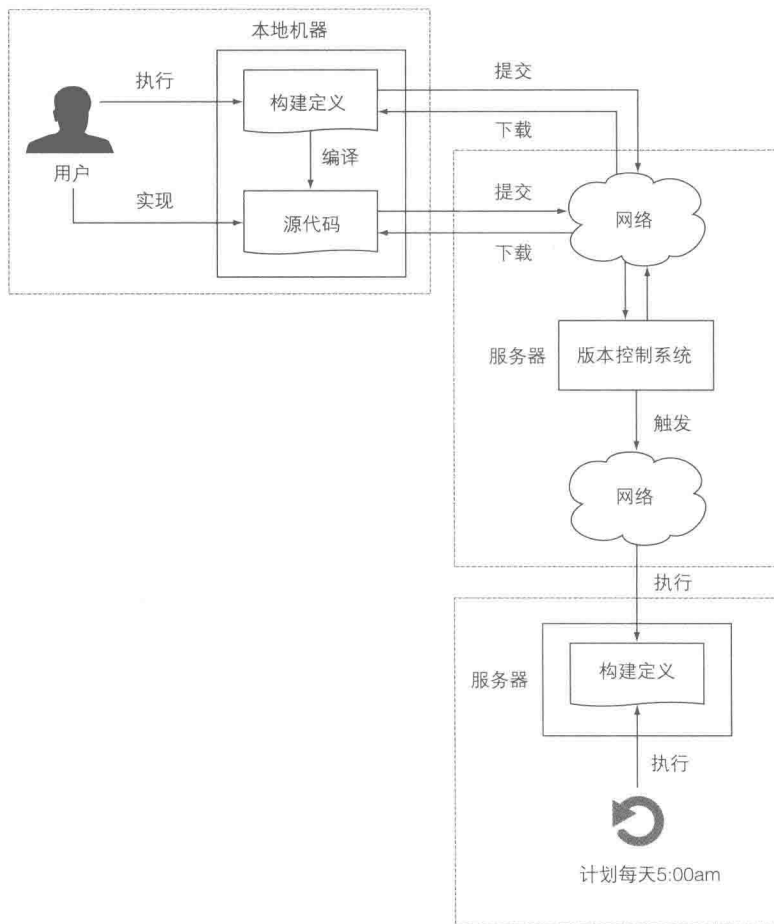


图 1.3 计划每天 5:00am 初始化构建

实现预定义和触发构建的实践方式通常也叫作持续集成（CI）。你会在第 13 章学到更多关于持续集成的知识。

在认识了项目自动化的好处和类型之后，是时候来讨论一下那些能让你实现这些功能的工具了。

## 1.4 构建工具

当然，你可能会问自己，为什么需要另外一个工具来实现项目自动化。用可执行脚本的方式完全可以写一些逻辑，例如 shell 脚本。回想一下我们之前讨论过的关于项目自动化的目的。你想要有一个工具，它能够帮助你创建一个可重复的、可靠的、

便携的且不需要手动干预的构建。一个 shell 脚本可不是那么容易就能从 UNIX 系统迁移到 Windows 系统上的，所以这样就无法满足你的需求了。

### 1.4.1 什么是构建工具

你所需要的是一个可编程的工具，它能够让你以可执行和有序的任务来表达自动化需求。假设你想要编译源代码，将生成的 class 文件拷贝到某个目录，然后将该目录组装成可交付的软件。这个可交付的软件可以是一个 ZIP 文件，比如，它可以被发布到某一个运行时环境中。图 1.4 展示了所描述场景中的任务和它们执行的顺序。



图 1.4 一个以预定义顺序执行任务的常见场景

每个任务都代表着一个工作单元——例如，编译源代码。顺序是非常重要的。如果所需要的 class 文件没有被编译出来，那么你是不能创建 ZIP 构件的。因此，编译任务必须先被执行。

#### 有向非循环图

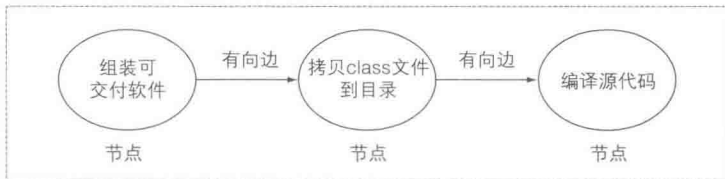
本质上，任务和它们的相互依赖被模块化成一个有向非循环图（DAG）。DAG 是计算机科学里的一种数据结构，包含下面两个元素。

- 节点：一个工作单元；就构建工具而言，它指的是一个任务（例如，编译源代码）。
- 有向边：有向边也叫作箭头，表示节点之间的关系。在这里，箭头表示依赖关系。如果一个任务的定义依赖于另一个任务，那么所依赖的任务就必须先被执行。发生这种情况常常是因为一个任务依赖于另一个任务的输出。这里有个例子——要执行任务“组装可交付软件”，你需要先执行任务“拷贝 class 文件到目录”和“编译源代码”。

每个节点都知道自己的执行状态。一个节点——表示一个任务——只能被执行一次。例如，如果两个不同的任务都依赖于任务“编译源代码”，那么你会想要执行一次这个任务。图 1.5 以 DAG 图的形式展示了这种场景。

你也许已经注意到，在图 1.4 中，节点与任务显示的方向相反。这是因为顺序由节点的依赖决定。作为一个开发人员，你没有必要与 DAG 图打交道。这个工作是由构建工具来完成的。在本章的后面，你会看到基于 Java 的构建工具是如何在实践中使用这些概念的。

有向非循环图



任务依赖

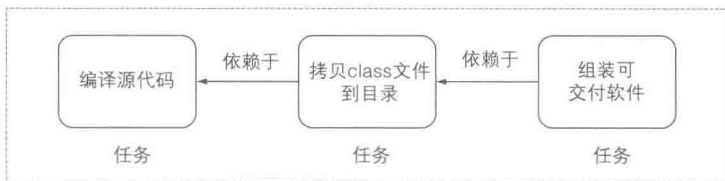


图 1.5 有向非循环图表示的任务

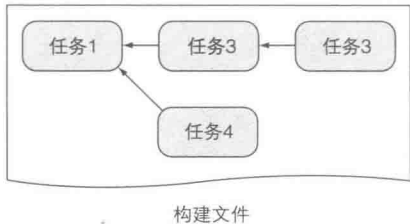
### 1.4.2 构建工具的剖析

理解构建工具中组件的交互、构建逻辑的实际定义，以及输入和输出的数据是非常重要的。让我们一起来探讨一下构建工具中每一个元素以及它们的职责。

#### 构建文件

构建文件包含了构建所需的配置信息、定义外部依赖，例如第三方类库，还包含了以任务形式实现某个特殊目的的指令和它们的相互依赖关系。图 1.6 展示了一个描述 4 个任务和它们之间相互依赖的构建文件。

在前面场景中我们讨论的任务——编译源代码、拷贝文件到目录以及组装 ZIP 文件——都可以定义在构建文件中。在通常情况下，会使用脚本语言来表达构建逻辑。这就是为什么一个构建文件也叫作构建脚本的原因。



#### 构建的输入和输出

一个任务会接收一个输入，然后执行一系列步骤，最后产生一个输出。某些任务也许不需要输入，也不需要产生一个必要的输出。在复杂的任务依赖关系中，也许会使用一个依赖任务的输出作为输入。图 1.7 展示了在任务关系图中输入的消耗和输出的产生。

图 1.6 构建文件说明的构建规则由任务以及它们之间的依赖表示

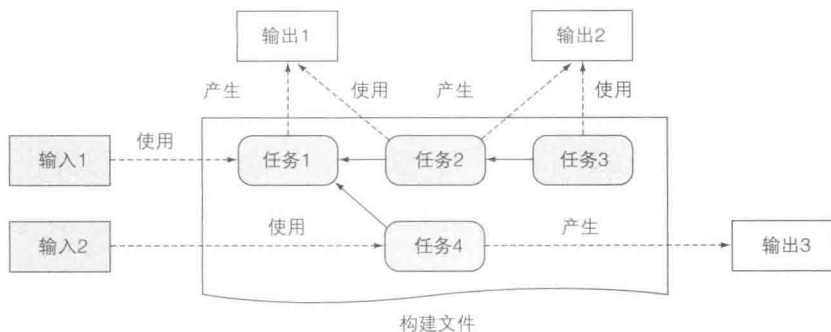


图 1.7 任务的输入和输出

下面我给出的例子遵循这个流程。我们将源代码文件作为输入，将它们编译为 class 文件，并组装成可交付软件作为输出。编译和组装过程各表示一个任务。只有先编译了源代码，组装可交付软件才有意义。因此，两个任务需要保证它们的顺序。

### 构建引擎

构建文件的一步步指令或者规则集必须被翻译成构建工具可以理解的内部模型。构建引擎会在运行时处理构建文件，解析任务之间的依赖，设置好执行所需要的全部配置，如图 1.8 所示。

一旦内部模型建立好了，引擎就会按照正确的顺序去执行一系列任务。某些构建工具还允许你通过 API 去访问这个模型，以便在运行时获取构建信息。

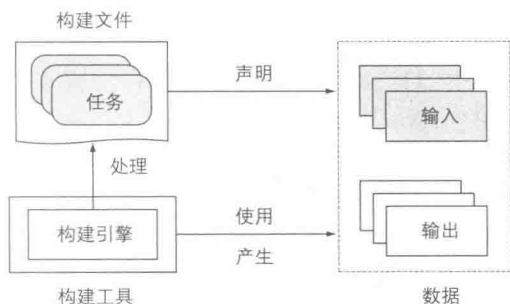


图 1.8 构建引擎将规则集翻译成内部模型表现方式，并在构建运行时被访问

### 依赖管理器

依赖管理器用于处理你在 build 文件中声明的依赖定义，从工件仓库（例如，本地文件系统、一个 FTP 或者 HTTP 服务器）中解析它们，并使它们对项目可用。依赖通常是指外部依赖，一种 JAR 文件形式的可重用类库（例如，Log4j 对日志的支持）。该仓库就像是依赖的储藏所，通过标识符组织和描述它们，例如名字和版本。一个典型的仓库可以是 HTTP 服务器或者本地文件系统。图 1.9 展示了依赖管理器在构建工具架构中所处的地位。

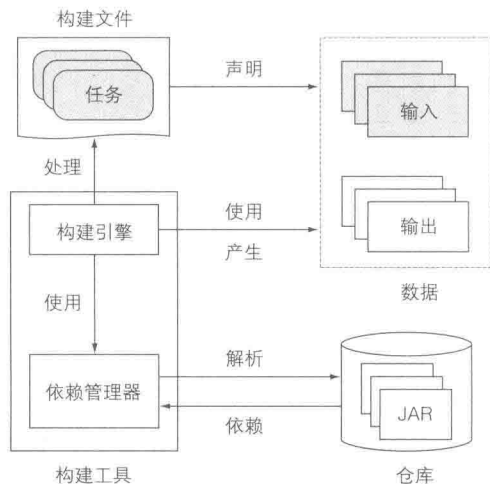


图 1.9 构建可以访问依赖管理器获取外部依赖

许多类库还依赖于其他类库，这叫作传递依赖。依赖管理器可以通过存储在仓库中的元信息自动地解析传递依赖。但是一个构建工具并不要求提供这样的依赖管理组件。

## 1.5 Java构建工具

在本节，我们会看到两个流行的基于 Java 的构建工具：Ant 和 Maven。我们会讨论它们的特性，看看实际的样例脚本，并概述每个工具的缺点。我们首先来看最早的工具——Ant。

### 1.5.1 Apache Ant

Apache Ant (Another Neat Tool) 是一个用 Java 编写的开源构建工具。其主要目的是在 Java 项目中为常用任务提供自动化，例如编译源代码、运行单元测试、打包 JAR 文件和生成 Javadoc 文档。另外，它还为文件系统和存档操作提供了许多不同的预定义任务。如果任何一个任务不满足需求，那么你就可以用 Java 写新的任务来扩展构建。

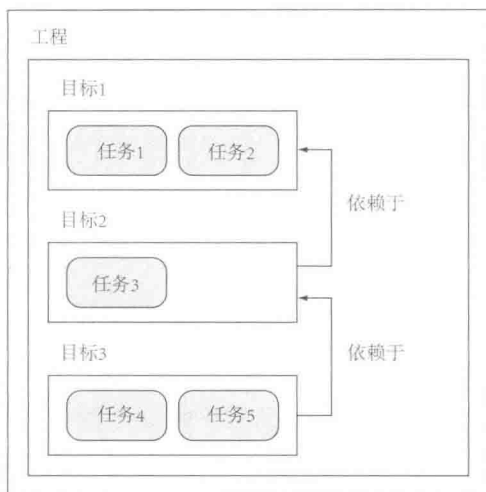
虽然 Ant 的核心是用 Java 编写的，但是 build 文件是通过 XML 表示的，这样就可以在任何运行时环境下使用了。Ant 不提供依赖管理器，所以你需要自己管理外部依赖。然而，Ant 可以和另一个 Apache 项目 Ivy 很好地集成，它是一个完善且独立的依赖管理器。要集成 Ant 和 Ivy 需要一些额外的工作，而且要为每个独立的项目手动配置。让我们一起看一个样例构建脚本。

#### 构建脚本术语

要理解 Ant 的构建脚本，你需要先了解一些命名规范。一个构建脚本由三个基本元素组成：一个 project、多个 target 和可用的 task。图 1.10 展示了每个元素之间的关系。



构建脚本

图 1.10 Ant 的构建脚本层级结构：  
project、target、task

在 Ant 中, *task* 是一段可执行的代码——例如, 创建一个目录或者移动一个文件。在构建脚本中, 通过预定义的 XML 标签名来使用一个 *task*。task 的行为可以由其暴露出来的属性配置。下面的代码片段展示了如何在构建脚本中调用 *javac* 的 *task* 来编译 Java 代码:

```
<javac srcdir="src" destdir="dest"/>
```

源代码目录和目标目录由 *srcdir* 和 *destdir* 属性配置, 源代码放在 *src* 目录, class 文件放在 *dest* 目录

Ant 自身带有许多预定义的 *task*, 然而你也可以用 Java 语言编写自己的 *task* 以扩展脚本功能。

*target* 是你想要执行的 *task* 的一个集合, 可以把它想成一个逻辑组。当在命令中运行 Ant 时, 提供了你想要运行的 *target* 的名字。通过声明 *target* 之间的依赖关系, 就可以创建一个完整的命令链。下面的代码片段展示了两个依赖的 *target*:

```
<target name="init">
  <mkdir dir="build"/>
</target>
```

名字为 *init* 的 *target* 使用 *mkdir* 任务  
创建 *build* 文件夹

```
<target name="compile" depends="init">
  <javac srcdir="src" destdir="build"/>
</target>
```

名字为 *compile* 的 *target* 通过 *javac* 的 Ant 任务编译 Java 源代码。该 *target* 依赖于 *init*, 所以如果你在命令行运行, *init* 会先执行

*project* 是对于所有 Ant 项目都必要的容器。它是 Ant 脚本的顶级元素, 包含一个或多个 *target*。在每个构建脚本中, 你只能定义一个 *project*。下面的代码片段展

示了 project 和 target 的关系：

```
<project name="example-build">
  <target name="init">
    <mkdir dir="build"/>
  </target>

  <target name="compile" depends="init">
    <javac srcdir="src" destdir="build"/>
  </target>
</project>
```

project 包含了一个或多个 target 和可选属性，例如 name，来描述这个 project

有了对 Ant 层级结构的基本理解，让我们来看一个完整场景的构建脚本样例。

### 构建脚本样例

假设你想要写这样一个脚本，用 Java 编译器去编译 src 目录下的 Java 源代码，并将结果放在输出目录 build 中。Java 源代码中有一个类依赖于 Apache Commons Lang 类库。将类库中的 JAR 文件放置在 classpath 路径下，这样编译器就知道该类库的存在了。在编译完成之后，你想要组装一个 JAR 文件。对于每一个工作单元，源代码编译和 JAR 文件的组装都会被分配到一个独立的 target 中。你还需要添加另外两个 target 去做初始化和对输出目录进行清理。图 1.11 展示了该 Ant 构建脚本的结构。

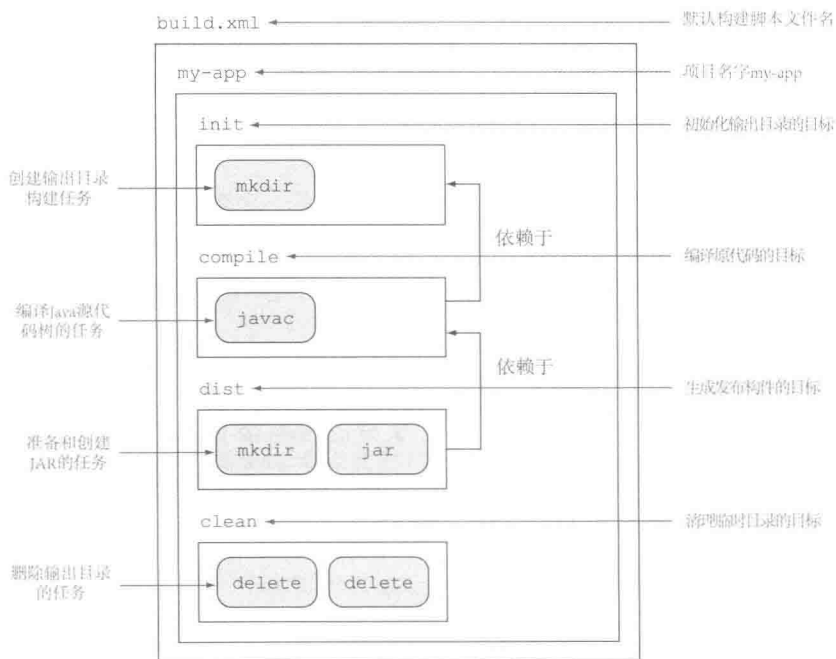


图 1.11 Ant 构建脚本样例的项目层级结构

让我们深入到业务中，是时候用 Ant 构建脚本去实现这个例子了。下面的清单展示了整个项目，以及实现该目标所需要的 target。

**清单 1.1 含有编译源代码和组装 JAR 文件任务的 Ant 脚本**

```
<project name="my-app" default="dist" basedir=".">
  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist" location="dist"/>
  <property name="version" value="1.0"/>

  <target name="init">
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init" description="compile the source">
    <javac srcdir="${src}" destdir="${build}"
      ➡ classpath="lib/commons-lang3-3.1.jar"
      ➡ includeantruntime="false"/>
  </target>

  <target name="dist" depends="compile">
    ➡ description="generate the distribution"
    <mkdir dir="${dist}"/>
    <jar jarfile="${dist}/my-app-${version}.jar" basedir="${build}"/>
  </target>

  <target name="clean" description="clean up">
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>
</project>
```

给构建设置全局属性，比如说源代码、输出和发布目录

创建由编译 target 使用的构建目录结构

从 src 目录编译源代码到 build 目录

创建发布目录

组装所有东西到 build 目录构建 myapp-1.0 的 JAR 文件

删除 build 和 dist 目录树

Ant 没有对如何定义构建的结构强加任何限制。这样让适应一个现有的项目结构变得简单。例如，在样例脚本中，源代码目录和输出目录是随意选择的。通过修改相关的属性可以非常轻松地改变它们。对于 target 的定义也是一样的；对于每个 target，哪个逻辑需要被执行，以及它们的执行顺序，你拥有完全灵活的选择性。

## 缺点

尽管有这样的灵活性，但是你也应该注意到一些缺点：

- 使用 XML 作为构建逻辑的定义语言相比于其他更简明的定义语言，会导致构建脚本过于臃肿和啰唆。
- 复杂的构建逻辑会导致又长又难以维护的构建脚本。当尝试用标记语言去定义类似 if-then/if-then-else 的逻辑语句时，它完全就成了一种负担。
- Ant 没有提供任何指导来告诉你如何建设项目。在一个企业级配置中，这常常会导致一个 build 文件每一次看上去都不一样。常用功能时常被到处拷贝。

项目中每一个新的开发人员都需要去理解构建中每一个独立的部分。

- 你想要知道在构建中有多少个类被编译或者多少个 task 被执行。Ant 没有暴露任何的 API 能够让你在运行时获取内存模型中的信息。
- 在没有 Ivy 的情况下，使用 Ant 很难管理依赖。在通常情况下，你需要将 JAR 文件提交到版本控制系统中，并且手动管理组织结构。

## 1.5.2 Apache Maven

在企业应用的多个项目中使用 Ant，对可维护性有很大的影响。灵活性带来了许多由项目间拷贝产生的重复代码。Maven 团队认识到标准化项目布局 and 统一构建生命周期的必要性。Maven 选择约定优于配置的思想，这意味着它为你的项目配置和行为提供了有意义的默认值。项目自然而然就知道去哪些目录寻找源代码以及构建运行时有哪些 task 去执行。如果你的项目遵从默认值，那么只需要写几行 XML 就可以建立一个完整的项目。另外，Maven 也拥有为应用产生包含 Javadoc 在内的 HTML 格式项目文档的能力。

Maven 的核心功能可以通过开发定制的插件来扩展。Maven 的社区非常活跃，几乎支持构建的每个方面，从集成其他工具到生成报告，你都能够找到合适的插件。如果找不到满足需求的插件，你也可以自己去写。

### 标准的目录布局

通过引入一个默认的项目布局，Maven 确保每个拥有 Maven 知识的开发人员可以立刻知道去哪里找什么类型的文件。例如，Java 应用程序源代码的目录是 src/main/java。所有默认的目录都是可配置的。图 1.12 展示了一个 Maven 项目的默认布局。

Maven 默认项目布局

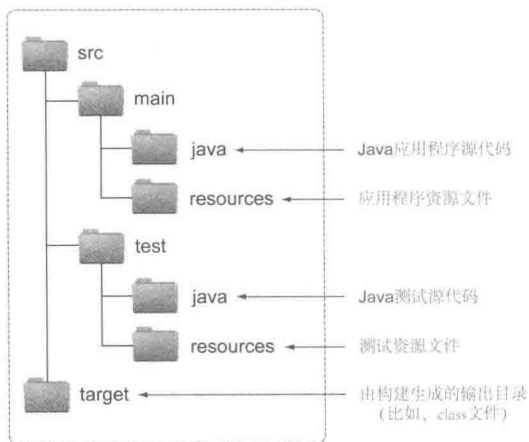


图 1.12 Maven 的默认项目布局定义了 Java 源代码、资源文件和测试代码的位置

## 构建生命周期

Maven 基于构建生命周期的思想。每个项目都确切知道有哪些步骤去执行构建、打包和发布应用程序，包括如下功能：

- 编译源代码
- 运行单元测试和集成测试
- 组装工件（例如，JAR 文件）
- 将工件部署到本地仓库
- 将工件发布到远程仓库

在构建生命周期中每个步骤都称作一个阶段(*phase*)。这些阶段会被有序地执行。当在命令行中运行构建时，你想要执行的阶段是固定的。假设你调用打包这个阶段，Maven 会自动确定它所依赖的阶段如编译源代码和运行测试事先被执行。图 1.13 展示了 Maven 构建所预定义的阶段和它们执行的顺序。

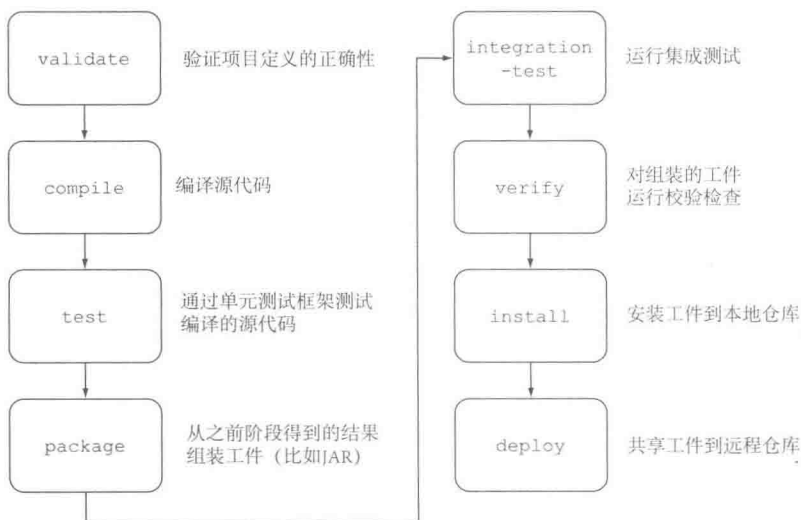
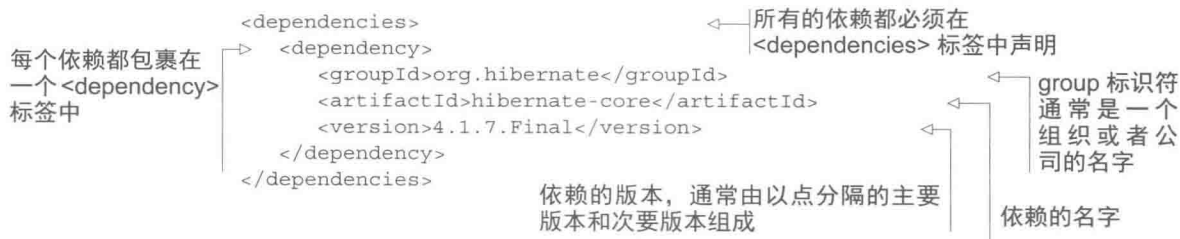


图 1.13 Maven 最重要的构建生命周期阶段

## 依赖管理

在 Maven 项目中，所需要的外部依赖库都在构建脚本中定义。例如，如果项目需要 Hibernate，那么你可以在依赖配置块中简单地定义它的独立工件坐标，比如组织名、工件名和版本。下面的代码片段展示了如何声明一个 4.1.7 Final 版本的 Hibernate 核心库：



在运行时, 声明的类库和它们的传递依赖会由 Maven 的依赖管理器下载, 保存到本地缓存中, 这样你的构建就可以使用它们 (例如, 编译源代码)。Maven 预配置从 Maven Central 下载依赖。接下来构建会从本地缓存中重用已存在的工件, 因此不用再连接 Maven Central。Maven Central 是 Java 社区中最流行的二进制工件仓库。图 1.14 展示了 Maven 工件的获取过程。

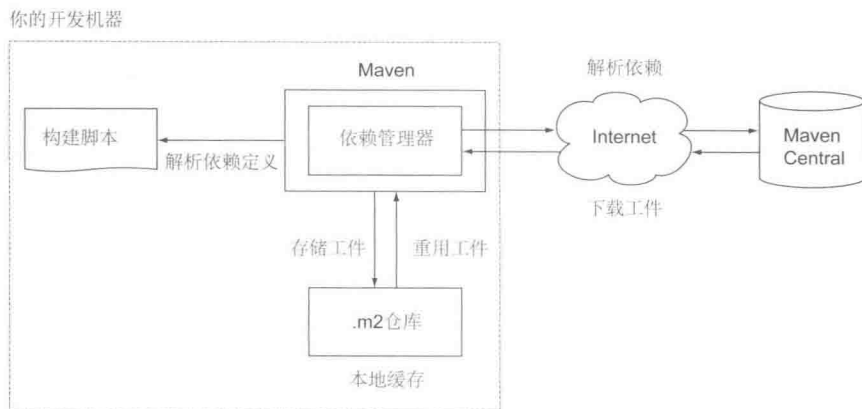


图 1.14 通过与 Maven Central 的交互为构建解析和下载依赖

Maven 中的依赖管理不仅限于外部库。你也可以将其他 Maven 项目定义为依赖。这种需求出现的原因是软件分解成了多个模块, 每个模块都是完成某项功能的组件。图 1.15 展示了一种传统的三层结构的模块化架构。在这个例子中, 表现层包含了在页面渲染数据的代码, 业务层是真实的业务对象, 而集成层则从集成库中获取数据。

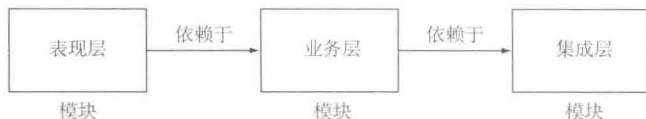


图 1.15 软件项目的模块化架构

## 构建脚本样例

下面的清单展示了一个 Maven 构建脚本样例，名字是 `pom.xml`，它实现了和 Ant 构建脚本相同的功能。记住，在这里要遵循默认约定，所以 Maven 会去 `src/main/java` 目录中寻找源代码，而不是 `src`。

清单 1.2 构建标准 Java 项目的 Maven POM 文件



## 缺点

和 Ant 一样，也要知道 Maven 的缺点：

- Maven 推荐一个默认的结构和生命周期，常常会太过限制，也许不适合你的项目需求。
- 为 Maven 写定制的扩展过于累赘。你需要去学习 Mojos (Maven 的内部扩展 API)，如何提供一个插件描述符 (又是 XML)，以及相关的特殊注解，以便提供扩展实现所需要的数据。
- Maven 的早期版本 (低于 2.0.9) 会自动尝试更新它们自己的核心插件，例如，将单元测试的支持插件升级到最新版本。这可能会导致脆弱和不稳定的构建。

## 1.5.3 对下一代构建工具的需求

在上面小节中，我们了解了 Ant 和 Maven 工具的特性，以及它们的优势和缺点。很明显，你常常需要在选择构建工具时妥协于它们所支持的功能。要么你选择完全灵活且可扩展，但很难实现项目标准化，有一堆公式化代码，并且没有依赖管理支持的 Ant，要么选择 Maven，它能提供约定优于配置的方式和无缝的依赖管理器集成，但过于限制思维和拥有臃肿的插件系统。

如果一个构建工具能够折中，是不是很棒？下面的一些特性是一个演变的构建工具应该提供的：

- 具有表达性、声明式、可维护的构建语言。
- 标准化的项目布局 and 生命周期，但是具有完全的灵活性和对默认值的完全可配置性。
- 拥有易用且灵活的方式去实现定制逻辑。
- 支持构建由多个项目组成的项目结构。
- 支持依赖管理。
- 能很好地集成和迁移现有的构建基础设施，包括能够引入现有的 Ant 构建脚本和可以将现有的 Ant 或 Maven 逻辑转换成其自身规则集的工具。
- 强调可扩展性和高效率的构建。如果你的项目需要长时间构建（比如 2 小时或更长时间），这一点是很重要的，某些大型的企业级项目就是这样。

本书会向你介绍一个工具，它提供了这些很棒的特性，它的名字叫作 Gradle。我们会一起学习如何使用 Gradle 和探索它能提供的所有优点。

## 1.6 总结

对于开发和测试人员来说，没有项目自动化的工作，都是重复、单调和易犯错的。伴随软件交付过程中的每一步——从源代码编译到打包软件，再到发布至测试和产品环境——都必须手动操作。项目自动化帮你消除手动操作介入的负担，让你的团队更有效率，带你进入到一个一键式和故障安全的软件发布过程。

在本章，我们了解了不同类型的项目自动化——按需构建，预定构建和触发构建——覆盖了各种特殊用例。你应该了解到不同类型的项目自动化是不相互排斥的。实际上，它们是互相补充的。

项目构建工具是项目自动化的一个关键因素。它允许你声明一系列有序的规则，允许你在初始化构建时执行。通过分析构建工具的内部原理，我们讨论了构建工具的活动组件。构建引擎（构建工具的可执行度）处理构建脚本中预定义的规则集，并将它翻译成为可执行的任务。每个任务也许需要输入数据。最终，产生构建的输出。依赖管理器是构建工具架构中的一个可选组件，它允许你声明项目正常运行时所需要的外部库。

通过深入了解两个流行的 Java 构建工具：Ant 和 Maven 的实现方式，我们看到了构建工具的具体特性。Ant 提供了一种非常灵活且通用的方式来定义构建逻辑，但是没有对标准化项目提供指导或者有意义的默认任务，以便可以在项目中重复使用。它也没有提供一个开箱即用的依赖管理器，需要自己去管理外部依赖。Maven，则正好相反，它遵循约定优于配置的原则，支持有意义的默认配置和标准的构建生



命周期。Maven 的内置特性支持对外部库和直接对 Maven 项目的自动化依赖管理。但 Maven 的败笔却落在定制逻辑和非约定项目的结构和任务的易扩展性上。你知道，一个高级的构建工具需要在灵活性和可配置约定上折中，这样才能满足现代化软件项目的需求。

在下一章中，我们会来了解 Gradle 是如何满足构建需求的。



# 下一代构建工具：Gradle

## 本章涵盖

- 理解 Gradle 和其他构建工具的不同
- 介绍 Gradle 的引人注目的特性集
- 安装 Gradle
- 编写和执行简单的 Gradle 脚本
- 在命令行中运行 Gradle

多年来，构建只有编译和打包的简单需求。但是现代软件开发的规模改变了，因此有了对自动化构建的需求。

今天，大多数项目都包含有多而杂的技术栈、混合的多种编程语言，并且使用多种测试策略。随着敏捷实践的崛起，构建不得不更早地支持代码集成，以及频繁和简单地交付软件到测试和产品环境。

现有的构建工具不能够以一种简单但是可定制的方式去满足这些要求。多少次你注视着 XML 文件，只是想要弄清楚构建是怎么工作的？而且为什么不能以更简单的方式向构建中添加定制逻辑？通常，当你向一个构建脚本中添加逻辑时，总摆脱不了一种使用了变通方式或者非常规方式实现的感觉。我深知你的痛苦。一定有

一种更好的方式,即以一种可表达且可维护的方式去做这些事情。确实有这样的方式,那就是 Gradle。

Gradle 是基于 JVM 构建工具的新一代版本。它从现有的构建工具如 Ant 和 Maven 中学到了很多东西,并且把它们的最优思想提升到更高层次。遵循基于约定的构建方式,Gradle 可以用一种声明式的方式为你的问题领域建模,它使用一种强大的且具有表达性的基于 Groovy 的领域特定语言(DSL),而不是 XML。因为 Gradle 是基于 JVM 的,它允许你使用自己最喜欢的 Java 或者 Groovy 语言来编写定制逻辑。

在 Java 世界里,有大量类库和框架可以使用。依赖管理可以自动地从仓库中下载工件,并为项目代码所用。Gradle 从现有的依赖管理解决方案的缺点中学习,提供了一套自己的依赖管理实现方式。不仅高度可配置,而且也尽可能地与现有的依赖管理设施(如 Maven 和 Ivy)相兼容。Gradle 管理依赖的能力不仅限于外部库。随着项目大小和复杂度的增加,你会想要以模块的方式来组织代码,以清晰地定义它们的职责。Gradle 对多项目构建的定义和组织提供了强有力的支持,以及对项目之间的依赖建模。

我知道,所有这些听起来都让你感觉看到了希望,但是你当前还陷在遗留的构建当中。Gradle 不会把你留在烂摊子里面,它会让你的迁移变得简单。Ant 可以在运行时装载,因此不需要任何额外的设置。Gradle 允许团队利用他们已经累积的 Ant 知识,以及在已有构建基础设施中的投入。想象一下,在 Gradle 构建脚本中直接使用已经存在的 Ant 任务和脚本的可能性。遗留的构建逻辑能够被重用或者逐渐迁移。Gradle 的确减轻了你不少的负担。

要开始使用 Gradle,你所需要的就是对 Java 编程语言有一个较好的理解。如果你是第一次接触项目自动化或者以前没有使用过构建工具,那么阅读第 1 章是一个好的开始。本书会告诉你如何有效地使用 Gradle 进行构建和交付真实世界的项目。

在本章中,我们会将 Gradle 提供的特性和现有的基于 JVM 语言的构建工具进行对比。之后,你会了解到 Gradle 是如何在持续交付的部署管道中帮助你实现自动化软件交付的。要初次体验使用 Gradle 的感觉,你需要首先安装 Gradle,然后编写一个简单的构建脚本,并在命令行中运行它。现在,跟我一起去探索振奋人心的 Gradle 世界吧。

## 2.1 为什么要用 Gradle, 为什么是现在

如果你曾经与构建系统打过交道,那么当你想到曾经遇到过的挑战时,沮丧也许是其中一种感觉。难道构建工具不应该很自然地帮助你完成项目自动化的目标吗?相反,你不得不向可维护性、可用性、灵活性、可扩展性或者性能妥协。

假设当前的情况是你在给项目构建一个发布版本，而你想要拷贝一个文件到特定的位置。为了确定版本，你需要在描述项目的元数据中检查一个字符串。如果它匹配某种数字模式（例如，1.0-RELEASE），你就将文件从 A 点拷贝到 B 点。从局外人的观点看，这也许听起来像是一件不太重要的事情。如果你不得不依赖于 XML，许多传统构建工具的构建语言，那么用它来表达逻辑就变成噩梦。构建工具给出的答案是通过非标准扩展机制来添加脚本功能。最终变成将脚本代码与 XML 混合或者从构建逻辑中触发外部脚本。可以想象，你将会需要越来越多的定制代码。结果就是，你不可避免地引入了偶然的复杂性，而降低了构建的可维护性。难道不应该一开始就使用一种具有可表达性的语言来定义构建逻辑吗？

再举一个例子。Maven 遵循约定优于配置的规范，为 Java 项目引入了一个标准化的项目布局和构建生命周期。如果你想要确保一个待开发的项目——一个对之前的工作没有任何限制的项目，具有统一的项目结构，那么这是一个非常棒的方式。然而，你也许比较幸运，需要在许多遵循不同约定的遗留项目上工作。Maven 严格遵循的约定之一就是项目需要生成一个工件，比如 JAR 文件。但是你如何在改变项目结构的情况下，从一个项目源中生成两个不同的 JAR 文件呢？仅仅为了这个目的，你就不得不创建两个分开的项目。而且，即使你大费周折地这么做了，也无法改变构建过程需要适应工具，而不是工具去适应构建过程的事实。

也许在现有的解决方案中，你只遇到一部分问题。通常，你需要牺牲非功能性的需求来为企业级自动化领域建模。但是，还是别忍受这些缺点了——让我们看看 Gradle 是如何解决这些问题的。

### 2.1.1 Java 构建工具的演变

让我们看看这些年构建工具是如何演变的。正如第 1 章所讨论的，有两个工具统领着 Java 项目的构建：Ant 和 Maven。经过这么多年，这两个工具都有大步提高和扩展的特性集。虽然它们都非常流行而且变成行业标准，但是却有一个弱点：构建逻辑必须用 XML 描述。XML 是非常好的层级数据描述语言，但是对于描述程序流程和构建逻辑却存在不足之处。随着构建脚本复杂度的增加，维护构建代码就成为了噩梦。

Ant 的第一个正式版本是在 2000 年发布的。每一个工作元素（在 Ant 的术语中叫 *target*）可以被组合和重用。多个 *target* 可以被链接，将单个的工作单元组合成一个完整的工作流。例如，你也许有一个 *target* 是 Java 源代码编译，另外一个 *target* 是将 class 文件打包创建 JAR 文件。构建一个 JAR 文件只有在完成代码编译之后才有意义。在 Ant 中，你让打包 JAR 的 *target* 依赖于编译的 *target*。Ant 在如何组织项目结构方面没有给出任何指导。虽然它拥有最大程度的灵活性，但是 Ant 使得每个构建脚本都是唯一的而且很难理解。项目中需要的外部库通常要提交到版本控制

系统中，因为没有高级的机制可以自动地将它们从一个中心位置下载下来。早期的 Ant 版本需要很多的准则以避免重复代码。它的扩展机制很弱。结果就是，复制和粘贴，这样很差的编码实践成为了唯一的选择。为了统一项目布局，企业需要强制推行一些标准。

Maven 1 发布于 2004 年 7 月，它尝试去简化这个过程。它提供了一个标准化的项目和目录结构，以及依赖管理。遗憾的是，定制逻辑太难实现了。如果你想要打破 Maven 的约定，则需要写插件，叫作 *Mojo*，这通常是唯一的解决方案。*Mojo* 这个名字暗示了这是一种直接、简单和迷人的方式来扩展 Maven。但事实上，在 Maven 中写插件是累赘和非常复杂的。

后来，Ant 通过 Apache 的类库 Ivy 引入了依赖管理来追赶 Maven 的脚步，它可以完全和 Ant 集成，声明式地指定项目编译和打包过程中所需要的依赖。Maven 的依赖管理器，和 Ivy 一样，支持解析传递依赖。当我谈到传递依赖时，指的是你指定的依赖自身所需要的类库。一个典型的传递依赖的例子是，XML 解析库 Xerces 需要 XML API 库才能正常工作。Maven 2 发布于 2005 年 10 月，它让约定优于配置的思想更进一步。由多个模块组成的项目可以将模块定义成相互的依赖。

这段时间有很多人在寻找现有构建工具的替代品。我们看到了从使用 XML 到更具表达性和可读性构建语言的转移。Gant 是带有这种思想的构建工具，它是在 Ant 的基础上用 Groovy 写的 DSL。使用 Gant，用户可以将 Groovy 语言的特性与现有的 Ant 知识结合而不需要写任何 XML。即使它不是 Maven 核心项目的一部分，项目 Maven Polyglot 也提出了相似的方法，允许你写自己构建定义逻辑，该逻辑使用 Groovy、Ruby、Scala 或者 Clojure 语言编写在项目对象模型（POM）文件中。

我们正处在应用开发新纪元的开端：多语言编程。今天许多应用都使用了多种编程语言，每一种语言都最适合实现一个特定的问题领域。很常见的一种情况是，使用客户端语言比如 JavaScript 与混合的多种后端语言如 Java、Groovy 和 Scala 进行通信，而这些后端语言进而会调用由 C++ 编写的遗留系统。最重要的是使用正确的工具做正确的事情。尽管结合多种编程语言有很多好处，但是你的构建工具也需要流畅地支持基础设施。JavaScript 需要被合并、最小化和压缩，而你的服务器端和遗留代码则需要被编译、打包和部署。

Gradle 恰好符合这一代的构建工具，满足现代构建工具的许多需求（图 2.1）。它提供了具有表达性的 DSL、约定优于配置的方法和强大的依赖管理。它摒弃了 XML，引入了动态语言 Groovy 来定义构建逻辑。听起来很不错，不是吗？请继续阅读和学习 Gradle 的特性集，以及如何让你的老板买账。



图 2.1 Gradle 结合了其他构建工具的最佳特性

### 2.1.2 为什么应该选择 Gradle

如果你是一个开发者，那么自动化项目就是你日常开发的一部分。难道你就不想把构建代码看作和其他软件代码一样，让它能够被扩展、测试和维护吗？让我们把软件工程搬回到构建中。Gradle 构建脚本是声明式的、可读的，并且清晰地表达它们的意图。用 Groovy 而不是 XML 写代码，挥洒着 Gradle 基于约定构建的哲理，大大地降低构建脚本的大小而且更易读（见图 2.2）。

看到用 Gradle 实现相同的目标所需要编写的代码时确实让人感到惊讶。使用 Gradle 时，你不需要做出妥协。而像 Maven 这样的构建工具提出的项目布局就是“要么我的方式，要么复杂的方式”，Gradle 的 DSL 提供了灵活性去适应非约定项目布局。

#### Gradle 的座右铭

“让不可能成为可能，让可能变得简单，让简单变得优雅”（适当引用 Moshé Feldenkrais）。

不要改变一个正在运行的系统，你说呢？你的团队已经花费大量的时间来建立项目构建代码基础设施。Gradle 并不强迫你完全迁移所有的构建逻辑。它和其他构建工具如 Ant 和 Maven 有非常好的集成，这是 Gradle 优先级列表中的最高优先级。在第 9 章，我们会深入了解 Gradle 的集成特性和可能的迁移策略。

市场似乎注意到了 Gradle。在 Spring 2010 会议上，Gradle 因为最具创新性开源项目被授予 Springy 大奖（<http://www.springsource.org/node/2871>）。ThoughtWorks，一个声誉很好的软件开发咨询公司，会周期性地发布关于新技术、语言和工具的报告——他们称作技术雷达。技术雷达的目的是帮助软件行业的决策者理解发展趋势和他们对市场的影响。在 2013 年 5 月出版的最新报告中（<http://thoughtworks>。

fileburst.com/assets/technology-radar-may-2013.pdf), Gradle 被标记为采纳状态, 说明这项技术应该被行业所采纳。

#### Maven

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

#### Gradle

```
apply plugin: 'java'
group = 'com.mycompany.app'
archivesBaseName = 'my-app'
version = '1.0-SNAPSHOT'

repositories {
    mavenCentral()
}

dependencies {
    testCompile 'junit:junit:4.11'
}
```

图 2.2 对比 Maven 和 Gradle 构建脚本的大小和可读性

### 被 ThoughtWorks 承认

“有两件事情会让你感到与类似于 Ant 和 Maven 这样基于 XML 的构建工具工作时觉得疲劳：太多的尖括号和粗糙的插件架构。虽然语法问题可以通过升级换代来解决，但随着项目变得越来越复杂，插件架构严重地限制了构建工具的优雅成长。我们感觉插件是一种错误的抽象级别，而更喜欢像 Gradle 和 Rake 那样基于语言的构建工具，因为它们提供了更细粒度的抽象和更长期的灵活性。”



Gradle 很早就有采纳者了，甚至在 1.0 版本发布之前。像 Groovy 和 Hibernate 这样流行的开源项目已经完全切换到 Gradle 并作为它们构建的支柱。每一个 Android 项目都使用 Gradle 作为默认的构建系统。Gradle 给商业市场也带来了影响。像 Orbitz、EADS 和 Software AG 这样的公司也使用 Gradle，这里只列出了几个公司。VMware，作为 Spring 和 Grails 背后的公司，对选择 Gradle 做出了巨大的投资。他们的许多软件产品，比如 Spring 框架和 Grails，都是建立在对 Gradle 能够完成交付的信任上的。

## 2.2 Gradle引人注目的特性集

让我们进一步看看 Gradle 那些与竞争对手不同的特性集：引人注目的特性集（见图 2.3）。总结起来，Gradle 是一个为企业准备的构建系统，由具有声明式和表达性的 Groovy 的 DSL 支持。它结合了灵活性和基于约定优于配置思想的扩展性，以及对传统依赖管理的支持。它背后有一个专业的服务公司（Gradleware）和强大的社区参与，Gradle 成为了许多开源项目和企业构建方案的第一选择。

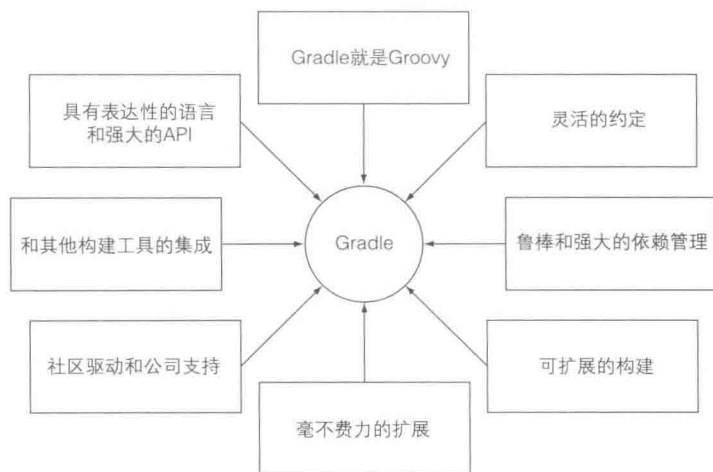


图 2.3 Gradle 引人注目的特性集

### 2.2.1 可表达性的构建语言和底层的 API

在构建脚本中解锁 Gradle 强大特性的关键需要你去探索和应用它的领域模型，如图 2.4 所示。

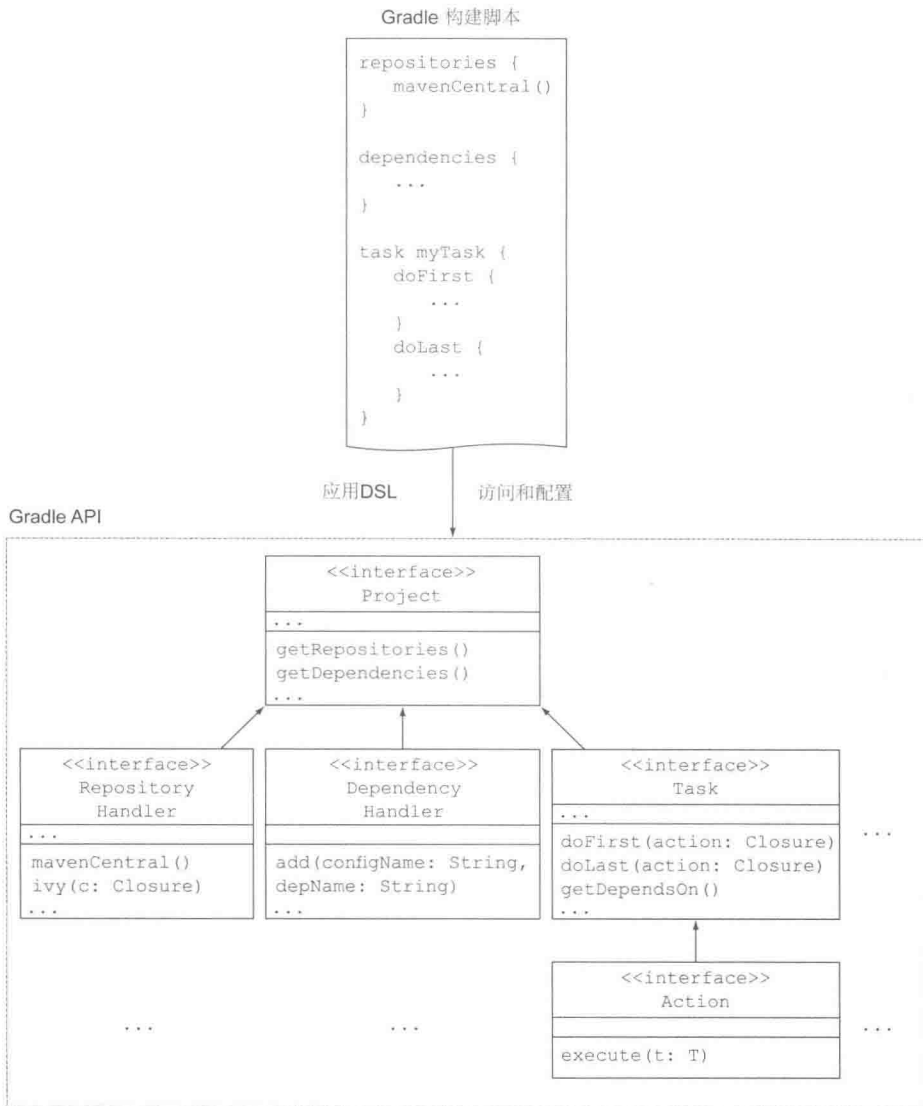


图 2.4 构建脚本应用 Gradle 的 DSL 和对 Gradle API 的访问

正如你在图中所看到的，一个构建脚本直接映射到 Gradle API 中 `Project` 类型的一个实例。相应的，在构建脚本中 `dependencies` 配置块触发 `project` 实例的 `dependencies()` 方法。和 Java 世界的大部分 API 相似，可以在 Gradle 的网站 <http://www.gradle.org/docs/current/javadoc/index.html> 看到 HTML 版本的 Javadoc 文档。谁会知道？实际上你是在和代码打交道。在完全不知情的情况下，在内存中生成了一个代表构建逻辑的对象。在第 4 章中，我们会探索许多 Gradle 的 API 类，以及它

们是如何在构建脚本中表现的。

在 Gradle 脚本中每一个元素都有一个与 Java 类一对一的映射。然而，某些元素被 Groovy 的语法糖衣包装着。在许多情况下 Groovy 化的类相对于 Java，代码更加简洁，并且允许使用如闭包这样的新语言特性。

Gradle 不可能知道满足企业级构建的所有需求。通过暴露钩子（hook）到生命周期阶段，Gradle 允许监控和配置构建脚本的执行行为。假设你有一个非常独特的需求，当单元测试失败时，需要发送一封邮件给开发团队。你想要发送邮件的方式（例如，通过 SMTP 或者第三方邮件服务提供商）和收件人列表需要在构建中指定。其他使用 Gradle 的构建也许对这个特性完全没有兴趣。通过写一个定制测试监听器，在测试执行的生命周期事件完成后通知它，你就可以轻松地将这个特性加入到构建中。

通过暴露用 Groovy 语言实现的 DSL，Gradle 为它的建模建立了一个词库。当处理复杂的领域问题时，比如构建软件，Gradle 就成为一种强有力的工具，可以用一种通用的语言去表达构建逻辑。让我们来看一些例子。最常见的构建就是执行一个工作单元。Gradle 将这个工作单元描述成任务（task）。Gradle 的标准 DSL 部分就是能够确切地定义任务，从编译到打包 Java 源代码。这是一种用自身词汇构建 Java 项目的语言，且这种语言与其他环境无关。

另一个例子就是你能够表达对外部库的依赖，这是构建工具需要解决的常见问题。Gradle 的开箱即用特性提供了两个配置块，允许你定义依赖以及远程仓库。如果标准的 DSL 元素不能满足你的需求，则可以通过 Gradle 的扩展机制引入你自己的词汇。

刚开始听起来可能会有点云里雾里，但是一旦你跨过学习该构建语言的第一道障碍，创建可维护和声明式的构建就非常简单了。一个比较好的起点是 Gradle 构建语言指南 <http://www.gradle.org/docs/current/dsl/index.html>。Gradle 的 DSL 是可扩展的。你可能想要改变现有任务的行为或者添加自己的术语来表述业务领域。Gradle 提供了许多选项让你做这件事。

### 2.2.2 Gradle 就是 Groovy

像 Ant 和 Maven 这样杰出的构建工具使用 XML 来定义它们的构建逻辑。我们都知道，XML 很容易读和写，但是如果内容太多，就不容易维护了。XML 并不具有很强的表达性。这使得它很难定义复杂的定制逻辑。Gradle 采用一种不同的方式。Gradle 的 DSL 是由 Groovy 实现的，它提供了基于 Java 的语法糖。结果就是产生了一种具有可读性和表达性的构建语言。所有你写的脚本都是 Groovy。能够用编程语言来表达你需要的构建是 Gradle 的一大亮点。你不必成为 Groovy 的专家才能开始

写脚本。因为 Groovy 是在 Java 的基础上实现的, 你可以通过尝试使用 Groovy 的语言特性来逐步迁移。你甚至可以完全用纯 Java 代码来编写定制逻辑——Gradle 完全不担心。久经沙场的 Groovy 会确保你使用 Groovy 比使用 Java 在效率上提升好几个数量级。一本非常好的参考书是 *Groovy in Action, Second Edition*, Dirk König 等著 (Maning, 2009), 对于 Groovy 的初学者, 请看附录 B。

### 2.2.3 灵活的约定

Gradle 最主要的思想之一就是针对你的项目给予引导和有意义的默认值。Gradle 中的每个 Java 项目都确切地知道源代码和测试类文件的位置, 知道如何编译代码, 运行单元测试, 生成 Javadoc 报告, 以及发布代码。所有这些任务都完全集成到了构建生命周期中。如果你坚持使用约定, 那么只需要一点配置改变。事实上, 你的构建脚本只需要一行。真的! 你想要了解更多关于使用 Gradle 构建 Java 项目的内容吗? 没有问题——我们在第 3 章中讲解了这部分内容。图 2.5 举例说明了 Gradle 是如何给 Java 项目引入约定和生命周期任务的。

构建生命周期任务

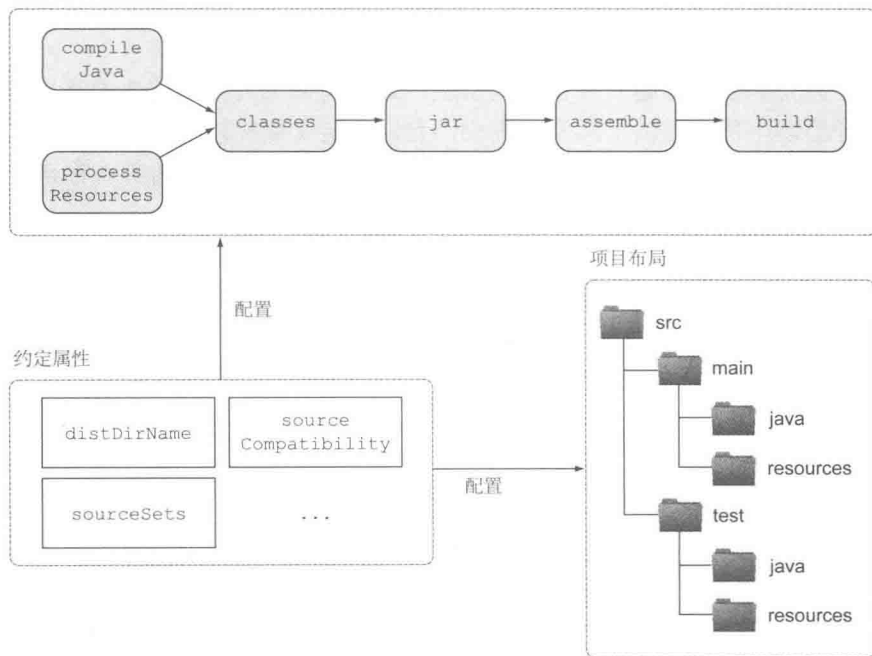


图 2.5 在 Gradle 中, Java 项目的构建基于约定并带有有意义的默认值。改变默认值非常的简单, 通过约定属性即可改变

对于一个 Java 项目，Gradle 已经提供了默认的有意义的任务。例如，你可以编译 Java 产品源代码，运行测试和组装 JAR 文件。每个 Java 项目都以一个标准的目录布局开始。它定义了在哪里可以找到源代码、资源文件和测试代码。可以通过约定属性来改变它们的默认值。

同样的思想也可以应用到像 Scala、Groovy、Web 等项目中。Gradle 称这个概念为约定构建。构建脚本开发人员并不需要知道它是怎么工作的。相反，你能够完全集中精力到什么需要配置上。Gradle 的约定和 Maven 提供的约定相似，但不会让你感觉到被限制。Maven 非常武断，它建议一个工程只包含一个 Java 源代码目录且只产生一个 JAR 文件。这对于许多企业级项目不一定合适。Gradle 允许你轻松地打破约定。与这个现象对立的是，Ant 不会给你任何关于如何组织构建脚本方面的指导，以确保最大程度的灵活性。Gradle 则折中选择，既提供约定，又给予你改变约定的能力。Szczepan Faber，Gradle 的核心工程师之一，在他的博客中这样写道：“Gradle 是不武断工具上的一个武断的框架。”（*Monkey Island*，“武断与否”，2012 年 6 月 2 日，<http://monkeyisland.pl/2012/06/02/opinionated-or-not/>。）

## 2.2.4 鲁棒和强大的依赖管理

软件项目通常不是自包含的。在通常情况下，你的应用代码需要使用第三方类库来解决特殊问题。如果已经有了 Hibernate，为什么你还会想要给持久性框架重造轮子呢？在一个组织内部，你可能是其他团队所实现组件或模块的消费者。外部依赖可以通过仓库获得，而且仓库类型高度依赖于你公司的偏爱。选项可以从纯的文件系统到完全成熟的企业仓库。外部依赖可能还存在对其他类库或资源的引用。我们称为传递性依赖。

Gradle 提供了一个基础设施去管理解析、获取和存储依赖的复杂性。一旦它们被下载并存储到本地缓存中，你的项目就可以使用了。企业构建的一个关键需求是可重现性。回想第 1 章中的 Tom 和 Joe 的故事。你还记得上一次你的同事说的“但是在我的机器上是工作的”吗？构建在不同的机器上产生相同的结果，不受本地缓存内容的影响。像 Ivy 和 Maven 这样的依赖管理器，在它们当前的实现中也不能完全保证可重现性。为什么？当一个依赖被下载并存储在本地缓存中时，它还不算是待构建工件的源。在某些情况下仓库因为项目而改变，缓存的依赖被认为是已解析过的，即使工件的内容可能有一点点不同。最坏的情况是，它会导致构建失败，而且很难调试。对于 Ivy 另一个常见问题是依赖的快照版本，处于开发阶段的工件，约定的名字是 -SNAPSHOT，不会在本地缓存中更新，即便它在仓库中已经改变和标记为改变。还有更多的情况说明当前的解决方案是有缺陷的。Gradle 提供了它自己的可配置的、可靠的和有效的依赖管理方案。我们会在第 5 章中进一步学习。

大型的企业级项目通常是由多个不同功能的模块组成的。在 Gradle 的世界里，每个子模块都被当作一个项目，里面会定义对外部库或者其他模块的依赖。此外，每个子模块都可以独立运行。Gradle 帮你找到哪个子项目的依赖需要重新构建，而不需要将子项目的工件存储到本地缓存中。

## 2.2.5 可扩展的构建

对于某些公司，大型的项目可能拥有上百个模块。构建和测试少量代码的改变会消耗很多时间。从个人经验中你可能知道运行清理任务来删除老的 class 文件和资源文件就证明了这一点。构建工具不知道找出改变的内容和它们的依赖，这常常让你很受伤。你需要的工具应该是足够聪明的，知道只重新构建项目中改变的部分。Gradle 支持通过指定任务的输入和输出进行增量性构建。它准确地找出哪些任务需要跳过，哪些需要构建或者部分构建。同样的思想也应用到多模块项目中，叫作部分构建。因为你的构建清晰地定义了子模块之间的依赖关系，Gradle 会负责重新构建需要的部分。不再是默认执行 clean 任务！

自动化单元测试、集成测试和功能测试是构建过程的一部分。将只需要短时间运行的测试和那些需要准备资源和外部依赖的测试分离是有道理的。Gradle 支持测试的并行执行。这个特性是完全可配置的，并且确保你确实正在利用处理器的内核。优点不止这些。在之后的版本中，Gradle 还将支持在多台机器上执行分布式测试。实在不好意思，这里要告诉你，构建时让你刷 Twitter 的日子已经过去了。

开发人员在开发过程中会多次运行构建。那意味着每次都启动一个新的 Gradle 进程，载入所有的内部依赖和运行构建逻辑。你会注意到，在实际执行脚本之前会有几秒钟的等待时间。要提高启动的效率，Gradle 可以以守护进程模式运行。实际上，Gradle 命令会 fork 出一个守护进程，它不仅会执行你的构建，而且会持续地在后台运行。后续的构建调用会交给这个守护进程以避免启动时的消耗。这样，你就会看到一个更快速的初始化构建执行。

## 2.2.6 轻松的可扩展性

大部分企业构建都不一样，它们也不会解决相同的问题。一旦完成了建立基本构建脚本的初始化阶段，你就会开始实现定制逻辑。Gradle 不会对于如何实现代码给出任何建议。相反，根据你的具体用例，它会提供给你不同的选择。最简单的实现定制逻辑的方式是实现一个任务。任务可以直接在构建脚本中定义而不需要特殊的配置。如果你觉得太过复杂，你也许想要找一种可以在类定义中写定制化逻辑的方法，这样可以让维护和编写代码更简单。如果你想要在多个构建或者项目中分享可重用代码，插件是最好的方式。它是 Gradle 最强大的扩展机制，插件可以让你完

全访问 Gradle 的 API，而且可以像任何其他软件一样，编写、测试和发布。写一个插件非常的简单，完全不需要一大堆额外的描述符。

## 2.2.7 和其他构建工具集成

如果能够和其他现有的构建工具集成，难道不是相当节省时间吗？Gradle 与它的前辈 Ant、Maven 和 Ivy 可以做很好的集成，如图 2.6 所示。

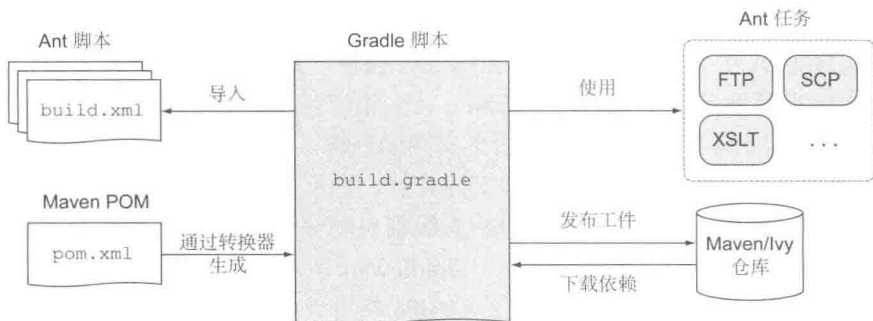


图 2.6 Gradle 提供了和其他构建工具的深入集成，以及对 Ant 或 Maven 构建进行逐渐迁移的支持

如果你是 Ant 的使用者，Gradle 并不强制你完全迁移构建基础设施。相反，它允许你导入现有的构建逻辑并重用标准的 Ant 任务。Gradle 构建和 Maven 及 Ivy 仓库 100% 兼容。你可以从中获取依赖，也可以发布工件。对已有的 Maven 构建，Gradle 提供了一个转换器可以将 Maven 构建逻辑转换为 Gradle 的构建脚本。

现有的 Ant 脚本可以无缝地导入到 Gradle 构建中，而且就像你使用其他外部 Gradle 脚本一样使用它。在运行时，Ant 的 target 直接映射到 Gradle 的 task。Gradle 装载了 Ant 的类库并将一个叫作 AntBuilder 的帮助类暴露到你的脚本中，而且是完全嵌入到 Gradle 的 DSL 中的。它仍然和 Ant 的 XML 相似，但是没有了难看的括号。Ant 用户会感觉到宾至如归，因为他们不需要立刻转换为 Gradle 的语法。从 Ant 到 Gradle 的迁移也是不需要太伤脑筋的。你可以通过重用现有的 Ant 逻辑，在感受 Gradle 特性的同时，小步地迁移。

Gradle 旨在与 Maven 集成时能够做到与 Ant 相同的程度。但在写这本书的时候，它还没有做到。长期而言，Maven 的 POM 和插件都会被看作 Gradle 原生的一部分。Maven 和 Ivy 仓库已经成为今天构建基础设施的一个重要部分。想象一下，一个没有 Maven Central 帮助你获取项目依赖的世界。从仓库中获取依赖只是一部分功能；发布构件也是很重要的。只需要一点点的配置，Gradle 就能将项目的工件上传提供给公司内部或者公共仓库。

## 2.2.8 社区和公司的推动

Gradle 基于 Apache License 2.0, 是免费使用的。在 2008 年 4 月第一次发布之后, 一个充满生气的社区围绕它快速形成。在过去的 7 年中, 开源软件开发者为 Gradle 核心代码做出了主要的贡献。将代码放在 GitHub 上证明是正确的。只要代码的核心提交者对需要提交的代码做一个仔细的审查, 代码就可以提交了。如果你是 Maven 这样构建工具的使用者, 那么可能习惯于广泛的可重用插件。除了 Gradle 中标准的插件, Gradle 社区几乎每天都会发布新的功能。通读本书, 你会看到许多 Gradle 自带的标准插件。附录 A 中给出了更多标准和第三方插件。每个社区主导的软件项目都需要一个论坛来帮助及时回答问题。Gradle 社区论坛是 <http://forums.gradle.org/gradle>。你可以确信你的问题一定会在当天得到有用的回答。

Gradleware 是 Gradle 背后的技术服务和支持公司。它不仅提供了关于 Gradle 的专业建议, 它的目标更是指向了更大范围的企业级别自动化咨询。这个公司由在该领域有非常丰富经验的工程师支撑着。最近, Gradleware 开始开放免费的在线研讨会以激起新来人员的兴趣, 以及加深有经验的 Gradle 使用者的知识。

## 2.2.9 锦上添花：额外的特性

难道你不讨厌给不同的项目安装新的运行时环境? Gradle 包装器是救星! 它允许你在任何需要运行构建的机器上从一个指定的仓库下载和安装一个 Gradle 运行时的新拷贝。这个过程是在第一次构建执行时自动触发的。包装器对于给一个发布团队分享你的构建或者在持续集成服务器上运行构建是非常有用的。

Gradle 也装载了一个丰富的命令行接口。使用命令行选项, 你可以控制所有的东西, 从指定日志级别, 到排除测试, 再到显示帮助信息。这没有什么特别的, 其他工具也提供了。只是某些特性比较突出。Gradle 允许命令是驼峰形式的缩写。举例来说, 一个名字是 `runMyAwesomeTask` 的命令可以 `rMAT` 的缩写形式调用。很方便, 是不是? 虽然本书中展示的大部分例子都是在 shell 中运行命令, 但是记住 Gradle 也提供了开箱即用的图形用户界面。

## 2.3 更大的场景：持续交付

能够构建源代码仅仅是软件交付过程中的一个方面。更重要的是, 你想要将产品发布到产品环境以创造业务价值。在这个过程中, 你想要运行测试, 构建发布产品, 分析代码以保证质量, 也可能要指定一个目标环境并部署。

自动化的整个过程有太多好处了。最重要的是, 手动交付软件是很慢的、易出错而且让人神经紧张。我确信我们中的每个人都会因为一个部署出错要整夜待命。



随着敏捷开发方法的出现，开发团队能够更快地交付软件。2～3 周的发布周期变得很平常。像 Etsy 和 Flickr 等一些组织甚至每天多次部署代码到产品环境中！最佳情况是，你希望将软件发布到指定环境只需要简单地按一个按钮。像自动化测试、持续集成和部署这样的实践已经进入到持续交付的一般概念中了。

在本书中，我们会了解 Gradle 是如何帮助项目实现由构建到部署的。它会帮助你自动化实现持续交付需要的所有任务，从编译代码到部署，或者调用外部工具来帮助实现这个过程。如果想要深入了解持续交付和所有相关方面的内容，我推荐 Jez Humble 和 David Farley 所著的 *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* (Addison Wesley, 2010)。

### 2.3.1 从构建到部署自动化项目

持续交付引入了部署管道的概念，也叫作构建管道。部署管道是一种将软件从版本控制部署到产品环境过程的技术实现。这个过程是由多个阶段组成的，如图 2.7 所示。

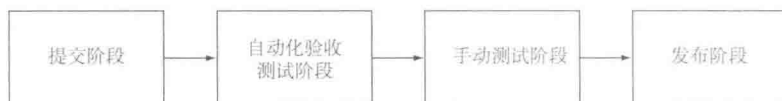


图 2.7 部署管道阶段

- 提交阶段：报告项目技术健康程度。这个阶段的主要利益相关者是开发团队，因为它提供了关于破坏原有功能代码的反馈和帮助找到“代码坏味道”。这个阶段的工作是编译源代码，运行测试，执行代码分析和准备发布。
- 自动化验收测试阶段：通过运行自动化测试判断功能性和非功能性需求是否得到满足。
- 手动测试阶段：验证系统在实际的测试环境中可用。这个阶段通常会包含 QA 人员在用户故事和用例级别验证需求。
- 发布阶段：要么以打包方式将软件交付给终端用户，要么将软件部署到产品环境。

让我们看看部署管道的哪些阶段能够从项目自动化中受益。很明显手动测试阶段不在进一步讨论的范围内，因为它仅仅包含手动的任务。本书主要集中在在提交和自动化验收测试阶段中使用 Gradle 上。我们要关注的具体 task 有：

- 编译源码
- 运行单元测试和集成测试
- 执行静态代码分析和产生测试覆盖率报告

- 构建待发布产品
- 准备目标环境
- 部署产品
- 执行冒烟和自动化功能测试

图 2.8 展示了每个阶段中任务的顺序。然而没有硬性规定会阻止你跳过某些具体的任务，它只是推荐你遵循这个顺序。例如，你可以编译源码，构建待发布产品，然后部署到目标环境，中间不运行任何测试或者静态代码分析。然而，这么做会增加未发现的代码缺陷和产生差质量代码的风险。

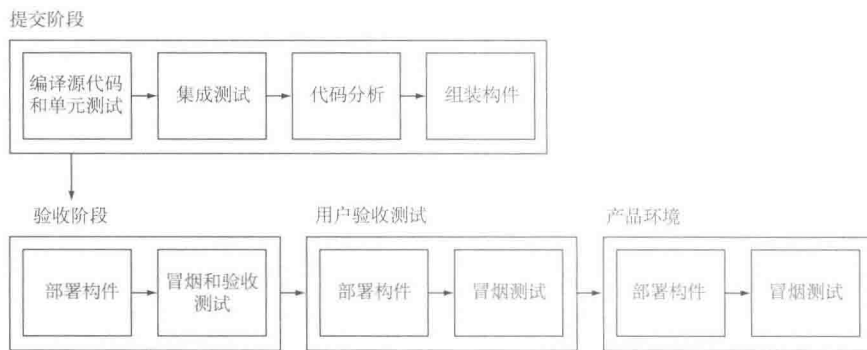


图 2.8 在构建管道阶段执行的任务

像技术基础设施准备、自动化部署、冒烟测试这样的内容也可以放在发布阶段。实际上，在产品环境中应用这些技术远比在一个可控制的测试环境中复杂。在产品环境中，你也许要与集群和分布式服务器设施打交道，零故障发布展示和自动回滚前一个版本。

要涵盖这些进阶内容会超出本书讨论的范围。不过，网络上有一些关于部署管理工具很棒的例子，你也许会想要看看，例如 Asgard，一个基于 Web 的云管理和部署工具，是由 Netflix (<https://github.com/Netflix/asgard>) 构建和使用的。纯理论内容介绍够了——该开始试着安装 Gradle，构建第一个项目了。在第 3 章中，我们会进一步了解如何用 Gradle 实现和运行一个复杂的 Java 项目。

## 2.4 安装 Gradle

首先，确保你已经安装了 JDK 1.5 或以上版本。虽然有些操作系统预装了 Java，但是还是要确保你安装了一个有效的版本。检查 JDK 版本，运行 `java -version` 命令。

开始使用 Gradle 很简单。直接从 Gradle 主页 <http://gradle.org/downloads> 下载

发布版。作为初学者，选择包含文档和样例源码的压缩文件比较好。将下载文件解压缩到某个目录下。在 shell 命令中引用 Gradle 运行时，你需要创建一个环境变量 `GRADLE_HOME` 并且将二进制文件加入到 shell 执行路径下：

- *Mac OS X 和 \*nix*：要确保 Gradle 在 shell 中可用，将下面两行命令添加到初始化脚本中（比如，`~/.profile`）。这些命令假设你将 Gradle 安装在了 `/opt/gradle` 目录下：

```
export GRADLE_HOME=/opt/gradle world
export PATH=$PATH:$GRADLE_HOME/bin
```

- *Windows*：在环境变量对话框中，定义环境变量 `GRADLE_HOME`，更新路径设置（图 2.9）。

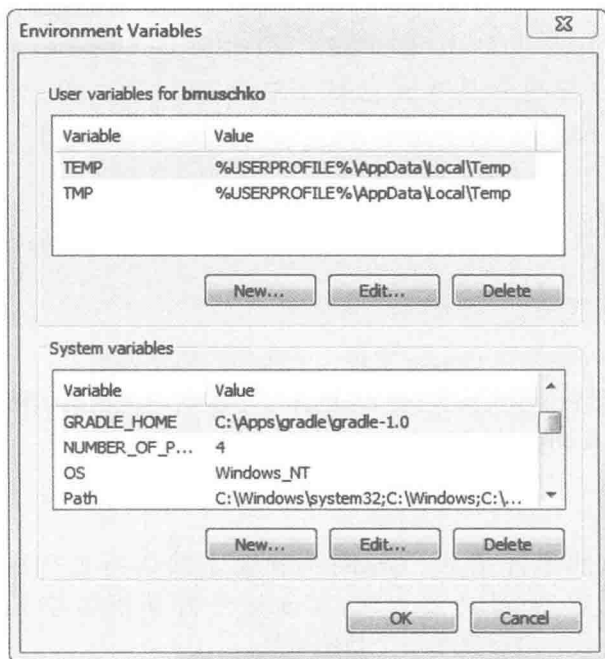


图 2.9 在 Windows 中设置环境变量

验证 Gradle 已经被正确安装。检查 Gradle 的运行时版本，运行 `gradle -v` 命令。你会看到关于 Gradle、JVM 和操作系统的元信息。下面的例子显示了成功安装 Gradle 1.7 的信息。

```
$ gradle -v
```

```
-----  
Gradle 1.7  
-----
```

```
Build time:    2013-08-06 11:19:56 UTC  
Build number: none  
Revision:     9a7199efaf72c620b33f9767874f0ebced135d83  
  
Groovy:       1.8.6  
Ant:          Apache Ant(TM) version 1.8.4 compiled on May 22 2012  
Ivy:          2.2.0  
JVM:         1.6.0_51 (Apple Inc. 20.51-b01-457)  
OS:          Mac OS X 10.8.4 x86_64
```

### 设置 Gradle 的 JVM 选项信息

和其他 Java 应用一样, Gradle 同样使用由环境变量 `JAVA_OPTS` 设置的 JVM 选项。如果你想要传递特定参数给 Gradle 运行时, 则使用环境变量 `GRADLE_OPTS`。假设你想要增加默认的最大堆内存到 1GB, 则可以这样设置:

```
GRADLE_OPTS="-Xmx1024m"
```

更好的方式是将变量添加到 `$GRADLE_HOME/bin` 目录下的 Gradle 启动脚本中。

现在一切都准备好了, 你可以开始使用 Gradle 实现一个简单的构建脚本了。大部分流行的 IDE 都提供了 Gradle 插件, 你所需要的只是选择一个喜欢的编辑器。第 10 章会讨论 Gradle 插件对 IntelliJ、Eclipse 和 NetBeans 这样的 IDE 的支持。

## 2.5 开始使用 Gradle

每个 Gradle 构建都是以一个脚本开始的。Gradle 构建脚本默认的名字是 `build.gradle`。当在 shell 中执行 `gradle` 命令时, Gradle 会去寻找名字是 `build.gradle` 的文件。如果找不到, 就会显示一个帮助信息。

让我们在 Gradle 中实现经典的“Hello world!”例子。首先, 你需要创建一个名字为 `build.gradle` 的文件。在文件中, 定义一个独立的原子性工作。在 Gradle 的词汇中, 叫作 `task` (任务)。在这个例子中, `task` 叫作 `helloWorld`。要打印信息“Hello world!”, 需要使用 Gradle 的通用语言 Groovy, 将 `println` 命令添加到 `task` 的 `action` (动作) `doLast` 中。Groovy 中的 `Println` 方法更简短, 它相当于 Java 中的 `System.out.println` 方法。

```
task helloWorld {  
    doLast {  
        println 'Hello world!'  
    }  
}
```

运行该 task :

```
$ gradle -q helloWorld
Hello world!
```

正如所期望的,当运行脚本时,你会看到输出“Hello world!”。通过 `-q` 定义可选命令行选项 `quiet`,告诉 Gradle 只输出该 task 相关的信息。

在完全不需要了解 Gradle 的情况下,你已经开始使用 Gradle 的 DSL 了。task 和 action 是这门语言重要的元素。名字为 `doLast` 的 action 几乎自表达了它的含义。它是 task 执行的最后一个 action。Gradle 还允许使用一种更精简的方式来指定相同的逻辑。使用左移符号 `<<` 来简单地代表 `doLast`。下面的代码片段展示了第一个例子的修改版本:

```
task helloWorld << {
    println 'Hello world!'
}
```

打印“Hello world!”就这么简单。在下面的清单中展示了一些更高级的特性。通过一点点练习,让我们增强对 Gradle 的信心。跟我一起念:摇滚吧,Gradle!

### 清单 2.1 动态任务定义和任务链

```
task startSession << {
    chant()
}

def chant() {
    ant.echo(message: 'Repeat after me...')
}

3.times {
    task "yayGradle$it" << {
        println 'Gradle rocks'
    }
}

yayGradle0.dependsOn startSession
yayGradle2.dependsOn yayGradle1, yayGradle0
task groupTherapy(dependsOn: yayGradle2)
```

① 隐含对 Ant 任务的使用

② 动态任务的定义

③ 任务依赖

一开始你可能没有注意到,这个清单中隐藏了很多高级特性。代码中引入了关键字 `dependsOn` 来说明 task 之间的依赖 ③。Gradle 会确保被依赖的 task 总会在定义该依赖的 task 之前执行。实际上, `dependsOn` 是 task 的一个方法。第 4 章中会涵盖一些内部 task,这里不用太深入细节。

我们之前谈到的一个特性是 Gradle 和 Ant 有很好的集成 ①。因为拥有对 Groovy 语言特性的完全访问权,你还可以使用 `chant()` 方法来打印消息。这个方

法可以非常方便地在 task 中调用。每个脚本都带有一个 `ant` 属性，它赋予了直接访问 Ant 任务的能力。在这个例子中，你可以使用 Ant 的任务 `echo` 打印出“Repeat after me”信息。

Gradle 提供的一个漂亮的特性是定义动态 task，这意味着可以在运行时指定它们的名称。你的脚本在循环 ❷ 中使用 Groovy 在 `java.lang.Number` 中扩展的 `times` 方法创建 3 个新的 task。Groovy 自动地暴露一个隐式变量 `it` 来指定循环迭代的次数。你使用这个计数器来构建 task 的名字。对于第一轮迭代，task 可以叫作 `yayGradle0`。

现在运行 `gradle groupTherapy`，你会看到下面的输出：

```
$ gradle groupTherapy
:startSession
[ant:echo] Repeat after me...
:yayGradle0
Gradle rocks
:yayGradle1
Gradle rocks
:yayGradle2
Gradle rocks
:groupTherapy
```

如图 2.10 所示，Gradle 以正确的顺序执行这些 task。你也许已经注意到例子中省去了 `quiet` 命令行选项，这表示运行该 task 时会得到更多的信息。



图 2.10 任务依赖图

多亏了 `group therapy` 这个 task，你不用担心 Gradle 只是另一个不能满足需要的构建工具。在下一章中，你会从一个完整的 Java 应用的角度体验 Gradle 的核心概念。现在，让我们来习惯一下 Gradle 的命令行。

## 2.6 使用 Gradle 的命令行

在前面的小节中，在命令行执行了 `helloWorld` 和 `groupTherapy` 任务，这将会成为整本书中大部分例子的运行方式。虽然使用 IDE 对于新手更方便，但是对 Gradle 的命令行选项和帮助任务的深入理解会让你在以后的日子里工作得更加高效。

### 2.6.1 列出项目中所有可用的 task

在上一小节中展示了如何通过 `gradle` 命令运行一个指定的 `task`。要运行一个 `task`，需要知道它的具体名字。如果可以在不用看源代码的情况下知道所有可用的 `task`，是不是很棒？Gradle 提供了一个叫作 `tasks` 的帮助任务来帮助你查看构建脚本和显示每个可以使用的 `task`，包括描述该 `task` 作用的信息。以 `quiet` 模式运行 `gradle tasks` 的输出结果如下：

```
$ gradle -q tasks

-----
All tasks runnable from root project
-----

Build Setup tasks
-----
setupBuild - Initializes a new Gradle build. [incubating]
wrapper - Generates Gradle wrapper files. [incubating]

Help tasks
-----
dependencies - Displays the dependencies of root project 'grouptherapy'.
dependencyInsight - Displays the insight into a specific dependency in root
  => project 'grouptherapy'.
help - Displays a help message
projects - Displays the sub-projects of root project 'grouptherapy'.
properties - Displays the properties of root project 'grouptherapy'.
tasks - Displays the tasks runnable from root project 'grouptherapy' (some of
  => the displayed tasks may belong to subprojects).

Other tasks
-----
groupTherapy
```

构建的 `setup` 任务帮助你初始化 Gradle 的构建(比如, 生成 `build.gradle` 文件)

1 帮助任务组列出了任务的名字和它们的描述

2 没有分类的任务没有被分配到一个任务组中

3 没有描述信息的任务没有自表达性；在第 3 章中你会学习如何给任务添加描述信息

To see all tasks and more detail, run with `--all`.

关于输出，有几点需要说明。Gradle 提供了任务组的概念，你可以把它看作是多个 `task` 的集群。每个构建脚本都会默认暴露 `Help tasks` 任务组 ❶。如果某个 `task` 不属于一个任务组，那么它就会显示在 `Other tasks` ❷中。在这里可以找到 `groupTherapy` ❸。在第 4 章中，我们会学习如何将 `task` 添加到一个任务组中。

你可能会想在构建脚本中定义的其他 `task` 去哪了。在输出的底部，你会看到一条提示，要获得关于 `task` 的更多信息，请使用 `--all` 选项。运行之后可以获得更多信息：

```
$ gradle -q tasks --all
```

```
-----
All tasks runnable from root project
-----
```

```
Build Setup tasks
-----
```

```
setupBuild - Initializes a new Gradle build. [incubating]
```

```
wrapper - Generates Gradle wrapper files. [incubating]
```

```
Help tasks
-----
```

```
dependencies - Displays the dependencies of root project 'grouptherapy'.
```

```
help - Displays a help message
```

```
projects - Displays the sub-projects of root project 'grouptherapy'.
```

```
properties - Displays the properties of root project 'grouptherapy'.
```

```
tasks - Displays the tasks runnable from root project 'grouptherapy' (some of
➡ the displayed tasks may belong to subprojects).
```

```
Other tasks
-----
```

```
groupTherapy
  startSession
  yayGradle0
  yayGradle1
  yayGradle2
```

❶ 依赖关系图的根 task



以执行顺序用缩进的方式列出依赖任务的名字

--all 选项是决定 task 执行顺序的好办法。要减少杂音，Gradle 很聪明地隐藏了作为根 task 的依赖任务 ❶。为了有更好的可读性，依赖任务以缩进方式有序地显示在根 task 的下面。

## 2.6.2 任务执行

在之前的例子中，通过给命令 gradle 添加一个参数来执行指定的任务。Gradle 的命令行实现会确保任务以及它们的所有依赖都被执行。你也可以在命令行中通过参数定义，一次执行多个任务。运行 gradle yayGradle0 groupTherapy 命令将会先执行 yayGradle0 任务，然后执行 groupTherapy 任务。

任务通常只会执行一次，无论它们是在命令行中指定的还是作为另一个任务的依赖。让我们看看输出是怎么样的：



```
$ gradle yayGradle0 groupTherapy
:startSession
[ant:echo] Repeat after me...
:yayGradle0
Gradle rocks
:yayGradle1
Gradle rocks
:yayGradle2
Gradle rocks
:groupTherapy
```

毫无意外。和运行 `gradle groupTherapy` 时的输出一样。正确的顺序被保存并且每个任务只执行一次。

### 任务名字缩写

Gradle 最有用的特性之一就是能够以驼峰式的缩写在命令行上运行任务。如果你想要以缩写方式运行之前的例子，则只需要输入 `gradle yG0 gT`。当你要运行名字特别长的任务或者多个任务参数时，这个特性非常有用。记住，任务名字的缩写必须是唯一的，Gradle 才能找到相应的任务。比如下面的场景：

```
task groupTherapy << {
    ...
}

task generateTests << {
    ...
}
```

在定义有 `groupTherapy` 和 `generateTests` 任务的 Gradle 中，使用 `gT` 缩写运行任务会导致下面的错误：

```
$ gradle yG0 gT

FAILURE: Could not determine which tasks to execute.

* What went wrong:
Task 'gT' is ambiguous in root project 'grouptherapy'. Candidates are:
➤ 'generateTests', 'groupTherapy'.

* Try:
Run gradle tasks to get a list of available tasks.

BUILD FAILED
```

### 在执行时排除一个任务

有时候你想要在构建运行时排除一个指定任务。Gradle 提供了一个命令行选项 `-x` 来实现。假设你想要排除任务 `yayGradle0`：

```
$ gradle groupTherapy -x yayGradle0
:yayGradle1
Gradle rocks
:yayGradle2
Gradle rocks
:groupTherapy
```

Gradle 排除了任务 `yayGradle0` 和它的依赖任务 `startSession`, 这个概念叫作智能排除。现在, 你应该越来越熟悉 Gradle 的命令行了, 让我们探索更多有用的功能吧。

### 2.6.3 命令行选项

在本小节中, 我们会探索最重要的通用命令行选项、控制构建脚本日志级别的标志位, 以及给项目提供参数的方法。gradle 命令允许你同时定义一个或者多个选项。假设你想要将日志级别改变到 `INFO`, 则可以使用 `-i` 选项, 或者如果想要打印出在执行期间发生错误时的堆栈踪迹信息, 则可以使用 `-s` 选项。比如: `gradle groupTherapy -is` 或者 `gradle groupTherapy -i -s`。正如你所看到的, 可以轻松地组合多个命令行选项。通过 `-h` 选项, 你可以看到所有可用的选项, 或者参考本书的附录 A。这里不会介绍所有的可用选项, 但是最重要的命令行选项如下:

- `-?, -h, --help`: 打印出所有可用的命令行选项, 包含描述信息。
- `-b, --build-file`: Gradle 构建脚本的默认命名约定是 `build.gradle`。使用这个命令行选项可以执行一个特定名字的构建脚本 (比如, `gradle -b test.gradle`)。
- `--offline`: 通常, 构建中声明的依赖必须在离线仓库中存在才可用。如果这些依赖在本地缓存中没有, 那么运行在一个没有网络连接环境中的构建都会失败。使用这个选项可以让你以离线模式运行构建, 仅仅在本地缓存中检查依赖是否存在。

#### 参数选项

- `-D, --system-prop`: Gradle 是以一个 JVM 进程运行的。和所有的 Java 进程一样, 你可以提供一个系统参数, 就像 `-Dmyprop=myvalue` 这样。
- `-P, --project-prop`: 项目参数是构建脚本中可用的变量。你可以使用这个选项直接向构建脚本中传入参数 (比如, `-Pmyprop=myvalue`)。

#### 日志选项

- `-i, --info`: 在默认设置中, Gradle 构建不会提供大量的输出信息。通过这个选项可以将 Gradle 的日志级别改变到 `INFO` 以获得更多信息。如果你想要知道构建中发生了什么, 这个选项非常有用。

- `-s, --stacktrace`: 如果构建在运行中出现错误, 你会想要知道错误是从哪里开始的。`-s` 选项在有异常抛出时会打印出简短的堆栈跟踪信息, 帮助你进行调试。
- `-q, --quiet`: 减少构建出错时打印出来的错误日志信息。

### 帮助任务

- `tasks`: 显示项目中所有可运行的 `task`, 包括它们的描述信息。项目中应用的插件可能会提供一些额外的 `task`。
- `properties`: 显示出项目中所有可用的属性。某些属性是由 Gradle 的 `project` 对象提供的, `project` 对象是一个构建的本质表现形式。其他的属性都是用户定义的, 要么来自于属性文件或者命令行选项, 要么是直接在构建脚本中定义的。

## 2.6.4 Gradle 守护进程

当 Gradle 成为日常工作的一部分时, 你会发现需要重复地运行构建。如果你在一个 Web 应用上工作, 就更是如此。当你改变一个类时, 就需要重新构建, 然后部署到服务器, 刷新页面看浏览器上的变化。许多开发人员喜欢测试驱动的开发方式。为了持续地获得关于代码质量的反馈, 他们会不断地运行单元测试以更早地发现代码缺陷。不管哪种方式, 你都会发现效率很重要。每次初始化一个构建时, JVM 都要启动一次, Gradle 的依赖要载入到类加载器中, 还要建立项目对象模型。这个过程需要花上好几秒钟。Gradle 守护进程是这个问题的救星!

守护进程以后台进程方式运行 Gradle。一旦启动, `gradle` 命令就会在后续的构建中重用之前创建的守护进程, 以避免启动时造成的开销。让我们回到之前的构建脚本例子中。在我的机器上, 成功地完成运行 `groupTherapy` 任务要花上 3 秒钟。我们希望提高启动和执行的效率。在命令行中启动 Gradle 守护进程很简单: 在运行 `gradle` 命令时加上 `--daemon` 选项。你可能会注意到在启动守护进程时也花费一点点额外的时间。为了验证守护进程在运行, 你可以查看系统进程列表:

- *Mac OS X 和 \*nix*: 在 shell 中运行命令 `ps | grep gradle`。
- *Windows*: 通过 “Ctrl+Shift+Esc” 打开任务管理器, 点击 “进程” 标签。

后续触发的 `gradle` 命令都会重用守护进程。现在来试一把, 运行 `gradle groupTherapy --daemon`。哇, 一秒钟就完成启动和执行了。记住, 守护进程只会被创建一次, 即便你在命令行中加了 `--daemon` 选项。守护进程会在 3 小时空闲时间之后自动过期。任何时候你都可以选择在执行构建时不使用守护进程, 只需要添加命令行选项 `--no-daemon` 即可。要手动停止守护进程, 可以执行 `gradle --stop` 命令。简而言之, 这就是 Gradle 守护进程。要更深入地了解所有的可配

置选项和更复杂的内容，可以参考 Gradle 在线文档 [http://gradle.org/docs/current/userguide/gradle\\_daemon.html](http://gradle.org/docs/current/userguide/gradle_daemon.html)。

## 2.7 总结

现有的构建工具不能满足今天行业的需求。在竞争对手最优思想的基础上进一步提高，在不以牺牲灵活性和可描述性为代价的条件下，Gradle 提供了基于约定的构建方法、可靠的依赖管理和多项目构建的支持。

在本章中，我们了解了在持续交付的构建管道中，Gradle 是如何在每个阶段发挥作用的。在本书中，我们通过提供实际的例子讲解了持续交付的每个阶段。

接下来，你实际感受了 Gradle 的强大特性。安装了 Gradle，写了第一个简单的构建脚本，并成功执行。通过实现一个更为复杂的构建脚本，发现使用 Gradle 的 DSL 去定义任务的依赖是多么简单。了解到 Gradle 命令行的机制和让 Gradle 更为高效的命令行选项。Gradle 提供了许多不同的命令行开关来改变运行时的行为，将参数传递给项目，改变日志级别。我们也探索了如果需要持续地执行 task，应该如何运行 Gradle 来节省时间，比如在测试驱动开发过程中。

在第3章中，我会向你展示如何用 Gradle 构建一个完整的 Web 项目。从一个简单、独立的 Java 应用开始，你会通过添加 Web 组件来扩展代码，使用 Gradle 的容器内部署支持特性来有效地实现解决方案。不仅如此，我还打算向你展示如何增强一个 Web 构建让它做好企业级准备，让构建可以在不需要安装 Gradle 运行时的情况下，在多台机器上切换。

# 通过范例学习构建 Gradle项目

## 本章涵盖

- 使用 Gradle 构建全栈 Java 项目
- 有效地练习 Web 应用开发
- 定制默认规约适应自定义需求
- 使用 Gradle 包装器

第 2 章介绍了 Gradle 的特性集并和其他 JVM 构建工具进行了对比。介绍了一些简单的例子，让你对该工具的具有表达性的构建语言有了第一印象。通过运行你的第一个构建脚本，看到了在命令行上有效运行构建是多么的简单。现在是时候通过构建真实的 Java 项目来巩固新学到的知识了。

当开始开发一个全新的应用时，Java 并没有指导你如何标准化项目结构。你可能会问自己在哪里放源文件、配置文件和类库文件。如果想要将应用程序代码与测试代码分开，应该怎么办？Gradle 通过引入预定义的项目布局为类似 Java 的项目提供了基于约定的构建方法。被那种有不同目录结构的遗留应用困住了？没问题！Gradle 可以根据你的需要调整约定。

在本章中我们通过构建一个 Java 项目来探索 Gradle 标准化范例的内部工作方

式，并且学习如何将它修改成非约定方式的用例。下一步，你会通过 Web 组件来扩展应用并引入有效的工具来提升开发速度。然后，我们会学习 Gradle 的包装器来丰富这一章，包装器可以让你在不需要安装 Gradle 运行时的情况下，创建可以迁移和可重现的构建。

## 3.1 介绍学习案例

这一小节通过介绍一个简单的应用来说明 Gradle 的使用：一个 To Do 应用程序。贯穿着整本书，我们会在构建管道的每个阶段，应用学习的内容来说明 Gradle 的特性。这个用例以一个没有界面的纯 Java 应用开始，通过控制台简单地控制输入。在本章之后，你会通过添加组件来扩展本应用以学习更多的高级概念。

To Do 应用程序会帮你了解 Gradle 的能力。你会学习如何使用 Gradle 的标准插件来启动、配置和运行项目。在本章的最后，你会基本理解 Gradle 是如何帮助构建基于 Web 的 Java 项目的。

### 3.1.1 To Do 应用程序

今天的世界是很忙碌的。许多人需要同时负责很多事情，无论是工作还是生活。通常，你会发现自己不知所措，失去控制。保持有组织性地集中在高优先级的事情上的关键就是有一个妥善维护的 to-do 列表。当然，你可以在纸上写下你的计划，但是如果可以随时随地地查看这些计划不是更方便吗？互联网几乎是无所不在，通过手机或者公共访问点都可以访问。你将构建一个自己的基于 Web 的好看的 To Do 应用，如图 3.1 所示。

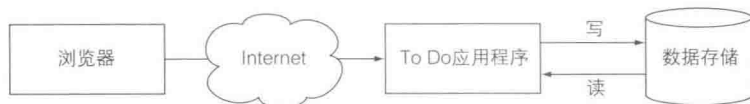


图 3.1 To Do 应用可以通过互联网访问并管理数据仓库中的执行项目

### 3.1.2 任务管理用例

现在你知道自己的最终目标了，让我们一起来确定一下完成它需要的用例吧。每个任务管理系统都是由一个有序的执行项目或者任务列表组成的。每个任务都有一个标题来表示需要完成的事情。任务可以在列表中添加和删除，以及标记为活动或者完成状态。列表也应该允许修改任务标题，以便让描述更加准确。当一个任务发生改变时应该自动地持久化到数据库中。

为了让任务有顺序，应该包含一个选项可以根据它们的状态是活动还是已完成来筛选。现在，我们就以这个最小的特性为起点。图 3.2 显示了用户界面的截图。



图 3.2 基于 Web 的 To Do 应用的用户界面和执行项目

让我们从用户界面退一步回来，从最基础开始构建这个应用。在该应用的第一个版本中，你会通过实现由命令行控制的基本功能来打下基础。下一小节，我们会集中在应用组件和它们之间的交互上。

### 3.1.3 检查组件交互

我们发现一个 To Do 应用要实现典型的增删改查（CRUD）功能。对于数据持久化，你需要一个领域模型来表示数据。你需要创建一个名为 `ToDoItem` 的 Java 类，这是一个 POJO。为了保证第一轮迭代的解决方案尽可能的简单，我们不会引入数据库来存储数据。相反，我们会把它放在内存里，这样实现相对简单。实现持久化协议的类叫作 `InMemoryToDoRepository`。缺点是应用程序关闭后，数据就消失了。本书的后面，我们会展示如何以一种更好的方式去实现持久化。

每个独立运行的 Java 程序都需要实现一个主类（Main Class），作为应用程序的入口。你的主类叫作 `ToDoApp`，它会一直运行直到用户决定退出。你会向用户展现一个命令菜单，通过这个菜单，用户可以通过输入字符触发一个指定操作来管理他们的 to-do 列表。每个操作命令都映射到一个枚举类，叫作 `CommandLineInput`。类 `CommandLineInputHandler` 代表着用户交互与命令执行的黏合剂。图 3.3 说明了用户想要列出所有任务所需要执行的操作。

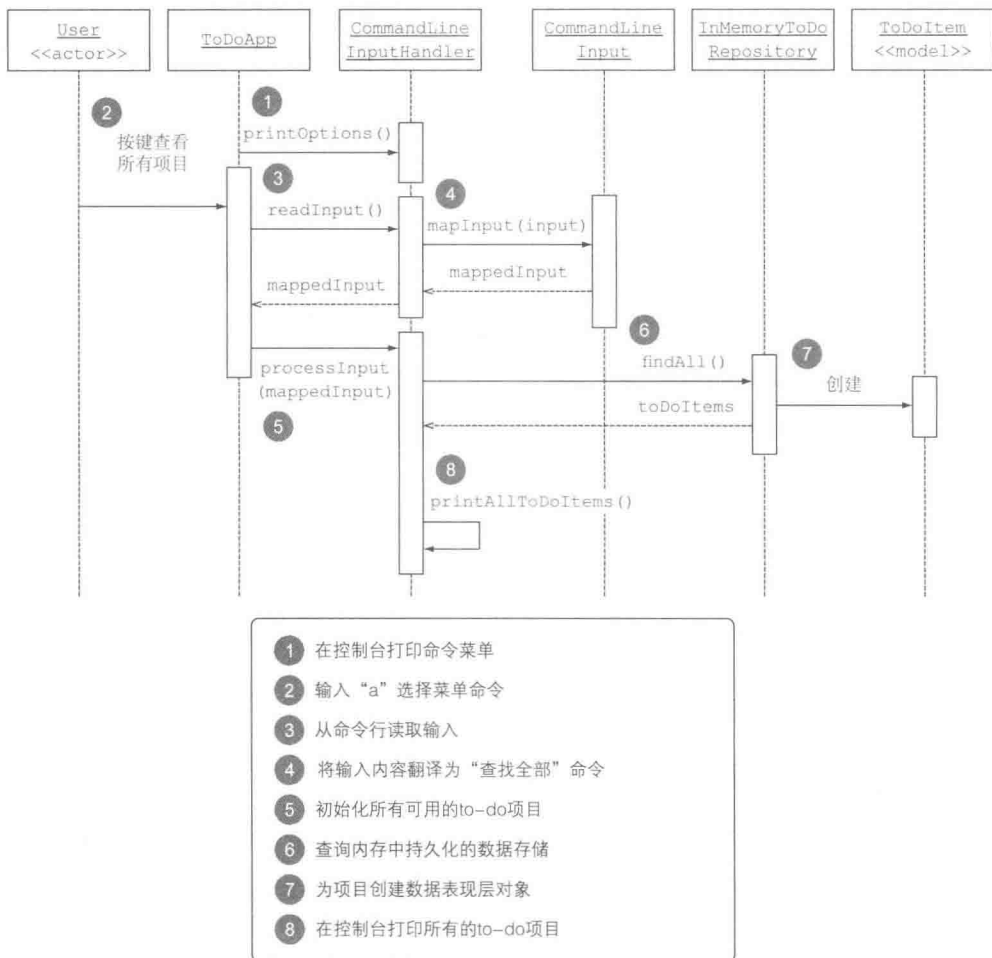


图 3.3 按照有序图列出所有的任务

现在你可以准备开始实现应用功能了。下一小节，我们会从代码入手。

### 3.1.4 构建应用功能

上一小节，我们确定了类、它们的作用以及它们之间的交互。现在是时候开始实现它们了。首先，我们看看 to-do 执行项目的模型。

#### To Do 模型类

每一个 `ToDoItem` 类的实例都代表着 to-do 列表中的一个执行项目。属性 `id` 是每一个项目的唯一标识，也让你能够在内存中存储和读取它。另外，领域模型



类暴露了 name 和 completed 属性。为了使代码简洁，getter 和 setter 方法以及 compareTo 方法从下面的代码片段中省去了：

```
package com.manning.gia.todo.model;

public class ToDoItem implements Comparable<ToDoItem> {
    private Long id;
    private String name;
    private boolean completed;

    (...)
}
```

现在让我们来看看仓库类是怎么读 / 写该领域模型的。

### 领域模型在内存中的持久化

在内存中存储数据很方便，而且可以简化实现。在本书的后面，你也许会想要提供更复杂的实现像数据库或文件系统一样。为了能够方便地切换实现，你需要创建一个接口，ToDoRepository，如下面的清单所示。

#### 清单 3.1 仓库类接口

```
package com.manning.gia.todo.repository;

import com.manning.gia.todo.model.ToDoItem;
import java.util.Collection;

public interface ToDoRepository {
    List<ToDoItem> findAll();
    ToDoItem findById(Long id);
    Long insert(ToDoItem toDoItem);
    void update(ToDoItem toDoItem);
    void delete(ToDoItem toDoItem);
}
```

接口定义了你想要的所有增删改查操作。你可以查找所有存在的 to-do 项目，通过 ID 查询，插入一个新的项目，更新或删除它们。接下来，你需要创建一个可扩展的、线程安全的接口实现类。下面的清单显示了一个叫作 InMemoryToDoRepository 的类，它将所有的 to-do 项目放在 ConcurrentHashMap 的实例中。

#### 清单 3.2 to-do 项目的持久化类

```
package com.manning.gia.todo.repository;

import com.manning.gia.todo.model.ToDoItem;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;
import java.util.concurrent.atomic.AtomicLong;
```

```

public class InMemoryToDoRepository implements ToDoRepository {
    private AtomicLong currentId = new AtomicLong();
    private ConcurrentMap<Long, ToDoItem> toDos
    = new ConcurrentHashMap<Long, ToDoItem>();
    @Override
    public List<ToDoItem> findAll() {
        List<ToDoItem> toDoItems = new ArrayList<ToDoItem>(toDos.values());
        Collections.sort(toDoItems);
        return toDoItems;
    }
    @Override
    public ToDoItem findById(Long id) {
        return toDos.get(id);
    }
    @Override
    public Long insert(ToDoItem toDoItem) {
        Long id = currentId.incrementAndGet();
        toDoItem.setId(id);
        toDos.putIfAbsent(id, toDoItem);
        return id;
    }
    @Override
    public void update(ToDoItem toDoItem) {
        toDos.replace(toDoItem.getId(), toDoItem);
    }
    @Override
    public void delete(ToDoItem toDoItem) {
        toDos.remove(toDoItem.getId());
    }
}

```

线程安全的标识符序列号

用来保护 to-do 项目的有效内存结构

通过标识符对 to-do 项目进行排序

如果 to-do 项目还不存在，就只将它放到 Map 中

如果存在于 Map 中，就替换掉该项目

如果存在于 Map 中，就移除该项目

到目前为止，你看到了 to-do 项目的数据结构，以及在内存中存储和获取数据的实现。要启动这个 Java 程序，你需要创建一个主类（Main Class）。

### 应用程序入口

ToDoApp 类会将应用的选项打印到控制台，读取用户的输入，将单个字符翻译成一个命令对象并处理，如下面清单所示。

列表 3.3 实现主类

```

package com.manning.gia.todo;

import com.manning.gia.todo.utils.CommandLineInput;
import com.manning.gia.todo.utils.CommandLineInputHandler;

public class ToDoApp {
    public static final char DEFAULT_INPUT = '\u0000';

    public static void main(String args[]) {
        CommandLineInputHandler commandLineInputHandler = new

```

```

    ➡ CommandLineInputHandler();
    char command = DEFAULT_INPUT;

    while(CommandLineInput.EXIT.getShortCmd() != command) {
        commandLineInputHandler.printOptions();
        String input = commandLineInputHandler.readInput();
        char[] inputChars = input.length() == 1 ? input.toCharArray() : new
        ➡ char[] { DEFAULT_INPUT };
        command = inputChars[0];
        CommandLineInput commandLineInput =
        ➡ CommandLineInput.getCommandLineInputForInput(command);
        commandLineInputHandler.processInput(commandLineInput);
    }
}

```

只要用户输入  
exit 命令，应  
用就停止运行

执行的  
CRUD  
命令

输入的  
单个字  
符和对  
应命令  
对象的  
映射

至此，我们讨论了应用程序的组件和它们在某一个特殊用例下的交互：查找一个用户的所有 to-do 项目。清单 3.3 粗略展示了每个组件的职责和它们是如何在内部工作的。如果你不理解这里展示的每一个实现细节，也不用担心。在这里更重要的是自动化这个项目。在本章的后继内容中，我们会仔细了解如何用 Gradle 配置这个项目，编译源代码，组装 JAR 文件，以及运行该程序。是时候让 Gradle 登场了。

## 3.2 构建Java项目

在上一小节中，我们确认了要写一个独立运行的 To Do 应用所需要的 Java 类。要组装一个可执行程序，需要编译源代码，class 文件需要打包成 JAR 文件。JDK 提供了像 javac 和 jar 这样的工具来帮助实现这些任务。除非你是受虐狂，否则你绝对不会想要一次次手动执行这些任务。

Gradle 插件作为驱动力能够自动化这些任务。插件通过引入特定领域的约定和任务来扩展你的项目。Java 插件是 Gradle 自身装载的一个插件。Java 插件提供的基本功能远比源代码编译和打包多。它为你的项目建立了一个标准的项目布局，并确保有意义、有顺序地执行任务。现在，为你的项目创建一个构建脚本并使用 Java 插件。

### 3.2.1 使用 Java 插件

在第 1 章里，你知道了每个 Gradle 项目都是以创建名字为 build.gradle 的文件开始的。创建这个文件，然后像下面这样告诉它要使用 Java 插件：

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

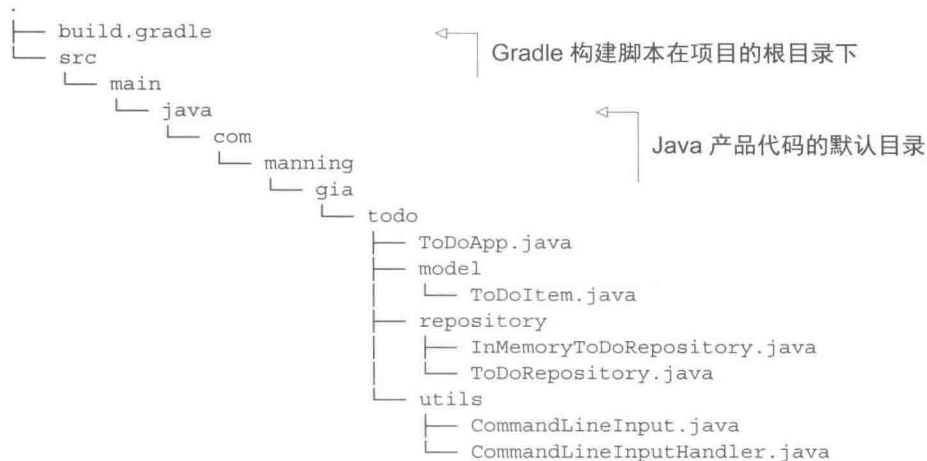
一行代码足够构建你的 Java 代码，但是 Gradle 怎么知道去哪里找源文件呢？Java 插件引入的约定之一就是源代码的位置。在默认情况下，插件会到 src/

main/java 目录下查找。你需要将所有关于 To Do 应用程序的类文件放在正确的目录下。

### 自动化项目生成

如果不需要你手动创建源文件目录，是不是很棒？Maven 有一个概念叫作 *project archetypes*，它是一个插件，用来从现有的模板生成项目结构。遗憾的是，在写这本书的时候，这个功能还没有成为 Gradle 的核心特性。Gradle 模板插件是由 Gradle 社区提出的，为的就是解决这个问题。该插件可以在 <https://github.com/townsfolk/gradle-templates> 上找到。build setup 插件实现了初始化 Gradle 项目的功能，你甚至不需要构建脚本就可以使用。这个插件允许你生成项目文件（和之后你会了解的其他相关文件）。要生成 Gradle 构建脚本，只需要在命令行执行 `gradle setupBuild` 即可。

当创建源文件时，记住所使用的包 `com.manning.gia.todo`，将其作为子文件夹拷贝到源文件目录的根目录下。在创建完构建脚本和将源代码移动到正确的位置之后，你的项目结构应该是这样的：



### 构建项目

你可以开始构建项目了。Java 插件提供的一个任务叫作 `build`。这个 `build` 任务会以正确的顺序编译你的源代码，运行测试，组装 JAR 文件。运行 `gradle build` 命令，你应该得到类似于下面的输出：

```

$ gradle build
:compileJava                                ← 编译 Java 产品代码
:processResources UP-TO-DATE
:classes
:jar                                         ← 组装 JAR 文件
:assemble
:compileTestJava UP-TO-DATE                 ← 编译 Java 测试代码
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:test                                       ← 运行单元测试
:check
:build

```

每一行输出都代表着 Java 插件提供的一个可执行任务。你也许注意到某些任务被标记为 UP-TO-DATE 消息。这意味着这个任务被跳过了。Gradle 的增量式构建支持自动鉴别不需要被运行的任务。特别是在大型的企业级项目中，这个特性是节省时间的好帮手。在第 4 章中，你会学习如何将这个概念应用到你自己的任务中。在命令行的输出中，你可以看到被跳过的具体任务：`compileTestJava` 和 `testClasses`。因为你在默认目录 `src/test/java` 下提供了单元测试，那么 Gradle 非常乐意去执行。如果你想要了解如何给应用程序写测试并将它们集成到构建中，请看第 7 章。下面是构建之后项目的结构：

```

.
├── build
│   ├── classes
│   │   └── main
│   │       └── com
│   │           └── manning
│   │               └── gia
│   │                   └── todo
│   │                       ├── ToDoApp.class
│   │                       ├── model
│   │                       │   └── ToDoItem.class
│   │                       ├── repository
│   │                       │   ├── InMemoryToDoRepository.class
│   │                       │   └── ToDoRepository.class
│   │                       └── utils
│   │                           ├── CommandLineInput.class
│   │                           ├── CommandLineInputHandler$1.class
│   │                           └── CommandLineInputHandler.class
│   ├── dependency-cache
│   ├── libs
│   │   └── todo-app.jar
│   ├── reports
│   │   └── tests
│   │       ├── base-style.css
│   │       ├── css3-pie-1.0beta3.htc
│   │       ├── index.html
│   │       ├── report.js
│   │       └── style.css
└──

```

默认目录包含有已编译的 Java 的 class 文件

组装包含 Java 的 class 文件和代码清单的 JAR 文件；文件的名字继承自项目的目录名



在项目的根目录下，你会看到有一个 `build` 目录，里面包含了构建运行的所有输出，包括 `class` 文件、测试报告和 `JAR` 文件，还有一些像清单（manifest）一样的对构建有用的临时文件。如果你之前使用过 `Maven`，它有一个标准的输出目录叫作 `target`，结构和 `build` 目录类似。构建输出目录的名字是可配置的属性。你已经看到根据约定构建 `Java` 项目是多么的简单，完全不需要额外的配置。`JAR` 文件生成在 `build/libs` 目录下，而且是可以运行的。理解 `JAR` 文件的名称继承自项目名称是很重要的。在这个例子中，只要你不重新配置，项目中使用的目录名称就是 `todo-app`。让我们来运行一下 `To Do` 应用。

## 运行项目

运行一个 `Java` 应用程序是非常简单的。就目前而言，你只需要在项目的根目录下使用 `JDK` 的 `java` 命令：

```
$ java -cp build/classes/main com.manning.gia.todo.ToDoApp
```

```
--- To Do Application ---
Please make a choice:
(a)ll items
(f)ind a specific item
(i)nsert a new item
(u)pdate an existing item
(d)elele an existing item
(e)xit
>
```

`Java` 程序启动后，会打印出所有可用的 `to-do` 项目，等待你从命令行输入。

## 对独立运行 `Java` 程序的支持

`Gradle` 能够进一步简化构建一个独立运行的 `Java` 程序。有一个标准的 `Gradle` 扩展值得一提，它就是 `application` 插件。该插件提供了能够绑定和简化运行应用的任务。

就是这样——使用 `Gradle`，你不费吹灰之力就实现和构建了一个 `Java` 应用。只要使用标准约定，你所需要做的只是一行脚本。接下来，我们来看看如何定制基于约定构建的标准。

### 3.2.2 定制你的项目

Java 插件是一个很灵活的框架。它会给项目的许多方面设置有意义的默认值，比如项目结构。如果你看待开发世界的方式不同，Gradle 给予你定制这些约定的选项。如何知道什么是可配置的？有一个好地方，就是 Gradle 构建语言指导 <http://www.gradle.org/docs/current/dsl/>。还记得第 2 章的命令行选项 `properties` 吗？运行 `gradle properties` 会给你一个可配置标准和插件属性的列表，同时还会显示它们的默认值。你可以通过扩展初始构建脚本来定制你的项目。

#### 修改项目和插件属性

在下面的例子中，你将给项目指定一个版本号，并且指定 Java 源代码的兼容性。之前，你通过 `java` 命令运行 To Do 应用。通过 `classpath` 命令行选项 `-cp build/classes/main` 告诉 Java 运行时去哪里找 class。为了能够从 JAR 文件启动应用，清单文件 `MANIFEST.MF` 需要包含信息头 `Main-Class`。下面的清单展示了如何在构建脚本中配置默认值和将头属性添加到 JAR 文件清单中。

清单 3.4 修改属性变量并添加 JAR 文件头

```
version = 0.1
sourceCompatibility = 1.6

jar {
    manifest {
        attributes 'Main-Class': 'com.manning.gia.todo.ToDoApp'
    }
}
```

定义项目版本

设置 Java 版本编译兼容 1.6

将 Main-Class 头添加到 JAR 文件代码清单中

在组装完 JAR 文件之后，你会看到版本号添加到了 JAR 文件的名字中。现在名字是 `todo-app-0.1.jar`，而不是 `todo-app.jar`。现在生成的 JAR 文件包含了主类头属性，你可以通过 `java -jar build/libs/todo-app-0.1.jar` 命令运行应用。接下来，我们会看到如何将现有的项目结构改成旧式（老式）的项目目录结构。

#### 改造遗留项目

很少有企业级软件项目是在一个干净的环境下开始的。和一个遗留系统集成，迁移已有项目的技术栈，或者坚持内部标准或者限制，实在是太常见了。构建工具必须足够灵活，可以通过改变默认配置来适应来自外部的限制。

在本小节中，我们会继续看几个例子，来说明如何定制 To Do 应用。让我们假设你是在一个完全不一样的目录结构下开始这个项目的。你需要把源代码放置在 `src` 目录下，而不是 `src/main/java`。同样的道理，也适用于改变默认的测试代码目录。另外，你想要让 Gradle 将输出结果放置在 `out` 目录下，而不是 `build`。下面清单展示了如何让你的构建适应一个定制的项目结构。

## 清单 3.5 改变项目默认结构

```
sourceSets {  
    main {  
        java {  
            srcDirs = ['src']  
        }  
    }  
    test {  
        java {  
            srcDirs = ['test']  
        }  
    }  
}  
buildDir = 'out'
```

用不同目录的列表代替约定的源代码目录

用不同目录的列表代替约定的测试代码目录

改变项目输出属性（路径）到 out 目录

定制一个构建的关键就是了解潜在的属性和 DSL 元素。接下来，我们来看看如何使用外部库的功能。

### 3.2.3 配置和使用外部依赖

让我们回想一下 `ToDoApp` 类中的 `main` 方法。你写了一些代码去读取用户从控制台的输入，然后将第一个字符翻译成一个 `to-do` 命令。这么做，你需要确保输入的字符串只能是一个字符长度。否则，将 `null` 赋值给它：

```
String input = commandLineInputHandler.readInput();  
char[] inputChars = input.length() == 1 ? input.toCharArray() : new  
char[] { DEFAULT_INPUT };  
command = inputChars[0];
```

我打赌你可以通过重用外部库来实现这一点。最好的选择就是 **Apache Commons Lang** 库的 `CharUtils` 类。它提供了一个叫作 `toChar` 的方法可以仅仅获取 `String` 的第一个字符并转换为 `char` 返回，或者如果是一个空值，就返回一个默认字符。下面这段代码显示了输入字符解析的升级版：

```
import org.apache.commons.lang3.CharUtils;  
  
String input = commandLineInputHandler.readInput();  
command = CharUtils.toChar(input, DEFAULT_INPUT);
```

那么，你应该如何告诉 Gradle 去引用 **Apache Commons Lang** 库？我们来看两个 DSL 配置元素：`repositories` 和 `dependencies`。

#### 定义仓库

在 Java 世界，依赖都是以 JAR 文件的形式发布和使用的。许多类库都可以在仓库中找到，仓库可以是一个文件系统或者一个中心服务器。Gradle 要求你定义至



少一个仓库来使用依赖。为此，你需要使用公共的可访问的仓库 Maven Central：

```
repositories {
    mavenCentral()
}
```

← 配置对 Maven Central 2 仓库 <http://repol.maven.org/maven2> 访问的快捷方式

定义好仓库之后，你就可以声明类库了。让我们看看依赖是如何定义的。

### 定义依赖

一个依赖是通过 `group` 标识符、名字和一个指定版本来确定的。如下面代码片段所示，你会使用类库的 3.1 版本：

```
dependencies {
    compile group: 'org.apache.commons', name: 'commons-lang3', version: '3.1'
}
```

在 Gradle 中，依赖是由 `configuration` 分组的。Java 插件引入的一种 `configuration` 是 `compile`。你可以通过 `configuration` 的名字看出它是给编译源代码使用的。

### 如何查找一个依赖

在 Maven Central 中查找依赖的详细信息是最直接的。在 <http://search.maven.org/> 上，仓库提供了一个使用简单的搜索入口。

### 解析依赖

Gradle 会自动检测到一个新的依赖添加到项目中。如果依赖没有被成功解析，那么就会在下一个需要使用该依赖的任务启动时去下载它——在这个例子中就是 `compileJava` 任务：

```
$ gradle build
:compileJava
Download http://repol.maven.org/maven2/org/apache/commons/
➡ commons-lang3/3.1/commons-lang3-3.1.pom
Download http://repol.maven.org/maven2/org/apache/commons/
➡ commons-parent/22/commons-parent-22.pom
Download http://repol.maven.org/maven2/org/apache/apache/9/
➡ apache-9.pom
Download http://repol.maven.org/maven2/org/apache/commons/
➡ commons-lang3/3.1/commons-lang3-3.1.jar
:processResources UP-TO-DATE
...
:build
```

← 元数据描述它所依赖的类库和工件

← 二进制工件：JAR 文件包含 Apache Commons Lang 的 class 文件

第 5 章会对依赖管理进行更深入的讨论。我知道 To Do 应用程序现在的形式不可能打动你。该是时候给它加些好看的用户界面了。

## 3.3 用Gradle做Web开发

在 Java 世界中，企业级版本（Java EE）的服务器端 Web 组件能够为在 Web 容器或应用服务器上运行的应用提供动态扩展的能力。Servlet 这个名字就已经说明，它为客户提供请求服务，给出响应。它就像 MVC 架构中的控制器。Servlet 的响应由视图组件渲染——JSP 技术。图 3.4 说明了在 Java Web 应用上下文中 MVC 架构模式。

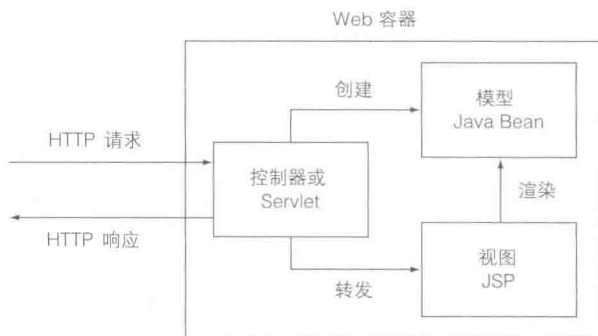


图 3.4 Java EE 提供组件来建立基于 MVC 架构模式的 Web 应用

一个 WAR 文件被用来绑定 Web 组件、编译 class 文件以及其他像部署描述器、HTML、JavaScript 和 CSS 文件这样的资源文件。它们一起组成了 Web 应用程序。为了运行一个 Java Web 应用，WAR 文件需要部署到服务器环境，一个 Web 容器中。

Gradle 提供了开箱即用的插件，用来组装 WAR 文件和将 Web 应用部署到本地 Servlet 容器中。之前，我们看到了如何应用和配置这些插件，你所需要的就是将一个独立运行的 Java 应用变成一个 Web 应用。

### 3.3.1 添加 Web 组件

Java 企业级的开发一直被广泛的 Web 框架占领，比如 Spring MVC 和 Tapestry。Web 框架被设计为抽象标准 Web 组件和减少样本代码。尽管有这些优势，但 Web 框架引入了陡峭的学习曲线，因为它引入了新的概念和 API。为了保证书中例子尽可能简单易懂，我们还是坚持使用标准的 Java 企业级 Web 组件。

在开始看代码之前，让我们来看看添加 Web 组件是如何改变前面章节中已编写的类之间的交互的。你需要创建一个叫作 `ToDoServlet` 的类。它负责接收 HTTP 请求，执行一个映射到某个 URL 的增删改查操作，并将请求转发到一个 JSP 页面。为了给用户带来一个流畅和舒适的体验，你要把 to-do 列表做成一个单页面应用。这意味着你只需要写一个 JSP 页面，并将它命名为 `todo-list.jsp`。该页面知道该如何动态地渲染 to-do 项目列表，并且提供像按钮一样的 UI 元素和链接启动增删

改查操作。图 3.5 显示了在新系统中获取和渲染所有 to-do 项目的流程。

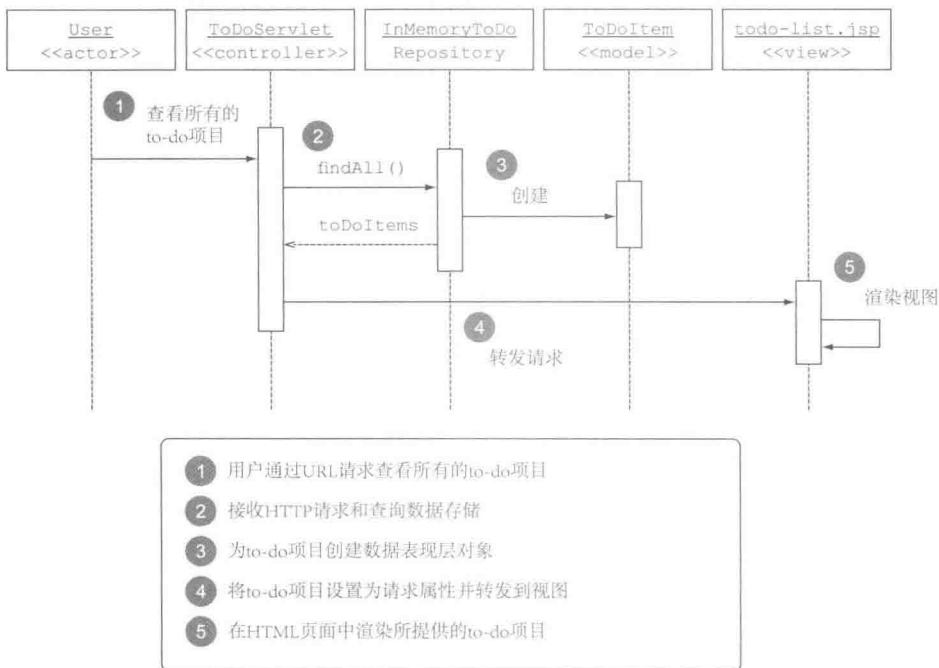


图 3.5 查找所有的 to-do 项目用例：用户通过浏览器发起一个 HTTP 请求，由 Servlet 提供服务，并将结果通过 JSP 渲染出来

正如你在图中所看到的，你可以重用类 `ToDoItem` 来表示领域模型和类 `InMemoryToDoRepository` 来存储数据。这两个类都可以无缝地与控制器和视图组件一起工作。让我们看看控制器是怎么工作的。

### Web 组件控制器

为了让事情变得简单和集中，你需要给所有想要暴露给客户端的 URL 写一个单独的入口点。下面的代码片段展示了 Web 组件控制器最重要的部分，`ToDoServlet` 类：

```
package com.manning.gia.todo.web;

import com.manning.gia.todo.model.ToDoItem;
import com.manning.gia.todo.repository.InMemoryToDoRepository;
import com.manning.gia.todo.repository.ToDoRepository;
import javax.servlet.*;
import java.io.IOException;
import java.util.List;
```

```
public class ToDoServlet extends HttpServlet {
    private ToDoRepository toDoRepository = new InMemoryToDoRepository();

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {
        String servletPath = request.getServletPath();
        String view = processRequest(servletPath, request);
        RequestDispatcher dispatcher = request.getRequestDispatcher(view);
        dispatcher.forward(request, response);
    }

    private String processRequest(String servletPath, HttpServletRequest
        request) {
        if(servletPath.equals("/all")) {
            List<ToDoItem> toDoItems = toDoRepository.findAll();
            request.setAttribute("toDoItems", toDoItems);
            return "/jsp/todo-list.jsp";
        }
        else if(servletPath.equals("/delete")) {
            (...)
        }
        (...)
        return "/all";
    }
}
```

获取请求 URL 的路径；路径以 / 开头

从 Servlet 转发请求到 JSP

为每个映射 URL 实现 CRUD 操作

当输入的请求 URL 不满足任何一种处理方式时，则跳转到 /all

每个进入的请求，你都需要获得它的 Servlet 路径，根据增删改查操作，在 `processRequest` 方法中处理该请求，并通过 `javax.servlet.RequestDispatcher` 将它转发给 JSP 页面 `todo-list.jsp`。

就是这样；你将任务管理程序转换成一个 Web 应用。在例子中，只是接触了代码中最重要的部分。如果需要深入了解，建议你查看完整的源代码。接下来，我们引入 Gradle。

### 3.3.2 使用 War 和 Jetty 插件

Gradle 对构建和运行 Web 应用都提供了扩展性支持。在这一部分，我们会学习两个 Web 应用程序部署插件：War 和 Jetty。War 插件扩展自 Java 插件，为 Web 应用部署和组装 War 包添加了约定与支持。在本地机器上运行一个 Web 应用程序是很简单的，启动快速应用部署（RAD），提供快速启动时间。在理想情况下，它不会要求你安装一个 Web 容器运行时环境。Jetty 是一个流行的轻量级开源 Web 容器，它支持所有的这些特性。它通过将一个 HTTP 模块添加到应用中来提供一个嵌入式实现。Gradle 的 Jetty 插件扩展了 War 插件，为部署一个 Web 应用到嵌入式容器和运行 Web 应用提供了对应的任务。

### 可选的嵌入式容器插件

Jetty 插件非常适合于本地 Web 应用开发。然而，你可能在产品环境中使用不同的 Servlet 容器。为了更早的在软件开发周期中提供最大的运行在不同运行时环境的能力，你需要提供其他的可用嵌入式容器实现。一个不错的选择是第三方 Tomcat 插件。

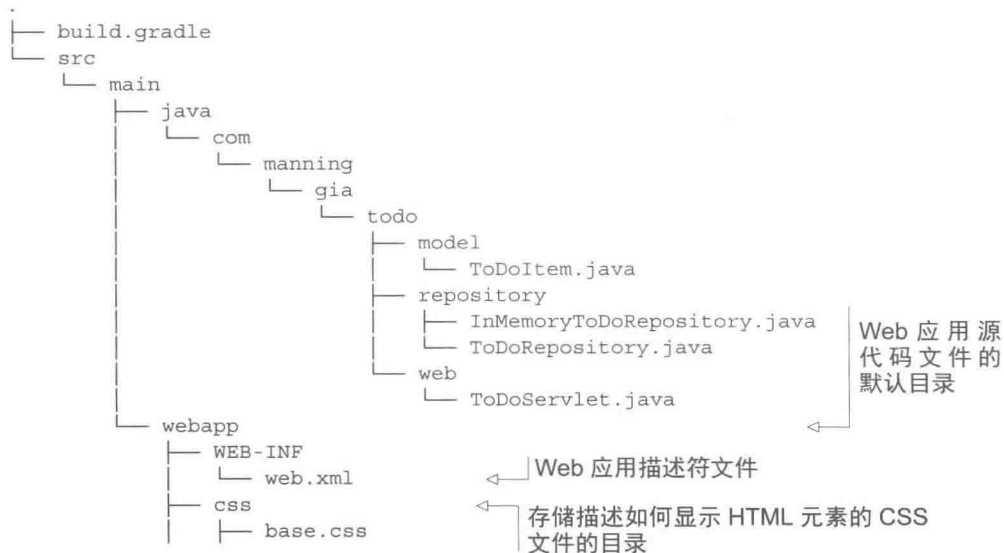
你已经从上一小节了解到该怎么做了。首先需要将插件应用到项目中，并使用默认约定，然后需要定制它们。让我们首先来了解一下 War 插件。

### War 插件

我之前提到过 War 插件扩展自 Java 插件。实际上，这意味着你不需要在构建脚本中应用 Java 插件。它会自动由 War 插件引入。不过即便你也应用了 Java 插件，这也不会对你的项目带来其他影响。应用插件是一个幂等操作，因为某一个指定的插件只会执行一次。当创建 build.gradle 文件时，使用插件只需要像这样：

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

它对于项目到底意味着什么？除了 Java 插件提供了约定外，你的项目也会意识到 Web 应用文件的源代码目录，并且知道如何组装一个 WAR 文件而不是 JAR 文件。Web 应用默认约定的源代码目录是 src/main/webapp。当所有的 Web 资源文件都放在了正确的地方后，你的项目结构应该像这样：





实现 Web 应用所需要的类并不是 Java 标准版本的一部分，例如 `javax.servlet.HttpServlet`。在运行构建之前，你需要确保声明了外部依赖。War 插件引入了两个新的依赖 configuration。Servlet 依赖使用到的 configuration 是 `providedCompile`。它表示该依赖在编译时需要，但是由运行时环境提供。这里的运行时环境是 Jetty。结果就是，被标记为 `provided` 的依赖不会打包到 WAR 文件中。像 JSTL 库这样的依赖，在编译时不需要，但是运行时需要。它们就会成为 WAR 文件的一部分。下面的 `dependencies` 闭包声明应用程序所需要的外部依赖：

```
dependencies {
    providedCompile 'javax.servlet:servlet-api:2.5'
    runtime 'javax.servlet:jstl:1.1.2'
}
```

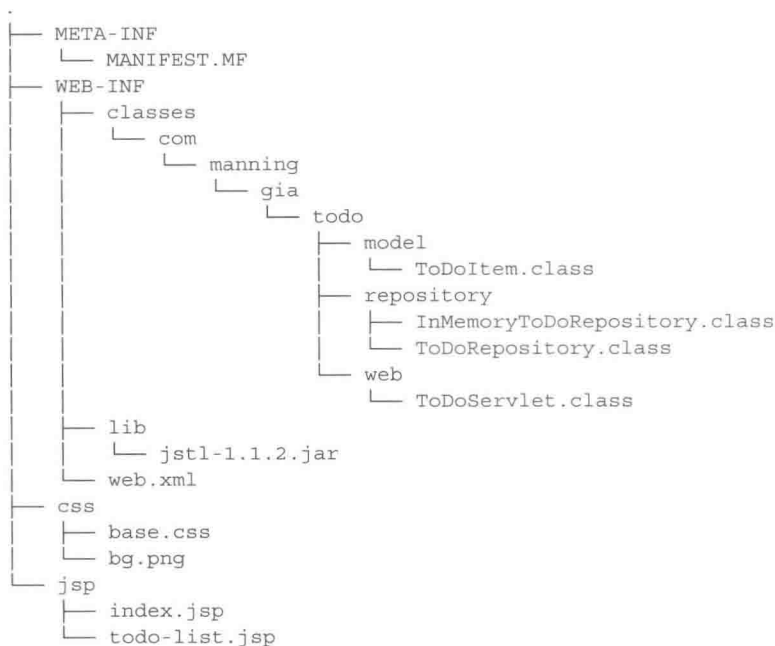
## 构建项目

在 Gradle 中构建一个 Web 应用就和构建一个独立运行的 Java 应用程序一样。运行 `gradle build` 命令后，组装的 WAR 文件可以在 `build/libs` 目录下找到。将一个独立运行的 Java 应用改变成一个 Web 应用，`jar` 任务被 `war` 任务代替，输出结果如下：

```
$ gradle build
:compileJava
:processResources UP-TO-DATE
:classes
:war
:assemble
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:test
:check
:build
```

由 War 插件提供的任务来组装 WAR 文件

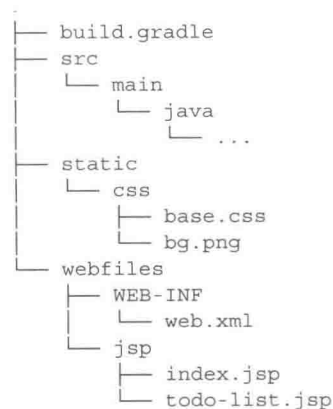
War 插件确保组装的 WAR 文件遵循由 Java EE 规范定义的标准结构。`war` 任务将 Web 应用源代码目录 `src/main/webapp` 的内容原封不动地拷贝到 WAR 文件的根目录。编译的 `class` 文件最终会放置在 `WEB-INF/classes` 下，通过依赖闭包定义的运行时类库会放置在 `WEB-INF/lib` 下。下面显示了在运行命令 `jar tf todo-webapp-0.1.war` 之后 WAR 文件的目录结构：



在默认情况下，WAR 文件的名称会继承自项目的目录名称。即便你的项目没有遵循 Gradle 的标准约定，该插件也依然可以构建 WAR 文件。让我们看一些定制化选项吧。

### 定制 War 插件

你已经了解到适应一个 Java 项目的定制化项目结构是多么简单。对于非约定的 Web 项目布局也是一样。在下面的例子中，我们会假设所有的静态文件都放置在 static 目录下，而且所有 Web 应用的内容都放置在 webfiles 目录下：



下面的代码片段显示了如何配置约定属性。War 插件暴露了 `webAppDirName` 约定属性，默认值是 `src/main/webapp`，重新赋值就可以轻松地切换到 `webfiles`。通过触发 `from` 方法就可以有选择性地需要的目录添加到 WAR 文件中，如下：

```
webAppDirName = 'webfiles'
war {
    from 'static'
}
```



改变 Web 应用的源代码目录

将 css 和 jsp 目录添加到 WAR 文件的根目录下

之前的例子仅仅显示了 War 插件配置选项的一部分。你可以轻松地加入其他的外部 JAR 文件，从一个非标准目录中使用 Web 部署描述符文件 (`web.xml`)，或者将另一个文件放到 `WEB-INF` 目录中。如果你在找某个配置参数，那么查看 War 插件的 DSL 指南是第一选择。

你已经看到如何在一个标准结构或定制化目录布局的 Web 项目中构建 WAR 文件。现在我们来将文件部署到 Servlet 容器中。接下来，你将在本地开发机器上启动 Jetty 运行应用。

### 在嵌入式 Web 容器中运行

一个嵌入式 Servlet 容器并不了解你的应用，直到你提供了 Web 应用程序确切的 classpath 和相关的源代码目录。通常，你需要用编程的方式做这件事。在这里，Jetty 插件帮你做了这件事。因为 War 插件暴露了所有的信息，Jetty 插件可以在运行时访问它们。这是 Gradle 中一个典型的例子，通过 Gradle 的 API，一个插件可以访问另一个插件的配置。在你的构建脚本中，像这样使用插件：

```
apply plugin: 'jetty'
```

运行 Web 应用使用的任务是 `jettyRun`。即使没有 WAR 文件，它也会启动 Jetty 容器。在命令行中运行该任务的输出结果应该类似于下面这样：

```
$ gradle jettyRun
:compileJava
:processResources UP-TO-DATE
:classes
> Building > :jettyRun > Running at http://localhost:8080/todo-webapp-jetty
```

在输出结果的最后一行，插件告诉你 Jetty 所监听的请求 URL。打开你最爱的浏览器，并输入 URL。终于，你可以看到 To Do Web 应用程序。Gradle 会一直让应用程序运行，直到你按“Ctrl+C”键。Jetty 是如何知道在什么端口和上下文中运行应用的呢？还是约定。Jetty 插件运行一个 Web 应用的默认端口是 8080，上下文路



径 `todo-webapp-jetty` 是从项目名字继承的。当然，这些都是可配置的。

### 快速应用开发

每次改变项目代码都需要重新启动容器是非常麻烦和耗时的。Jetty 插件允许你在不启动容器的情况下，改变静态资源和 JSP 文件。另外，还针对 class 文件的改变配置像 Jrebel 这样的字节码交换技术来执行热部署。

### 定制 Jetty 插件

假设你不满意 Jetty 插件提供的默认值。已经有一个应用使用了 8080 端口，而你也讨厌输入那么长的上下文路径。下面的配置可以解决你的问题：

```
jettyRun {  
    httpPort = 9090  
    contextPath = 'todo'  
}
```

太棒了，你成功了。用这个配置启动应用，将暴露 `http://localhost:9090/todo`。Jetty 插件还提供了更多的配置选项。查看插件的 API 文档是最好的选择。它会帮助你理解所有的可用配置选项。

## 3.4 Gradle包装器

你做好任务管理的 Web 应用的原型。向你的同事 Mike 展示，他说他想要加入你的开发团队，给应用添加更多的高级特性让它更上一层楼。代码已经被提交到版本控制系统（VCS）中，所以他把代码下载下来就可以开始工作了。

Mike 从来没有使用过 Gradle，所以他询问你怎么在机器上安装 Gradle 运行时和用什么版本。因为他没有安装过 Gradle，所以他也关心在 Windows 上安装和在 Mac 上安装有什么不同。根据使用其他构建工具的经验，Mike 意识到如果选错构建工具的版本或者运行时环境可能对构建结果带来不好的影响。在本机上运行成功，但是在别的机器上运行失败，实在是太平常了。在煎熬了好几个小时后，他发现原因是运行时版本不兼容。

针对这个问题，Gradle 提供了一个非常方便和实用的解决方案：Gradle 包装器。它是 Gradle 的核心特性，能够让机器在没有安装 Gradle 运行时的情况下运行 Gradle 构建。它也让构建脚本运行在一个指定的 Gradle 版本上。它是通过自动从中心仓库下载 Gradle 运行时，解压和使用来实现的。最终的目标是创造一个独立于系统、系统配置和 Gradle 版本的可靠和可重复的构建。

### 什么时候使用包装器

使用包装器被认为是最佳实践，对每一个 Gradle 项目都是必需的。由包装器支持的 Gradle 脚本非常适合作为自动化发布过程的一部分，比如持续集成和交付。

让我们看看如何为 Mike 和其他想要加入团队的开发者配置包装器。

#### 3.4.1 配置包装器

在项目中配置包装器，你只需要做两件事情：创建一个包装器任务和执行任务生成包装器文件（图 3.6）。

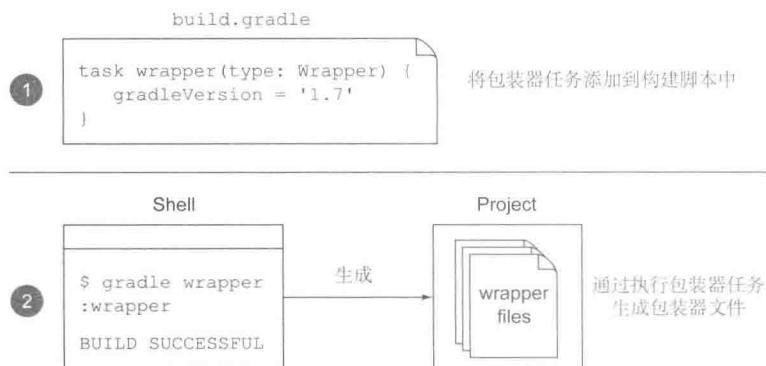


图 3.6 两步配置包装器：添加包装器任务和执行任务

为了能够让项目可以下载压缩过的 Gradle 运行时文件，定义一个类型为 `Wrapper` 的任务，通过 `gradleVersion` 属性指定你想要使用的 Gradle 版本：

```
task wrapper(type: Wrapper) {  
    gradleVersion = '1.7'  
}
```

不要求该任务的名字为 `wrapper`——任何名字都可以。然而，`wrapper` 这个名字通常在 Gradle 的在线文档中使用，并且作为默认约定。执行任务：

```
$ gradle wrapper  
:wrapper
```

结果是，你会发现包装器文件在构建脚本旁边：



记住，你只需要在项目中运行一次 `gradle wrapper` 命令。从那时开始，你就可以用包装器的脚本执行构建了。下载下来的包装器文件应该提交到版本控制系统中。为了记录构建使用过包装器，将 `wrapper` 任务保留在项目中也是有用的。通过改变 `gradleVersion` 的值和重新运行 `wrapper` 任务，它会帮助你升级包装器的版本。你也可以使用前面提到的 `build setup` 插件，而不需要手动去创建一个包装器任务和执行它来下载相关文件。执行命令 `gradle wrapper` 会根据 Gradle 运行时的当前版本来生成包装器文件：

```
$ gradle wrapper
:wrapper
```

接下来，你会使用生成的包装器脚本来启动 Gradle 脚本。

### 3.4.2 使用包装器

作为包装器发布内容的一部分，它提供了一个命令执行脚本。对于 `*nix` 系统，它是一个叫作 `gradlew` 的 `shell` 脚本；对于 Windows 操作系统，它是 `gradlew.bat`。使用它们运行构建和使用已安装的 Gradle 运行时运行构建是一样的。图 3.7 说明了当用包装器脚本运行一个任务时发生了什么。

让我们回到我们的好朋友 Mike 那儿。他从版本控制系统中将代码下载下来。在项目的源代码中，他找到了包装器文件。就像 Mike 在 Windows 环境中开发自己的代码一样，他需要通过包装器的批处理文件执行任务。下面是启动 Jetty 容器运行应用时的控制台输出：

```

> gradlew.bat jettyRun
Downloading http://services.gradle.org/distributions/gradle-1.7-bin.zip
...
Unzipping C:\Documents and Settings\Mike\.gradle\wrapper\dists\gradle-1.7-
bin\35oej0jnbfh6of4dd0553ledaj\gradle-1.7-bin.zip to C:\Documents and
Settings\Mike\.gradle\wrapper\dists\gradle-1.7-
bin\35oej0jnbfh6of4dd0553ledaj
Set executable permissions for: C:\Documents and
Settings\Mike\.gradle\wrapper\dists\gradle-1.7-
bin\35oej0jnbfh6of4dd0553ledaj\gradle-1.7\bin\gradlew.bat
:compileJava
:processResources UP-TO-DATE
:classes
> Building > :jettyRun > Running at http://localhost:9090/todo

```

从远程服务器下载包装器发布版本

解压已压缩的包装器文件到预定义目录

给包装器批处理文件设置执行权限

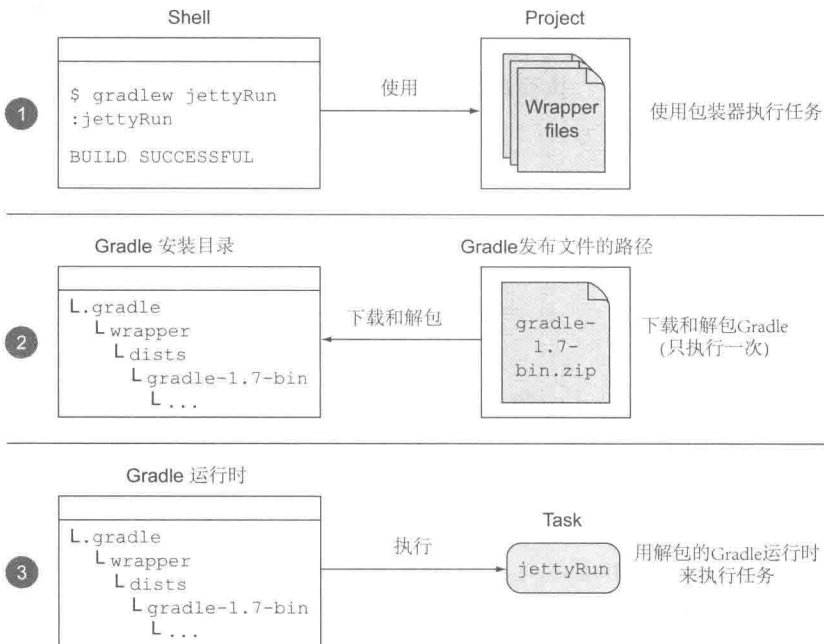


图 3.7 当包装器任务被执行时，Gradle 运行时会被下载、解包并使用

发布的压缩文件从 Gradle 项目维护的中心服务器上下载下来，存储在 Mike 的本地文件系统的 `$HOME_DIR/.gradle/wrapper/dists` 目录下。Gradle 包装器也会负责解包该发布文件并设置相应的权限来执行该批处理文件。下载过程只需要执行一次。后续的构建运行都会重用这个解包的运行时安装程序：

```
> gradlew.bat jettyRun
:compileJava
:processResources UP-TO-DATE
:classes
> Building > :jettyRun > Running at http://localhost:9090/todo
```

关键是什么？由 Gradle 包装器执行的构建脚本提供了与本地已安装的 Gradle 完全相同的任务、特性和行为。而且，你不必遵循 Gradle 包装器提供的默认约定。它的配置选项是非常灵活的。我们会在下一小节来了解它们。

### 3.4.3 定制包装器

某些企业有非常严格的安全策略，特别是当你为政府机构工作时，访问外网是禁止的。

在这种情况下，如何让你的项目使用 Gradle 包装器呢？重新配置。你可以改变默认配置，将它指向一个存有运行时发布文件的企业内部服务器。同时你还可以改变本地的存储路径：

```
task wrapper(type: Wrapper) {
    gradleVersion = '1.2'
    distributionUrl = 'http://myenterprise.com/gradle/dists'
    distributionPath = 'gradle-dists'
}
```

请求的 Gradle 版本

包装器被解压缩后存放的  
相对路径

获取 Gradle 包装  
器的 URL

简单直接，是吧？包装器还有很多可用的选项等待你去了解。你可以在 Gradle 包装器的 DSL 文档 <http://gradle.org/docs/current/dsl/org.gradle.api.tasks.wrapper.Wrapper.html> 中找到更详细的信息。

## 3.5 总结

在第 2 章中，我们了解到如何通过将任务添加到项目中来表达和执行简单的逻辑。在本章中，我们进一步了解了更多的内容。我们实现了一个全栈 Java 应用并用 Gradle 构建它。许多 Java 项目在本质上都是相似的。它们需要编译 Java 源代码，运行测试，打包 JAR 文件。幸运的是，我们不需要为实现它们自己写任务。通过使用 Gradle 的插件，我们几乎不需要在构建脚本中写代码。

我们从使用 Gradle 自带的 Java 插件起步。将插件应用到项目中可以添加预配置的任务和含有默认约定的框架包装的标准化项目结构。灵活的约定满足了定制项目的需求。通过引入插件，我们了解了如何通过选项定制默认约定。可以通过 Gradle

DSL 和 API 文档了解这些选项的更多内容。

在对 Java Web 应用开发的基础做了简单的重述之后，我们讨论了如何通过兼容 Java EE 的 Web 组件来扩展样例程序。通过 War 和 Jetty 插件，Gradle 帮助简化了 Web 开发。War 插件帮助组装 WAR 文件，而 Jetty 插件提供了一种有效部署到轻量级 Servlet 容器的方法。你看到约定优于配置的范例也可以轻松地动态改变。你也学习到使用包装器是每个 Gradle 项目的最佳实践方式。它不仅允许你在没有安装 Gradle 运行时的机器上构建项目，还帮助你解决了版本不兼容的问题。

本章展现的插件还提供了更多我们没有讨论过的功能。想要详细了解，请参考在线用户手册（[http://www.gradle.org/docs/current/userguide/standard\\_plugins.html](http://www.gradle.org/docs/current/userguide/standard_plugins.html)）。本章将结束本书的第 1 部分。在第 2 部分中，我们会深入研究 Gradle 的许多核心概念。下一章会集中在 Gradle 的构建块、任务的输入 / 输出和构建的生命周期上。

## 第2部分

# 掌握基本原理

在第 1 部分，你已经通过示例学习了 Gradle 的核心概念和特性。第 2 部分将进一步提高你的知识。我们将着眼于更高级的主题，比如依赖管理、使用 Gradle 对应用程序进行测试、使用插件来扩展构建，以及更多的内容。

第 4 章涵盖了 Gradle 建模构建的精髓构建块。你会学到如何声明新的 task，操作已有的 task，为复杂的逻辑实现合适的抽象。没有项目能够在不重用现有类库的情况下成功。在第 5 章中，你可以学到如何在构建脚本中声明和管理依赖，也涵盖了依赖报告以及解决版本冲突。模块化软件项目为项目构建添加了额外一层复杂度。第 6 章将讨论 Gradle 如何支持多项目构建。

在第 7 章中，我们将注意力放在测试上。你将看到编写、组织和执行单元测试、集成测试和功能测试是多么地容易，同时我们会介绍一些相应的工具，以供你日后自由选择。第 8 章将用示例演示 Gradle 的扩展机制。你将学到如何抽象复杂逻辑将应用程序部署到云端。我们将从自定义 task、脚本和二进制插件等方面来了解 Gradle 的扩展模型，从而实现自己的配置语言。因为 Gradle 总是与流行的构建工具如 Ant 和 Maven 密切合作，

所以第 9 章将致力于帮你将现有的构建逻辑从一种工具迁移到另一种工具，辨别集成点，并深入刻画迁移策略。

当你完成这一部分的时候，你就可以将 Gradle 的核心概念运用到实战项目中了。在第 3 部分，我们将讨论 Gradle 与其他构建工具配合使用的交付场景。



# 构建脚本概要

# 4

## 本章涵盖

- Gradle 的构建块及其 API 表示
- 声明 task 和操作 task
- task 的高级用法
- 实现和使用 task 类型
- 了解构建生命周期挂接

在第 3 章中，使用 Gradle 的核心插件从无到有地构建了一个完整的 Java Web 应用。你已经学到那些插件引入的默认约定，就是它们的自定义性和易用性以满足非标准构建需求。插件预配置 task 功能通过添加可执行的构建逻辑到项目中，从而成为插件的核心组件。

在这一章中，我们将探索最基础的 Gradle 构建块，也就是 project 和 task，以及它们是如何映射到 Gradle 的 API 中类上面的。这些类通过方法暴露了很多属性，从而帮助控制构建。你也会学到如何通过属性来控制构建的行为，以及结构化构建逻辑的好处等。

作为这一章的核心，你会通过实现一个实例来体验 task 是如何工作的，一步一

步的，你会获得相应的知识，从声明简单的 task 到编写自定义的 task 类。在这个过程中，我们将接触到如获取 task 属性，定义显式的和隐式的 task 依赖，添加增量式构建支持，以及使用 Gradle 的内置 task 类型等内容。

我们也会看看 Gradle 的构建生命周期，以更好地理解构建是如何配置和执行的。构建脚本可以响应生命周期的构建进度的通知。在本章的最后一部分中，我们将展示如何编写使用闭包和监听器实现的生命周期钩子。

## 4.1 构建块

每个 Gradle 构建都包含三个基本构建块：project、task 和 property。每个构建包含至少一个 project，进而又包含一个或多个 task。project 和 task 暴露的属性可以用来控制构建。图 4.1 展示了 Gradle 的核心组件之间的依赖关系。

Gradle 构建

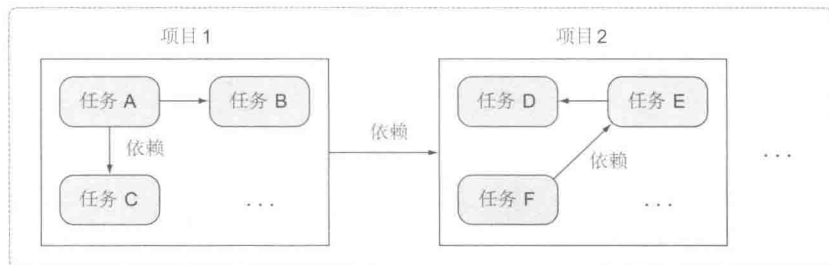


图 4.1 Gradle 构建中的两个基本概念是 project 和 task。在多项目构建中一个 project 可以依赖于其他的 project。相似的，task 可以形成一个依赖关系图来确保它们的执行顺序。

Gradle 使用领域驱动设计（DDD）的原理为其自己的领域构建软件建模。因此，在 Gradle API 中有相应的类来表示 project 和 task。让我们仔细看看每个组件及其对应的 API。

### 4.1.1 项目

在 Gradle 术语中，一个项目（project）代表一个正在构建的组件（比如，一个 JAR 文件），或一个想要完成的目标，如部署应用程序。如果你使用过 Maven，那么这个概念应该听起来很熟悉。Gradle 的 build.gradle 文件相当于 Maven 的 pom.xml。每个 Gradle 构建脚本至少定义一个项目。当构建进程启动后，Gradle 基于 build.gradle 中的配置实例化 org.gradle.api.Project 类，并且能够通过 project 变量使其隐式可用。图 4.2 显示了 API 接口及其最重要的方法。

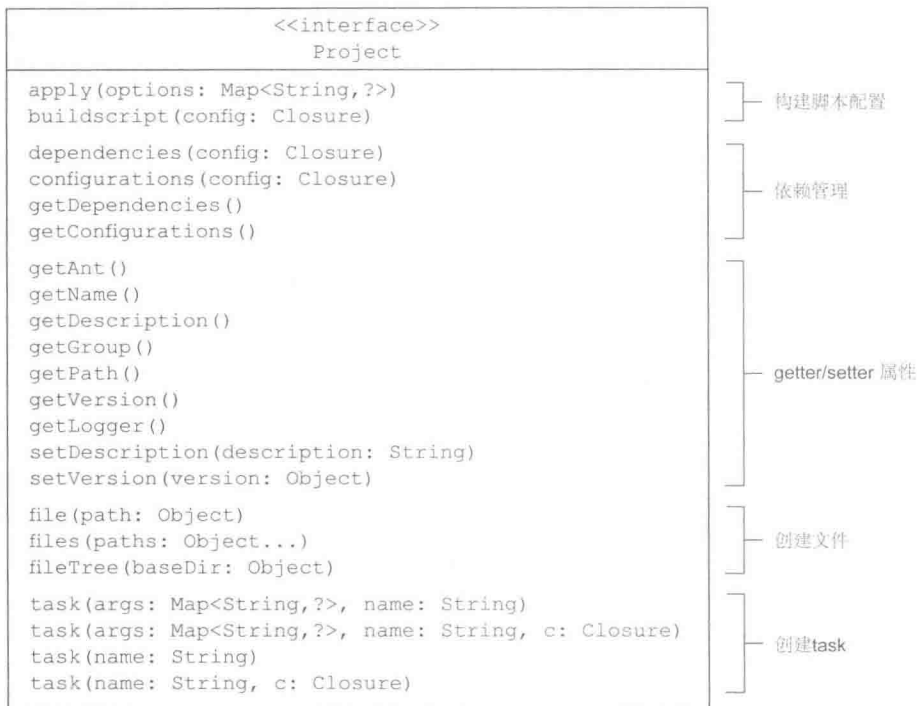


图 4.2 Gradle 构建的主要入口点——Project 接口

一个 project 可以创建新的 task，添加依赖关系和配置，并应用插件和其他的构建脚本。它的许多属性，如 name 和 description，可通过 getter 和 setter 方法访问。

为什么我们要过早地谈论 Gradle 的 API 吗？你会发现了解 Gradle 的基本知识之后，你想要更进一步地将概念应用到真实项目中。API 就是最有效地使用 Gradle 的关键。

在构建中 Project 实例，让你可以通过代码来访问 Gradle 的所有特性，比如 task 的创建和依赖管理。在本书中，通过调用对应的 API 方法你会使用到很多这些特性。记住，当访问属性和方法时，不需要使用 project 变量——它会假设你是指 Project 实例。下面的代码片段展示了如何合理地调用 Project 实例上的方法：

```

setDescription("myProject")
println "Description of project $name: " + project.description

```

在不显式使用 project 变量的情况下设置项目描述

在不使用 project 变量的情况下通过 Groovy 语法来访问 name 和 description 属性

在前面的章节中，你只需要处理单项目构建。Gradle 也支持多项目构建。软件开发最重要的原则之一是分离关注点。软件系统变得越复杂，就越想要将它分解为多个模块化的功能，这些模块可以相互依赖。每个分解的部分将被表示为一个 Gradle 项目，并且有自己独立的 build.gradle 脚本。为了简单起见，我们这里不详细介绍了。如果你想了解更多的话，请跳转到第 6 章，第 6 章完全致力于使用 Gradle 进行多项目构建。接下来，我们将看看 task 的特点，task 是 Gradle 的另一个核心构建块。

### 4.1.2 任务

你已经在第 2 章中创建了一些简单的任务（task）。尽管我使用的用例很琐碎，但你一定知道了 task 的一些重要功能：任务动作（task action）和任务依赖（task dependency）。任务动作定义了一个当任务执行时最小的工作单元。这可以简单到只打印文本如“Hello world!”或复杂到编译 Java 源代码，见第 2 章。很多时候，运行一个 task 之前需要运行另一个 task，尤其是当 task 的运行需要另一个 task 的输出作为输入来完成自己的行动时更是如此。比如，你已经看到过在打包成一个 JAR 文件之前需要先编译 Java 源代码。让我们看看 Gradle task 的 API 表示，org.gradle.api.Task 接口，如图 4.3 所示。

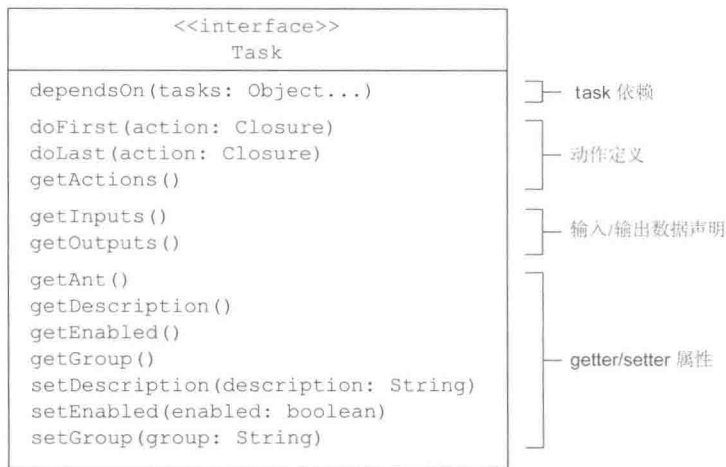


图 4.3 Gradle API 中的 Task 接口。task 可以定义依赖于其他 task、动作序列和执行条件。

### 4.1.3 属性

每个 Project 和 Task 实例都提供了可以通过 getter 和 setter 方法访问的属性。

一个属性可能是一个任务的描述或项目的版本。在本章后面，你会在实例中读和修改这些实例的属性值。通常，你需要定义自己的属性。比如，你可能想要声明一个变量，该变量引用了在同一个构建脚本中多次使用的一个文件。Gradle 允许用户通过扩展属性自定义一些变量。

## 扩展属性

Gradle 的很多领域模型类提供了特别的属性支持。在内部，这些属性以键值对的形式存储。为了添加属性，你需要使用 `ext` 命名空间。让我们看一个具体的例子。下面的代码片段演示了以不同的方式添加、读取和修改一个属性。

```
project.ext.myProp = 'myValue'
ext {
    someOtherProp = 123
}

assert myProp == 'myValue'
println project.someOtherProp
ext.someOtherProp = 567
```

只在初始声明扩展属性时需要使用 `ext` 命名空间

使用 `ext` 命名空间访问属性是可选的

类似地，额外的属性也可以通过属性文件来提供。

## Gradle 属性

Gradle 属性可以通过在 `gradle.properties` 文件中声明直接添加到项目中，这个文件位于 `<USER_HOME>/ .gradle` 目录或项目的根目录下。这些属性可以通过项目实例访问。记住，即使你有多个项目，每个用户也只能有一个 Gradle 属性文件在 `<USER_HOME>/ .gradle` 目录下。这是目前 Gradle 对它的限制。在这个属性文件中声明的属性对所有的项目可用。我们假设下面的属性是在 `gradle.properties` 文件中声明的：

```
exampleProp = myValue
someOtherProp = 455
```

你可以按照如下方式访问项目中的这两个变量：

```
assert project.exampleProp == 'myValue'

task printGradleProperty << {
    println "Second property: $someOtherProp"
}
```

## 声明属性的其他方式

对于前面两种方式，我们大多用来声明自定义变量及其值。Gradle 也提供了很多其他方式为构建提供属性，例如：

- 项目属性通过 `-P` 命令行选项提供
- 系统属性通过 `-D` 命令行选项提供
- 环境属性按照下面的模式提供

```
ORG_GRADLE_PROJECT_propertyName=someValue
```

在这里对于这些声明属性的可选方式不再给出具体的例子，但是在需要时你可以使用它们。如果你想进一步了解，在线的 Gradle 用户指南上有非常详细的例子可供参考。在本章的后续部分，你将充分使用 `task` 和 Gradle 的构建生命周期。

## 4.2 使用task

在默认情况下，每个新创建的 `task` 都是 `org.gradle.api.DefaultTask` 类型的，标准的 `org.gradle.api.Task` 实现。`DefaultTask` 里的所有属性都是 `private` 的。这意味着它们只能通过 `public` 的 `getter` 和 `setter` 方法来访问。幸运的是，Groovy 提供了一些语法糖，可以直接通过属性名来使用属性。在底层，Groovy 会为你调用这些方法。在本节中，我们将通过例子探索 `task` 最重要的特性。

### 4.2.1 项目版本管理

为了展示 `DefaultTask` 类的属性和方法，我将使用第 3 章的 `To Do` 应用程序上下文中来解释它们。现在你已经有了了一般的构建基础了，而且可以容易地添加特性。通常，一些特性会被分组发布。为了识别每一次发布，应该为可交付软件添加一个唯一的版本号。

许多企业级或开源项目都有自己的版本管理策略。回想一下你工作过的一些项目。通常，你会指定一个特定的版本编号方案（比如，通过点号分隔主版本和次版本号，如 1.2）。你也可能遇到一个项目版本中附加了一个 `SNAPSHOT` 指示器来表示构建项目工件处于开发状态。在第 3 章中你已经给项目指定了一个版本，这是通过为项目属性 `version` 设置一个字符串值来实现的。在简单的用例中非常适合使用 `String` 数据类型，但是如果要知道项目确切的次版本号该怎么办呢？必须解析字符串值，搜索点字符，过滤出标识次版本的子串。这岂不是很容易通过一个实际的类来表示项目版本？

你可以轻松地使用类的属性来设置、检索和修改编号方案的特定部分。你可以更进一步。通过外部化版本信息来持久化数据存储，比如一个文件或数据库，可以避免通过修改构建脚本本身来改变项目的版本。图 4.4 展示了构建脚本之间的交互，属性文件中包含了版本信息和数据表示类。在接下来的章节里，你会创建和学习如何使用所有的这些文件。

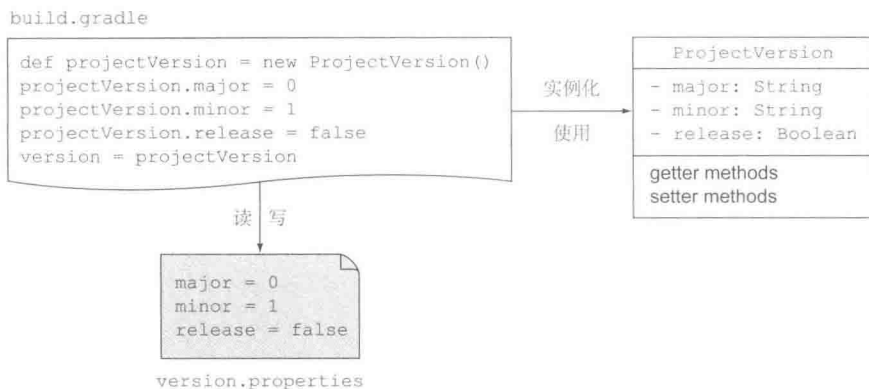


图 4.4 在构建脚本运行阶段，从属性文件中读取项目版本。ProjectVersion 数据类被实例化。每个版本类别都被转换成数据类的属性值。ProjectVersion 实例被赋值给项目的 version 属性。

如果你想要自动化项目生命周期，那么以编程方式控制版本管理方案将变得很有必要。举一个例子：你的代码已经通过了所有的功能测试并且准备组装发布。项目的当前版本是 1.2-SNAPSHOT。在构建最后的 WAR 文件之前，你要把它变成一个发布版本 1.2，并且自动部署到生产服务器上。这些步骤都可以通过创建 task 的方式进行建模：一个用于修改项目版本，一个用于部署 WAR 文件。让我们更进一步来学习关于 task 的知识，以实现灵活的项目版本管理。

## 4.2.2 声明 task 动作

动作 (action) 就是在 task 中合适的地方放置构建逻辑。Task 接口提供了两个相关的方法来声明 task 动作：doFirst(Closure) 和 doLast(Closure)。当 task 被执行的时候，动作逻辑被定义为闭包参数被依次执行。

通过添加一个 task printVersion 来简单地开始吧。这个 task 的作用就是打印出当前的项目版本。把这段逻辑放在 doLast 方法里面，如下面代码片段所示：

```
version = '0.1-SNAPSHOT'

task printVersion {
    doLast {
        println "Version: $version"
    }
}
```

在第 2 章中，我解释过左移操作符 (<<) 就是 doLast 方法的快捷版本。它们做的是完全相同的事情。当执行 gradle printVersion 命令时，你会看到正确的版本号：

```
$ gradle printVersion
:printVersion
Version: 0.1-SNAPSHOT
```

如果把这段逻辑放到 `doFirst` 方法里面，也会得到同样的结果：

```
task printVersion {
    doFirst {
        println "Version: $version"
    }
}
```

### 给现有 task 添加动作

到目前为止，你只给 `task printVersion` 添加了一个动作，或者是第一个动作或者是最后一个动作。但并不限于为每个 `task` 只添加一个动作。事实上，在 `task` 创建后，你可以根据需要添加很多动作。在内部，每个 `task` 都保持了一个动作列表。在运行时，它们按顺序执行。让我们看一个示例 `task` 的修改后版本：

```
task printVersion {
    doFirst {
        println "Before reading the project version"
    }
    doLast {
        println "Version: $version"
    }
}

printVersion.doFirst { println "First action" }
printVersion << { println "Last action" }
```

使用 `doLast` 别名在动作列表的最后添加闭包

声明一个包含 `doFirst` 和 `doLast` 的 `task`

在动作列表的开始添加 `doFirst` 闭包

如上所示，我们可以给现有的 `task` 添加一些动作。这在你想要为不是自己编写的 `task` 执行自定义逻辑时非常有用。比如，为 Java 插件的 `compileJava` `task` 添加一个 `doFirst` 动作来检查项目中至少包含一个 Java 源文件。

### 4.2.3 访问 DefaultTask 属性

接下来，你将改进输出版本号的方式。Gradle 提供了一个基于 SLF4J 日志库的 `logger` 实现。除了实现常规范围的日志级别（`DEBUG`、`ERROR`、`INFO`、`TRACE`、`WARN`）之外，Gradle 还增加了一些额外的日志级别。通过 `Task` 的方法可以直接访问 `logger` 实例。现在，要打印 `QUIET` 日志级别的版本号：

```
task printVersion << {
    logger.quiet "Version: $version"
}
```



由此可见，访问 task 的属性是多么容易呀！其实还有两个属性：group 和 description。它们都是 task 文档的一部分。

description 属性用于描述任务的作用，而 group 属性则用于定义 task 的逻辑分组。在创建 task 的时候，为这两个属性设置值作为参数：

```
task printVersion(group: 'versioning',
                  description: 'Prints project version.') << {
    logger.quiet "Version: $version"
}
```

或者，也可以通过调用 setter 方法来设置属性：

```
task printVersion {
    group = 'versioning'
    description = 'Prints project version.'

    doLast {
        logger.quiet "Version: $version"
    }
}
```

当运行 gradle tasks 时，我们可以看到 task 正确的分组和描述：

```
gradle tasks
:tasks
...

Versioning tasks
-----
printVersion - Prints project version.
...
```

尽管设置 task 的描述和分组是可选的，但是为所有的 task 指定值是一个好主意。这会帮助最终用户比较容易地去识别 task 的功能。接下来，我们将回顾一下 task 之间定义依赖关系的复杂性。

#### 4.2.4 定义 task 依赖

dependsOn 方法允许声明依赖一个或多个 task。你已经看到 Java 插件充分利用了这个概念，通过创建 task 依赖关系图来建模完整的 task 生命周期如 build task。下面清单显示了使用 dependsOn 方法应用 task 依赖的不同方式。

##### 清单 4.1 应用 task 依赖

```
task first << { println "first" }
task second << { println "second" }

task printVersion(dependsOn: [second, first]) << {
    logger.quiet "Version: $version"
}
```

指定多个 task 依赖

```
task third << { println "third" }  
third.dependsOn('printVersion')
```

← 声明依赖时按名称引用 task

在命令行通过调用 `task third` 来执行依赖链上的其他 task：

```
$ gradle -q third  
first  
second  
Version: 0.1-SNAPSHOT  
third
```

如果仔细看看 task 的执行顺序，你可能会对结果感到惊讶。task `printVersion` 声明了依赖 `first` 和 `second` task。难道你不期望 `second` 在 `first` 执行之前得到执行？在 Gradle 中，task 执行顺序是不确定的。

### task 依赖的执行顺序

理解 Gradle 并不能保证 task 依赖的执行顺序是很重要的。`dependsOn` 方法只是定义了所依赖的 task 需要先执行。Gradle 的思想是声明在一个给定的 task 执行之前什么该被执行，而没有定义它该如何执行。如果你之前使用过像 Ant 这种命令式地定义依赖的构建工具的话，那么 Gradle 的这个概念就有点难以理解了。在本章的后续部分你将看到，在 Gradle 中，执行顺序是由 task 的输入 / 输出规范自动确定的。这种构建设计有很多好处。一方面，你不需要知道整个 task 依赖链上的关系是否发生改变，这样可以提高代码的可维护性和避免潜在的破坏。另一方面，因为构建没有严格的执行顺序，也就是支持 task 的并行执行，这样可以极大地节约构建执行时间。

## 4.2.5 终结器 task

在实践中，你会发现所依赖的 task 执行后需要清理某种资源。一个典型的例子就是 Web 容器需要对已经部署的应用程序运行集成测试。针对这种情景 Gradle 提供了终结器 task（finalizer task），即使终结器 task 失败了，Gradle 的 task 也会按预期运行。下面的代码片段展示了如何通过使用 Task 方法 `finalizedBy` 来使用一个特定的终结器 task。

```
task first << { println "first" }  
task second << { println "second" }  
first.finalizedBy second
```

← 声明一个 task 被另一个 task 终结

你会看到执行 `first` 会自动触发 `second`：

```
$ gradle -q first  
first  
second
```

第 7 章将通过真实的例子更深入地介绍终结器 task 的概念。在下一节中，将编写一个 Groovy 类来细粒度地控制版本管理方案。

### 4.2.6 添加任意代码

现在是时候来讨论在 Gradle 构建脚本中定义通用的 Groovy 代码的功能了。在实践中，你可以在 Groovy 脚本或类中以习惯的方式来编写类和方法了。在本节中，你将创建一个表示项目版本的类。在 Java 中，遵循 bean 惯例的类被称为 POJO。根据定义，它们通过 getter 和 setter 方法来暴露属性。随着时间的推移，手工编写这些方法会变得非常烦琐。Groovy 也有等效的 POJO，即 POGO，只是需要声明属性，而不需要设置访问权限修饰符。它们的 getter 和 setter 方法本质上是在生成字节码时自动添加的，因此在运行时它们可以直接被使用。在下面的清单中，指定了一个 POGO 实例 `ProjectVersion`。实际的值是在构造器中设置的。

清单 4.2 通过 POGO 来表示项目版本

```
version = new ProjectVersion(0, 1)
class ProjectVersion {
    Integer major
    Integer minor
    Boolean release

    ProjectVersion(Integer major, Integer minor) {
        this.major = major
        this.minor = minor
        this.release = Boolean.FALSE
    }

    ProjectVersion(Integer major, Integer minor, Boolean release) {
        this(major, minor)
        this.release = release
    }

    @Override
    String toString() {
        "$major.$minor${release ? '' : '-SNAPSHOT'}"
    }
}
```

← 根据 `java.lang.Object` 表示的 `version` 属性；Gradle 总是调用 `toString()` 来获取 `version` 的值

← 只有当 `release` 属性值为 `false` 的时候，版本上才会添加 `-SNAPSHOT` 后缀

运行修改后的构建脚本，你会看到 `printVersion` task 产生的结果与之前的完全相同。遗憾的是，你仍然需要手动编辑构建脚本来改变版本类别。接下来，我们将版本信息存储在外部文件中并配置构建脚本来读取它。

### 4.2.7 理解 task 配置

在开始编写代码之前，你需要创建一个名为 `version.properties` 的属性文件，并且为每一个版本的类别如主版本和次版本设置不同的属性。下面的键值对表示最初的版本 `0.1-SNAPSHOT`：

```
major = 0
minor = 1
release = false
```

## 添加 task 配置块

清单 4.3 声明了一个名为 `loadVersion` 的 task，用于从属性文件中读取版本类别，并将新创建的 `ProjectVersion` 实例赋值给项目的版本属性。乍一看，这个 task 很像你之前定义的其他 task。但是如果仔细观察，就会发现你没有定义动作或者使用左移操作符 (`<<`)。Gradle 称之为 *task 配置*。

清单 4.3 编写一个 task 配置

```
ext.versionFile = file('version.properties')

task loadVersion {
    project.version = readVersion()
}

ProjectVersion readVersion() {
    logger.quiet 'Reading the version file.'

    if(!versionFile.exists()) {
        throw new GradleException("Required version file does not exist:
                                   ➡ $versionFile.canonicalPath")
    }

    Properties versionProps = new Properties()
    versionFile.withInputStream { stream ->
        versionProps.load(stream)
    }

    new ProjectVersion(versionProps.major.toInteger(),
        ➡ versionProps.minor.toInteger(), versionProps.release.toBoolean())
}
```

Project 接口提供了 `file` 方法；它会创建一个相对于项目目录的 `java.io.File` 实例

没有使用左移操作符定义 task 配置

Groovy 的文件实现通过添加新的方法来读取 `InputStream`

如果版本文件不存在，将抛出 `GradleException` 异常，带有相应的错误消息

在 Groovy 中，如果 `return` 是方法中最后一条语句的话，则可以将它省略

现在运行 `printVersion`，你会看到新创建的 `loadVersion` 先被执行。尽管 task 的名字没有被打印出来，但是你仍然知道这种情况，因为打印的日志是你自己添加的。

```
$ gradle printVersion
Reading the version file.
:printVersion
Version: 0.1-SNAPSHOT
```

你也许会问自己，为什么这个 task 被完全调用了？当然，你没有声明依赖关系，也没有在命令行调用它。原因就是 task 配置块永远在 task 动作执行之前被执行。完

全理解这种行为是了解 Gradle 构建生命周期的关键。让我们具体看看每一个构建阶段。

### Gradle 构建生命周期阶段

无论什么时候执行 Gradle 构建，都会运行三个不同的生命周期阶段：初始化、配置和执行。图 4.5 可视化了构建阶段的运行顺序和它们执行的代码。

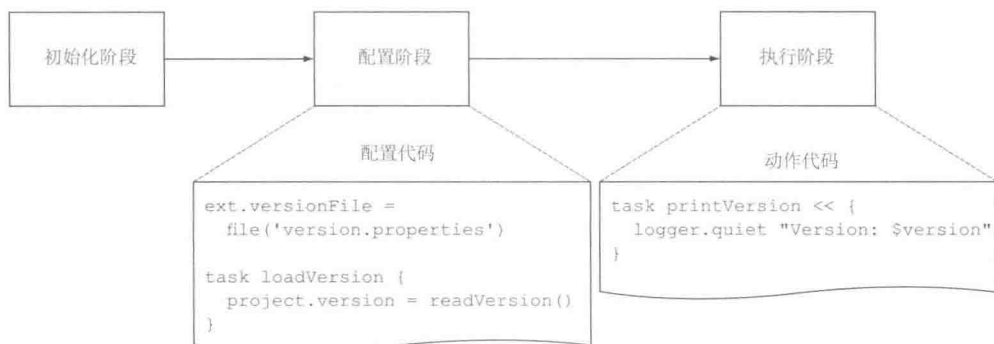


图 4.5 在 Gradle 构建生命周期中构建阶段的顺序

在初始化阶段，Gradle 为项目创建了一个 Project 实例。在给定的构建脚本中只定义了一个项目。在多项目构建中，这个构建阶段变得更加重要。根据你正在执行的项目，Gradle 找出哪些项目依赖需要参与到构建中。注意，在这个构建阶段当前已有的构建脚本代码都不会被执行。在第 6 章中，当你模块化 To Do 应用程序使其成为多项目构建的时候，这种情况会发生变化。

初始化阶段后面紧接着的是配置阶段。Gradle 构造了一个模型来表示任务，并参与到构建中来。增量式构建特性决定了模型中的 task 是否需要被运行。这个阶段非常适合于为项目或指定 task 设置所需的配置。

**注意：**项目每一次构建的任何配置代码都可以被执行——即使你只执行 gradle tasks。

在执行阶段，所有的 task 都应该以正确的顺序被执行。执行顺序是由它们的依赖决定的。如果任务被认为没有修改过，将被跳过。比如，如果 task B 依赖于 task A，那么当在命令行运行 gradle B 时执行顺序将是 A->B。

正如你所见，Gradle 的增量式构建特性紧紧地与生命周期相结合。在第 3 章中，你已经看到 Java 插件大量地使用了这个特性。如果 Java 源文件与最后一次运行的构建不同的话，则只运行 compileJava task。最终，这个特性将充分提高构建的性能。在下一节中，我们将介绍针对自己的 task 如何使用增量式构建特性。

## 4.2.8 声明 task 的 inputs 和 outputs

Gradle 通过比较两个构建 task 的 inputs 和 outputs 来决定 task 是否是最新的, 如图 4.6 所示。自从最后一个 task 执行以来, 如果 inputs 和 outputs 没有发生变化, 则认为 task 是最新的。因此, 只有当 inputs 和 outputs 不同时, task 才运行; 否则将跳过。

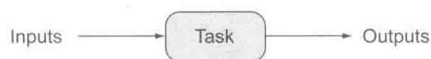


图 4.6 Gradle 通过 task 的 inputs/outputs 来决定它是否被执行

输入可以是一个目录、一个或多个文件, 或者是一个任意属性。一个 task 的输出是通过一个目录或 1 ~  $n$  个文件来定义的。inputs 和 outputs 在 DefaultTask 类中被定义为属性或者有一个直接类来表示, 如图 4.7 所示。

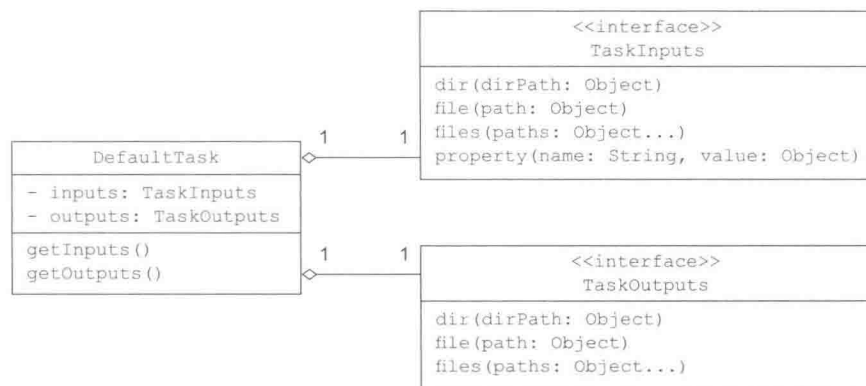


图 4.7 DefaultTask 类定义的 task 的 inputs 和 outputs

我们来看看这个特性。假设你想创建一个 task, 为产品发布准备项目的可交付版本。为此, 你想将项目版本从 SNAPSHOT 改变为 release。下面的清单定义了一个新的 task, 将 Boolean 值 true 赋值给版本属性 release。这个 task 也将版本变化传递到属性文件中。

### 清单 4.4 将项目版本切换为产品发布版本

```

task makeReleaseVersion(group: 'versioning', description: 'Makes project
                        ➡ a release version.') << {
    version.release = true
    ant.propertyfile(file: versionFile) {
        entry(key: 'release', type: 'string', operation: '=', value: 'true')
    }
}
  
```

Ant task 的 propertyfile 提供了一种便利的方式来修改属性文件

正如所期望的,运行这个 task 将改变版本属性,并将新值持续保存到属性文件中。下面的输出显示了这个行为:

```
$ gradle makeReleaseVersion
:makeReleaseVersion

$ gradle printVersion
:printVersion
Version: 0.1
```

makeReleaseVersion task 可能是另一个生命周期 task 的一部分,用来将 WAR 文件部署到生产服务器上。令人感觉痛苦的是部署出错了。可能是网络出现故障,导致无法联系上服务器。修复网络故障后,需要再次运行部署 task。因为 makeReleaseVersion task 被声明为依赖部署 task,所以它会自动重新运行。注意,虽然我们将项目版本标记为产品发布版本,但是 Gradle 并不知道。为了解决这一问题,需要声明它的 inputs 和 outputs, 如下面的清单所示。

#### 清单 4.5 通过 inputs/outputs 来添加增量式构建支持

```
task makeReleaseVersion(group: 'versioning', description: 'Makes project
→ a release version.') {
    inputs.property('release', version.release)
    outputs.file versionFile
    doLast {
        version.release = true
        ant.propertyfile(file: versionFile) {
            entry(key: 'release', type: 'string', operation: '=', value: 'true')
        }
    }
}
```

在配置阶段声明 inputs/outputs

声明版本的 release 属性作为输入

由于版本文件被修改了,所以它被声明作为输出文件属性

把你执行的代码放到 doLast 动作闭包中,并从 task 声明中移除左移操作符。这样,你就清晰地分离了它的配置和动作代码。

#### task 的 inputs 和 outputs 的执行

记住,task 的 inputs 和 outputs 是在配置阶段执行的用来连接 task 依赖。这就是它们需要在配置块中被定义的原因。为了避免出现所不期望的行为,请确保赋给 inputs 和 outputs 的值在配置期间是可访问的。如果你需要通过编程获得输出,则可以通过 TaskOutputs 上的 upToDateWhen (Closure) 方法来实现在。与常规的 inputs 和 outputs 相比,这个方法是在执行期间执行的。如果闭包返回 true,这个 task 则会被认为是最新的。

现在,如果你执行 task 两次,就会发现 Gradle 已经知道项目版本被设置为发布

版本，而且自动跳过 task 的第二次执行。

```
$ gradle makeReleaseVersion
:makeReleaseVersion

$ gradle makeReleaseVersion
:makeReleaseVersion UP-TO-DATE
```

如果你没有手动修改属性文件中的 `release` 属性，那么 `makeReleaseVersion` 将被标记为最新的，永远会被跳过执行。

至此，你已经使用 Gradle 的 DSL 在构建脚本中创建和修改了 task。每个 task 都得到在 Gradle 的配置阶段实例化的 task 对象支持。在大多数情况下，使用简单的 task 就可以完成工作。然而，有时候你可能想要完全控制 task 实现。在下一节中，你将以自定义 task 实现的方式来重写 `makeReleaseVersion` task。

### 4.2.9 编写和使用自定义 task

`makeReleaseVersion` task 中的动作逻辑相当简单。当前代码的可维护性显然不是问题。然而，当你工作在多项目中时，就会发现原本简单的 task 随着项目规模的快速增长需要添加更多的逻辑。此时就有了对类和方法中代码的结构化需求。你应该能够应用像在常规的产品源代码中习惯使用的编码惯例。Gradle 不建议你用某种特殊方式去编写 task。你可以完全控制构建源代码。你所选择的编程语言如 Java、Groovy 或者其他基于 JVM 的语言，完全取决于你对它们的熟悉程度。

自定义 task 包含两个组件：自定义的 task 类，封装了逻辑行为，也被称作任务类型，以及真实的 task，提供了用于配置行为的 task 类所暴露的属性值。Gradle 把这些 task 称为增强的 task。

可维护性是编写自定义 task 类的优势之一。因为你正在操作一个实际的类，通过单元测试任何方法都是完全可测试的。测试构建代码不在本章讨论范围之内。如果你想进一步了解，请跳到第 7 章。与简单的 task 相比，增强的 task 的另一个优势是可重用性。自定义 task 所暴露的属性可以在构建脚本中进行单独设置。记住了增强的 task 的诸多好处，让我们讨论一下如何编写自定义的 task 类。

#### 编写自定义的 task 类

正如本章前面所提到的，Gradle 为构建脚本中的每个简单的 task 都创建了一个 `DefaultTask` 类型的实例。当创建一个自定义 task 时，你需要做的是，创建一个继承 `DefaultTask` 的类。下面的清单展示了如何将 `makeReleaseVersion` 的逻辑表示为使用 Groovy 编写的自定义类 `ReleaseVersionTask`。



## 清单 4.6 自定义 task 的实现

```

class ReleaseVersionTask extends DefaultTask {
    @Input Boolean release
    @OutputFile File destFile

    ReleaseVersionTask() {
        group = 'versioning'
        description = 'Makes project a release version.'
    }

    @TaskAction
    void start() {
        project.version.release = true
        ant.propertyfile(file: destFile) {
            entry(key: 'release', type: 'string', operation: '=', value: 'true')
        }
    }
}

```

通过注解声明 task 的输入 / 输出

在构造器中设置 task 的 group 和 description 属性

使用注解声明将被执行的方法

编写一个继承 Gradle 默认 task 实现的自定义 task

在这个清单中，没有使用 `DefaultTask` 的属性来声明它的输入和输出。而是使用 `org.gradle.api.tasks` 包下的注解。

## 通过注解表示输入和输出

task 的输入和输出注解为你的实现添加了语义糖。它们不仅与 `TaskInputs` 和 `TaskOutputs` 方法有相同的效果，而且它们还能够充当自动文档。稍微一看，你就会知道什么数据会被当作输入，该 task 会生成什么样的输出工件。探索这个包的 Javadocs 文档时，你会发现 Gradle 提供了各种各样的注解。

在自定义的 task 类中，你可以使用 `@Input` 注解声明输入属性 `release`，使用 `@OutputFile` 注解定义输出文件。为属性添加输入和输出注解并不是唯一的选择，你也可以为 `getter` 方法添加注解。

## task 输入验证

`@Input` 注解会在配置期间验证属性值。如果值为 `null`，Gradle 会抛出 `TaskValidationException` 异常。为了允许输入为 `null` 值，我们需要给它添加 `@Optional` 注解

## 使用自定义 task

通过创建一个动作方法和暴露它的配置属性，你实现了一个自定义的 task 类。但你会怎么使用它呢？在构建脚本中，你需要创建一个 `ReleaseVersionTask` 类型的 task，并且通过为它的属性赋值来设置输入和输出，如下面的清单所示。认为这是创建一个特定类的新实例，并且在构造器中为它的属性设置值。

清单 4.7 ReleaseVersionTask 类型的 task

```
task makeReleaseVersion(type: ReleaseVersionTask) {
    release = version.release
    destFile = versionFile
}
```

← 定义一个增强的 ReleaseVersionTask 类型的 task

└─ 设置自定义 task 属性

正如所期望的，增强的 makeReleaseVersion task 与简单的 task 的运行结果表现完全一致。与简单的 task 实现相比，增强的 task 的一个巨大优势在于所暴露的属性可以被单独赋值。

### 应用自定义 task 的可重用性

假设你想要在另一个项目中使用自定义 task。在那个项目中，需求是不同的。其版本 POGO 通过暴露不同的属性来表示版本管理方案，如下面的清单所示。

清单 4.8 不同的版本 POGO 实现

```
class ProjectVersion {
    Integer min
    Integer maj
    Boolean prodReady

    @Override
    String toString() {
        "$maj.$min${prodReady? '' : '-SNAPSHOT'}"
    }
}
```

此外，项目所有者决定命名版本文件为 project-version.properties，而不是 version.properties。如何让增强的 task 满足这些需求呢？其实你只需要给所暴露的属性赋予不同的值就可以了，如下面的清单所示。自定义的 task 类可以灵活地处理需求的变化。

清单 4.9 为 makeReleaseVersion task 设置单独的属性值

```
task makeReleaseVersion(type: ReleaseVersionTask) {
    release = version.prodReady
    destFile = file('project-version.properties')
}
```

← POGO 版本表示使用 prodReady 属性来指示发布标识

← 指定不同的版本文件对象

Gradle 封装了大量开箱即用的自定义类作为常用功能，比如复制和删除文件，或者创建一个 ZIP 归档。在下一节中，我们将仔细了解这些内容。

### 4.2.10 Gradle 的内置 task 类型

你还记得上一次手动部署产品出错的情形吗？我想你的脑海里依然会有这样生动的画面：愤怒的客户把电话打到了你的支持团队，老板敲着你的桌子问怎么回事，当时你和同事正疯狂地试图找出启动程序时堆栈跟踪抛出异常的根源。在手动发布过程中忘记一个简单的步骤就可能是致命的。

我们是专业人士，要以自动化构建生命周期的每一方面为傲。以自动化方式修改项目的版本管理方案只是建模发布过程的第一步。为了能够快速地从失败的部署中恢复过来，一个好的回滚策略是必需的。保留一份最新的稳定的应用程序可交付版本的备份非常重要。你可以为 To Do 应用程序使用 Gradle 封装的一些 task 类型来实现这个过程的各部分。

这就是你所需要做的。在将代码部署到产品环境中之前，你需要创建一个发布包。把它作为将来项目部署失败时回退的可交付版本。发布包是一个 ZIP 文件，包括了 Web 应用程序存档、所有的源文件和版本属性文件。创建发布包后，文件被复制到备份服务器上。备份服务器可以通过一个挂载的共享驱动器来访问或者通过 FTP 传输文件。因为我不想使这个实例过于复杂，所以你只需要将其复制到 build/backup 子目录中就行。图 4.8 展示了你想要执行的 task 的顺序。

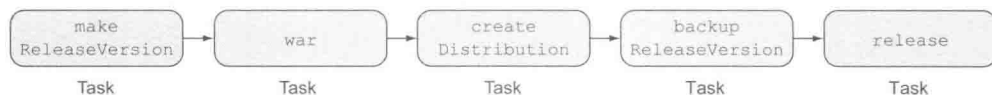


图 4.8 用于发布项目的 task 依赖关系

#### 使用 task 类型

Gradle 的内置 task 类型都是 DefaultTask 的派生类。因此，它们可以被构建脚本中的增强的 task 使用。Gradle 提供了广泛的 task 类型，但是在这个例子中只使用两个。下面的清单显示了在产品发布过程中用到的 task 类型 Zip 和 Copy。你可以在 DSL 指南中找到完整的 task 参考。

#### 清单 4.10 使用 task 类型来备份 ZIP 发布包

```
task createDistribution(type: Zip, dependsOn: makeReleaseVersion) {  
    from war.outputs.files  
  
    from(sourceSets*.allSource) {  
        into 'src'  
    }  
  
    from(rootDir) {  
        include versionFile.name  
    }  
}
```

隐式引用 War task 的输出

把所有源文件都放到 ZIP 文件的 src 目录下

为 ZIP 文件添加版本文件

```
task backupReleaseDistribution(type: Copy) {
    from createDistribution.outputs.files
    into "$buildDir/backup"
}

task release(dependsOn: backupReleaseDistribution) << {
    logger.quiet 'Releasing the project...'
}
```

← 隐式引用 createDistribution 的输出

在这个清单中，使用了不同的方式来告诉 Zip 和 Copy task 包括哪些文件以及它们被放到哪里了。这里使用的许多方法都继承自父类 AbstractCopyTask，如图 4.9 所示。想了解全部可用的 task 类型，请参考类的 Javadocs 文档。

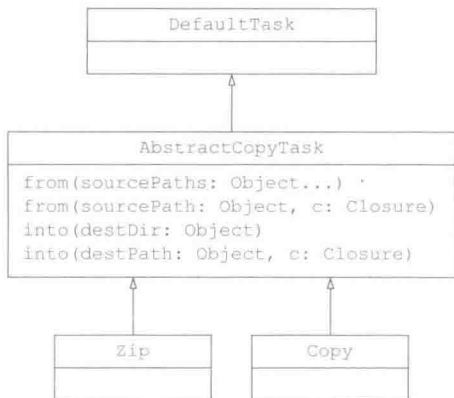


图 4.9 task 类型 Zip 和 Copy 的继承关系

你使用的 task 类型所提供的配置选项远多于这个实例中所显示的。同样的，想了解全部可用的配置选项，请参考 DSL 参考手册或者 Javadocs 文档。接下来，我们会深入地看看它们的 task 依赖。

### task 依赖推断

你可能已经注意到，清单中的两个 task 之间的依赖关系是通过 dependsOn 方法显式声明的。然而，一些 task 并不直接依赖其他 task（比如，createDistribution 对于 war）。Gradle 是如何知道事先执行依赖的 task 呢？其实通过使用一个 task 的输出作为另一个 task 的输入，Gradle 就可以推断出依赖关系了。因此，所依赖的 task 会自动运行。让我们来看看完整的 task 执行图：

```
$ gradle release
:makeReleaseVersion
:compileJava
:processResources UP-TO-DATE
:classes
:war
:createDistribution
:backupReleaseDistribution
:release
Releasing the project...
```

运行构建后,你可以在build/distributions目录下找到所生成的ZIP文件,这个目录是用于归档 task 的默认输出目录。通过设置 destinationDir 属性你可以很容易地指定一个不同的发布包输出目录。下面的目录树显示了构建所生成的相关工作件:

```

.
├── build
│   ├── backup
│   │   └── todo-webapp-0.1.zip
│   ├── distributions
│   │   └── todo-webapp-0.1.zip
│   └── libs
│       └── todo-webapp-0.1.war
├── build.gradle
├── src
└── version.properties

```

增量式构建支持内置的 task 类型。在不改变任何源文件的情况下,连续多次运行的 task 会被标记为最新的。接下来,你将了解到如何定义一个 task 的行为取决于灵活的 task 名称。

### 4.2.11 task 规则

有时候你可能会发现在某些情况下,自己所编写的多个 task 却做着类似的事情。例如,你想通过两个 task 扩展版本管理功能:一个用来增加项目的主版本,另一个对于次版本类别做同样的事情。假定这两个 task 都会将变化持续保存到版本文件中。如果你比较下面清单中这两个 task 的 doLast 行为,就会发现主要是复制了代码并应用了次版本的变化。

清单 4.11 声明 task 用来递增版本类别

```

task incrementMajorVersion(group: 'versioning', description: 'Increments
    ➡ project major version.') << {
    String currentVersion = version.toString()
    ++version.major
    String newVersion = version.toString()
    logger.info "Incrementing major project version: $currentVersion ->
        ➡ $newVersion"

    ant.propertyfile(file: versionFile) {
        entry(key: 'major', type: 'int', operation: '+', value: 1)
    }
}

task incrementMinorVersion(group: 'versioning', description: 'Increments
    ➡ project minor version.') << {
    String currentVersion = version.toString()
    ++version.minor
    String newVersion = version.toString()

```

使 用 Ant task  
的 propertyfile  
来增加属性文件  
中的特定属性

```
logger.info "Incrementing minor project version: $currentVersion ->
    $newVersion"

ant.propertyfile(file: versionFile) {
    entry(key: 'minor', type: 'int', operation: '+', value: 1)
}
```

使用 Ant task 的 propertyfile 来增加属性文件中的特定属性

如果你在一个版本为 0.1-SNAPSHOT 的项目上运行 `gradle incrementMajorVersion`, 将看到版本升到 1.1-SNAPSHOT。在 INTO 日志级别运行将会看到更详细的输出信息:

```
$ gradle incrementMajorVersion -i
:incrementMajorVersion
Incrementing major project version: 0.1-SNAPSHOT -> 1.1-SNAPSHOT
[ant:propertyfile] Updating property file: /Users/benjamin/books/
gradle-in-action/code/chapter4/task-rule/version.properties
```

两个独立的 task 工作得很好, 但是你肯定可以改进这个实现。最后, 你会对维护重复的代码失去兴趣。

### task 规则命名模式

Gradle 也引入了 task 规则的概念, 根据 task 名称模式执行特定的逻辑。该模式由两部分组成: task 名称的静态部分和一个占位符。它们联合起来就组成了一个动态的 task 名称。如果你想将 task 规则应用于前面的例子中, 那么命名模式看起来应该像这样: `increment<Classifier>Version`。在命令行上执行这个 task, 将以“驼峰命名法”(camel-case)指定类别占位符(比如, `incrementMajorVersion` 或 `incrementMinorVersion`)。

### task 规则实践

Gradle 的一些核心插件充分地利用了 task 规则。Java 插件定义的 task 规则之一是 `clean<TaskName>`, 用来删除指定 task 的输出。比如, 在命令行运行 `gradle cleanCompileJava` 会删除所有产品代码的 class 文件。

### 声明 task 规则

你只是了解了给 task 规则定义一个命名模式, 但是在构建脚本中应该如何声明一个 task 规则呢? 为了给项目添加 task 规则, 首先你需要获得对 `TaskContainer` 的引用。一旦有了这个引用, 你就可以调用 `addRule(String, Closure)` 方法了。第一个参数提供了描述信息(比如, task 命名模式), 第二个参数声明了要执行的闭包来应用规则。遗憾的是, 不能像简单的 task 一样直接通过 `Project` 的方法来创建 task 规则, 如图 4.10 所示。

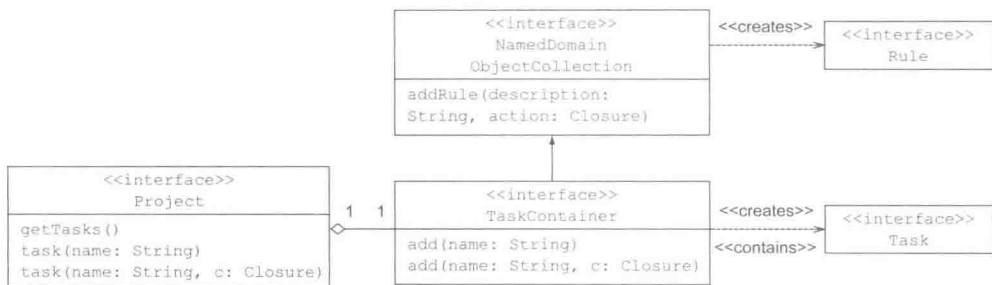


图 4.10 简单的 task 可以直接通过调用 project 实例的方法来添加。task 规则只能通过 task 容器来添加，所以你首先需要通过调用 getTasks() 方法来获取对它的引用。

对如何为项目添加 task 规则有了一个基本的了解后，你可以开始编写实际的闭包实现了。下面的清单演示了如何应用 task 规则使其变成富有表达性的工具来实现具有相似逻辑的 task 动作。

#### 清单 4.12 合并相似逻辑到 task 规则中

```

tasks.addRule("Pattern: increment<Classifier>Version - Increments the
    ➡ project version classifier.") { String taskName ->
    if(taskName.startsWith('increment') && taskName.endsWith('Version')) {
        task(taskName) << {
            String classifier = (taskName - 'increment' - 'Version')
            ➡ .toLowerCase()
            String currentVersion = version.toString()

            switch(classifier) {
                case 'major': ++version.major
                    break
                case 'minor': ++version.minor
                    break
                default: throw new GradleException("Invalid version
                    ➡ type '$classifier'. Allowed types: ['Major', 'Minor']")
            }

            String newVersion = version.toString()
            logger.info "Incrementing $classifier project version:
                ➡ $currentVersion -> $newVersion"

            ant.propertyfile(file: versionFile) {
                entry(key: classifier, type: 'int', operation: '+', value: 1)
            }
        }
    }
}

```

添加带有描述信息的 task 规则

从完整的 task 名称中提取类型字符串

给符合命名模式的 task 动态添加一个 doLast 方法

根据预定义模式检查 task 名称

在项目中添加 task 规则之后，你会发现当运行帮助 task tasks 时会列出一个具体的 task 组 Rules：

```
$ gradle tasks
...
Rules
-----
Pattern: increment<Classifier>Version - Increments project version type
```

task 规则不能像处理任何其他的简单的 task 或增强的 task 一样被独立分组。task 规则即使通过插件声明了，它也将永远显示在 Rules 组下。

#### 4.2.12 在 buildSrc 目录下构建代码

你已经看到了如何快速地构建脚本代码。在这一章中，你已经在构建脚本中创建了两个 Groovy 类：ProjectVersion 和自定义 task ReleaseVersionTask。这些类最适合被移动到项目的 buildSrc 目录下。将构建代码放在 buildSrc 目录下是一个良好的软件开发实践。你可以应用在其他项目中习惯使用的方式来结构化代码，甚至是编写测试代码。

Gradle 在 buildSrc 目录下使源文件结构标准化。Java 代码需要放在 src/main/java 目录下，Groovy 代码则放在 src/main/groovy 目录下。位于这些目录下的代码都会被自动编译，并且被加入到 Gradle 构建脚本的 classpath 中。buildSrc 目录是组织代码的最佳方式。因为要处理这些类，所以也可以把它们放到指定的包下。可以让它们成为 com.manning.gia 包的一部分。下面的目录结构显示了 Groovy 类的存放位置：

```
.
├── build.gradle
├── buildSrc
│   ├── src
│   │   ├── main
│   │   │   ├── groovy
│   │   │   │   ├── com
│   │   │   │   │   ├── manning
│   │   │   │   │   │   ├── gia
│   │   │   │   │   │   │   ├── ProjectVersion.groovy
│   │   │   │   │   │   │   └── ReleaseVersionTask.groovy
│   └── ...
└── version.properties
```

记住，提取类放到源文件中需要一些额外的工作。在构建脚本中定义一个类和一个独立的源文件的区别在于你需要从 Gradle API 导入类。下面的代码片段显示了包和导入的自定义 task ReleaseVersionTask 的声明：

```
package com.manning.gia

import org.gradle.api.DefaultTask
import org.gradle.api.tasks.Input
```



```
import org.gradle.api.tasks.OutputFile
import org.gradle.api.tasks.TaskAction

class ReleaseVersionTask extends DefaultTask {
    (...)
}
```

相应地，构建脚本需要从 `buildSrc` 导入编译过的类（比如，`com.manning.gia.ReleaseVersionTask`）。下面的控制台输出显示了在命令行调用这个 `task` 之前运行编译 `task`：

```
$ gradle makeReleaseVersion
:buildSrc:compileJava UP-TO-DATE
:buildSrc:compileGroovy
:buildSrc:processResources UP-TO-DATE
:buildSrc:classes
:buildSrc:jar
:buildSrc:assemble
:buildSrc:compileTestJava UP-TO-DATE
:buildSrc:compileTestGroovy UP-TO-DATE
:buildSrc:processTestResources UP-TO-DATE
:buildSrc:testClasses UP-TO-DATE
:buildSrc:test
:buildSrc:check
:buildSrc:build
:makeReleaseVersion UP-TO-DATE
```

`buildSrc` 目录被视为 Gradle 项目的指定路径。由于没有编写任何单元测试，所以跳过了编译和执行 `task`。第 7 章将完全致力于在 `buildSrc` 中编写测试类。

在前面章节中，你了解了简单的 `task` 工作的底细、自定义的 `task` 类和 Gradle API 提供的特定的 `task` 类型。我们学习了 `task` 动作和配置代码的区别，以及它们相应的用例，其中一个要点是 `task` 动作和配置代码在构建生命周期的不同阶段执行。本章后续部分将讨论当特定的生命周期事件被触发时如何编写执行代码。

## 4.3 挂接到构建生命周期

作为一个构建脚本开发人员，不能仅限于编写在不同的构建阶段执行的 `task` 动作或者配置逻辑。有时候当一个特定的生命周期事件发生时你可能想要执行代码。一个生命周期事件可能发生在某个构建阶段之前、期间或者之后。在执行阶段之后发生的生命周期事件是构建的完成。

假设你想在开发周期中尽可能早地获得失败构建的反馈信息。对失败构建一个典型的反应是发送邮件给团队中的所有开发人员，以使代码恢复正常。有两种方式可以编写回调生命周期事件：在闭包中，或者通过 Gradle API 所提供的监听器接口实现。Gradle 不会引导你采用哪种方式去监听生命周期事件，这完全取决于你的

选择。采用监听器实现最大的优势在于你处理的类通过编写单元测试是完全可测试的。下面为你提供有一个有用的生命周期钩子（hook）的想法，如图 4.11 所示。

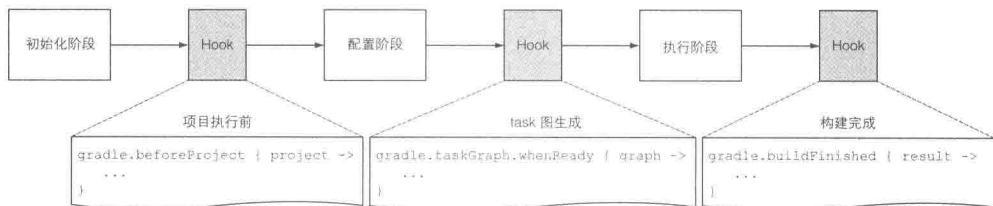


图 4.11 构建生命周期钩子示例

所有可用的生命周期钩子扩展列表超出了本书的讨论范围。许多生命周期回调方法被定义在 Project 和 Gradle 接口中。Gradle 的 Javadocs 为你的用例提供了合适的回调方法。

不要害怕使用生命周期钩子。它们不是 Gradle API 的秘密后门。相反，它们是特意提供的，因为 Gradle 无法预测企业级构建的需求。

在接下来的两节中，我将演示 task 执行图生成以后如何立即接收通知。为了完全理解构建 task 执行图时在底层发生了什么，我们首先来看看 Gradle 内部是如何运作的。

### Task 执行图的内部展示

在配置时，Gradle 决定了在执行阶段要运行的 task 的顺序。正如第 1 章所提到的，表示依赖关系的内部结构被建模为一个有向无环图（DAG）。图中的每个 task 被称为一个节点，并且每个节点通过有向边连接起来。你最有可能通过声明 dependsOn 关系或者通过利用隐式 task 依赖干预机制来创建这些节点之间的连接。重要的是要注意 DAG 没有一个闭环。换句话说，一个先前执行的 task 将永远不会被再次执行。图 4.12 展示了早期发布过程建模的 DAG 表示。

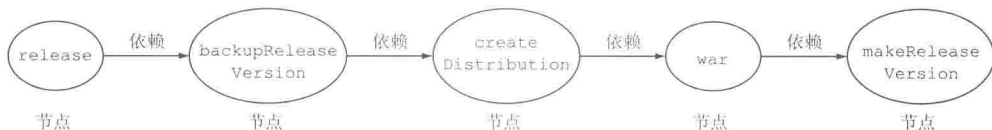


图 4.12 task 依赖关系通过 DAG 表示出来

现在，对 task 图的内部表示有了更好的了解，你可以在构建脚本中编写一些代码来应对它了。

### 4.3.1 挂接到 task 执行图

回想一下你所实现的 `makeReleaseVersion` task，它作为 `release` task 的依赖被自动执行。除了编写一个 task 改变项目版本来指示产品准备发布外，你也可以通过编写一个生命周期钩子来实现同样的目标。因为构建确切地知道 task 在得到执行之前哪些将参与构建，你可以查询 task 图来检查它的存在。图 4.13 显示了访问 task 执行图的相关接口及其方法。

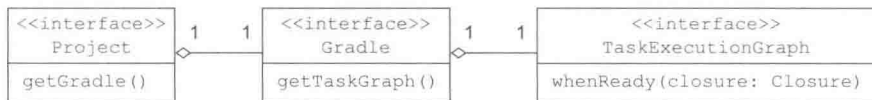


图 4.13 当 task 执行图生成完成后，TaskExecutionGraph 提供的 `whenReady` 方法会被调用

接下来你将在适当的地方放置生命周期钩子。在清单 4.13 中通过调用 `whenReady` 方法注册一个闭包扩展了构建脚本，task 图生成完成后，它将立即被执行。因为你知道在执行图中任何 task 之前会运行逻辑行为，所以可以完全删除 `makeReleaseVersion` task 和省略 `createDistribution` 中的 `dependsOn` 声明。

清单 4.13 通过生命周期钩子实现发布版本功能

```

gradle.taskGraph.whenReady { TaskExecutionGraph taskGraph ->
    if(taskGraph.hasTask(release)) {
        if(!version.release) {
            version.release = true
            ant.propertyfile(file: versionFile) {
                entry(key: 'release', type: 'string', operation: '=',
                    ➡ value: 'true')
            }
        }
    }
}

```

查看执行图中是否包含 task release

注册的生命周期钩子在 task 图生成后被调用

或者，用监听器的方式来实现这个逻辑。下面我们看看是如何实现的。

### 4.3.2 实现 task 执行图监听器

通过监听器挂接到构建生命周期只需两个简单的步骤。首先，通过在构建脚本中编写一个类来实现特定的监听器接口。其次，注册监听器实现。

用于监听 task 执行图事件的接口是由 `TaskExecutionGraphListener` 接口提供的。在写作本书的时候，你只需要实现一个方法：`graphPopulate(TaskExecutionGraph)`。图 4.14 显示了监听器实现 `ReleaseVersionListener` 的方式。

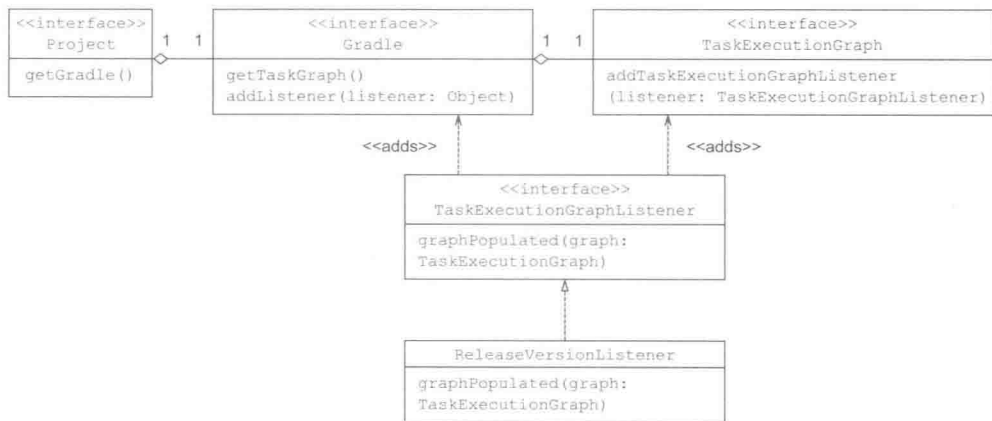


图 4.14 注册 TaskExecutionGraphListener 的方式。监听器可以通过通用的 addListener 方法或者指定监听器类型实例的方法进行注册。

记住，如果将监听器添加到构建脚本中，那么你就不能直接访问 Project 实例了。但是，你可以充分地使用 Gradle 的 API。下面的清单显示了如何通过调用 release task 上的 getProject() 方法来访问 project。

#### 清单 4.14 通过生命周期监听器实现发布版本功能

```

class ReleaseVersionListener implements TaskExecutionGraphListener {
    final static String releaseTaskPath = ':release'

    @Override
    void graphPopulated(TaskExecutionGraph taskGraph) {
        if(taskGraph.hasTask(releaseTaskPath)) {
            List<Task> allTasks = taskGraph.allTasks
            Task releaseTask = allTasks.find {it.path == releaseTaskPath }
            Project project = releaseTask.project

            if(!project.version.release) {
                project.version.release = true
                project.ant.propertyfile(file: project.versionFile) {
                    entry(key: 'release', type: 'string', operation: '=',
                        value: 'true')
                }
            }
        }
    }
}

def releaseVersionListener = new ReleaseVersionListener()
gradle.taskGraph.addTaskExecutionGraphListener(releaseVersionListener)

```

在执行图中从一系列 task 中过滤 release task

确定 release task 是否包含在执行图中

每个 task 都知道自己所属的 project

注册监听器到 task 执行图

并不局限于在构建脚本中注册生命周期监听器。在任何 task 执行之前，都可以应用生命周期逻辑来监听 Gradle 事件。在下一节中，我们将通过初始化脚本定制构建环境来探讨挂接到生命周期。

### 4.3.3 初始化构建环境

假设你想要在构建完成后获得构建结果。当构建完成后，你想知道构建成功了还是失败了。你也希望能够确定执行了多少 task。Gradle 的核心插件之一，build-announcements 插件，提供了一种将通告发送到本地通知系统如 Snarl

(Windows) 或 Growl (Mac OS X) 的方式。这个插件根据你的操作系统自动选择正确的通知系统。如图 4.15 所示为 Growl 显示的通知。



图 4.15 在 Mac OS X 上 Growl 显示的构建通告

你可以把这个插件独立地运用到每个项目上，但是为什么不使用 Gradle 提供的强大机制呢？初始化脚本会在任何构建脚本逻辑解析执行之前运行。编写一个初始化脚本把插件应用到项目中，不需要人工干预。在 `<USER_HOME>/gradle/init.d` 下创建初始化脚本，如下面的目录树所示：

```
.
├── .gradle
│   └── init.d
│       └── build-announcements.gradle
```

Gradle 会执行在 `init.d` 下以 `.gradle` 为扩展名的所有初始化脚本。因为想在任何构建脚本代码执行之前使用插件，所以我们选择使用最合适生命周期回调方法来处理这种情况：`Gradle#projectsLoaded(Closure)`。下面的代码片段显示了如何将 `build-announcements` 插件应用到构建的根项目中：

```
gradle.projectsLoaded { Gradle gradle ->
    gradle.rootProject {
        apply plugin: 'build-announcements'
    }
}
```

← 当构建项目创建好时执行闭包

请注意，一些生命周期事件只有在适当的位置声明才会发生。比如，如果将生命周期挂接闭包 `Gradle#projectsLoaded(Closure)` 声明在 `build.gradle` 文件中，那么将不会发生这个事件，因为项目创建发生在初始化阶段。

## 4.4 总结

每个 Gradle 构建脚本都包含两个基本的构建块：一个或多个项目（project）和任务（task）。这两个元素都与 Gradle 的 API 密切相关，并且它都有一个直接类表示。在运行时，Gradle 会通过构建定义创建一个模型，将它存储在内存中，并且可以通过方法访问。你了解到属性是控制构建动作的一种方式。项目暴露了标准的属性。除此之外，你也可以在 Gradle 的领域模型对象上定义额外的属性（比如，在项目和任务级别）来声明任意的用户数据。

在本章后部分，你了解了 task 的底细。作为一个例子，你实现了构建逻辑来控制存储在外部属性文件中的项目版本编号方案。你首先将简单的 task 添加到构建脚本中。构建逻辑可以被直接定义在 task 的动作闭包中。每个 task 都继承自 `org.gradle.api.DefaultTask` 类。因此，它通过父类的方法加载了很多功能。

对初学者而言，理解构建生命周期及其阶段执行顺序是至关重要的。Gradle 明确区分了 task 动作和 task 配置。task 动作，通过闭包 `doFirst` 和 `doLast` 或者简写的 `<<` 定义，在执行阶段运行。那些不在 task 动作中定义的代码被认为是配置，因此在配置阶段提前执行。

接下来，我们将注意力转向实现非功能性需求：构建执行性能、代码可维护性和可重用性上。通过声明输入和输出数据，对已有的 task 实现添加增量式构建支持。在初始的和后续的构建 task 之间如果输入数据没有发生变化，那么执行将被跳过。实现增量式构建支持其实非常简单和容易。如果做得正确，它可以极大地减少构建的执行时间。复杂的构建逻辑最好在自定义的 task 类中实现结构化，这可以给你带来面向对象编程的所有好处。你练习编写了一个自定义的 task 类，用来将现有的逻辑移植到 `DefaultTask` 实现中。通过移动可编译的代码，你也可以清理 `buildSrc` 目录下的构建脚本。Gradle 提供了广泛的可重用的 task 类型如 `Zip` 和 `Copy`。通过建模一个 task 依赖关系链合并这两种类型来发布项目。

访问 Gradle 的内部并不局限于模型。你可以注册构建生命周期钩子（hook），当触发目标事件时来执行代码。作为一个例子，通过闭包和监听器实现，你编写了 task 执行图生命周期钩子。通过初始化脚本可以应用公共代码，比如构建中的生命周期监听器。

你已经初步了解了声明对外部类库的依赖机制。在下一章中，我们将深入讨论如何处理依赖关系，以及依赖冲突到底是如何解决的。

# 5 依赖管理

## 本章涵盖

- 理解自动化依赖管理
- 声明和组织依赖
- 了解各种类型的仓库
- 了解和调整本地缓存
- 依赖报告 and 解决版本冲突

在第 3 章中，你学习了如何声明一个 Servlet API 依赖为 To Do 应用程序实现 Web 组件。Gradle 的 DSL 配置闭包使得声明依赖和从仓库中获取依赖变得很容易。首先，需要使用 `dependencies` 脚本来定义构建所依赖的类库。其次，使用 `repositories` 闭包告诉构建从哪里获取依赖。有了这些信息，Gradle 会自动地解决依赖关系，下载所需要的依赖，将它们存储在本地缓存中，并且在构建中使用它们。

本章涵盖了 Gradle 对依赖管理的强力支持。我们将仔细看看关键的 DSL 配置元素，用于对依赖进行分组和针对不同类型的仓库。

依赖管理听起来很容易，但是如果遇到依赖冲突，就会变得非常麻烦。传递性依赖，声明的依赖需要依赖其他的依赖关系，是一把双刃剑，有利有弊。过于复杂

的依赖关系图可能会引起多个版本之间依赖的混乱，导致不可靠。Gradle 为分析依赖关系提供了依赖报告。这样你就能知道依赖的来源并解决不同版本之间的依赖冲突。

Gradle 有它自己的依赖管理实现。Gradle 摒弃了像 Ivy 和 Maven 这样的依赖管理工具的缺点，Gradle 注重性能、构建可靠性以及可重复性。

## 5.1 依赖管理概述

几乎所有基于 JVM 的软件项目都需要依赖外部类库来重用现有的功能。比如，如果你正在开发一个基于 Web 的项目，则很可能需要依赖一个流行的开源框架如 Spring MVC 或 Play 来提高开发效率。Java 类库通常以 JAR 文件的形式存在。JAR 文件规范不要求你指定类库版本。然而，将版本号附加到 JAR 文件名上来标识一个特定的发布版本（比如，spring-web-3.1.3.RELEASE.jar）是常见的做法。随着项目由小变大，项目所依赖的模块和第三方类库会越来越多。组织和管理好 JAR 文件显得至关重要。

### 5.1.1 不完善的依赖管理技术

因为 Java 语言没有提供或建议使用任何工具来管理版本依赖，项目团体将不得不想出自己的办法来保存和获取依赖。你可能遇到过以下常见的做法：

- 手动将 JAR 文件拷贝到开发机器上。这种处理依赖关系是最原始、非自动化而且最容易出错的方式。
- 使用 JAR 文件共享存储器（比如，一个共享网络驱动器的文件夹），它被安装在开发人员的机器上，或者通过 FTP 检索二进制文件。这种方式需要开发人员建立到二进制仓库的连接。新的依赖关系需要在有写权限和访问许可的情况下手动进行添加。
- 检查所下载的 JAR 文件，使其和版本控制系统中的项目源代码符合。这种方法不需要任何额外的设置，只需要将源代码和依赖绑定在一起。当更新仓库中的本地副本时，整个团队都可以检索到变化。缺点是一些不必要的二进制文件占用了仓库的大量空间。只要源代码发生变化，类库副本的变化就需要频繁地检入。如果你正在开发的多个项目彼此互相依赖，就更是如此了。

### 5.1.2 自动化依赖管理的重要性

尽管所有这些方法都能工作，但绝不是最佳解决方案，因为它们都没有提供一种标准化的方式来命名和组织 JAR 文件。至少你需要知道类库的确切版本和传递性



依赖。为什么这如此重要？

### 知道依赖的确切版本

使用项目时，如果你不清楚项目依赖的版本，那么项目的维护将极其困难。如果没有精心记录，你永远无法确定项目所依赖的类库版本实际支持哪些特性。将类库升级到一个新版本就会变成一个猜谜游戏，因为你不知道是从哪个版本升级的，甚至有可能降低了某些类库的版本。

### 管理传递性依赖

传递性依要在项目开发的早期阶段就需要关注。这些一级依赖类库是项目正常工作的保障。流行的 Java 开发栈像 Spring 和 Hibernate 的组合可以很容易地从一开始就引入 20 多个附加的类库。一个类库可能需要许多其他类库才能正常工作。图 5.1 显示了 Hibernate 的核心类库的依赖图。

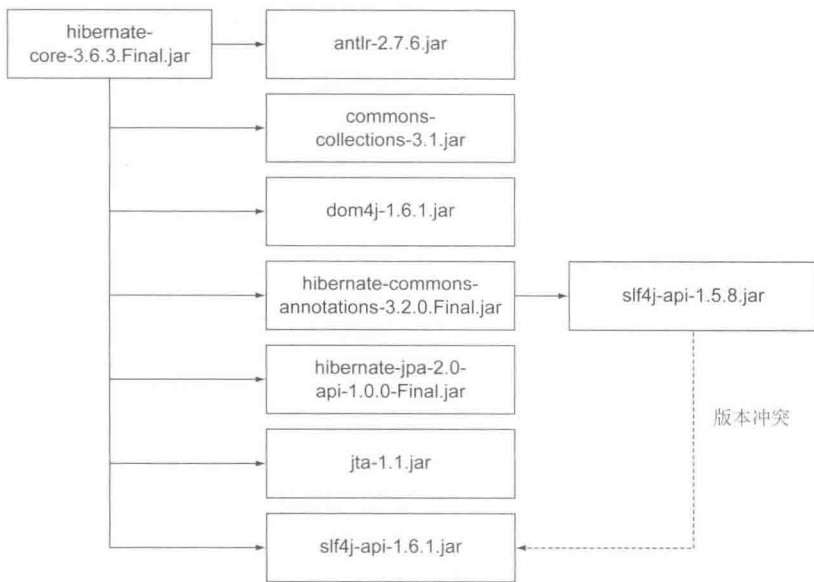


图 5.1 Hibernate 核心类库的依赖图

尝试手动确定对一个特定类库的所有依赖性传递是非常困难的。通常，在类库的文档中这种信息无处可寻，最后徒劳无功，还是找不到确切的依赖关系。结果是，你可能遇到像编译错误和运行时类加载错误的问题。

我想我们需要一个更完善的解决方案来管理依赖。最好能够声明依赖关系和各自的版本作为项目元数据。作为自动化过程的一部分，这些依赖可以从中心位置获取，并且安装到项目中。让我们看看现有的支持这些特性的开源解决方案。

### 5.1.3 使用自动化依赖管理

在 Java 世界里，支持声明和自动化管理依赖的主要占主导地位的两个项目是：Apache Ivy，一个纯粹的依赖管理器，常常与 Ant 项目结合使用，以及 Maven，包含有一个依赖管理器作为构建基础环境的一部分。我们不会深入到这些解决方案的细节中。本节的目的解释自动化依赖管理的概念和机制。

在 Ivy 和 Maven 中，依赖配置是通过 XML 描述符文件来表达的。配置由两部分组成：依赖标识符加各自的版本和二进制仓库的地址（比如，依赖仓库的 HTTP 地址）。依赖管理器解析这些信息并自动从目标仓库中把依赖下载到本地机器上。类库可以定义传递性依赖作为元数据的一部分。依赖管理器在获取依赖的过程中能够分析这些信息并解析依赖性。如果依赖的版本发生冲突，正如 Hibernate 核心类库例子所展示的那样，依赖管理器将尝试解决冲突。一旦下载，类库就会存储在本地缓存中。现在，所配置类库在开发机器上是可用的，可以在构建中使用了。后续构建会首先检查本地缓存中的类库，以避免对仓库的不必要请求。图 5.2 展示了自动化依赖管理的关键要素。

本地机器

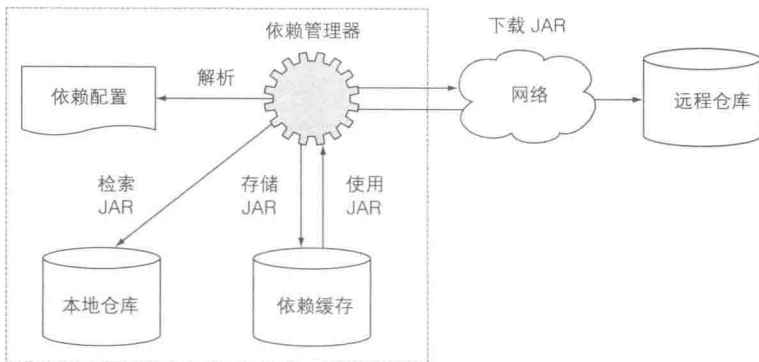


图 5.2 剖析自动化依赖管理

使用依赖管理器让你摆脱了手动复制或组织 JAR 文件的困扰。Gradle 提供了一个强大的适合上述架构的依赖管理实现。它通过 Gradle 的具有表述性的 DSL 来描述依赖配置，支持传递性依赖管理，并且与已有的仓库结合得很好。在深入细节之前，我们来看看针对依赖管理你可能面对的一些挑战，以及如何应对它们。

### 5.1.4 自动化依赖管理的挑战

尽管依赖管理大大简化了对外部类库的处理，但是有时候你会发现处理的某种不足可能会损害构建的可靠性和可重复性。

## 潜在不可用的中央托管仓库

这种情况在依赖于开源类库的企业级软件中不足为奇。许多这样的项目发布版本到中央托管仓库中。最广泛使用的仓库之一是 **Maven Central**。如果构建只依赖于 **Maven Central**，你的系统就可能会出现单点故障。万一仓库连接失败，而在本地缓存中又没有所需要的依赖，就有可能出现错误。

为了避免出现这种情况，你可以配置构建使用自定义的内部仓库，让你完全控制服务器的可用性。如果你想要进一步了解，可以直接跳到第 14 章，其中将讨论如何设置和使用开源的和商业的仓库管理器如 **Sonatype Nexus** 和 **JFrog** 的 **Artifactory**。

## 坏元数据和依赖缺失

前面我们提到元数据被用来声明传递性依赖。依赖管理器分析这些信息，建立依赖图，并且帮助你解决了所有的嵌套依赖关系。使用传递性依赖管理能够节省很多时间，而且具有可追溯性。

遗憾的是，元数据和仓库都不能能保证元数据中所声明的工件实际存在、被正确定义甚至是所需要的。你可能会遇到依赖缺失这样的问题，尤其是那些没有质量保证的仓库，对于 **Maven Central** 这是已知的问题。图 5.3 展示了 **Maven** 仓库中工件开发和使用的生命周期。

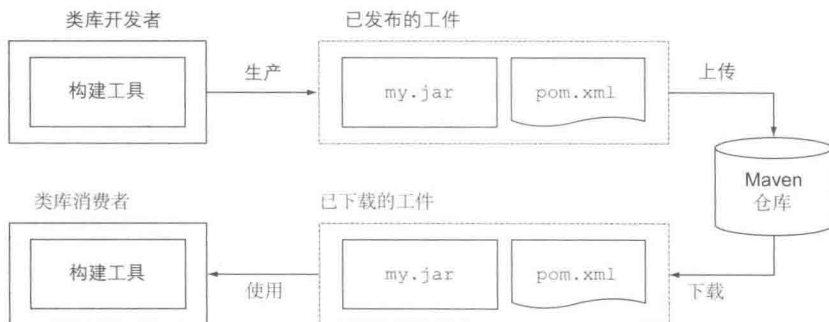


图 5.3 坏元数据让传递性依赖的使用变得更复杂。在 **Maven** 仓库中依赖元数据通过项目对象模型 (POM) 文件来表示。如果类库的开发者提供了不正确的元数据，那么使用者将会继承这个错误。

**Gradle** 允许在依赖图的任何阶段除去传递性依赖。或者，你也可以忽略它所提供的元数据，并且设置自己的传递性依赖定义。

你会发现在不同版本的传递性依赖图中都有流行类库。这通常出现在一些常用功能上，比如日志框架。依赖管理器试图找到一个智能的解决方案，根据某种解决策略选择一个合适的版本以避免版本冲突。当然有时候你需要调整选择。为此，你就可以知道哪些依赖引入了传递性依赖的哪个版本。**Gradle** 提供了很有价值的依赖报告来解决这些问题。稍后，我们将仔细看看这些报告。现在我们通过一个完整的

例子来看看 Gradle 是如何实现这些想法的。

## 5.2 通过例子学习依赖管理

在第 3 章中，我们了解了如何使用 Jetty 插件将 To Do 应用程序部署到一个嵌入式的 Jetty Servlet 容器中。Jetty 是一个在开发期间使用非常方便的容器。凭借 Jetty 的轻量级容器实现，它提供了快速的启动时间。许多企业在产品环境下使用其他的 Web 应用程序容器。假设你想要让构建支持将 Web 应用程序部署到不同的容器中，比如 Apache Tomcat。

开源项目 Cargo (<http://cargo.codehaus.org/>) 提供了支持将 Web 应用程序部署到各种 Servlet 容器和应用程序服务器中的功能。在构建项目中，Cargo 支持两种实现。一方面，你可以使用 Java API，细粒度地访问 Cargo 配置的每个方面。另一方面，你可以选择执行一组预配置的 Ant task，其封装了 Java API。因为 Gradle 能够与 Ant 很好地集成，所以我们的例子将基于 Cargo Ant task 来演示。

让我们重温一下图 5.1，看看这些组件在 Gradle 用例上下文中是如何变化的。在第 3 章中你了解了项目的依赖管理是通过 dependencies 和 repositories 两个 DSL 配置块来配置的。配置块的名称直接映射到 Project 接口的方法。在用例中，你将使用 Maven Central，因为它不需做任何额外的设置。图 5.4 显示了在 build.gradle 文件中通过 Gradle 的 DSL 提供的依赖定义。依赖管理器在运行时执行这个配置，从中央仓库下载所需要的工件，并将它们存储在本地缓存中。因为你没有使用本地仓库，所以在图中没有显示。

本地机器

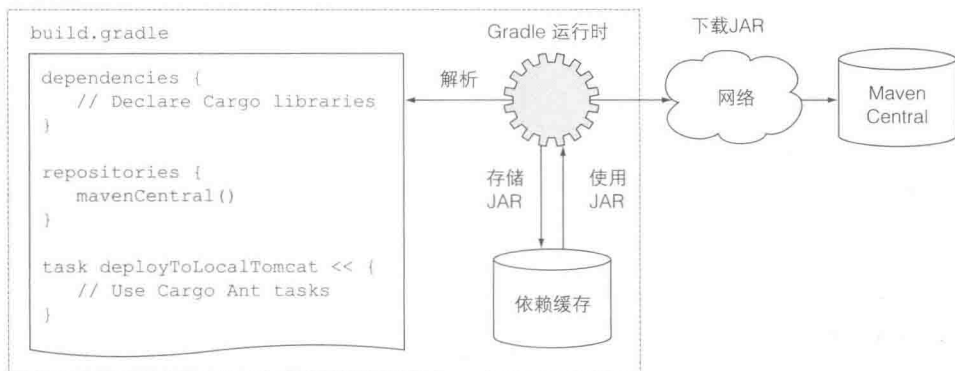


图 5.4 在 Gradle 构建中声明依赖 Cargo 类库

本章后续部分将讨论 Gradle 构建脚本的每个配置元素。你不仅会学到如何将它们应用到 Cargo 例子中，而且还将了解如何应用依赖管理来实现项目需求。首先让

我们来看一个在例子上下文中越来越重要的概念：依赖配置。

## 5.3 依赖配置

在第3章中，你看到了插件可以引入配置来定义依赖的作用域。Java 插件引入了各种标准配置来定义 Java 构建生命周期所应用的依赖。例如，通过 `compile` 配置添加编译产品源代码所需的依赖。在 Web 应用程序的构建中，你使用了 `compile` 配置来声明依赖 Apache Commons Lang 类库。为了更好地理解配置信息是如何存储、配置和访问的，让我们看看 Gradle 的 API 中的负责接口。

### 5.3.1 理解配置 API 表示

配置可以直接在项目的根级别添加和访问；你可以使用插件所提供的配置，或者声明自己的配置。每个项目都有一个 `ConfigurationContainer` 类的容器来管理相应的配置。配置在行为方面可以表现得很灵活。你可以控制依赖解决方案中是否包含传递性依赖，定义解决策略（例如，如何解决工件版本冲突），甚至可以使配置扩展。图 5.5 显示了相关的 Gradle API 接口和方法。

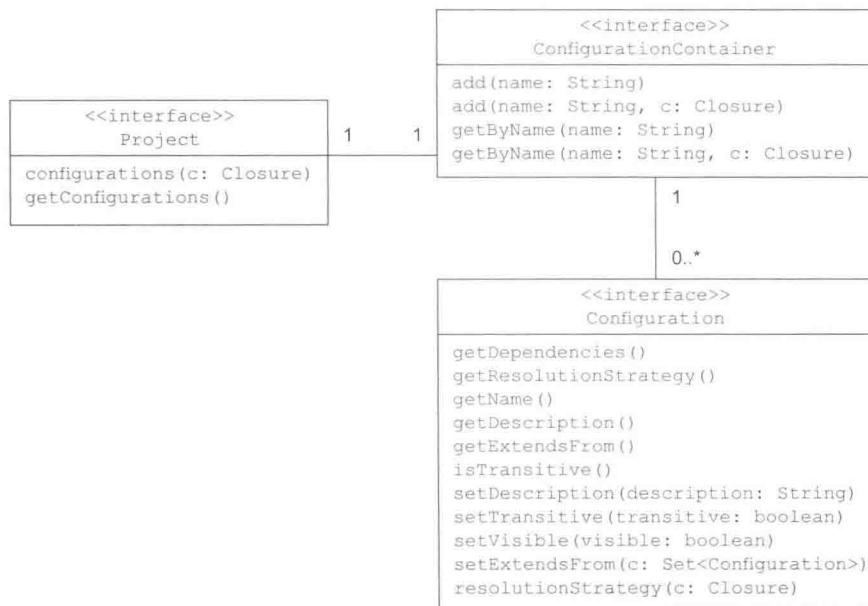


图 5.5 通过 Project 实例添加和访问配置

考虑配置的另一种方式是按照逻辑分组。通过配置分组依赖与 Java 类分包的概

念相似。包针对其所包含的类提供了唯一的命名空间，这同样适用于配置。通过逻辑分组把职责相同的配置放到一起。

Java 插件提供了 6 个现成的配置：`compile`、`runtime`、`testCompile`、`testRuntime`、`archives` 和 `default`。能不能只使用其中一个配置来声明依赖 Cargo 类库呢？通常可以，但会混淆相关的应用程序代码和为部署应用程序所编写的基础环境代码之间的依赖。在运行时将不必要的类库添加到发布包中可能导致不可预见的副作用，最好是避免发生这种情况。例如，使用 `compile` 配置将导致 WAR 文件中包含 Cargo 类库。接下来，我将展示如何自定义配置 Cargo 类库。

### 5.3.2 自定义配置

为了明确 Cargo 所需的依赖，你需要声明一个名为 `cargo` 的新配置，如下面的清单所示。

清单 5.1 定义 Cargo 类库的配置

```
configurations {  
    cargo {  
        description = 'Classpath for Cargo Ant tasks.'  
        visible = false  
    }  
}
```

通过名称定义新的配置

设置配置的描述信息和可见性

现在，处理的只是单个 Gradle 项目。对于这个项目限制配置的可见性是明智的选择，为多项目设置做好准备。如果你了解更多的关于多项目构建的信息，请查看第 6 章。不要让不必要的配置涉及其他项目。当你列出项目的依赖时，描述信息会直接反映出来：

```
$ gradle dependencies  
:dependencies
```

```
-----  
Root project  
-----
```

```
cargo - Classpath for Cargo Ant tasks.  
No dependencies
```

为项目的配置容器添加一个配置后，可以直接通过名称来访问。接下来，将使用 `cargo` 配置使构建脚本可以访问第三方 Cargo Anttask。

### 5.3.3 访问配置

从本质上讲, Ant task 是 Java 类, 遵循 Ant 的扩展端点来定义自定义逻辑。为了添加一个非标准的 Ant task, 如 Cargo 部署 task, 你需要使用 Taskdef Ant task。为了解决 Ant task 实现类, Cargo 的 JAR 文件中包含了需要分配的实现类。如下清单显示了通过名称访问配置是多么容易啊。这个 task 使用的依赖很明确, 并将它们指定给 Cargo Ant task 的 classpath 路径。

清单 5.2 通过名称访问 cargo 配置

```
task deployToLocalTomcat << {
    FileTree cargoDeps = configurations.getByName('cargo').asFileTree
    ant.taskdef(resource: 'cargo.tasks', classpath: cargoDeps.asPath)

    ant.cargo(containerId: 'tomcat7x', action: 'run',
               output: "$buildDir/output.log") {
        configuration {
            deployable(type: 'war', file: 'todo.war')
        }

        zipUrlInstaller(installUrl: 'http://archive.apache.org/dist/
                               tomcat/tomcat-7/v7.0.32/bin/
                               apache-tomcat-7.0.32.zip')
    }
}
```

以文件树的方式获取 cargo 配置的所有依赖

使用 Cargo Ant task 来自动化下载 Tomcat 7 发布包, 部署 WAR 文件, 并在容器中运行

使用完全限定依赖的连接路径来解决 Cargo Ant task 的定义

如果你不理解代码示例中的内容也不用着急。最重要的是认识到 Gradle API 方法允许你访问配置。其余的代码主要是通过 Gradle 的 DSL 来表达 Ant 的特定配置。第 9 章给出了在 Gradle 中使用 Ant task 的内幕。随着部署 task 被设置完成, 现在时候把 Cargo 依赖指派给 cargo 配置了。

## 5.4 声明依赖

第 3 章让我们了解到项目正常运行需要外部类库。DSL 配置块 dependencies 通常用来将一个或多个依赖指派给配置。外部依赖并不是为项目声明的唯一依赖。表 5.1 中给出了各种类型的依赖概述。在本书中我们将讨论和应用其中的许多依赖类型。一些依赖类型将在本章中介绍, 另一些依赖类型在其他章节中介绍更有意义。这个表涉及了多个用例。

表 5.1 Gradle 项目的依赖类型

类 型	描 述	了解更多信息
外部模块依赖	依赖仓库中的外部类库，包括它所提供的元数据	5.4.2 节
项目依赖	依赖其他 Gradle 项目	6.3.3 节
文件依赖	依赖文件系统中的一系列文件	5.4.3 节
客户端模块依赖	依赖仓库中的外部类库，具有声明元数据的能力	没有涵盖的，请参考在线手册
Gradle 运行时依赖	依赖 Gradle API 或者封装 Gradle 运行时的类库	8.5.7 节

在这一章中，我们将介绍外部模块依赖和文件依赖，但是首先看看在 Gradle API 中依赖支持是如何表现的。

### 5.4.1 理解依赖 API 表示

每个 Gradle 项目都有依赖处理器实例，由 `DependencyHandler` 接口来表现。通过使用项目的 `getter` 方法 `getDependencies()` 来获得对依赖处理器的引用。表 5.1 中列出的每种依赖类型都是通过项目的 `dependencies` 配置块中的依赖处理器的方法来声明的。每个依赖都是 `Dependency` 类型的一个实例。`group`、`name`、`version` 和 `classifier` 属性明确地标识了一个依赖。图 5.6 展示了项目、依赖处理器和真实依赖之间的关系。

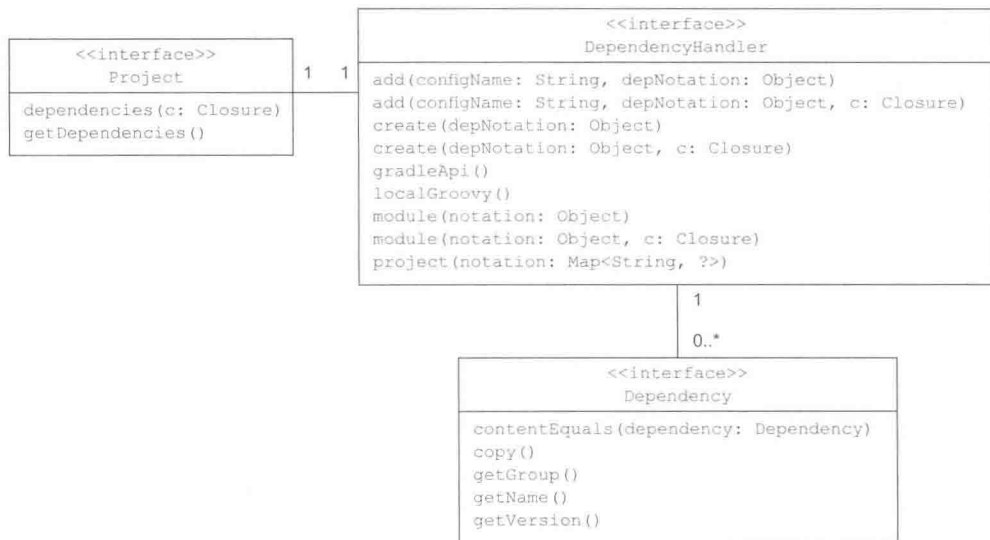


图 5.6 在项目级别添加的不同类型的依赖



我们首先来看看如何声明外部模块依赖、它们的标记方法，以及如何配置它们来满足需求。

## 5.4.2 外部模块依赖

在 Gradle 术语中，外部类库通常以 JAR 文件的形式存在，被称为外部模块依赖。它们代表依赖的模块在项目结构之外。这种类型依赖的特点是在仓库中能够通过属性明确地标识。在接下来的内容中，我们将讨论每一个属性。

### 依赖属性

当依赖管理器在仓库中查找一个依赖时，会通过属性组合来找到它。一个依赖至少需要提供一个名称。我们来回顾一下在 5.1.2 节通过 Hibernate 核心库介绍的依赖属性：

- `group`：这个属性通常用来标识一个组织、公司或者项目。`group` 可以使用点标记法，但不是强制性的。例如 Hibernate 库，`group` 是 `org.hibernate`。
- `name`：一个工件的名称唯一地描述了依赖。Hibernate 核心库的名字就是 `hibernate-core`。
- `version`：一个类库可以有很多版本。版本字符串一般包含主版本和次版本。比如选择 Hibernate 核心库的版本是 `3.6.3-Final`。
- `classifier`：有时一个工件也定义了另一个属性，即 `classifier`，用来区分具有相同 `group`、`name` 和 `version` 属性的工件，但它需要进一步规范（例如，运行时环境）。Hibernate 核心库没有提供 `classifier`。

我们已经回顾了一些依赖属性，现在来进一步看看 Gradle 是如何在构建脚本中声明这些依赖属性的。

### 依赖标记

可以在项目中使用如下语法来声明依赖：

```
dependencies {  
    configurationName dependencyNotation1, dependencyNotation2, ...  
}
```

首先给出配置名称以便指派依赖，然后设置一系列依赖标记。依赖标记分为两种。使用 `map` 形式给出属性名称及其值，或者以字符串形式使用快捷标记，用冒号分隔每一个属性（见图 5.7）。我们通过例子来看看这两种标记方法。

org.hibernate:hibernate-core:3.6.3-Final

group                      name                      version

图 5.7 依赖属性的快捷标记

定义配置之后，你可以很容易地使用它来指派相关的 Cargo 依赖。在项目

使用 Cargo，需要提供 JAR 文件，其包含了 Cargo API、核心容器实现和 Cargo Ant task。幸运的是，Cargo 提供了一个独立的 JAR 文件 UberJar，它包含了 API 和容器功能，使得依赖管理更加容易。下面的清单显示了如何将相关的 Cargo 依赖指派给 cargo 配置。

### 清单 5.3 将 Cargo 依赖指派给 cargo 配置

```
ext.cargoGroup = 'org.codehaus.cargo'
ext.cargoVersion = '1.3.1'

dependencies {
    cargo group: cargoGroup, name: 'cargo-core-uberjar',
        version: cargoVersion
    cargo "$cargoGroup:cargo-ant:$cargoVersion"
}
```

依赖声明使用 map 形式包含 group、name 和 version 属性

以字符串形式快捷声明依赖

如果你需要处理项目中的大量依赖，那么将常用的依赖属性作为扩展属性是有帮助的。在示例代码中你可以创建和使用 Cargo 的 group 和 version 依赖属性作为扩展属性。

Gradle 不会为你选择一个默认的仓库。在没有配置仓库的情况下运行 `deployToLocalTomcat` task 将会导致错误，如下面的控制台输出所示：

```
$ gradle deployToLocalTomcat
:deployToLocalTomcat FAILED

FAILURE: Build failed with an exception.

* Where: Build file '/Users/benjamin/gradle-in-action/code/
chapter5/cargo-configuration/build.gradle' line: 10

* What went wrong:
Execution failed for task ':deployToLocalTomcat'.
> Could not resolve all dependencies for configuration ':cargo'.
   > Could not find group:org.codehaus.cargo, module:cargo-core-
       uberjar, version:1.3.1.
       Required by:
           :cargo-configuration:unspecified
   > Could not find group:org.codehaus.cargo, module:cargo-ant,
       version:1.3.1.
       Required by:
           :cargo-configuration:unspecified
```

到目前为止，我们还没有讨论不同类型的仓库以及如何配置它们。为了让这个例子正常运行，我们添加如下 `repositories` 配置块：

```
repositories {
    mavenCentral()
}
```

不需要完全理解这个复杂的代码片段。重要的是会配置项目使用 Maven Central 来下载 Cargo 依赖。在本章后面部分，你将学习到如何配置其他的仓库。

## 检查依赖报告

当你运行 `dependencies` 帮助 task 时，会显示出来完整的依赖树。依赖树显示了在构建脚本中声明的顶层依赖，以及它们的传递性依赖：

```
$ gradle dependencies
:dependencies

-----
Root project
-----

cargo - Classpath for Cargo Ant tasks.
+--- org.codehaus.cargo:cargo-core-uberjar:1.3.1
|   +--- commons-discovery:commons-discovery:0.4
|       \--- commons-logging:commons-logging:1.0.4
+--- jdom:jdom:1.0
+--- dom4j:dom4j:1.4
|   +--- xml-apis:xml-apis:1.0.b2 -> 1.3.03
|   +--- jaxen:jaxen:1.0-FCS
|   +--- saxpath:saxpath:1.0-FCS
|   +--- msv:msv:20020414
|   +--- relaxngDatatype:relaxngDatatype:20020414
|       \--- isorelax:isorelax:20020414
+--- jaxen:jaxen:1.0-FCS (*)
+--- saxpath:saxpath:1.0-FCS (*)
+--- msv:msv:20020414 (*)
+--- relaxngDatatype:relaxngDatatype:20020414 (*)
+--- isorelax:isorelax:20020414 (*)
+--- com.sun.xml.bind:jaxb-impl:2.1.13
|   \--- javax.xml.bind:jaxb-api:2.1
|       +--- javax.xml.stream:stax-api:1.0-2
|       \--- javax.activation:activation:1.1
+--- javax.xml.bind:jaxb-api:2.1 (*)
+--- javax.xml.stream:stax-api:1.0-2 (*)
+--- javax.activation:activation:1.1 (*)
+--- org.apache.ant:ant:1.7.1
|   \--- org.apache.ant:ant-launcher:1.7.1
+--- org.apache.ant:ant-launcher:1.7.1 (*)
+--- xerces:xercesImpl:2.8.1
|   \--- xml-apis:xml-apis:1.3.03 (*)
+--- xml-apis:xml-apis:1.3.03 (*)
\--- commons-logging:commons-logging:1.0.4 (*)
\--- org.codehaus.cargo:cargo-ant:1.3.1
    \--- org.codehaus.cargo:cargo-core-uberjar:1.3.1 (*)

(*) - dependencies omitted (listed previously)
```

在构建脚本中声明的顶层依赖

指明请求的和所选择的版本来解决类库的版本冲突问题

在构建脚本中声明顶层依赖

标记在依赖图中排除的传递性依赖

如果你仔细查看依赖树，就会看到标有星号的依赖被排除了。这意味着依赖管

理器选择的是相同的或者另一个版本的类库，因为它被声明作为另一个顶层依赖的传递性依赖。有趣的是，UberJar 就是这样的，你甚至不需要在构建脚本中声明它。Ant task 类库会自动确保依赖类库被引入进来。针对版本冲突 Gradle 默认的解决策略是获取最新的版本——也就是说，如果依赖图中包含了同一个类库的两个版本，那么它会自动选择最新的。比如 xml-api 类库，Gradle 会选择 1.3.03 而不是 1.0.b2，通过一个箭头 (->) 来指示。正如你所看到的，它在分析依赖报告中的信息是很有帮助的。当你想要找出哪个顶层依赖声明了某个传递性依赖，以及为什么选择或者排除了某个版本的类库时，依赖报告给出了最好的答案。接下来，我们将看看如何排除传递性依赖。

### 排除传递性依赖

当处理一个像 Maven Central 一样的公共仓库时，你可能会遇到维护得很差的依赖元数据。Gradle 可以让你完全控制传递性依赖，因此你可以决定排除所有的传递性依赖或者有选择性地排除某些依赖。假设你想要显式地指定 xml-api 类库的不同版本，而不是使用 Cargo 的 UberJar 所提供的传递性依赖。在实践中，基于 API 或者框架的一个特定版本来构建自己的一些功是很常见的。下面的清单显示了如何使用 ModuleDependency 中的 exclude 方法来排除传递性依赖。

#### 清单 5.4 排除一个依赖

```
dependencies {  
    cargo('org.codehaus.cargo:cargo-ant:1.3.1') {  
        exclude group: 'xml-apis', module: 'xml-apis'  ← 通过快捷或 map 标记来  
    }                                                    声明排除依赖  
    cargo 'xml-apis:xml-apis:2.0.2'  
}
```

注意排除属性与常用的依赖标记略有不同。你可以使用 group 和 / 或 module 属性。Gradle 不允许你只排除某个特定版本的依赖，所以 version 属性是不可用的。

有时依赖的元数据声明的传递性依赖在仓库中不存在。因此，构建会失败。当你想要完全控制传递性依赖的时候才会发生这种情况。Gradle 让你通过使用 transitive 属性来排除所有的传递性依赖，如下面的清单所示。

#### 清单 5.5 排除所有的传递性依赖

```
dependencies {  
    cargo('org.codehaus.cargo:cargo-ant:1.3.1') {  
        transitive = false  
    }  
    // Selectively declare required dependencies  
}
```

到目前为止，你只声明了对外部类库的某一特定版本的依赖。让我们看看如何获取最新版本的依赖或在版本范围内选择最新的依赖。

### 动态版本声明

动态版本声明有一个特定的语法。如果你想使用最新版本的依赖，则必须使用占位符 `latest.integration`。例如，为 Cargo Ant task 声明最新版本的依赖，可以使用 `org.codehaus.cargo:cargo-ant:latest-integration`。或者，声明版本属性，通过使用一个加号（+）标定它来动态改变。下面的清单显示了如何获取 Cargo Ant 类库最新的 1.x 版本。

清单 5.6 在最新的 Cargo 1.x 版本下声明一个依赖

```
dependencies {  
    cargo 'org.codehaus.cargo:cargo-ant:1.+'  
}
```

Gradle 的帮助 task 清晰地指出了所选择的版本：

```
$ gradle -q dependencies  
  
-----  
Root project  
-----  
  
cargo - Classpath for Cargo Ant tasks.  
\--- org.codehaus.cargo:cargo-ant:1.+ -> 1.3.1  
    \--- ...
```

另一种方法是在版本范围内选择最新的依赖。要了解更多的语法相关信息，请查看 Gradle 的在线手册。

### 什么时候使用动态版本？

简单来说最好是少用或者不用。可靠的和可重复的构建是最重要的。选择最新版本的类库可能会导致构建失败。更糟糕的是，在不知情的情况下，你可能引入了不兼容的类库版本和副作用，这样很难找到原因并且只在应用程序的运行时发生。因此，声明确切版本的类库应该成为惯例。

## 5.4.3 文件依赖

如前所述，项目不使用自动的依赖管理在源代码或本地文件系统中组织外部类库。尤其是在将项目迁移到 Gradle 上时，你不想立即改变构建的方方面面。Gradle 通过配置文件依赖使其变得非常简单。通过引用本地文件系统中的 Cargo 类库来体

验一下这种情况。下面的清单显示了一个 task，用于将从 Maven Central 获取的依赖拷贝到用户 home 目录下的 libs/cargo 子目录中。

#### 清单 5.7 将 Cargo 依赖拷贝到本地文件系统中

```
task copyDependenciesToLocalDir(type: Copy) {
    from configurations.cargo.asFileTree
    into "${System.properties['user.home']}/libs/cargo"
}
```

← Gradle API 提供的语法糖；  
如同调用 configurations.  
getByName('cargo').asFileTree

运行这个 task 之后，你可以在 dependencies 配置块中声明 Cargo 类库。下面的清单展示了如何把 JAR 文件指派给 cargo 配置作为文件依赖。

#### 清单 5.8 声明文件依赖

```
dependencies {
    cargo fileTree(dir: "${System.properties['user.home']}/libs/cargo",
        ➡ include: '*.jar')
}
```

因为你不是在与一个要求你以某一特定模式声明依赖的仓库打交道，你也不需要定义一个 repositories 配置块。接下来，我们将重点放在 Gradle 支持的各种仓库类型以及如何配置它们。

## 5.5 使用和配置仓库

Gradle 特别强调了对现有仓库的支持。你已经知道了如何在构建中使用 Maven Central。通过调用 mavenCentral() 方法，你可以配置构建使用最流行的 Java 二进制仓库。除了预配置的仓库支持以外，如果需要，你也可以指定一个任意的 Maven 或者 Ivy 仓库的 URL 并且配置来使用身份验证。或者，可以使用一个简单的文件系统仓库来解决依赖关系。如果找到了依赖的元数据，它也会从仓库下载。表 5.2 中显示了不同类型的仓库，以及可以从哪个章节中了解到更多信息。

表 5.2 Gradle 项目的仓库类型

类 型	描 述	了解更多信息
Maven 仓库	本地文件系统或远程服务器中的 Maven 仓库，或者预配置的 Maven Central	5.5.2 节
Ivy 仓库	本地文件系统或远程服务器中的 Ivy 仓库，具有特定的结构模式	5.5.3 节
扁平的目录仓库	本地文件系统仓库，没有元数据支持	5.5.4 节

你可以跳到相应的章节，去了解你想要在项目中使用的仓库。在下一节中，我们来看看 Gradle API 对定义和配置仓库的支持，然后再将它们应用到实际的例子中。

### 5.5.1 理解仓库 API 表示

在项目中定义仓库的关键是 `RepositoryHandler` 接口，它提供了添加各种类型仓库的方法。从项目上看，这些方法在 `repositories` 配置块中被调用。你可以声明多个仓库。当依赖管理器试图下载依赖及其元数据的时候，它会按照声明的顺序来检查仓库。仓库提供了依赖优先原则。对于特定的依赖后续的仓库声明不会被进一步检查。如图 5.8 所示，每个仓库的接口针对特定类型的仓库都暴露了不同的方法。

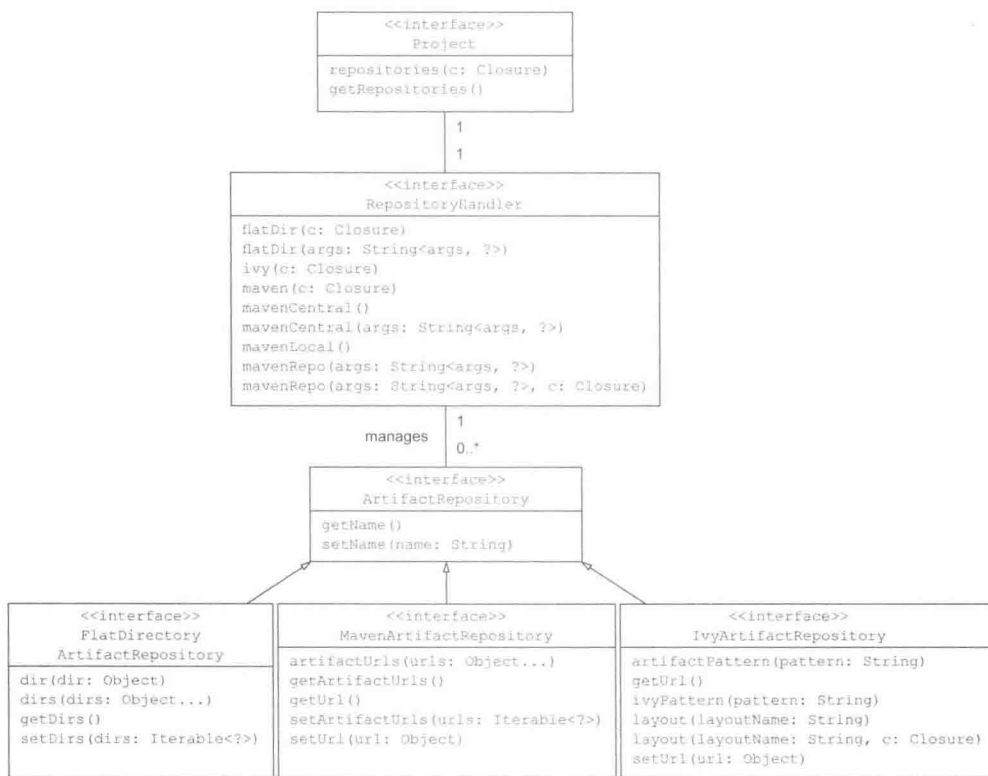


图 5.8 Gradle API 中用来配置各种类型仓库的相关接口。Gradle 支持扁平的目录、Maven、Ivy 三种仓库的实现。

Gradle 不会偏好任何仓库类型。最好是根据项目需要来声明最合适的仓库。在下一节中，我们来看看声明 Maven 仓库的语法。

## 5.5.2 Maven 仓库

Maven 仓库是 Java 项目中最常用的仓库类型之一。类库通常以 JAR 文件的形式表现。元数据用 XML 表示，并且使用 POM 文件描述了类库相关信息及其传递性依赖。所有的工件都存储在仓库里的一个预定义的目录结构中。当在构建脚本中声明一个依赖时，其属性可以用来获取它在仓库中的确切位置。依赖属性 `group` 中的点字符指示出 Maven 仓库中的子目录。图 5.9 显示了如何映射 Cargo Ant 的依赖属性来确定仓库中 JAR 和 POM 文件的位置。

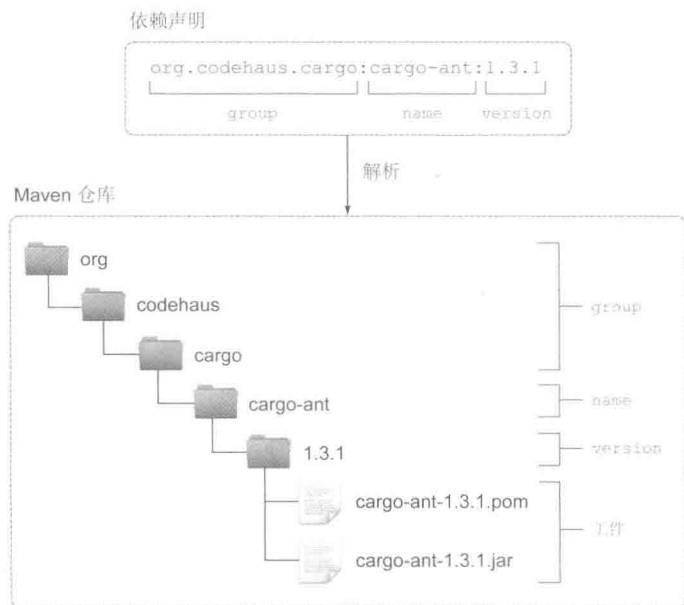


图 5.9 在 Maven 仓库中依赖声明是如何映射到工件的

`RepositoryHandler` 接口提供了两个方法来允许你定义预配置的 Maven 仓库。`mavenCentral()` 方法用来将 Maven Central 引用添加到一系列仓库中，`mavenLocal()` 方法用来在文件系统中关联一个本地 Maven 仓库。让我们来回顾一下这两种仓库类型并且讨论在项目中何时可以使用它们。

### 添加预配置的 Maven Central 仓库

Maven Central 是构建中经常使用的仓库。Gradle 想让它尽可能容易地被构建开发人员使用，因此 Gradle 提供了一种快捷方式来声明 Maven Central。而不必每次都定义 URL `http://repo1.maven.org/maven2`，你可以调用 `mavenCentral()` 方法，如下面的代码片段所示：



```
repositories {  
    mavenCentral()  
}
```

对于定义本地 Maven 仓库也有相似的快捷方式，默认在 `<USER_HOME>/.m2/repository` 目录下是可用的。

### 添加预配置的本地 Maven 仓库

当 Gradle 获取依赖时，它会定位仓库、下载并存储在本地缓存中。这个缓存在本地文件系统中的位置与 Maven 存储所下载的工件的目录是不同的。你可能想知道何时可以使用一个本地 Maven 仓库和 Gradle 打交道，尤其是在项目中使用多个构建工具的时候。想象一下你正在处理一个使用 Maven 生成类库的项目，另一个 Gradle 项目也想使用这个类库。尤其是在开发阶段，你会检查实现周期的变化，并且尝试应用这些变化。为了防止每一次小小的改动都必须将类库发布到远程 Maven 仓库，Gradle 提供了一个本地 Maven 仓库，如下面的仓库声明所示：

```
repositories {  
    mavenLocal()  
}
```

请注意，使用本地 Maven 仓库应该局限于这个特定的用例，因为它可能导致不可预见的副作用。你可以显式地声明工件依赖只在本地文件系统中可用。在工件不存在的情况下，在其他机器或持续集成服务器上运行脚本可能导致构建失败。

### 添加一个自定义的 Maven 仓库

我们有很多理由需要一个并非 Maven Central 的仓库。也许一个特定的依赖是简单的但不可使用，或者你想要通过设置企业级仓库以确保构建是可靠的。仓库管理器为你提供了一个以 Maven 结构来配置仓库的机会。这意味着它遵循了我们前面讨论的工件存储模式。此外，也可以通过要求用户提供认证信息来限制访问你的仓库。Gradle API 支持两种方式来配置一个自定义仓库：`maven()` 和 `mavenRepo()`。下面的清单显示了如果工件在 Maven Central 中不可用，如何选择一个公共的 Maven 仓库。

#### 清单 5.9 声明一个自定义的 Maven 仓库




```
repositories {  
    mavenCentral()  
    maven {  
        name 'Custom Maven Repository',  
        url 'http://repository-gradle-in-action.forge.cloudbees.com/release/'  
    }  
}
```

在本章中我们没有讨论每一个可用的配置选择，请参考在线文档来了解更多信息。接下来我们看看 Ivy 仓库与 Maven 仓库如何不同，以及它的配置。

### 5.5.3 Ivy 仓库

Maven 仓库中的工件必须以一种固定的布局存储。任何结构偏差都可能导致依赖关系发生错误。另一方面，Ivy 仓库提出了一个完全可自定义的默认布局。在 Ivy 里，仓库依赖元数据被存储在 `ivy.xml` 文件中。Gradle 在构建中提供了各种方法来配置 Ivy 仓库及其特定的布局，这超出了本书的涵盖范围。我们来看一个例子。假设你想要从 Ivy 仓库中来获取 Cargo 依赖。下面的清单展示了如何定义仓库的基础 URL，以及工件和元数据布局模式。

清单 5.10 声明 Ivy 仓库

```
repositories {  
    ivy {  
        url 'http://repository.myenterprise.com/ivy/bundles/release'  Ivy 仓库的  
                                                基础 URL  
        layout 'pattern', {  
            artifact '[organisation]/[module]/[revision]/[artifact]-  
                [revision].[ext]'  工件  
                                                模式  
            ivy '[organisation]/[module]/[revision]/ivy-[revision].xml'  元数据  
                                                模式  
        }  
    }  
}
```

因为 Maven 仓库中存在 POM，所以不需要强制使用 Ivy 元数据来解决传递性依赖。Ivy 仓库完美地解决了依赖，而不必遵循标准的 Maven 工件模式。例如，你可以决定将 JAR 文件放到 Web 服务器的一个特定目录下，并通过 HTTP 来提供服务。接下来，我们看看 flat 目录仓库。

### 5.5.4 扁平的目录仓库

flat 目录仓库是最简单和最基本的仓库形式。在文件系统中它是一个单独的目录，只包含 JAR 文件，没有元数据。如果你常常手动维护项目中的类库并且准备把项目迁移到自动化依赖管理工具中，那么你会对这种方法感兴趣。

当声明依赖时，你只能使用 `name` 和 `version` 属性。不能使用 `group` 属性，因为它会导致产生不明确的依赖关系。下面的清单显示了如何以 `map` 和快捷方式来声明从 flat 目录仓库中获取的 Cargo 依赖。

清单 5.11 从 flat 目录仓库中获取的 Cargo 依赖声明

```
repositories {  
    flatDir(dir: "${System.properties['user.home']}/libs/cargo",  
           name: 'Local libs directory')  
}  
  
dependencies {  
    cargo name: 'activation', version: '1.1'  
    cargo name: 'ant', version: '1.7.1'  
    cargo name: 'ant-launcher', version: '1.7.1'  
    cargo name: 'cargo-ant', version: '1.3.1'  
    cargo name: 'cargo-core-uberjar', version: '1.3.1'  
    cargo name: 'commons-discovery', version: '0.4'  
    cargo name: 'commons-logging', version: '1.0.4'  
    cargo name: 'dom4j', version: '1.4'  
    cargo name: 'isorelax', version: '20020414'  
    cargo ':jaxb-api:2.1', ':jaxb-impl:2.1.13', ':jaxen:1.0-FCS',  
           ':jdom:1.0', ':msv:20020414', ':relaxngDatatype:20020414',  
           ':saxpath:1.0-FCS', ':stax-api:1.0-2', ':xercesImpl:2.8.1',  
           ':xml-apis:1.3.03'  
}
```

依赖属性 name  
和 version 的用法

没有 group 属性的  
依赖声明快捷  
方式的用法

这个清单同样完美地演示了使用元数据来自动化声明传递性依赖的好处。就 flat 目录仓库来说，你没有这些信息，因此你需要单独地声明每个依赖，这种方式耗时耗力。

## 5.6 理解本地依赖缓存

至此，我们已经讨论了如何声明依赖和配置各种类型的仓库来解决工件的依赖问题。Gradle 会自动确定你想要执行的 task 所需要的依赖，从仓库中下载工件，并将它们存储在本地缓存中。任何后续构建都将重用这些工件。在本节中，我们将深入分析缓存结构，确定缓存在底层是如何工作的，以及如何调整其行为。

### 5.6.1 分析缓存结构

让我们通过 Cargo 类库的例子来探索本地缓存结构。当运行部署 task 时，Gradle 会下载 JAR 文件，但是它们被放在哪里了呢？如果你查看 Gradle 论坛，你会发现有很多人也在问这个问题。使用 Gradle API 可以找出答案。下面的清单显示了如何打印完整的、指派给 cargo 配置的所有依赖的连接路径。

## 清单 5.12 打印所有的 Cargo 依赖的连接文件路径

```
task printDependencies << {
    configurations.getByNames('cargo').each { dependency ->
        println dependency
    }
}
```

如果你运行这个 task，就会发现所有的 JAR 文件都被存储在 /Users/benjamin/.gradle/caches/artifacts-15/filestore 目录中：

```
$ gradle -q printDependencies
/Users/benjamin/.gradle/caches/artifacts-15/filestore/
org.codehaus.cargo/cargo-core-uberjar/1.3.1/jar/
3d6aff857b753e36bb6bf31eccf9ac7207ade5b7/cargo-core-uberjar-1.3.1.jar
/Users/benjamin/.gradle/caches/artifacts-15/filestore/
org.codehaus.cargo/cargo-ant/1.3.1/jar/
a5a790c6f1abd6f4f1502fe5e17d3b43c017e281/cargo-ant-1.3.1.jar
...
```

这条路径和你的机器上的可能会稍有不同。我们来仔细研究一下这条路径并赋予它更多的意义。在本地缓存中用来存储依赖的 Gradle 根目录是 <USER\_HOME>/ .gradle/caches。这条路径接下来的 artifact-15 是一个标识符，用来指定 Gradle 版本。区分元数据存储方式的变化是必需的。

记住，这个结构可能会在 Gradle 的新版本中发生改变。缓存实际上被分为两个部分。filestore 子目录包含了从仓库下载的原始二进制文件。此外，你会发现一些二进制文件，其中存储了关于已下载工件的元数据。在日常业务中，你永远不需要查看它们。下面的目录树显示了根级别的本地依赖缓存的内容：



filestore 目录是依赖的自然表现形式。group、name 和 version 属性直接映射到文件系统中的子目录。在下一节中，我们将讨论 Gradle 缓存给构建带来的好处。

## 5.6.2 显著的缓存特性

Gradle 缓存的真正能力在于其元数据。它让 Gradle 实现了额外的优化，使得构建更智能、更快捷、更可靠。下面我们来逐一地讨论其特性。

### 依赖来源的存储

想象这样一个场景：你在脚本中声明了一个依赖。首次运行构建时，下载依赖并存储在缓存中。后续构建将愉快地使用缓存中的这个可用的依赖。构建成功。如果仓库结构变化了（例如，其中一个属性重命名了或者依赖移动或删除了）将发生什么事情呢——工件的使用者将无法控制某些情况？在许多其他的依赖管理工具如 Maven 和 Ivy 的帮助下，构建会工作得很好，因为依赖存在于本地缓存中，并且可以被获取到。然而，当其他开发人员在不同的机器上运行构建时，构建将会失败。问题就是导致构建不一致。针对这种情况，Gradle 采用了不同的方法。它知道依赖的来源，并将这些信息存储在缓存中。因此，构建变得更加可靠。

### 工件变化检测

Gradle 试图减少到远程仓库的网络传输。这不仅仅针对下载依赖的情况。如果无法从一个仓库中获取依赖，元数据被存储在缓存中。Gradle 使用这些信息来避免每次运行构建时都对仓库进行检查。

### 减少工件下载和提高变化检测

Gradle 与 Maven 的本地仓库紧密集成，可以避免必须下载已有的工件。如果依赖可以在本地获取，就会重用它。使用其他版本的 Gradle 存储工件也是如此。

Gradle 通过比较本地和远程的校验和来检测仓库中的工件是否发生变化。没有发生变化的工件不会再次下载，并且会重用本地缓存中的。想象一下仓库中的工件发生了变化，但校验和仍是相同的。如果仓库管理员用相同版本的工件取代了原来的，就会发生这种情况。最终，构建将使用一个过时版本的工件。Gradle 依赖管理器试图通过考虑额外的信息来消除这种情况。例如，可以通过比较 HTTP 头参数 contentlength 的值或者最后的修改日期来确保工件的唯一性。这是 Gradle 的实现相比于其他的依赖管理器如 Ivy 的优势所在。

### 离线模式

如果构建声明了远程仓库，Gradle 就不得不检查仓库看依赖是否发生变化。有

时这种行为是不可取的；例如，如果你在差旅中，访问不了互联网。你可以通过运行命令行选项 `--offline` 告诉 Gradle 在离线模式下不要检查远程仓库。只使用本地缓存中的依赖，而不是强行通过网络解决依赖。如果所需要的依赖在缓存中不存在，那么构建将会失败。

## 5.7 解决依赖问题

版本冲突是一个棘手的问题。如果你的项目有很多依赖，而且你选择自动解决传递性依赖，那么版本冲突几乎是不可避免的。Gradle 解决版本冲突默认的策略是选择最新的依赖版本。依赖报告是最有用的工具，它可以帮助选择所需依赖的版本。在下一节中，我将介绍如何解决版本冲突并调整 Gradle 的依赖解决策略以适应特定的用例。

### 5.7.1 应对版本冲突

Gradle 不会自动通知你项目遇到了版本冲突问题。必须不断地运行依赖报告以找出冲突并不是一个实际的解决方法。你可以更改默认的解决策略，当遇到版本冲突问题时让构建失败，如下面的代码示例所示：

```
configurations.cargo.resolutionStrategy {  
    failOnVersionConflict()  
}
```

失败对于调试目的很有帮助，特别是在建立项目的早期阶段和改变依赖关系时。运行项目的任何 task 都会显示版本冲突信息，如下面的示例输出所示：

```
$ gradle -q deployToLocalTomcat  
  
FAILURE: Build failed with an exception.  
  
* Where:  
Build file '/Users/benjamin/Dev/books/gradle-in-action/code/chapter4/  
cargo-dependencies-fail-on-version-conflict/build.gradle' line: 10  
  
* What went wrong:  
Execution failed for task ':deployToLocalTomcat'.  
> Could not resolve all dependencies for configuration ':cargo'.  
   > A conflict was found between the following modules:  
       - xml-apis:xml-apis:1.3.03  
       - xml-apis:xml-apis:1.0.b2
```

### 富 API 访问确切的依赖图

在内存中，Gradle 为确切的依赖图创建了一个模型。Gradle 的解决结果 API 可以让你更细粒度地控制所需要和选择的依赖。ResolutionResult 接口是了解 API 的一个良好的开端。

## 5.7.2 强制指定一个版本

管理项目越多，你就越会觉得需要标准化的构建环境。你想要共享通用的 task 或者确保所有项目都使用一个指定版本的类库。例如，你想要统一所有的 Web 项目使用 Cargo 1.3.0 版本部署，尽管依赖声明可能需要不同的版本。使用 Gradle，可以很容易地实现这样的企业级策略。它使你能够强制指定顶层依赖的版本，以及传递性依赖。

下面的代码片段演示了如何为 cargo 配置重新配置默认的解决策略，强制依赖 1.3.0 版本的 Ant task：

```
configurations.cargo.resolutionStrategy {
    force 'org.codehaus.cargo:cargo-ant:1.3.0'
}
```

现在运行依赖报告 task，你会发现所需要的 Cargo Ant 版本被全局的强制执行模块版本所代替：

```
$ gradle -q dependencies
```

```
-----
Root project
-----
```

```
cargo - Classpath for Cargo Ant tasks.
```

```
\--- org.codehaus.cargo:cargo-ant:1.3.1 -> 1.3.0
```

```
    \--- org.codehaus.cargo:cargo-core-uberjar:1.3.0
```

```
        +--- ...
```

← 强制执行模块版本优先

## 5.7.3 使用依赖观察报告

配置的解决策略的变化，如前所示，是完全适合于初始化脚本的，它可以全局级别强制执行。构建脚本用户可能不知道为什么 Cargo Ant task 选择了这个特定的版本。他们唯一能看到的是依赖报告中指出选择了不同的版本。有时候你可能想知道是什么迫使这个版本被选择的。

Gradle 提供了不同类型的报告：依赖观察报告，它解释了依赖图中的依赖是如何选择的以及为什么。为了运行这个报告，你需要提供两个参数：配置名称（默认

是 compile 配置) 和依赖本身。下面的 dependencyInsight 帮助 task 的调用显示了原因, 以及所需要和选择的依赖 xml-apis:xml-apis 的版本:

```
$ gradle -q dependencyInsight --configuration cargo --dependency
➡ xml-apis:xml-apis
xml-apis:xml-apis:1.3.03 (conflict resolution)
+--- org.codehaus.cargo:cargo-core-uberjar:1.3.0
|    \--- org.codehaus.cargo:cargo-ant:1.3.0
|         \--- cargo
\--- xerces:xercesImpl:2.8.1
      \--- org.codehaus.cargo:cargo-core-uberjar:1.3.0 (*)

xml-apis:xml-apis:1.0.b2 -> 1.3.03
\--- dom4j:dom4j:1.4
      \--- org.codehaus.cargo:cargo-core-uberjar:1.3.0
            \--- org.codehaus.cargo:cargo-ant:1.3.0
                  \--- cargo

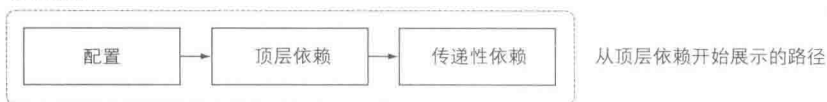
(*) - dependencies omitted (listed previously)
```

显示了一个特定依赖被选择的原因

显示了所需要和选择的特定依赖的版本

虽然依赖报告是从配置的顶层依赖开始的, 但是观察报告显示了依赖图是从特定依赖开始到配置的。因此, 观察报告展现的视图与普通的依赖报告正好相反, 如图 5.10 所示。

依赖报告



依赖观察报告

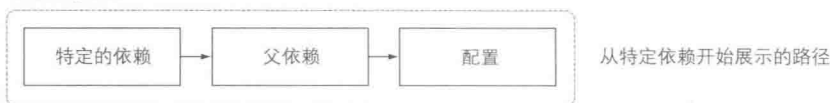


图 5.10 不同类型报告的依赖图视图

## 5.7.4 刷新缓存

为了避免一遍又一遍地为特定类型的依赖去访问仓库, Gradle 提供了特定的缓存策略。这种策略主要针对依赖的 SNAPSHOT 版本和使用动态版本模式声明依赖。一旦获取了依赖, 它们就会被缓存 24 小时, 这样使得构建更快捷、更有效。工件缓存时间到期后, 会再次检查仓库, 如果构件发生变化, Gradle 会下载一个最新版本的工作件。

你可以使用命令行选项 `--refresh-dependencies` 手动刷新缓存中的依赖。



这个标志强制检查配置仓库中的工件版本是否发生变化。如果校验和发生了变化了，依赖将被再次下载，并取代缓存中的现有副本。过了一段时间后添加一次命令行选项会变得很麻烦，或者可能忘了标记。你可以配置一个构建改变缓存的默认行为。

假如你总是想要获得使用 `org.codehaus.cargo:cargo-ant:1.+` 声明的最新的 Cargo Ant task 的 1.x 版本。你可以设置缓存动态依赖版本 0 秒超时，如下面的代码片段所示：

```
configurations.cargo.resolutionStrategy {  
    cacheDynamicVersionsFor 0, 'seconds'  
}
```

你或许有很好的理由不想缓存一个外部模块的 SNAPSHOT 版本。例如，组织中的另一个团队使用了在多项目中共享的可重用的类库。在开发过程中代码变动很大，并且你总是想要获得最新最好的代码。下面的代码块修改了解决策略，用来配置不缓存 SNAPSHOT 版本：

```
configurations.compile.resolutionStrategy {  
    cacheChangingModulesFor 0, 'seconds'  
}
```

## 5.8 总结

大多数项目，如开源项目或企业级产品不是完全独立的。它们依赖外部类库或其他项目构建的组件。手动管理这些依赖无法满足现代软件开发的需求。一个项目越复杂，就越难找出依赖之间的关系、解决潜在的版本冲突，甚至不知道为什么需要一个特定的依赖。

有了自动依赖管理，你可以在项目中通过唯一标识符来声明依赖，而不必手动控制工件。在运行时，仓库中依赖的工件会被自动解析、下载并存储到一个本地缓存中，使其对项目可用。自动依赖管理并不是没有挑战，我们讨论了潜在的陷阱以及如何应对它们。

Gradle 提供了强大的依赖管理机制。你了解了如何声明不同类型的依赖，通过配置进行分组，从多种类型的仓库中下载它们。本地缓存是 Gradle 的依赖管理机制不可分割的一部分，并负责高性能、可靠的构建。我们分析了它的结构，并讨论了它的基本特性。了解了如何解决依赖版本冲突和调整缓存是稳定和可靠构建的关键。使用 Gradle 的依赖报告能够很好地理解依赖图，以及为什么选择一个特定版本的依赖，并且它来自何处。我们展示了如何改变默认的解决策略和缓存行为，并使其满足适当的场合需求。

在下一章中，我们将通过模块化代码把 To Do 应用程序提高到更高的层次，并将学习如何使用 Gradle 的多项目构建支持来定义独立组件之间的依赖性，让它们整体发挥作用。

# 多项目构建

## 本章涵盖

- 将项目源代码组织成子项目
- 为多项目层次结构建模构建
- 通过 Project API 配置项目行为
- 在项目之间声明依赖
- 通过 Settings API 自定义构建

每一个活跃软件项目的代码库都会随着时间的推移而增长。最初拥有几个类的小项目很快就会变成一个具有不同责任的包和类的集合。为了提高可维护性和防止紧密耦合，你可以基于特定的功能和逻辑边界将代码分组成模块。模块通常具有层次结构而且可以定义为相互依赖。构建工具需要满足这些需求。

Gradle 对构建模块化项目提供了强大的支持。因为在 Gradle 中每个模块都是一个项目，我们称之为多项目构建（相对于 Maven 中的多模块构建）。本章介绍了通过 Gradle 建模和执行多项目构建的技术。本章结束后，你就会知道如何应用这种技术最适合自己的项目需要和适当地建模构建。

通过 To Do Web 应用程序来介绍 Gradle 对多模块构建的支持。你可以通过解构

现有的项目结构开始，分解成各个独立的功能子项目。新建的项目布局将作为建模构建的基础。接下来我们会回顾组织构建逻辑选项，你将了解部分 Gradle API 来定义独立或公共的项目行为。最后，你将学习通过声明项目依赖如何控制项目执行顺序，以及如何从根项目执行一个子项目或所有参与子项目的完整构建。本章不仅会介绍多项目构建的结构，还将让你了解到如何降低构建的执行时间。让我们先从将现有的 To Do 应用程序项目结构重构成模块化架构开始。

## 6.1 模块化项目

在企业级项目中，包的层次结构和类之间的关系变得非常复杂。将代码分离成模块是一项艰巨的任务，因为它需要你能够清楚地识别功能边界——例如，分离业务逻辑与数据持久化逻辑。

### 6.1.1 耦合与内聚

实现项目关注点分离很容易，起决定作用的两个主要因素是：耦合和内聚。耦合用来衡量特定代码工件如类之间关系的强弱。内聚涉及了模块的组件之间关系的密切程度。代码的耦合性越低和内聚性越高，就越容易实现项目重构。讲解最佳软件设计实践超出了本书范围，但是你可以记住两个指导原则：最小化耦合和最大化内聚。

Spring 框架就是一个很好的模块化架构的例子。Spring 是一个开源框架，它为企业级 Java 应用程序提供了广泛的服务。例如，为简化 MVC Web 应用程序开发提供服务功能支持，或者以 JAR 文件形式提供分布式事务管理。如果服务需要不同模块提供的功能，那么它们会相互依赖。图 6.1 显示了所有的 Spring 3.x 版本的模块及其相互关系。

Spring 的架构看起来很复杂。它定义了很多相互依赖的组件。但在实际使用中，你不需要将整个框架的所有组件都引入到项目中。你可以选择使用框架的哪个服务。幸运的是，组件之间的依赖关系是通过元数据指定的。使用 Gradle 的依赖管理机制解决这些传递性依赖非常容易。

在接下来的章节中，你将模块化 To Do 应用程序并且使用 Gradle 的多项目特性来构建它。目前你使用的代码库是有限的，与 Spring 框架开发人员相比，这个任务要简单得多。我们将从为应用程序划分模块开始。

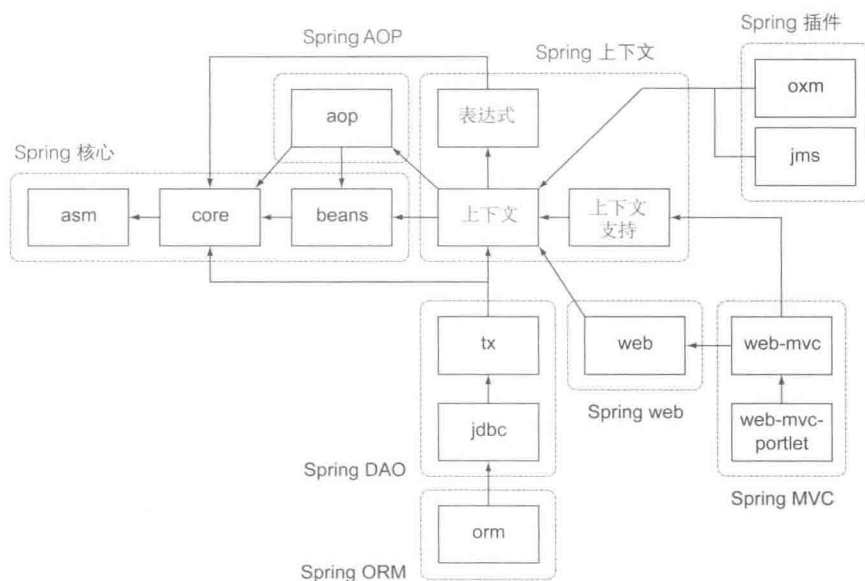
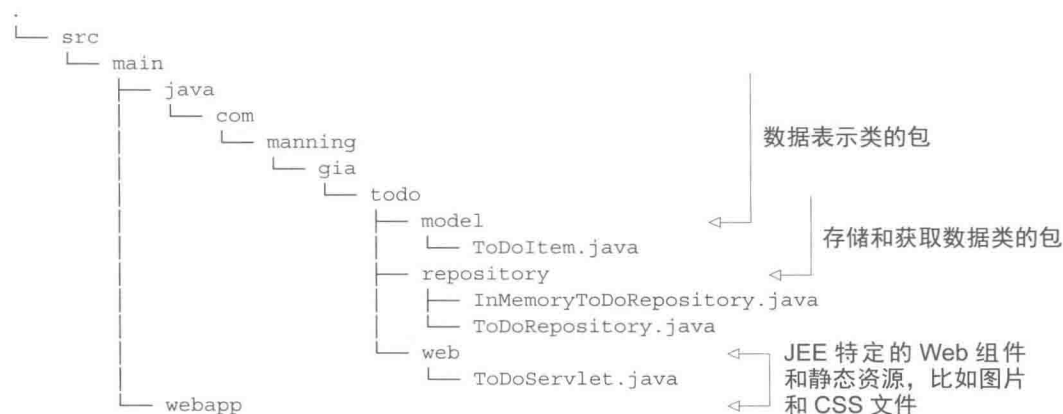


图 6.1 Spring 的模块化架构

### 6.1.2 模块划分

让我们回顾一下你为 To Do 应用程序编写的代码，找出它的自然边界。这些边界将帮助你应用程序代码分解成相应的模块。下面的目录树展示了现有的项目结构：



```

├── WEB-INF
│   └── web.xml
├── css
│   ├── base.css
│   └── bg.png
└── jsp
    ├── index.jsp
    └── todo-list.jsp

```

通过将特定功能的类分组到不同的包里，你已经做好了应用程序关注点的分离。你可以使用这些包作为指导来找到应用程序的功能边界：

- *Model* : to-do 项商品的数据表示
- *Repository* : to-do 项存储和获取
- *Web* : 处理 HTTP 请求和在浏览器中渲染 to-do 项及功能的 Web 组件

即使在非常简单的应用程序中，这些模块之间也会存在相互关系。例如，*Repository* 模块中的类使用 *Model* 数据类来传输数据存储器输入和输出的数据。图 6.2 给出了所有涉及的模块和它们之间相互关系的示意图。

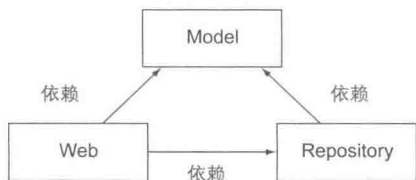


图 6.2 To Do 应用程序中所涉及的模块

### 6.1.3 模块化重构

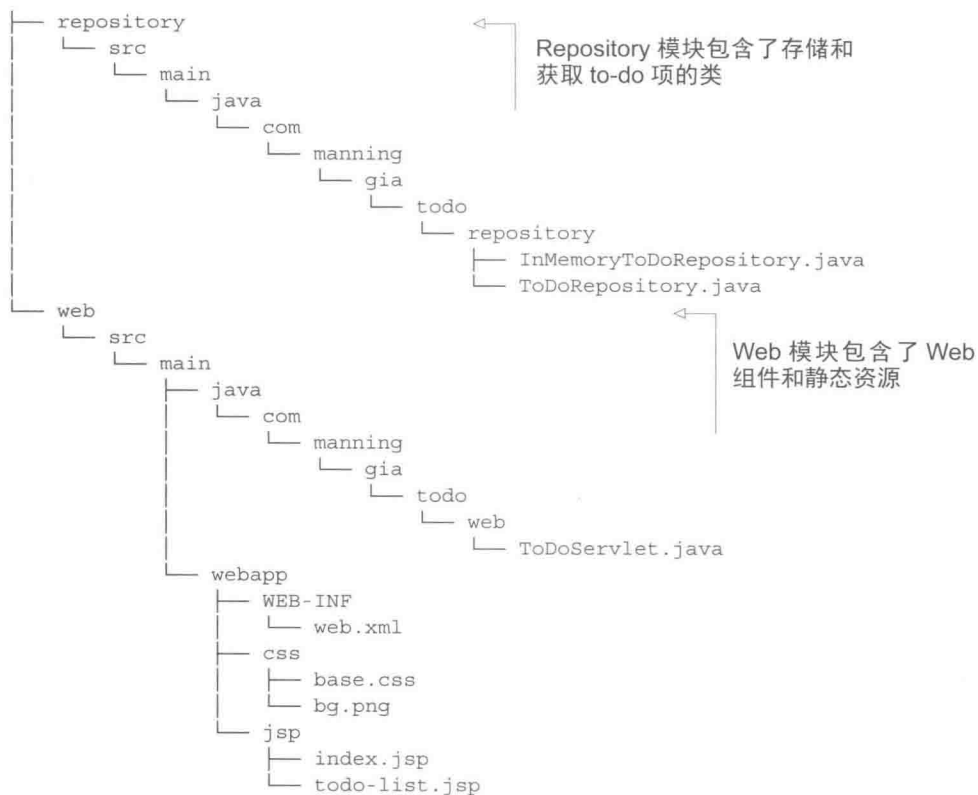
将现有的项目结构重构成模块其实不难。对于每个模块，需要创建一个具有适当名称的子目录，并将相关的文件移动到该子目录下。每个模块都有一个默认的源目录 `src/main/java`，并应该保持完好无损。*Web* 模块是唯一需要默认的 *Web* 应用程序源目录 `src/main/webapp` 的模块。下面的目录树显示了模块化的项目结构：

```

├── model
│   └── src
│       ├── main
│       │   └── java
│       │       ├── com
│       │       │   ├── manning
│       │       │   │   ├── gia
│       │       │   │   │   ├── todo
│       │       │   │   │   │   ├── model
│       │       │   │   │   │   │   └── ToDoItem.java

```

Model 模块包含了 To Do 数据表示类



就是这样——你已经模块化了 To Do 应用程序。现在是时候考虑构建基础环境了。

## 6.2 组装多项目构建

在上一节中，为项目定义了一个分层目录结构。整个项目由一个根目录和每个模块的子目录组成。在本节中，你将学习如何使用 Gradle 构建这样的项目结构。

首先应该从处于目录树根级别的 `build.gradle` 文件开始。创建一个空的构建脚本，通过运行 `gradle projects` 来查看参与构建的所有项目：

```
$ gradle projects
:projects
```

```
-----
Root project
-----
```

```
Root project 'todo'
No sub-projects
```

Gradle 显示你现在只应对一个项目。如果创建一个构建具有多个项目，那么 Gradle 会展示一个多项目构建。这是因为你把每个模块当成一个独立的 Gradle 项目。从现在开始，我们不再使用模块术语了，只谈论项目。

位于顶层目录中的总体项目被称为根项目，它存在于多项目构建中。它协调构建子项目，并且为子项目定义一些共同的或特定的行为。图 6.3 用图形概述了你想要实现的项目层次结构。

目前我们只处理了 Gradle 单项目构建配置。你看到把代码分离成多个项目并不难，所缺少的就是对表示根项目及其子项目的构建支持。在多项目构建中是通过 *settings* 文件来声明子项目的。

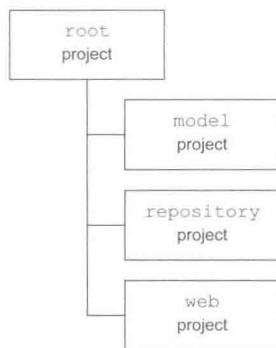


图 6.3 To Do 应用程序的多项目层次结构，它定义了三个子项目

## 6.2.1 settings 文件介绍

*settings* 文件声明了所需的配置来实例化项目的层次结构。在默认情况下，这个文件被命名为 *settings.gradle*，并且和根项目的 *build.gradle* 文件放在一起。下面的清单显示了 *settings* 文件的内容。若想使每个子项目都成为构建的一部分，则可以调用带有项目路径参数的 *include* 方法。

### 清单 6.1 通过路径添加子项目的 *settings* 文件

```
include 'model'
include 'repository', 'web'
```

将项目的字符串数组传给方法调用，而不是为每个单独的子项目调用 *include* 方法

将给定的子项目添加到构建中，传给 *include* 方法的参数是项目路径，不是文件路径

这个代码片段中所提到的项目路径是相对于根目录的项目目录。你也可以建模更深层次的项目结构。使用冒号 (:) 字符来分隔每一个子项目的层次结构。例如，如果你想要映射 *model/todo/items* 目录结构，则可以通过 *model:todo:items* 方式来添加子项目。

添加 *settings* 文件后，执行 *projects* 帮助 task，将生成不同的结果：

```
$ gradle projects
:projects
```

```
-----
Root project
-----
```



```
Root project 'todo'
+--- Project ':model'
+--- Project ':repository'
\--- Project ':web'
```

子项目以缩进的层次树结构显示

通过添加一个 `settings` 文件，你创建了一个包含有一个根项目和三个子项目的多模块构建。不需要额外的配置。我们将深入了解 `settings` 文件的细节。你可能已经猜到了，可以使用 Gradle API 表示来查询和修改构建配置。

## 6.2.2 理解 settings API 表示

Gradle 组装构建之前，它会创建一个 `Settings` 类型的实例。`Settings` 接口是 `settings` 文件的直接表示。它的主要作用是添加 `Project` 实例参与多项目构建。除了组装多项目构建之外，你可以在 `build.gradle` 脚本中做任何事情，因为你可以直接访问 Gradle 和 `Project` 接口。图 6.4 显示了 `Settings` 接口的相关方法及其联系。

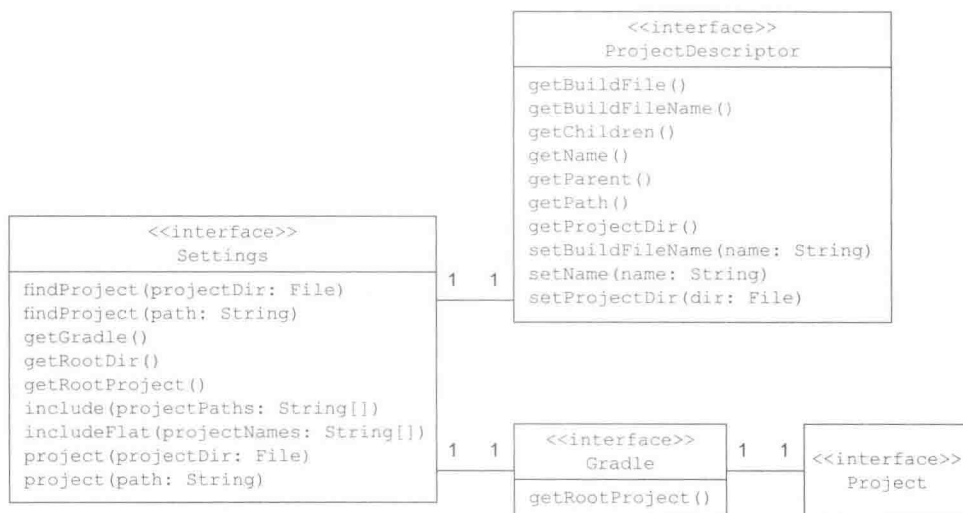


图 6.4 `Settings` API 表示。你可以使用 `Settings` 实例来获取项目描述符，或者通过 `Gradle` 接口来获取项目实例。

这里传递的重要信息是你可以 `settings.gradle` 文件中面向 `Settings` 接口的实例进行编码。`Settings` 接口中的任何方法都可以被直接调用，就像调用 `include` 一样。

接下来，我们将讨论在构建生命周期中什么时候执行 `settings` 文件，以及按照什么规则来解析这个文件。

### 通过构建文件访问 Settings

`settings` 被加载和解析后，如果你需要通过 `build.gradle` 文件访问 `Settings` 实例，则可以注册一个生命周期闭包或监听器。`Gradle#settingsEvaluated(Closure)` 方法是一个好的起点，它提供了 `Settings` 对象作为闭包参数。

## 6.2.3 settings 执行

回想第 4 章，我们讨论了一个构建的三个不同的生命周期阶段。你可能已经在想在什么阶段解析和执行 `settings` 文件的代码。它在任何 `Project` 实例配置之前的初始化阶段执行，如图 6.5 所示。

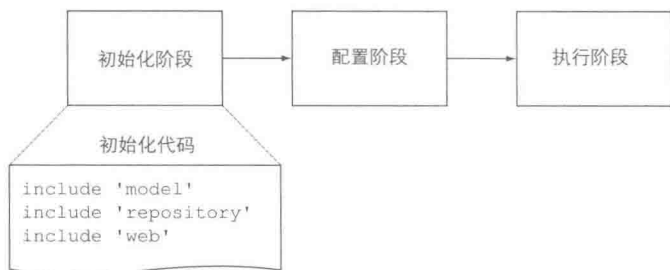


图 6.5 settings 文件在初始化阶段解析和执行

在执行一个构建时，Gradle 会自动知道一个子项目是否是一个单项目或多项目构建的一部分。让我们来看看 Gradle 如何通过一系列规则来确定 `settings` 文件的存在。

## 6.2.4 获取 settings 文件

只要根项目目录或任何子项目目录中包含有构建文件，Gradle 就允许你从相应位置运行构建。Gradle 如何知道一个子项目是一个多项目构建的一部分呢？它需要找到 `settings` 文件，这个文件指示了子项目是否包含在多项目构建中。图 6.6 显示了 Gradle 通过两步找到 `settings` 文件。

在步骤 1 中，Gradle 在与当前目录同层次的 master 目录下搜索 `settings` 文件。如果没有找到 `settings` 文件，Gradle 将从当前目录开始在父目录中查找 `settings` 文件。对于 `subproject2` 来说，搜索顺序为 `subproject1 > root`。

如果找到了 `settings` 文件，并且在它的定义中包含了这个项目，那么该项目就被认为是多项目构建的一部分。否则，这个项目将作为一个单项目构建执行。

在 `settings` 文件获取过程中，步骤 2 适用于之前设置的层次项目布局。我们回

头检查一下步骤 1 中所示的项目布局。

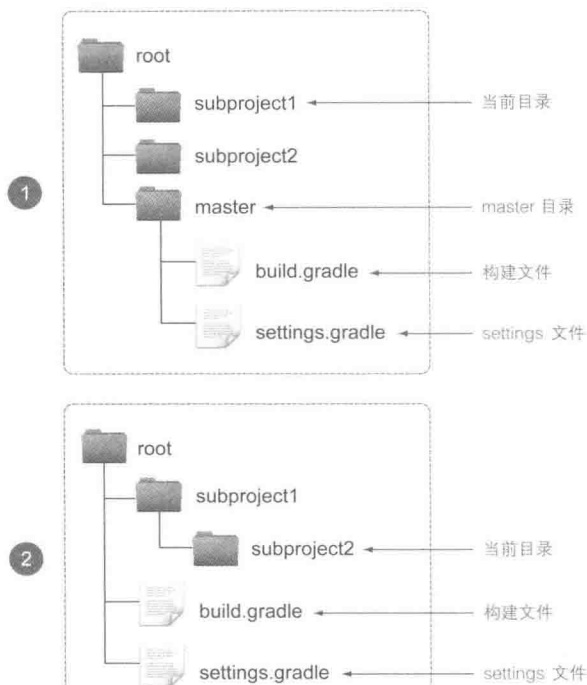


图 6.6 通过两步找到 settings 文件

### 控制 settings 文件的搜索行为

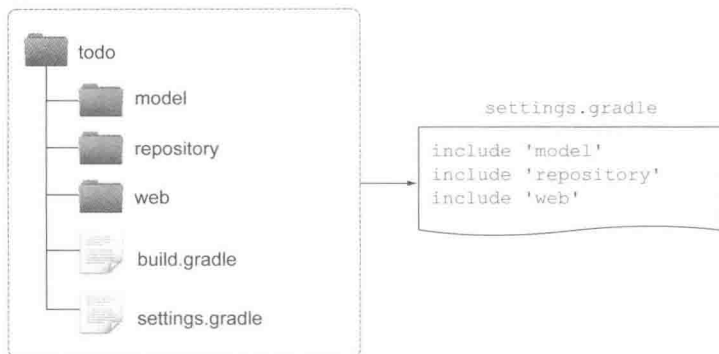
有两个命名行参数可以帮助我们确定 settings 文件的搜索行为：

- `-u, --no-search-upward`: 告诉 Gradle 不去父目录中搜索 settings 文件。
- `-c, --settings-file`: 指定 settings 文件的位置。当 settings 文件名没有遵循标准的命名约定时，你可能想要使用这个选项。

## 6.2.5 分层布局与扁平布局

Gradle 项目结构可以是分层布局或平面布局，如图 6.7 所示。我们所讨论的多项目平面布局是指所有参与的项目与根项目处于同一个目录级别。因此，这意味着子项目的嵌套级别只能深一层。你可以自由选择项目布局。我个人更喜欢层次化项目布局，因为它让你能够更细粒度地控制对组件的建模。

分层布局



平面布局

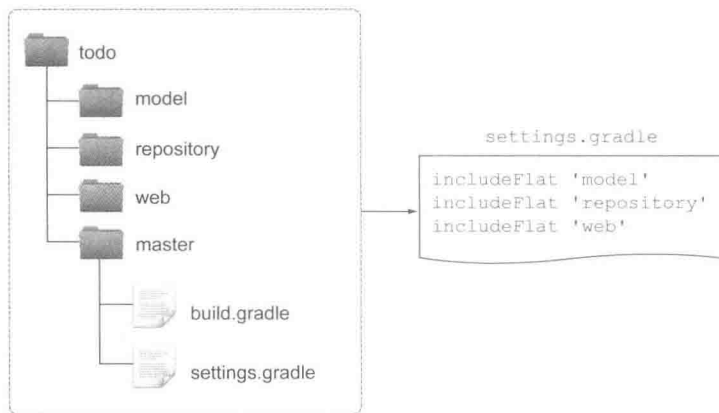


图 6.7 比较项目层次布局和平面布局以及它们的 settings 文件配置

图 6.7 比较了设置 To Do 应用程序分层布局与平面布局的差异。你需要为构建文件和 settings 文件创建与其他子项目并排的专用目录，而不是把它们放在项目的根目录下。正如我们在上一节所讨论的，选择 master 目录，以便可以从子项目执行构建。为了指示你想要包括与根项目同一嵌套层次的项目，可以在 settings 文件中使用 includeFlat 方法。

在下一节中，你将为每个子项目配置构建逻辑。

## 6.3 配置子项目

至此，你已经基于功能职责分离了应用程序代码并重新安排到各个子项目中。现在，你将采取类似的方法以可维护方式来组织构建逻辑。以下几点是真实多项目构建的共同需求：

- 根项目和所有子项目应该使用相同的 group 和 version 属性值。

- 所有子项目都是 Java 项目，并且都需要 Java 插件来保证正常运行，所以只需要对子项目应用插件，而不是根项目。
- web 子项目是唯一需要声明外部依赖的项目。这个项目类型来自于其他子项目，它产生一个 WAR 文件，而不是 JAR 文件，并且使用 Jetty 插件来运行应用程序。
- 在子项目之间建模依赖关系。

在本节中，你将了解到在多项目构建中如何定义特定的和公共的行为来避免重复配置。一些子项目可能依赖其他项目的编译的源代码——在应用程序中，model 项目的代码被 repository 项目所使用。通过声明项目依赖，可以确保引入的类在 classpath 中是可用的。在编写 build.gradle 文件之前，我们先看看与多项目构建相关的 Project API 的方法。

### 6.3.1 理解 Project API 表示

在第 4 章中，我解释了 Project API 的属性和方法，你可能会在日常业务中使用到它们。为了实现多项目构建，你需要了解一些新的方法，如图 6.8 所示。

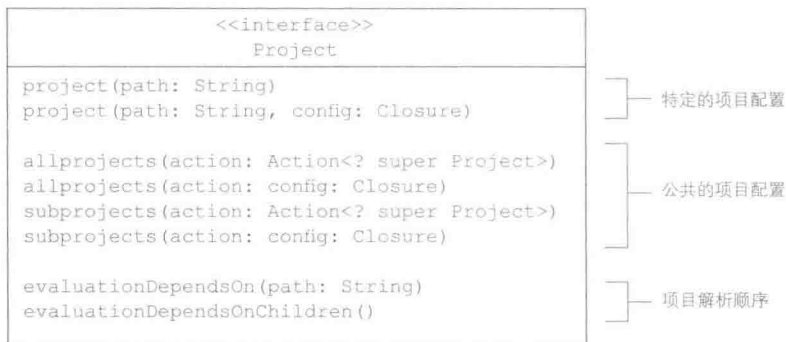


图 6.8 用于实现多项目构建的 Project API 的重要方法

为了声明特定的项目构建代码，使用了 `project` 方法。至少必须提供项目路径（例如，`:model`）。

很多时候，你想要为所有的项目或只有子项目定义一些公共的行为。为实现这些用例，Project API 提供了两个专门的方法：`allprojects` 和 `subprojects`。假设你想要把 Java 插件应用到所有子项目中，因为你需要编译 Java 源代码。你可以通过在 `subprojects` 闭包参数中定义代码来实现。

在多项目构建中项目的默认执行顺序是基于它们的字母名称的。为了显式地控制在构建生命周期的配置阶段的执行顺序，你可以使用项目执行方法 `evaluationDependsOn` 和 `evaluationDependsOnChildren`。对于需要确保为一个项目设置的属性可以被其他项目使用的情况尤其如此。在本章中我们不会讨论这些方法；对于特殊的用例，可能参考 Gradle 在线手册。

在这一章中，你将使用所有提到的方法来配置多项目构建。首先，我们把现有的构建代码应用到指定的子项目中。

### 6.3.2 定义特定的行为

可以通过 `project` 方法来定义特定的项目行为。为了给三个子项目——`model`、`repository`、`web` 搭建构建基础环境，你需要为它们分别创建一个项目配置块。下面的清单显示了在 `build.gradle` 文件中的项目定义。

清单 6.2 定义特定的项目构建逻辑

```
ext.projectIds = ['group': 'com.manning.gia', 'version': '0.1']

group = projectIds.group
version = projectIds.version

project(':model') {
    group = projectIds.group
    version = projectIds.version
    apply plugin: 'java'
}

project(':repository') {
    group = projectIds.group
    version = projectIds.version
    apply plugin: 'java'
}

project(':web') {
    group = projectIds.group
    version = projectIds.version
    apply plugin: 'java'
    apply plugin: 'war'
    apply plugin: 'jetty'

    repositories {
        mavenCentral()
    }

    dependencies {
        providedCompile 'javax.servlet:servlet-api:2.5'
        runtime 'javax.servlet:jstl:1.1.2'
    }
}
```



声明额外属性 `projectIds`，通过 `map` 的形式存储 `group` 和 `version` 键 - 值对；这些属性可以在子项目中使用

通过 `project` 方法配置每一个子项目，实际的配置在闭包中

你会发现解决方案并不完美。即使定义了一个额外属性，将 `group` 和 `version` 属性值指定给每一个子项目，也还是有一些重复代码，并且 Java 插件必须应用于每一个子项目。现在，只是让项目可以运行。稍后我们将改进代码。

### 属性继承

在一个项目中定义的属性会自动被其子项目继承，这个概念也适用于其他构建工具如 Maven。在清单 6.2 中，额外属性 `projectIds` 是在根项目中声明的，它对于 `model`、`repository` 和 `web` 子项目都是可用的。

在多项目构建中，你可以从根目录为某个子项目执行 `task`。你只需要确定连接项目路径和 `task` 名称即可。记住，路径是用冒号 (:) 表示的。例如，为子项目 `model` 执行 `build` `task`，可以通过在命令行引用完整的路径来实现：

```
$ gradle :model:build
:model:compileJava
:model:processResources UP-TO-DATE
:model:classes
:model:jar
:model:assemble
:model:compileTestJava UP-TO-DATE
:model:processTestResources UP-TO-DATE
:model:testClasses UP-TO-DATE
:model:test
:model:check
:model:build
```

这个自包含的子项目 `model` 能够正常工作，因为它并没有依赖其他子项目的代码。如果为子项目 `repository` 执行相同的 `task`，结果会报出编译错误。为什么会这样？因为子项目 `repository` 使用了子项目 `model` 的代码。为了让它正常运行，我们需要声明一个编译时依赖项目。

### 6.3.3 声明项目依赖

声明依赖另一个项目和声明依赖外部类库看起来很相似。在这两种情况下，依赖必须在 `dependencies` 配置块的闭包中声明。必须为项目依赖指定一个特定的配置——在这个例子中，配置 `compile` 是 Java 插件提供的。下面的清单列出了对所有子项目的项目依赖声明。

清单 6.3 声明项目依赖

```
project(':model') {
    ...
}

project(':repository') {
    ...
    dependencies {
        ...
    }
}
```

model 子项目不需要声明任何外部依赖或项目依赖

```
    compile project(':model')
  }
}

project(':web') {
    ...

    dependencies {
        compile project(':repository')
        providedCompile 'javax.servlet:servlet-api:2.5'
        runtime 'javax.servlet:jstl:1.1.2'
    }
}
```

使用 : model 方式声明编译时依赖项目

使用 : repository 方式声明编译时依赖项目

子项目 repository 依赖于子项目 model，子项目 web 依赖于兄弟项目 repository。建模项目依赖关系就是这样的。这样做有三个重要的含义：

- 一个项目依赖关系的实际依赖是它自己创建的库。对于子项目 model 的情况，它就是 JAR 文件。这就是为什么一个项目依赖关系也被称为库依赖的原因。
- 对另一个项目的依赖也要将其传递性依赖添加到 classpath 中。这意味着外部依赖和其他项目依赖也被添加进去。
- 在构建生命周期的初始化阶段，Gradle 确定了项目的执行顺序。对另一个子项目的依赖意味着这个子项目必须被事先构建，毕竟你要依赖它的库。

### 从根项目执行 task

经过初始化阶段后，Gradle 有了一个项目依赖的内部模型。它知道子项目 repository 依赖于 model，子项目 web 依赖于 repository。你不需要从一个特定的子项目执行 task——你可以为构建的所有项目执行 task。假设你想要从根项目执行 build task。事实上 Gradle 知道子项目的执行顺序，你所期望的构建如图 6.9 所示。

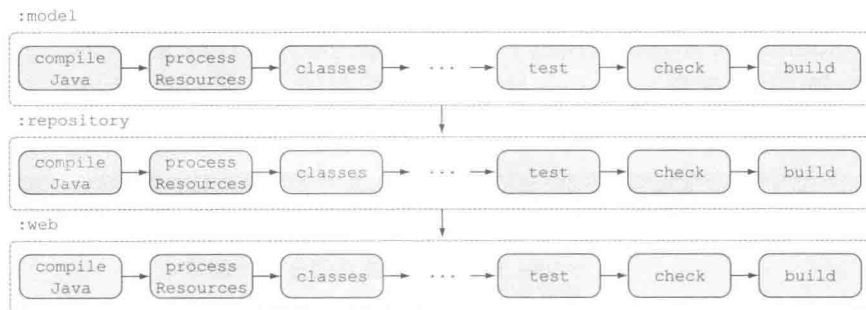


图 6.9 从根项目运行 build task 时多项目 task 的执行顺序



你可以在项目的根级别执行 `build` task 来证明这一点：

```
$ gradle build
:model:compileJava
:model:processResources UP-TO-DATE
:model:classes
:model:jar
:model:assemble
:model:compileTestJava UP-TO-DATE
:model:processTestResources UP-TO-DATE
:model:testClasses UP-TO-DATE
:model:test
:model:check
:model:build
:repository:compileJava
:repository:processResources UP-TO-DATE
:repository:classes
:repository:jar
:repository:assemble
:repository:compileTestJava UP-TO-DATE
:repository:processTestResources UP-TO-DATE
:repository:testClasses UP-TO-DATE
:repository:test
:repository:check
:repository:build
:web:compileJava
:web:processResources UP-TO-DATE
:web:classes
:web:war
:web:assemble
:web:compileTestJava UP-TO-DATE
:web:processTestResources UP-TO-DATE
:web:testClasses UP-TO-DATE
:web:test
:web:check
:web:build
```

从根项目执行 task 节省了项目的构建时间。Gradle 从所有子项目执行 task，包括支持增量式构建。这种做法很方便，而且确保在 `classpath` 中总是有最新的类文件，你可以细粒度地控制何时构建所有依赖的子项目。

### 6.3.4 多项目部分构建

拥有几十个甚至上百个依赖的子项目的复杂多项目构建，将大大影响平均执行时间。Gradle 会遍历所有的项目依赖并确保它们是最新的。在开发阶段，通常你知道在什么子项目中哪些源文件发生了变化。从技术上讲，不需要重新构建没有发生变化的子项目。针对这些情况，Gradle 提供了部分构建特性。部分构建是通过命令行选项 `-a` 或 `--no-rebuild` 启用的。假设在子项目 `repository` 中只改变了代码，但不想重新构建子项目 `model`。通过使用部分构建，可以省去检查子项目

model 的成本并且降低构建执行时间。如果你正工作在一个拥有上百个子项目依赖的企业级项目中，你会感激在执行构建时节省的每一秒。下面的命令行输出显示了该选项的用法：

```
$ gradle :repository:build -a
:repository:compileJava
:repository:processResources UP-TO-DATE
:repository:classes
:repository:jar
:repository:assemble
:repository:compileTestJava UP-TO-DATE
:repository:processTestResources UP-TO-DATE
:repository:testClasses UP-TO-DATE
:repository:test
:repository:check
:repository:build
```

在一个单项目中如果只改变了文件，`--no-rebuild` 选项可以很好地发挥作用。作为日常开发实践的一部分，你想要从仓库中拉取源代码的最新版本，并集成团队成员所做的更改。为了确保代码不会出现异常，你需要重新构建和测试当前项目所依赖的项目。通常 `build task` 只编译依赖项目的代码，组装成 JAR 文件，并且使其作为其他项目的依赖。为了运行测试，可以执行 `buildNeeded task`，如下面的命令行输出所示：

```
$ gradle :repository:buildNeeded
:model:compileJava
:model:processResources UP-TO-DATE
:model:classes
:model:jar
:model:assemble
:model:compileTestJava UP-TO-DATE
:model:processTestResources UP-TO-DATE
:model:testClasses UP-TO-DATE
:model:test UP-TO-DATE
:model:check UP-TO-DATE
:model:build
:model:buildNeeded
:repository:compileJava
:repository:processResources UP-TO-DATE
:repository:classes
:repository:jar
:repository:assemble
:repository:compileTestJava UP-TO-DATE
:repository:processTestResources UP-TO-DATE
:repository:testClasses UP-TO-DATE
:repository:test UP-TO-DATE
```

```
:repository:check UP-TO-DATE
:repository:build
:repository:buildNeeded
```

项目的任何改变都可能对依赖于它的其他项目产生副作用，在 `buildDependents` task 的帮助下，通过构建和测试依赖的项目来验证代码变化所产生的影响。下面的命令行输出显示了它在实际中的使用：

```
$ gradle :repository:buildDependents
:model:compileJava
:model:processResources UP-TO-DATE
:model:classes
:model:jar
:repository:compileJava
:repository:processResources UP-TO-DATE
:repository:classes
:repository:jar
:repository:assemble
:repository:compileTestJava UP-TO-DATE
:repository:processTestResources UP-TO-DATE
:repository:testClasses UP-TO-DATE
:repository:test UP-TO-DATE
:repository:check UP-TO-DATE
:repository:build
:web:compileJava
:web:processResources UP-TO-DATE
:web:classes
:web:war
:web:assemble
:web:compileTestJava UP-TO-DATE
:web:processTestResources UP-TO-DATE
:web:testClasses UP-TO-DATE
:web:test UP-TO-DATE
:web:check UP-TO-DATE
:web:build
:web:buildDependents
:repository:buildDependents
```

### 6.3.5 声明跨项目的 task 依赖


在上一节中，你看到了从根项目执行一个特定的 task 来调用跨所有子项目具有相同名称的所有 task，以及由声明的编译时项目依赖所确定的 build task 执行顺序。如果你的项目不依赖于其他项目，或者为根项目和一个或多个子项目定义了一个具有相同名称的 task，情况就不一样了。

#### 默认的 task 执行顺序

假设你在根项目以及所有子项目中定义了一个名为 `hello` 的 task，如下面的清单所示。在每个 `doLast` 动作中，在控制台上打印出一条消息来指示当前项目。

**清单 6.4 跨项目的 task 定义，没有依赖**

```
task hello << {  
    println 'Hello from root project'  
}  
  
project(':model') {  
    task hello << {  
        println 'Hello from model project'  
    }  
}  
  
project(':repository') {  
    task hello << {  
        println 'Hello from repository project'  
    }  
}
```



为根项目和所有子项目  
声明一个具有相同名字  
的 task

如果从根项目运行这个 hello task，你将看到如下输出：

```
$ gradle hello  
:hello  
Hello from root project  
:model:hello  
Hello from model project  
:repository:hello  
Hello from repository project
```


这些 task 都没有声明依赖另一个 task。那么 Gradle 如何知道按什么顺序来执行 task 呢？很简单：在多项目构建中位于根级别的 task 总是首先执行。对于子项目来说，执行顺序完全取决于项目名称的字母顺序：model 在 repository 之前。记住，在 settings 文件中子项目的声明顺序完全不影响其执行顺序。

**控制 task 执行顺序**

你可以通过声明一个跨项目的 task 依赖来确定 task 执行顺序。为此，你需要引用到不同项目中 task 的路径。下面的清单演示了如何确保子项目 repository 中的 hello task 在子项目 model 之前被执行。

**清单 6.5 声明跨项目的 task 依赖**

```
task hello << {  
    println 'Hello from root project'  
}  
  
project(':model') {  
    task hello(dependsOn: ':repository:hello') << {  
        println 'Hello from model project'  
    }  
}  
  
project(':repository') {  
    task hello << {
```



声明一个 task 依赖于  
子项目仓库中的 task

```
println 'Hello from repository project'
}
}
```

如果从根项目执行 `hello` task，你会发现依赖的 task 以正确的顺序被执行：

```
$ gradle hello
:hello
Hello from root project
:repository:hello
Hello from repository project
:model:hello
Hello from model project
```

控制跨不同项目的 task 执行顺序并不局限于具有相同名称的 task。通过使用相同的机制也可以控制具有不同名称的 task 执行顺序。你所需要做的就是当声明 task 依赖时引用完整的路径。

你已经有了运行多项目构建的基础，并且基本了解了如何控制 task 执行顺序。接下来，我们将讨论定义公共行为的方法，从而提高代码的可读性和可重用性。

### 6.3.6 定义公共行为

在清单 6.2 中，你需要将 Java 插件分别应用于每个子项目。你还创建了一个名为 `projectIds` 的额外属性来定义 `group` 和 `version`。你使用这个额外属性把它们的值赋给根项目和子项的 `Project` 属性。在相当小的项目中，这似乎不是什么大问题，但是如果在拥有超过 10 个子项目的大项目中这样做，就会变得很无趣了。

在本节中，我们将通过使用 `allprojects` 和 `subprojects` 方法来改进现有代码。如何将每个方法应用到多项目构建中，图 6.10 提供了一个直观表示。

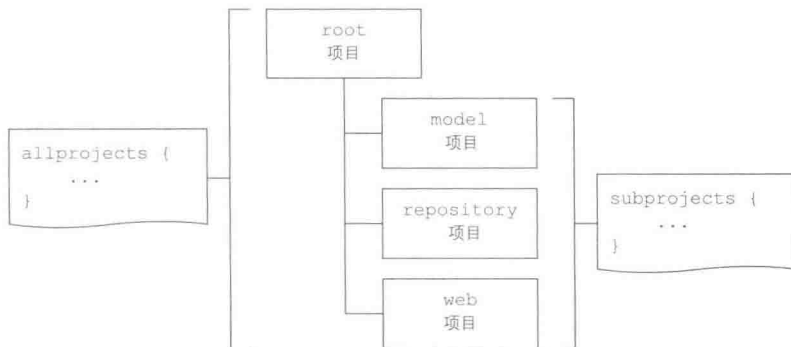


图 6.10 通过 `Project` API 定义公共项目行为

对于项目这意味着什么呢？你可以使用 `allprojects` 方法来设置根项目和

子项目的 `group` 和 `version` 属性。由于根项目没有定义 Java 代码，所以不需要使用 Java 插件，只有子项目需要 Java 插件。你可以使用 `subprojects` 方法将插件应用于所有的子项目。下面的清单展示了在多项目构建中 `allprojects` 和 `subprojects` 方法的用法。

清单 6.6 配置公共项目行为

```
allprojects {  
    group = 'com.manning.gia'  
    version = '0.1'  
}  
  
subprojects {  
    apply plugin: 'java'  
}  
  
project(':repository') {  
    dependencies {  
        compile project(':model')  
    }  
}  
  
project(':web') {  
    apply plugin: 'war'  
    apply plugin: 'jetty'  
  
    repositories {  
        mavenCentral()  
    }  
  
    dependencies {  
        compile project(':repository')  
        providedCompile 'javax.servlet:servlet-api:2.5'  
        runtime 'javax.servlet:jstl:1.1.2'  
    }  
}
```

为根项目和所有子项目设置 `group` 和 `version` 属性

设置 Java 插件只应用于子项目

执行这个构建脚本将产生与前面的构建一样的结果。然而，通过定义公共项目行为使构建脚本变得很清晰，减少了重复的代码，并且提高了构建的可读性。

## 6.4 独立的项目文件

到目前为止，我们所定义的多项目构建只包含一个 `build.gradle` 文件和 `settings.gradle` 文件。随着你在 `build.gradle` 文件中添加新的子项目和 `task`，代码的可维护性将变差。辛苦地读完一页页代码来扩展或修改构建逻辑很无趣。你可以通过为每个项目创建单独的 `build.gradle` 文件来更进一步地分离关注点。

### 6.4.1 为每个项目创建构建文件

我们从设置构建基础环境开始。为每个子项目创建一个遵循默认命名约定的构建文件。下面的目录树显示了最终结果：



项目文件准备好之后，现在你可以把构建逻辑从主构建文件中分离了，并将它移动到合适的位置。

### 6.4.2 定义根项目的构建代码

除去特定子项目的代码后，根级别构建文件的内容看起来相当简单。你只需要保持 `allprojects` 和 `subprojects` 配置块即可，如下面的清单所示。

清单 6.7 根项目的 `build.gradle` 文件

```
allprojects {
    group = 'com.manning.gia'
    version = '0.1'
}

subprojects {
    apply plugin: 'java'
}
```

代码的其余部分都被移动到了子项目的构建文件中。接下来，我们把重点放在定义子项目的构建逻辑上。

### 6.4.3 定义子项目的构建代码

记得子项目 `model` 没有定义特定的项目构建逻辑。事实上，你也不需要声明

一个 project 配置块。因此，你不需要在子项目的 build.gradle 文件中声明任何代码。Gradle 知道这个子项目是多项目构建的一部分，因为在 settings 文件中包含了它的声明。

子项目 repository 和 web 的构建文件不会引入任何新的代码。你可以简单地使用现有的 project 配置块并拷贝到正确的位置。每个项目都有一个专门的 Gradle 文件来表明你正在处理哪个项目。因此，把代码放到一个闭包里是可选的。下面的清单显示了子项目 repository 的构建文件的内容。

#### 清单 6.8 子项目 repository 的 build.gradle 文件

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

子项目 web 的 build.gradle 文件看起来很类似，如下面的清单所示。

#### 清单 6.9 子项目 web 的 build.gradle 文件

```
apply plugin: 'war'  
apply plugin: 'jetty'  
  
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    compile project(':repository')  
    providedCompile 'javax.servlet:servlet-api:2.5'  
    runtime 'javax.servlet:jstl:1.1.2'  
}
```

运行这个多项目构建所产生的结果与在主构建文件中有相同的代码的结果是一样的。上面的构建代码的可读性和可维护性得到了显著的提高。在下一节中，我们将通过示例讨论如何自定义项目。

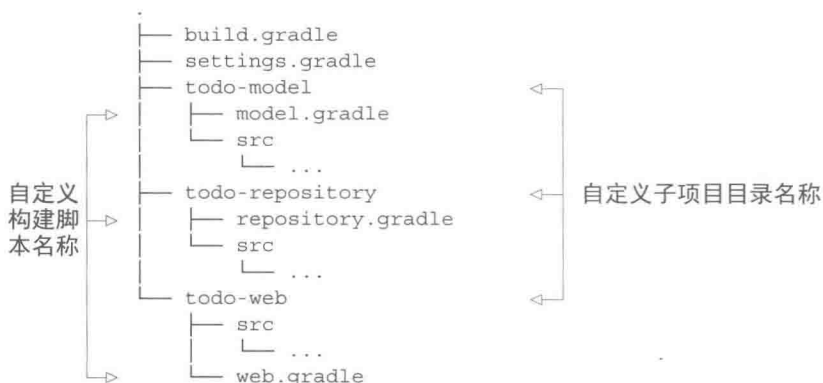
## 6.5 自定义项目

标准的 Gradle 构建文件名是 build.gradle。在拥有许多子项目的多项目构建中，也许你想要使用更具表达性的名称来命名构建文件。在集成开发环境中同时编辑多个 build.gradle 文件并经常在它们之间切换则很容易混淆。这一节将介绍如何配置项目来使用自定义的构建文件名。

假设你想要构建这样的项目结构：每个子项目的目录名称都有前缀 todo- 和一个富有含义的名称。例如，子项目 repository 的目录将被命名为 todo-



repository。构建文件名应该表现出实际项目的职责。下面的目录树显示了你要实现的最终结果：



使得这个项目结构能够工作的关键在于 settings 文件。它提供了更多的功能，而不仅仅是告诉构建应该包括哪些子项目。事实上，它本身是一个构建脚本，在构建生命周期的解析阶段执行。在 6.2.2 节介绍的 Settings API 的帮助下，你可以直接访问根项目和它的子项目。下面的清单显示了如何遍历所有的子项目来指定自定义的构建文件名。此外，你还可以为根项目设置一个自定义的名称。

清单 6.10 在 Settings 文件中自定义项目脚本名称

```

include 'todo-model', 'todo-repository', 'todo-web'
rootProject.name = 'todo'
rootProject.children.each {
    it.buildFileName = it.name + '.gradle' - 'todo-'
}
  
```

通过路径包含子项目

设置根项目的名称

通过根项目遍历所有可访问的子项目

通过使用子项目的名称，添加文件扩展名 .gradle 并去除前缀 todo-，为子项目设置自定义的构建文件名

虽然这个例子可能不适用于真实的项目，但是配置多项目构建的可能性是无止境的。在大多数情况下，配置多项目构建是非常容易的。请记住，Settings API 是最好的帮手。

## 6.6 总结

模块化项目能够提高系统的质量属性——即，可重用性、可维护性和分离关注点。有两条指导性建议可以帮助你实现目标：最小化耦合和最大化内聚。

在这一章中，你已经把 To Do 应用程序代码库分成了模块。你创建了三个模块，一个用于保存模块类，一个用于处理持久化数据，一个用于暴露 Web 应用程序功能。

Gradle 把每个模块都作为一个单独的项目。每个项目都可以声明依赖其他项目。Gradle 的工具箱为建模和执行无论是分层的还是平面项目结构的多项目构建提供了强大支持。在构建生命周期的初始化阶段执行的 `settings` 文件，确定了哪些项目属于构建的一部分。

Project API 提供了声明特定项目构建代码的方法。它还允许配置公共的或特定子项目的构建行为。在当前项目层次结构中项目之间的依赖关系是通过使用与声明外部依赖相同的依赖机制来声明的。

组织多项目构建代码非常灵活。你可以选择使用一个主构建脚本、每个项目单独的构建脚本，或者两者混合一起使用。具体使用哪种方式取决于项目需求。然而，构建中包括的子项目越多，将构建逻辑分离到独立的脚本中越能提高代码的可维护性。

`settings` 文件中可用的 Settings API 适合于在非常规的多项目结构布局中使用。示例中展示了不遵循默认的命名约定，使用自定义的构建脚本名称非常简单。

下一章将专门介绍 Gradle 对测试的支持。我们将探讨使用不同的测试框架来编写单元测试、集成测试和功能测试。我们还将讨论如何为自己的构建脚本编写测试代码。

# 7 Gradle测试

## 本章涵盖

- 理解自动化测试
- 使用不同的框架编写和执行测试
- 配置和优化测试执行行为
- 在构建中支持单元测试、集成测试和功能测试

在前面的章节中，你实现了一个简单但功能齐全的 Web 应用程序，学习了如何使用 Gradle 构建并运行它。测试代码在软件开发生命周期中是一件重要的事情。通过检查软件是否按预期运行来确保软件的质量。在本章中，我们将关注 Gradle 对组织、配置和执行测试代码的支持。特别是为 To Do 应用程序编写单元测试、集成测试和功能测试，并将它们集成到构建中。

Gradle 集成了许多 Java 和 Groovy 的单元测试框架。在这一章的最后，你将使用 JUnit、TestNG 和 Spock 编写测试，并且将这些测试作为构建生命周期的一部分来执行。还将调整默认的测试执行行为。你将了解到控制测试日志输出和添加钩子或监听器对测试生命周期事件做出反应是多么简单的一件事。我们也将探索如何通过 fork 测试进程来提高测试套件的性能。集成测试和功能测试需要更复杂的工具配

置。你将学习如何使用第三方工具 H2 和 Geb 来启动测试代码。

在开始实施测试构建之前，让我们来快速回顾一下不同类型的测试，以及它们各自的优缺点。

## 7.1 自动化测试

我们不打算详细讨论为什么自动化测试方式有利于保证项目的质量。有许多很好的书涵盖了这个话题。长话短说：如果你想构建可靠的、高质量的软件，自动化测试是开发工具箱的关键组成部分。此外，它将有助于减少手工测试的成本，提高开发团队重构现有代码的能力，并帮助在开发生命周期中尽早地发现代码中的缺陷。

### 7.1.1 自动化测试类型

并不是所有的自动化测试都一样。它们通常在使用范围、实现难度和执行时间上存在不同。我们将自动化测试分为三种类型——单元测试、集成测试和功能测试：

- 单元测试与产品代码实现一起作为一个 task 来执行，目的是测试代码的最小单元。在基于 Java 的项目中，这个单元是一个方法。在单元测试中，你需要避免与其他类或外部系统交互（例如，数据库或文件系统）。在被测试代码中对其他组件的引用，通常用测试替身独立出来，这是测试中替代组件的一个通用术语，如 Stub（打桩）或 Mock（模拟）。单元测试很容易编写，可以快速执行，并在开发过程中针对代码的正确性提供宝贵的反馈。
- 集成测试用来测试一个完整的组件或子系统。你想要确保多个类之间的交互是否按预期运行。集成测试的一个典型场景就是验证产品代码和数据库之间的交互。验证的结果是，依赖的子系统、资源以及服务在测试执行中都是可以访问的。集成测试通常比单元测试需要更长的执行时间，而且更加难以维护，失败的原因可能更难以诊断。
- 功能测试通常用于测试应用程序的端到端功能，包括从用户的角度与所有外部系统的交互。当我们谈论用户的角度时，通常指的是用户界面。功能测试是最难实现的和运行最慢的，因为它们需要模拟用户交互。以 Web 应用程序为例，功能测试的工具需要能够点击链接，将数据输入表单字段中，或者在浏览器窗口中提交表单。因为用户界面会随着时间的推移发生改变，维护功能测试代码就会变得乏味而耗时。

### 7.1.2 自动化测试金字塔

你可能在想哪种测试类型最适合项目以及如何扩展。在最理想的情况下，你可

以混合使用这些测试来确保代码能够在不同的架构层级上正常工作。然而，所编写的测试数量是需要花费时间和精力来实现和维护的。所编写的测试越简单，执行就越快，就会得到更高的投资回报率（ROI）。为了优化投资回报率，代码库应该包含大量的单元测试、少量集成测试以及更少的功能测试。测试的分布及其相关 ROI 可以通过自动化测试金字塔最好地展现出来，这是 Mike Cohn 在他的书 *Succeeding with Agile: Software Development Using Scrum* (Addison Wesley, 2009) 中提出来的。图 7.1 显示了 Cohn 的自动化测试金字塔的改编版本。

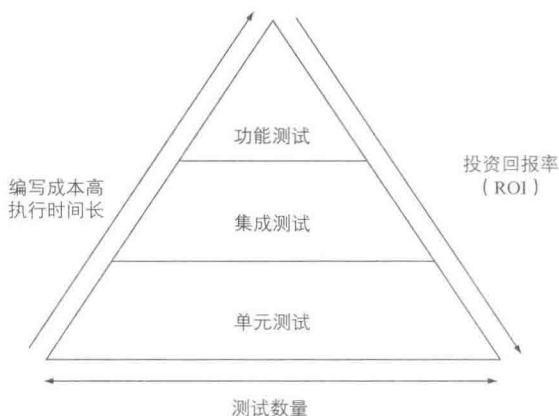


图 7.1 自动化测试金字塔

在本章的后继部分，我们将探索如何使用 Gradle 进行自动化单元测试、集成测试和功能测试。Gradle 的许多开箱即用的测试功能都是通过 Java 插件提供的。让我们仔细看看这些特性。

## 7.2 测试Java应用程序

一般情况下，在 Java 项目中测试代码都是用 Java 编写的。流行的开源测试框架如 JUnit 和 TestNG 可以帮助你编写可重复和结构化的测试。执行这些测试，你需要像编译产品源代码那样，先编译测试代码。测试代码的目的就是仅仅执行测试用例。我们不想将编译测试类放在产品系统中，混合产品源代码和测试代码并不是一个好主意。理想的情况是，项目中有一个专门的目录用来存放测试源代码，以及用另一个目录作为编译测试类的目标目录。

Gradle 的 Java 插件帮你做了这些繁重的工作。它为测试源代码和所需的资源文件引入了一个标准的目录结构，将测试代码的编译和执行集成到构建的生命周期中，并且与几乎所有流行的测试框架兼容得很好。这对于兼容的构建工具如 Ant，在实现相同的功能时有了重大的改进。你只要编写 10~20 行代码就可以很轻松地项目建立一个测试框架。如果这还不够，则只能给每个想使用这个框架的项目拷贝相同

的代码了。

### 7.2.1 项目布局

在第3章中，我们讨论了放置产品源代码的默认目录结构：`src/main/java`和`src/main/resources`。测试源代码也遵循相同的模式。把测试代码源文件放到`src/test/java`目录，把测试代码所需的资源文件放到`src/test/resources`目录。编译测试源代码后，最终的`class`文件处于输出目录`build/classes/test`下，非常完美地与编译的产品`class`文件分离了。

所有的测试框架至少生成一个工件来表明测试执行的结果。XML通常是记录结果的通用格式。在`build/test-results`目录下可以找到这些文件。XML文件的可读性不是很好。它们通常需要被其他质量保证工具做进一步的处理，我们将在第12章中具体介绍。许多测试框架允许将结果转换成一个报告。例如，JUnit会默认生成一个HTML报告。Gradle将测试报告放在`build/reports/test`目录下。图7.2概述了Java插件所提供的标准测试目录。

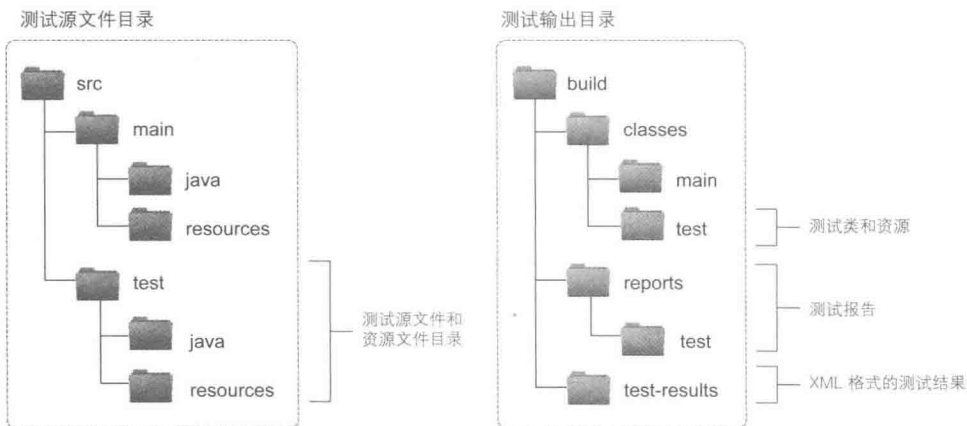


图 7.2 标准的测试源文件目录和输出目录

针对上面所讨论的测试框架，如何让 Gradle 使用一个特定的框架呢？你需要声明依赖外部类库。

### 7.2.2 测试配置

Java 插件引入了两个新的配置，用来声明测试代码编译或执行所需的依赖类库：`testCompile`和`testRuntime`。让我们来看一个例子，声明了一个编译时依赖 JUnit：

```
dependencies {
    testCompile 'junit:junit:4.11'
}
```

另一个配置 `testRuntime`，用于声明不需要在测试编译阶段使用，但是需要在运行时使用的依赖。记住，测试配置中的依赖不影响产品代码的 `classpath`。换句话说，它们不会参与编译或打包的过程。然而，测试配置扩展了用来处理产品源代码所需依赖的特定配置，如图 7.3 所示。配置 `testCompile` 会自动指定 `compile` 的依赖。配置 `testRuntime` 扩展了 `runtime` 和 `testCompile`，以及它们的父配置。

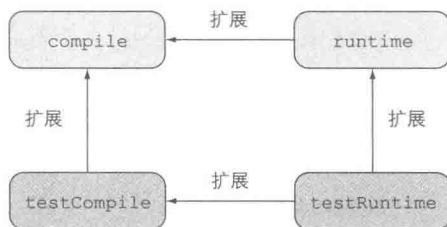


图 7.3 测试配置的继承层级

## 7.2.3 测试 task

在执行前面的例子时，你可能已经注意到 task 图中包含了 4 个 task，它们总是最新的，并因此被跳过。这是因为你没有编写任何 Gradle 需要编译或执行的测试代码。图 7.4 展示了 Java 插件所提供的测试 task，以及它们是如何适应现有的 task 顺序的。

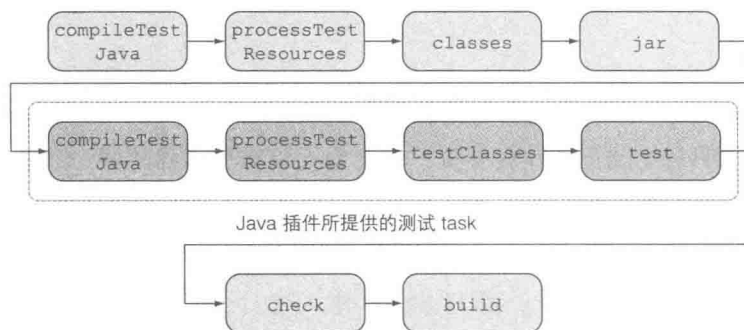


图 7.4 测试 task 与构建生命周期无缝集成

如图中所示，测试的编译和执行发生在产品代码编译和打包之后。如果你想避免执行测试阶段，则可以在命令行上运行 `gradle jar` 或者让 task 定义依赖于 `jar` task。

## 7.2.4 自动化测试检测

在 `build/classes/test` 目录下的编译测试类中，Gradle 如何指定运行哪些类呢？答案是这个目录下所有的类文件只要满足下面的描述条件就可以执行：

- 继承 `junit.framework.TestCase` 或 `groovy.util.GroovyTestCase` 的任何类或父类。
- 使用 `@RunWith` 注解的任何类或父类。
- 至少包含一个使用 `@Test` 注解的方法的任何类或父类（注解的实现可以来自 JUnit 或 TestNG）。

如果上述条件不满足或者扫描到的类是抽象类，那么它就不会被执行。现在是在一个完整的例子中使用你所学过的知识了。在下一节中，你将在不同测试框架的帮助下编写单元测试，并且使用 Gradle 来执行测试。

## 7.3 单元测试

作为 Java 开发人员，你可以从众多的测试框架中选择一个。在本节中，将使用传统的测试框架 JUnit 和 TestNG，但也会看看新成员 Spock。如果你刚开始了解这些测试框架，请参考它们的在线文档，因为我们不会介绍如何编写一个测试基础知识。

### 7.3.1 使用 JUnit

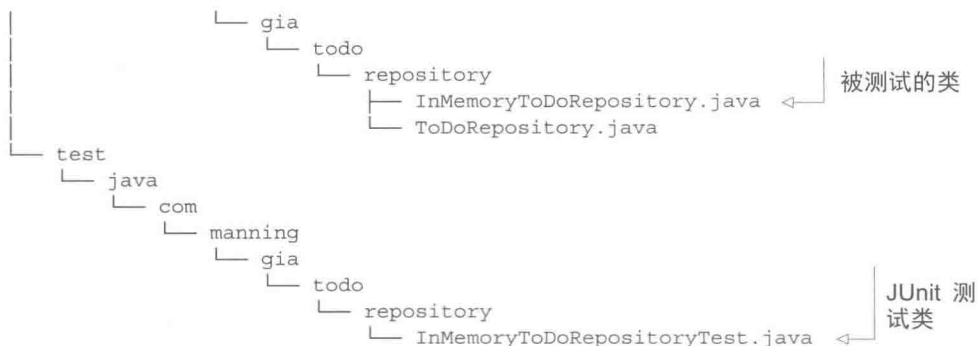
为 To Do 应用程序的存储实现 `InMemoryToDoRepository.java` 编写一个 JUnit 测试。为了强调测试框架和与 Gradle 集成之间的相同点和不同点，所有的单元测试都将验证同一个类的功能的正确性。同时，你需要让测试和构建代码适应特定测试框架的需求。

#### 编写测试类

假设需要为子项目 `repository` 编写测试类。这个测试文件应该放在标准的测试源目录下。在 `src/test/java` 目录下创建一个名为 `InMemoryToDoRepositoryTest.java` 的 Java 测试类：

```
.
├── build.gradle
└── src
    ├── main
    │   └── java
    │       └── com
    │           └── manning
    └── test
```





在测试驱动开发理念中，你先编写一个断言失败的测试。如果测试成功，也就是你的假设是正确的，它将给你信心。下面的清单显示了 JUnit 测试用例的实现，验证插入功能的正确性。

#### 清单 7.1 使用 Junit 编写测试类

```

package com.manning.gia.todo.repository;

import com.manning.gia.todo.model.ToDoItem;
import org.junit.Before;
import org.junit.Test;

import java.util.List;

import static org.junit.Assert.*;

public class InMemoryToDoRepositoryTest {
    private ToDoRepository inMemoryToDoRepository;

    @Before
    public void setUp() {
        inMemoryToDoRepository = new InMemoryToDoRepository();
    }

    @Test
    public void insertToDoItem() {
        ToDoItem newToDoItem = new ToDoItem();
        newToDoItem.setName("Write unit tests");
        Long newId = inMemoryToDoRepository.insert(newToDoItem);
        assertNull(newId);

        ToDoItem persistedToDoItem = inMemoryToDoRepository.findById(newId);
        assertNotNull(persistedToDoItem);
        assertEquals(newToDoItem, persistedToDoItem);
    }
}

```

使用这个注解标记的方法  
总会在每个类的测试方法  
之前执行

使用这个注解标记的方法  
将作为测试用例运行

错误的断言故意放  
在这里，用来触发  
失败测试

测试类准备就绪了，让我们来看看如何为构建添加测试支持。

## 添加依赖

你已经了解了 `testCompile` 配置。下面的清单显示了如何将 JUnit 4.11 版本的依赖指定给配置。现在 `testCompile` task 就可以使用 JUnit 来编译测试源文件了。

清单 7.2 在子项目 `repository` 中声明依赖 JUnit

```
project(':repository') {
    repositories {
        mavenCentral()
    }

    dependencies {
        compile project(':model')
        testCompile 'junit:junit:4.11'
    }
}
```

声明依赖 JUnit

所有的配置就是这些。在项目中你可以使构建使用 JUnit 作为测试框架了。接下来，通过执行 `test` task 来证明关于失败断言的假设。

## 执行测试

在上一节中，了解了 `test` task 首先要编译产品源代码，然后创建 JAR 文件，而测试源代码编译和测试执行紧随其后。下面的命令行输出表明测试断言错误导致构建失败：

```
$ gradle :repository:test
:model:compileJava
:model:processResources UP-TO-DATE
:model:classes
:model:jar
:repository:compileJava
:repository:processResources UP-TO-DATE
:repository:classes
:repository:compileTestJava
:repository:processTestResources UP-TO-DATE
:repository:testClasses
:repository:test

com.manning.gia.todo.repository.InMemoryToDoRepositoryTest
> testInsertToDoItem FAILED
    java.lang.AssertionError at InMemoryToDoRepositoryTest.java:24

1 test completed, 1 failed
:repository:test FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':repository:test'.
> There were failing tests. See the report at:
  file:///Users/ben/dev/gradle-in-action/code/chapter07/junit-test-
  failing/repository/build/reports/tests/index.html

失败的测试方法名

发生异常的测试文件和代码行

测试结果总结，包括成功、失败和跳过的测试用例

HTML 格式的测试报告地址
```

在控制台输出中，你会看到一个断言的失败。这正是你所期望的结果。它展示的信息并没有说明测试失败的原因。你只知道在第 24 行断言失败。如果你有庞大的测试套件，那么只能通过测试报告来查找测试失败的原因。你可以通过在 INFO 日志级别运行 task 来获取更多的测试输出信息：

```
$ gradle :repository:test -i
...
com.manning.gia.todo.repository.InMemoryToDoRepositoryTest
  > testInsertToDoItem FAILED
    java.lang.AssertionError: expected null, but was:<1>
    at org.junit.Assert.fail(Assert.java:88)
    at org.junit.Assert.failNotNull(Assert.java:664)
    at org.junit.Assert.assertNull(Assert.java:646)
    at org.junit.Assert.assertNull(Assert.java:656)
    at com.manning.gia.todo.repository.InMemoryToDoRepositoryTest
    .testInsertToDoItem(InMemoryToDoRepositoryTest.java:24)
...
```

通过命令行选项改变日志级别，并不是控制测试日志输出的唯一方法。在本章后面，我们将讨论在构建脚本中配置测试日志的选项。

在跟踪栈中，你可以看到失败断言发生在 InMemoryToRepositoryTest 类的第 24 行。假设 newId 的值是 null。事实是数据存储中的每条记录都应该被唯一地标识，所以这个属性应该有一个值。在测试方法中通过指定一个非空的 ID 值来修改这个断言：

```
assertNotNull(newId);
```

再次运行 test task，显示所有的测试都通过了：

```
$ gradle :repository:test
:model:compileJava
:model:processResources UP-TO-DATE
:model:classes
:model:jar
:repository:compileJava
:repository:processResources UP-TO-DATE
:repository:classes
:repository:compileTestJava
:repository:processTestResources UP-TO-DATE
:repository:testClasses
:repository:test
```

接下来，我们看看所生成的 HTML 报告。

## 检查测试报告

Gradle 产生了一个比 Ant 或 Maven 更具视觉吸引力的测试报告。正如之前所说

的，可以在 `build/reports/test` 目录下找到 HTML 报告。打开 HTML 页面，应该呈现如图 7.5 所示的类似截图信息。

### Test Summary



图 7.5 成功的 JUnit 测试 HTML 报告

报告总结了运行测试的数量、失败率和执行时间。可以通过点击选项卡来切换包和类的视图。在至少有一个测试失败的情况下，将显示另一个选项卡，可以查看失败断言的整个跟踪栈信息。

#### 可点击的报告 URL

导航到测试报告目录，然后双击 HTML 索引文件，随着时间的推移可能会变得非常无趣。当然，你可以把这个 URL 存在书签里，但是 Gradle 为这个手动操作提供了很好的快捷方式。在一些操作系统中，输出文件的 URL 在控制台中是可点击的，通过这种方式同样可以在浏览器中打开 HTML 报告。

- *Linux*：在终端直接点击
- *Mac OS*：Cmd + 双击
- *Windows*：不支持

这个特性不仅仅适用于失败的测试执行。任何产生报告文件的 task 在控制台都提供了一个可点击的 URL。

在 Gradle 中 JUnit 的标准的单元测试框架；然而，Gradle 允许你选择不同的解决方案。让我们讨论一下如何集成其他单元测试框架，甚至在一个项目中使用多个测试框架。

### 7.3.2 使用其他的单元测试框架

在项目中，你可能会选择使用其他的单元测试框架，而不是 JUnit。你选择的原因可能各有不同，但通常是基于功能集合的，如开箱即用的 Mock 支持或者用于编

写测试的语言等。在本节中，我们将介绍如何在构建中使用 TestNG 和 Spock 框架。我们不会详细讲解如何使用不同的单元测试框架来编写测试类。在本书的源代码以及在线文档中找到例子。我们只关注将这些框架集成到构建中的具体细节。

### 使用 TestNG

假设使用 TestNG 编写和我们之前讨论的相同的测试类。包名和类名必须相同。在测试类内部，使用特定的 TestNG 注解来标记相关方法。为了让构建执行 TestNG 测试，你需要做两件事：

- 声明依赖 TestNG 类库。
- 通过调用 `Test#useTestNG()` 方法来指定 TestNG 需要被用来执行测试。此外，也可以通过 `org.gradle.api.tasks.testing.testng.TestNGOptions` 类型的 Closure 参数来配置。请查看在线用户手册来了解更多的信息。

下面的清单展示了与 TestNG 集成的完整构建脚本。

清单 7.3 支持 TestNG 测试

```
project(':repository') {
    repositories {
        mavenCentral()
    }

    dependencies {
        compile project(':model')
        testCompile 'org.testng:testng:6.8'
    }

    test.useTestNG()
}
```

声明对 TestNG 的依赖

在项目中支持 TestNG 测试

运行 `gradle:repository:test` 后，你会注意到 task 执行顺序与 JUnit 例子是一样的。Gradle 早期版本所生成的测试报告的界面外观和 JUnit 报告有些不同，从 1.4 版本开始，测试报告看起来就完全一样了。

### 使用 Spock

Spock 是一个测试和规范框架，遵循了行为驱动开发（BDD）的概念。以 BDD 风格编写的测试用例会有一个明确的标题，并且按照 `given/when/then` 方式进行叙述。Spock 通过 Groovy 的 DSL 提供这些测试。结果就是测试用例非常具有可读性和表达性。

Spock 与 JUnit 完全兼容。每个测试类都需要扩展基类 `spock.lang.Specification`，它是 Spock 类库的一部分。使用 `@RunWith` 注解标记这个类，允

许使用专门的 JUnit Runner 实现来运行测试。

假设通过 Groovy 使用 Spock 编写测试类。为了能够在默认的源目录 `src/test/groovy` 下编译 Groovy 类，需要为项目应用 Groovy 插件。Groovy 插件需要你在项目中将想要使用的 Groovy 版本声明为依赖。因为你需要使用 Groovy 来编译测试源代码，所以必须在 `testCompile` 配置中指定相应的类库。除了 Groovy 类库，你也可以声明 Spock 类库。下面的清单展示了编译和执行 Spock 测试所需的配置。

清单 7.4 使用 Spock 来编写和执行单元测试

```
project(':repository') {  
    apply plugin: 'groovy'           ← 给项目添加 Groovy 支持  
  
    repositories {  
        mavenCentral()  
    }  
  
    dependencies {  
        compile project(':model')  
        testCompile 'org.codehaus.groovy:groovy:2.0.6' ← 为编译测试代码指定 Groovy 类库  
        testCompile 'org.spockframework:spock-core:0.7-groovy-2.0' ← 声明依赖 Spock 类库  
    }  
}
```

所生成的 HTML 测试报告的界面外观与 JUnit 和 TestNG 测试报告一致。不管你选择使用哪个测试框架，Gradle 都提供了相同的报告方法。你不必为选择使用哪个单元测试框架做决定。所有的这些测试框架在同一个项目都可以结合使用。

### 7.3.3 结合使用多个单元测试框架

对于长期的项目，随着时间的推移测试策略可能会发生变化。团队从一个测试框架切换到另一个框架是很平常的事情。显然，你不想用新的测试框架来重写所有现有的测试类。你想要保留它们，并且作为构建的一部分来运行。最重要的是，你想要生成一个汇集了所有测试结果的测试报告。那么该怎么做呢？

#### 定义测试 task

在前面的章节中，我们讨论了如何一次集成一个单元测试框架。假设你想要支持通过前面我们讨论的所有测试框架来编写单元测试。一个额外的需求就是对测试类的命名约定：

- *JUnit*：所有的测试类名字以 `*Test.java` 结尾。
- *TestNG*：所有的测试类名字以 `*NGTest.java` 结尾。
- *Spock*：所有的测试类名字以 `*Spec.groovy` 结尾。

你已经看到 TestNG 支持通过调用 `useTestNG()` 方法来配置。然而，默认的 `test` task 要么执行 JUnit 测试，要么执行 TestNG 测试。为了支持两者，你需要添加一个新的 Test 类类型的 task，命名为 `testNG`。这个新的 task 通过让 `test` task 依赖于它，从而很容易地集成到测试生命周期中，如图 7.6 所示。



图 7.6 将额外的测试 task 集成到构建生命周期中

结果就是一个构建执行了三种类型的测试类。JUnit 和 Spock 测试是通过 `test` task 执行的，TestNG 测试是通过 `testNG` task 执行的。下面的清单展示了通过对现有构建的最小变化来支持使用多个测试框架。

#### 清单 7.5 配置构建执行 JUnit、TestNG 和 Spock 测试

```

project(':repository') {
    apply plugin: 'groovy'

    repositories {
        mavenCentral()
    }

    dependencies {
        compile project(':model')
        testCompile 'junit:junit:4.11'
        testCompile 'org.testng:testng:6.8'
        testCompile 'org.codehaus.groovy:groovy:2.0.6'
        testCompile 'org.spockframework:spock-core:0.7-groovy-2.0'
    }

    task testNG(type: Test) {
        useTestNG()
    }

    test.dependsOn testNG
}

```

为了执行 TestNG 测试类增强的 task

创建 task 依赖于 TestNG 测试 task

#### 汇集 HTML 测试报告

通过 `gradle :repository:test` 执行构建暴露了一个缺点：HTML 报告索引页面不包含所有的测试结果。这是因为第二次运行生成的报告会覆盖第一次的。因此，它只包含了 JUnit 和 Spock 测试结果。这可以通过合并所有测试 task 的测试结果来解决。下面的清单显示了如何创建一个新的 `org.gradle.api.tasks.testing.TestReport` 类型的 task 来生成综合报告。

## 清单 7.6 汇集测试报告

```
task aggregateTestReports(type: TestReport) {
    destinationDir = test.reports.html.destination
    reportOn test, testNG
}

check.dependsOn aggregateTestReports
```

添加 TestReport 类型的 task 来汇集报告

将 task 集成到构建生命周期中

为了将这个 task 集成到构建生命周期中,你需要添加它作为 check task 的依赖,如图 7.7 所示。执行 build task 将自动汇集测试报告 task。

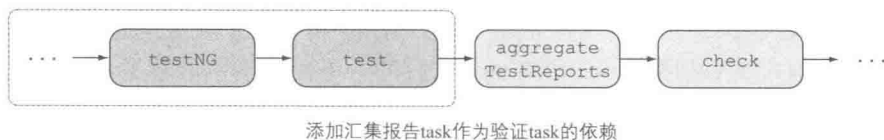


图 7.7 在构建生命周期中添加汇聚测试报告 task

运行 gradle build 后,你会发现在 build/reports/test 目录下汇集的 HTML 测试报告。它看起来类似于图 7.8 所示的截图。

### Package com.manning.gia.todo.repository

all > com.manning.gia.todo.repository



#### Classes

Class	Tests	Failures	Duration	Success rate
<a href="#">InMemoryToDoRepositoryNGTest</a>	1	0	0.005s	100%
<a href="#">InMemoryToDoRepositorySpec</a>	1	0	0.218s	100%
<a href="#">InMemoryToDoRepositoryTest</a>	1	0	0.001s	100%

图 7.8 汇集的 HTML 测试报告

在实际中,你会发现需要调整测试执行行为来满足自己的需要。下一节将探索可用的配置选项,以及如何将它们应用到构建中。

## 7.4 配置测试执行

测试执行是构建生命周期中一个重要的阶段。Gradle 在构建脚本中提供了各种各样的配置选项,以及命令行参数来控制运行时行为。如何以及何时应用这些选项



完全取决于构建的需要。本节将简单地概述常用功能和这些选项背后的 API 类。我们先从一些有用的命令行选项开始介绍。

### 7.4.1 命令行选项

拥有庞大的测试套件的项目要求细粒度地控制你想要执行的测试。有时你可能想要只运行一个测试或者特定包或项目下的测试。如果一个或多个测试失败了，你想要修复并且重新运行它们，而不是执行整个测试套件，那么就会出现这种情况。

#### 按模式执行测试

Gradle 提供了如下系统属性来应用特定的测试命名模式：`<taskName>.single=<testNamePattern>`。假设你想执行所有包中的 Spock 测试。项目中的 Spock 测试类有命名约定 `*Spec.groovy`（例如，`InMemoryToDoRepositorySpec.groovy`）。在命令行中，表达方式如下：

```
$ gradle -Dtest.single=**/*Spec :repository:test
```

这仅仅是一个简单的定义测试命名模式的例子。如果想要了解全部的模式选项，请参阅 Java 插件的在线文档。

#### 远程测试调试

有时测试失败的根本原因是很难识别的，特别是测试不能作为一个单元测试独自运行时。通过 IDE 远程调试测试是一个非常有用的工具。Gradle 为启用远程调试提供了一个方便的快捷方式：`<taskName>.debug`，这意味着你也可以同样对其他 task 进行调试。使用这个启动参数会在 5005 端口启动一个服务器 Socket，并且会阻塞 task 执行，直到你真正通过 IDE 连接上它：

```
$ gradle -Dtest.debug :repository:test
...
:repository:test
Listening for transport dt_socket at address: 5005
> Building > :repository:test
```

与此同时，你可以通过选择部署 IDE 在代码中设置断点，并连接到端口。一旦连接成功，就将恢复 task 执行，并且可以单步调试代码。每个 IDE 远程连接调试器的步骤各有不同，请查阅相应的文档来了解指令信息。

虽然这些命令行选项在日常业务中能够派上用场，但是你可能需要一种更持久的方式来配置测试执行：在构建脚本中配置。

## 7.4.2 理解测试 API 表示

API 的入口点是 `org.gradle.api.tasks.testing.Test` 类, 让你能够配置测试执行的特定行为。Test 类扩展了 `DefaultTask`, 可以用来在构建脚本中创建特定的测试 task。事实上, Java 插件提供的 test task 是一个预配置的增强的 Test 类型的 task。你可以通过其所暴露的 API 来更改默认的行为。图 7.9 显示了主要的测试 API 及其相关的类。该类图主要显示了本章所涉及的方法。想要深入了解 API, 请查阅 DSL 指南或 Javadocs。

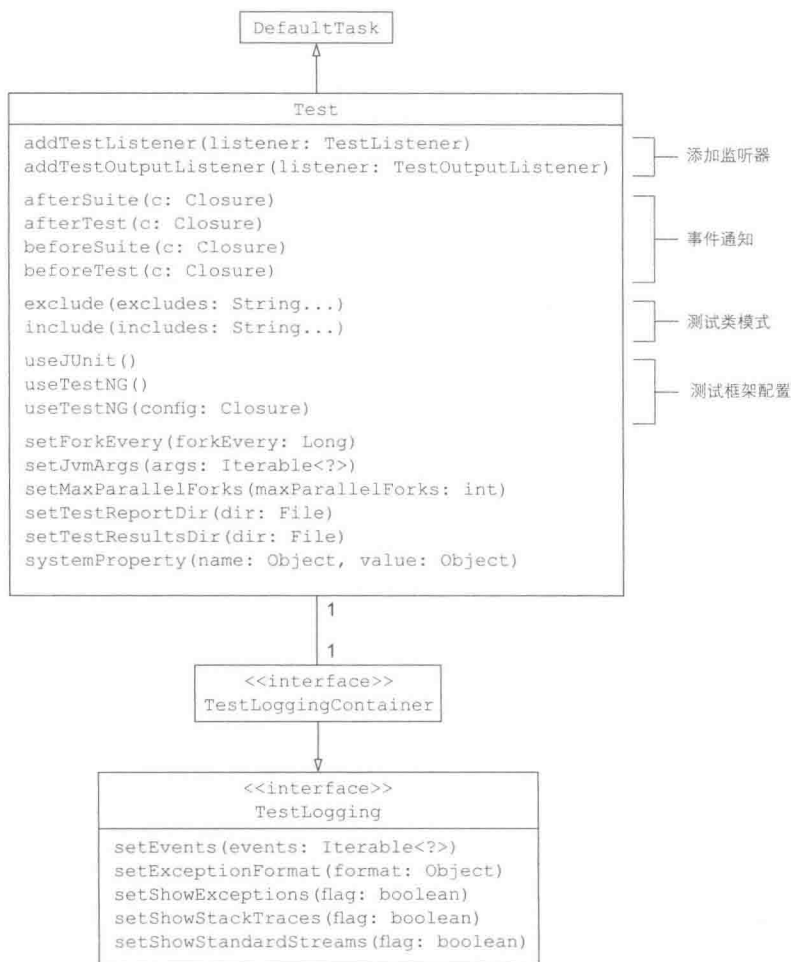


图 7.9 相关的 Test API 类

了解了该类图, 现在开始使用这些配置选项。接下来的场景会让你知道如何使用它们。

### 7.4.3 控制运行时行为

Gradle 会在一个 forked JVM 进程中运行测试。这样做，你可以得到与通常启动一个 Java 进程一样的所有好处。你可以通过传入选项来调整垃圾回收和性能调优，或者在代码中提供所需使用的系统属性。

假设你在测试方法上做了最小的改变。与其每次只插入一个 to-do 项，还不如通过系统属性 items 来配置可插入的 to-do 项的数量。下面清单中显示的 createAndInsertToDoItems 方法接受了所提供的系统属性值，并确定填入多少个 task 合适。

清单 7.7 获取系统属性来驱动测试执行

```
public class InMemoryToDoRepositoryTest {  
    ...  
  
    @Test  
    public void insertToDoItems() {  
        int items = System.getProperty("items") != null ?  
            Integer.parseInt(System.getProperty("items")) : 1;  
        createAndInsertToDoItems(items);  
        List<ToDoItem> toDoItems = inMemoryToDoRepository.findAll();  
        assertEquals(items, toDoItems.size());  
    }  
  
    private void createAndInsertToDoItems(int items) {  
        System.out.println("Creating " + items + " To Do items.");  
  
        for(int i = 1; i <= items; i++) {  
            ToDoItem toDoItem = new ToDoItem();  
            toDoItem.setName("To Do task " + i);  
            inMemoryToDoRepository.insert(toDoItem);  
        }  
    }  
}
```

解析所提供的系统属性

打印创建和插入的 to-do 项的数量

现在，你该如何告诉 Gradle 使用一个系统属性在测试中来驱动 to-do 项的创建呢？你可以简单地在 Test task 中调用 `systemProperty` 方法，并且提供一个名字和值作为参数。可以想象到，项目数量越多，就越容易占满内存。与此同时，你也可以通过调用 `jvmArgs` 方法调整 JVM 内存设置，来避免潜在的 `OutOfMemoryErrors` 错误。下面的清单展示了 test task 中的方法调用。

## 清单 7.8 提供系统属性和 JVM 参数

```
test {
    systemProperty 'items', '20'
    minHeapSize = '128m'
    maxHeapSize = '256m'
    jvmArgs '-XX:MaxPermSize=128m'
}
```

设置系统属性

提供 JVM Heap 设置

设置 JVM 的 Perm Gen 的最大大小

根据你为系统属性 `items` 提供的值，完成测试的时间可能会有所不同。关于所提供的值是否正确，你没有得到直接的反馈。如果仔细查看清单 7.7，你可能会注意到将所提供的项目数量打印到了标准输出流中。然而，当运行测试时，你不会看到输出。让我们看看如何通过控制测试日志来改变这种情况。

## 7.4.4 控制测试日志

控制日志对于在测试执行期间诊断问题是非常有帮助的。通过 `testLogging` 属性访问 `TestLoggingContainer` 接口是改变默认配置的关键。你可以进一步了解这个类，因为我们不会介绍所有选项。

## 标准流日志

在清单 7.7 中，你试图编写一条消息发送给标准输出流。Gradle 的一个 `Test` 配置选项是开关 `Boolean` 标志，将标准输出和错误消息打印到终端，如下面的清单所示。

## 清单 7.9 把日志标准流输出到终端

```
test {
    testLogging {
        showStandardStreams = true
    }
}
```

打开标准输出和错误流的日志

正如所期望的，运行 `gradle :repository:test`，在终端显示了 `System.out.println` 语句：

```
$ gradle :repository:test
...
:repository:test
com.manning.gia.todo.repository.InMemoryToDoRepositoryTest
> testInsertToDoItems STANDARD_OUT
    Creating 20 To Do items.
...
```

## 异常跟踪栈日志

前面你看到了通过在 INFO 日志级别运行构建如何打印失败测试的异常跟踪栈信息。这种方法的缺点是终端也包含了与诊断测试失败原因不相关的其他信息。你可以通过 `exceptionFormat` 方法永久地改变测试异常日志的格式。下面的清单提供了 `full` 值，用来告诉 Gradle 在 INFO 日志级别下运行构建时独立打印出全部的异常跟踪栈信息。

### 清单 7.10 显示异常跟踪栈信息

```
test {  
    testLogging {  
        exceptionFormat 'full'    ← 显示全部的异常跟踪栈信息  
    }  
}
```

## 测试事件日志

在 Gradle 的测试执行默认设置中，不会给出任何信息告诉你有多少测试被运行，以及有多少测试通过了、失败了或跳过了。只有在至少一个测试失败的情况下才会打印出总结信息。`events` 方法允许你传入一个想要记录的事件类型列表。下面的清单演示了测试开始、通过、跳过或失败时如何在终端记录一条信息。

### 清单 7.11 记录特定的测试事件

```
test {  
    testLogging {  
        events 'started', 'passed', 'skipped', 'failed' ← 在测试执行期间打印特定的测试事件  
    }  
}
```

执行测试并且打开 `started`、`passed`、`skipped` 和 `failed` 事件的日志，会产生下面的输出：

```
$ gradle :repository:test  
...  
:repository:test  
com.manning.gia.todo.repository.InMemoryToDoRepositoryTest  
  > testInsertToDoItem STARTED  
com.manning.gia.todo.repository.InMemoryToDoRepositoryTest  
  > testInsertToDoItem PASSED  
...
```

每个事件都用单独的一行和颜色编码来记录。没有发生的事件——在这个用例中，跳过的和失败的——没有被记录。甚至还记录了更多的事件。请参阅在线文档

来了解所有可用的选项。

### 7.4.5 并行执行测试

Gradle 在一个单独的、forked 进程中执行测试。执行拥有上千个测试用例的庞大的测试套件可能需要几分钟到几小时不等，因为它们是按顺序运行的。鉴于现在计算机具有极快的多核处理器，你应该充分利用它们的计算能力。

Gradle 提供了一种方便的方法来并行执行测试。你需要做的就是指定 forked JVM 进程的数量。此外，你可以为每个 forked 测试进程设置执行测试类的最大数量。下面的清单使用一个简单的公式来计算在机器中可用处理器的情况下 forked JVM 进程的数量。

#### 清单 7.12 配置 forked 测试进程

```
test {  
    forkEvery = 5  
    maxParallelForks = Runtime.getRuntime().availableProcessors() / 2  
}
```

在 forked 测试进程中执行  
测试类的最大数量

fork 测试进程的最大数量

让我们想象一下基于拥有 18 个测试类的测试套件的执行行为。下面的清单显示了可并行的测试进程数量，这个数量的计算是基于对 JVM 可用的虚拟或物理的逻辑核心数量的。我们假设这个数量是 4。因此，maxParallelForks 属性的值是 2。forkEvery 属性设置为 5，每个 forked 测试进程执行一组 5 个测试类。图 7.10 演示了在运行时如何执行测试。



图 7.10 每次使用两个 forked 进程执行测试

在这个例子中指定的数量并不是一成不变的。在项目中如何配置并行执行测试依赖于目标硬件和测试类型（CPU 或 I/O 限制）。试着设置数量值以找到平衡点。为了获得更多的关于如何找到最佳平衡点的信息，建议你阅读 Venkat Subramaniam 所写的 *Programming Concurrency on the JVM* (The Pragmatic Programmers, 2011)。

### 7.4.6 响应测试生命周期事件

在第4章中，你了解到可以很容易地挂接构建生命周期，在事件发生时执行相应的代码。Gradle 为任何 Test 类型的 task 暴露了生命周期方法。特别是可以监听下面的事件：

- `beforeSuite`：测试套件执行前
- `afterSuite`：测试套件执行后
- `beforeTest`：测试类执行前
- `afterTest`：测试类执行后

图 7.11 显示了当这些事件为 Java 插件所提供的默认 test task 注册时是如何应用到构建生命周期当中的。

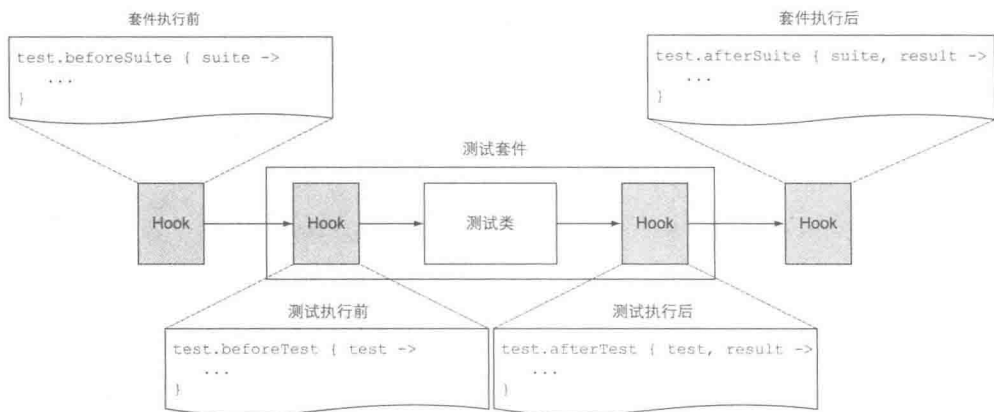


图 7.11 注册测试生命周期钩子

假设你想知道执行完测试套件中的测试需要多长时间。为了搞清楚这一点，你可以通过 `afterSuite` 方法挂接测试生命周期。下面的清单展示了如何使用传入闭包中的参数来计算执行时间，并将此信息作为通知发送到桌面。

清单 7.13 测试套件执行后执行代码

```

apply plugin: 'announce'

test.afterSuite { TestDescriptor suite, TestResult result -> {
    if (!suite.parent && result.getTestCount() > 0) {
        long elapsedTestTime = result.getEndTime() - result.getStartTime()
        announce.announce("""Elapsed time for execution of
            ↳ ${result.getTestCount()} test(s):
            ↳ $elapsedTestTime ms""", 'local')
    }
}

```

测试套件执行后添加通知的闭包

检查测试套件是否至少包含一个测试

使用 announce 插件显示关于测试执行时间的通知

这是一种简单的和直接的显示测试套件执行时间的方法。然而，你看不到先前运行测试套件的跟踪记录。你可以轻松地将这个数据发送到数据库，并且随着时间的推移做成可视化图。

注册测试事件方法非常适合于特别的功能。它的缺点是不能很容易地在项目之间共享。这时 `TestListener` 实现就可以发挥作用了。

### 7.4.7 实现测试监听器

`TestListener` 是一个用于监听测试执行事件的接口。所实现的功能与上一节讨论的相同。你需要在构建脚本中创建一个新的 `NotificationTestListener` 类。唯一需要实现的方法是 `afterSuite`。所有的其他方法都可以是空实现。下面的清单完整地展示了监听器的实现，以及如何注册包含 `test task` 的类。

清单 7.14 为默认的 `test task` 添加测试监听器

```
project(':repository') {
    apply plugin: 'announce'
    ...
    test.addTestListener(new NotificationTestListener(project))
}

class NotificationTestListener implements TestListener {
    final Project project

    NotificationTestListener(Project project) {
        this.project = project
    }

    @Override
    void afterSuite(TestDescriptor suite, TestResult result) {
        if(!suite.parent && result.getTestCount() > 0) {
            long elapsedTestTime = result.getEndTime() - result.getStartTime()
            project.announce.announce("Elapsed time for execution of
                                     ➡ ${result.getTestCount()} test(s):
                                     ➡ $elapsedTestTime ms", 'local')
        }
    }

    @Override
    void afterTest(TestDescriptor testDescriptor, TestResult result) {}

    @Override
    void beforeSuite(TestDescriptor suite) {}

    @Override
    void beforeTest(TestDescriptor testDescriptor) {}
}
```

为构建添加测试监听器

`TestListener` 实现用于通知用户关于测试套件执行的时间

在第 4 章中，你了解到如果把类放在 `buildSrc` 目录下，则可以很轻松地在项



目间共享它们。这同样适用于 `TestListener` 实现。

在本节中，我们将快速地了解相关配置选项。这些选项并不只针对用于处理单元测试的 `task`。它们也可以被应用到集成测试和功能测试中。接下来，我们将讨论如何为 `To Do` 应用程序编写集成测试并将它们集成到构建中。

## 7.5 集成测试

单元测试是验证系统中代码的最小单元、一个方法，可以孤立地正常工作。这使你可以实现快速运行、可重复的并且一致的测试用例。集成测试超出了单元测试的范围。它们通常集成系统的其他组件或外部的基础环境，如文件系统、邮件服务器或数据库。因此，集成测试通常需要更长的时间来执行。通常它们还依赖于一个系统的正确状态——例如，一个具有特定内容的现有文件——会使它们变得很难维护。

### 7.5.1 引入用例研究

集成测试的一个常见的场景是验证持久层是否按预期工作。目前，你的应用程序只将对象存储在内存中。可以通过 `SQL` 让它与数据库交互来改变现状。为了达到这个目的，你可以使用开源的数据库引擎 `H2` (<http://www.h2database.com/>)。H2 很容易搭建，并且提供了快速启动时间功能，非常适合于这个例子。

在第 3 章中，你为持久层提供了一个接口。这使得它很容易提供 `ToDoRepository` 的不同实现，而且方便在 `Web` 层交换它们。H2 完全兼容 `JDBC`。与数据库的任何交互都是在实现接口的新类 `H2ToDoRepository` 中实现的。我不想重复介绍使用 `JDBC` 类的细节，所以这里不会详细地讨论代码。如果你想要深入了解，可下载代码示例，其中包含了所有相关的代码。

### 7.5.2 编写测试类

把所有类型的测试都放在同一个目录下是很常见的做法。考虑到集成测试通常比单元测试需要更长的执行时间，你可以按照命名约定来划分测试。开发人员可以在他们的本地机器上重复运行单元测试，得到关于代码变化的快速反馈。

作为项目中集成测试的命名约定，让测试类的名字以 `IntegTest` 后缀结束。对于 H2 仓库实现的集成测试 `H2ToDoRepositoryIntegTest`，看起来与单元测试极其相似。唯一的最大不同之处是测试的类是 `H2ToDoRepository`，如下面的清单所示。集成测试代码与现有的单元测试类位于同一个包内。

清单 7.15 测试 H2 数据库持久化代码

```
package com.manning.gia.todo.repository;

import com.manning.gia.todo.model.ToDoItem;
import org.junit.Before;
import org.junit.Test;

import java.util.List;

import static org.junit.Assert.*;

public class H2ToDoRepositoryIntegTest {
    private ToDoRepository h2ToDoRepository;

    @Before
    public void setUp() {
        h2ToDoRepository = new H2ToDoRepository();
    }

    @Test
    public void testInsertToDoItem() {
        ToDoItem newToDoItem = new ToDoItem();
        newToDoItem.setName("Write integration tests");
        Long newId = h2ToDoRepository.insert(newToDoItem);
        newToDoItem.setId(newId);
        assertNotNull(newId);

        ToDoItem persistedToDoItem = h2ToDoRepository.findById(newId);
        assertNotNull(persistedToDoItem);
        assertEquals(newToDoItem, persistedToDoItem);
    }
}
```

应用程序的集成测试有命名约定 \*IntegTest

创建 H2 数据访问类的实例

这个测试类验证的断言与单元测试相同。在实际中，你会有更多的测试用例来测试与数据库的交互。接下来，你需要关注构建部分。

### 7.5.3 在构建中支持集成测试

你所编写的测试侧重于测试代码和数据库之间的集成点。这意味着你需要一个可访问的 H2 数据库，并且运行在正确的模式下。为每个环境提供一个数据库实例被认为是好的实践（例如，开发、QA 和产品环境）。在下面的内容中，假定你不必应对数据库的管理和配置。所有的事情都为你设置好了。在构建中，你想要支持三个基本需求：

- 提供独立的 task 来执行单元测试和集成测试。
- 分离单元测试与集成测试的运行结果和报告。
- 让集成测试作为验证生命周期 task check 的一部分。

在前面的章节中，你已经了解了实现这一目标的技术。你可以使用 Test API

的属性和方法。下面的清单展示了如何通过特定的命名模式来包括或排除测试类。同时也重载了测试结果和报告的默认目录。

清单 7.16 定义 task 来运行集成测试

```
project(':repository') {
    repositories {
        mavenCentral()
    }

    dependencies {
        compile project(':model')
        runtime 'com.h2database:h2:1.3.170'
        testCompile 'junit:junit:4.11'
    }

    test {
        exclude '**/*IntegTest.class'
        reports.html.destination = file("${reports.html.destination/unit}")
        reports.junitXml.destination = file("${reports.junitXml.destination/
            unit}")
    }

    task integrationTest(type: Test) {
        include '**/*IntegTest.class'

        reports.html.destination = file("${reports.html.destination/
            integration}")
        reports.junitXml.destination = file("${reports.junitXml.destination/
            integration}")

        check.dependsOn integrationTest
    }
}
```

定义单元测试结果和报告的输出目录

添加 JDBC 驱动作为运行时依赖来连接 H2 数据库

从默认的测试 task 中排除集成测试

按照类的命名约定只包含集成测试

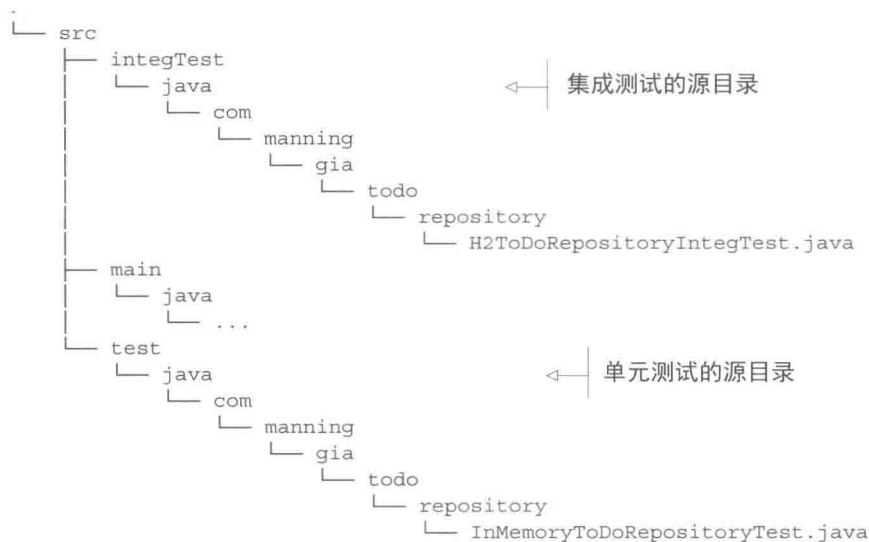
定义集成测试结果和报告的输出目录

添加集成测试作为 check task 的依赖

在命令行运行 `gradle :repository:build` 将调用测试 task 运行单元测试和集成测试。在同一个源文件夹中混合不同类型的测试刚开始时听起来可能是一个好主意。随着测试类数量的增加，项目中的测试源将难以导航和区分。你需要让团队中的每个开发人员都遵循测试类的命名约定。如果这种模式没有被严格遵守的话，那么一个测试类有可能会在一个不期望的构建阶段执行。正如你所了解的那样，集成测试通常比单元测试需要更长的时间来执行，因此这将是一个致命的缺点。有了 Gradle，你可以做得更好。你可以强制通过命名约定将单元测试和集成测试划分为不同的 source set。

### 7.5.4 为集成测试建立约定

假设你想把所有的单元测试放在 `src/test/java` 目录下，把集成测试移动到 `src/integTest/java` 目录下。为集成测试创建新的测试源目录之后，项目结构应该是这样的：



Gradle 为将不同类型的测试分离到源目录中提供了一个干净的解决方案。在第 3 章中，你了解了如何通过 Java 插件预配置默认的源目录。添加新的 source set 是另一个选择。这就是你需要为集成测试所做的事情。

#### 为集成测试定义一个 source set

每个 source set 定义中的源代码都需要在执行之前被编译，并且拷贝到正确的目录下。集成测试也是这样的。还记得使用 Ant 来实现同样的需求吗？你必须编写与编译和执行单元测试相似的代码。如何解决这个问题呢？一般是通过复制和粘贴代码，并且修改目标目录来实现的，这通常不是一个好的实践。高级 Ant 用户会考虑编写一个自定义 task。

使用 Gradle 可以有不同的方法。这就是 Gradle 的神奇之处。在项目中，定义要做什么：添加一个新的测试源代码目录。如何编译源代码，就交给 Gradle 了。事实上，Gradle 会自动帮你做决定，并且为新的 source set 隐式地添加一个新的编译 task。下面的清单显示了在项目中如何定义新的集成测试 source set。

清单 7.17 为集成测试定义一个 source set

```

sourceSets {
    integrationTest {
        java.srcDir file('src/integTest/java')
        resources.srcDir file('src/integTest/resources')
        compileClasspath = sourceSets.main.output
        runtimeClasspath = output + compileClasspath
    }
}

```

集成测试源目录

集成测试资源目录

指定编译时 classpath

指定运行时 classpath

在源代码示例中你可以看到 `source set` 需要一些额外的配置。它需要你指定编译时 `classpath`，包含产品代码类和所有指派给 `testRuntime` 配置的依赖。你还需要定义运行时 `classpath`，包含通过 `output` 变量可以直接访问的编译的集成测试类和编译时 `classpath`。

### 在集成测试 task 中使用 source set

任何 `Test` 类类型的 `task` 如果没有被另外配置的话，都将使用默认的配置。因为集成测试 `source set` 的类输出目录不同于默认目录，你需要将 `integrationTest` `task` 指定给这个目录。你还需要考虑重新配置该 `task` 的 `classpath`。下面的代码片段显示了 `integrationTest` `task`，并且指定了属性值：

```

task integrationTest(type: Test) {
    testClassesDir = sourceSets.integrationTest.output.classesDir
    classpath = sourceSets.integrationTest.runtimeClasspath
}

```

将测试 task 指定给可以找到测试类的目录

执行测试所需要的 classpath

试一试。下面的输出显示了在命令行运行 `build` `task` 的结果：

```

$ gradle :repository:build
...
:repository:assemble
:repository:compileIntegrationTestJava
:repository:processIntegrationTestResources UP-TO-DATE
:repository:integrationTestClasses
:repository:integrationTest
:repository:compileTestJava
:repository:processTestResources UP-TO-DATE
:repository:testClasses
:repository:test
:repository:check
:repository:build

```

通过添加一个新的 `source set`，Gradle 自动添加所需要的 `task` 进行编译和处理集成测试源代码；`task` 名称派生自 `source set` 名称。

构建代码准备就绪后，你就可以很好地在单元测试和集成测试之间分离关注点了。接下来，我们将讨论在本地机器上自动化设置数据库作为构建的一部分。

### 7.5.5 引导测试环境

与外部系统集成时存在一个不足之处，需要能够从执行构建的机器上访问它们。如果不能这样，集成测试就会失败。为了确保有一个稳定的测试环境，可以从构建中引导测试所需要的资源。

H2 提供了轻量级的基于 Java 的工具来管理和控制数据库，它可以被很容易地集成到构建中。假设你希望通过启动 H2 来建模集成测试的生命周期，通过 SQL 脚本重建整个结构，依靠数据库实例运行测试，最后关闭 H2 数据库。你需要建立的 task 看起来类似于图 7.12。

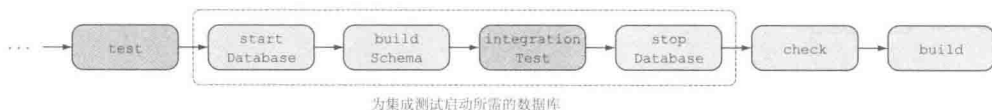


图 7.12 为集成测试启动、准备和停止数据库

在构建中引导测试环境是非常多样化的、特定产品，并且可以根据项目需求进行定制。你可能需要架设一台邮件服务器或者部署其他应用程序来暴露 Web 服务。重要的是你可以按照自己的需要进行控制。

讨论如何针对 H2 数据库实现这一目标的细节超出了本书的范围。然而，本书提供了一个工作示例的源代码，你可以尝试和探索。运行这个示例，你会发现命令行的输出类似于下面这样：

```
$ gradle :repository:build
...
:repository:compileTestJava
:repository:processTestResources UP-TO-DATE
:repository:testClasses
:repository:test
:repository:startDatabase
TCP server running at tcp://localhost:9092 (only local connections)
:repository:buildSchema
:repository:startAndPrepareDatabase
:repository:integrationTest
:repository:stopDatabase
Shutting down TCP Server at tcp://localhost:9092
:repository:check
:repository:build
```

在 TCP 端口 9092 启动本地 H2 数据库

从头开始重新构建数据库架构

集成测试运行完成后关闭本地 H2 数据库

对如何编写集成测试有了基本了解之后，我们将关注自动化测试金字塔的顶部：功能测试。

## 7.6 功能测试

功能测试是从终端用户的角度来验证软件满足需求。在 Web 应用程序的上下文中，这意味着模拟用户与浏览器之间的交互，如在文本框中输入值或点击链接。从历史上来看，功能测试难以编写，而且维护成本高。你需要一个工具可以自动启动浏览器，操作 Web 页面中的 DOM 元素，并且支持在不同的浏览器中运行这些测试。除此之外，你还需要把功能测试集成到构建中，并且以自动和可重复的方式运行它们。让我们看一个具体的用例和一个自动化工具，帮助你编写功能测试。

### 7.6.1 引入用例研究

当在 UI 层设计一个功能测试时，问自己以下一些问题很有益处：

- 想要测试什么功能？例如，一个 to-do 列表在项目数量超过 10 个时支持分页。
- 高级用户的工作流程是什么？例如，只有在用户插入了 11 个项目时 to-do 列表才能分页。
- 采取什么样的技术措施来实现这一目标？例如，用户打开一个浏览器并输入 URL/ 查看所有的 to-do 列表项。为了插入一个新的 to-do 项目，输入这个新的 to-do 项目的名称并按 Enter 键。这个 UI 交互调用了 URL/ 插入将 to-do 项目添加到列表中。重复这个步骤 11 次，验证显示了分页。

出于这个目的，我们选择一个简单的用例：打开 URL 显示 to-do 列表项。插入一个新的 to-do 项目，在文本框中输入“Write functional tests（编写功能测试）”，然后按 Enter 键。通过检查 to-do 列表项，验证这个新的项目被成功添加到列表中。这个测试假设 to-do 列表是空的。图 7.13 演示了页面工作流程。



图 7.13 记录页面工作流程

了解了这些 UI 交互，让我们看一个工具，用来帮助实现这些需求。一个开源

的可以经受住这些需求挑战的工具是 Geb (<http://www.gebish.org/>)。Geb 建立在流行的浏览器自动化框架 Selenium 之上，并且允许你使用可读性很强的 Groovy DSL 定义测试。测试类可以使用 JUnit、TestNG 或 Spock 框架来编写。这意味着如果你知道这些测试框架的用法，并且浏览了 Geb 的 DSL 文档，那么就可以很好地设置来编写第一个功能测试了。

现在，你可以假设先前描述的业务工作流的测试类是通过 Geb 使用测试框架 JUnit 建立的。本书不会教你如何使用 Geb 编写测试，因为它需要用一两章来讲解。把它留给你自己去探索所提供的代码示例。所有的测试都被配置为专门针对 Mozilla Firefox 工作的。如果你没有安装 Firefox，现在就安装。值得一提的是，Geb 也允许针对其他浏览器执行测试。为了了解更多的信息，请查看 Geb 在线文档和源代码示例。接下来，我们将为组织和执行测试准备构建。

## 7.6.2 在构建中支持功能测试

功能测试需要 Web 应用程序的一个运行良好的实例。许多组织为此专门提供了一个特定的运行时环境。假设你想要修改构建来支持功能测试，并且可以使用这样的环境。乍一看，对构建的需求看起来类似于为集成测试定义的需求：

- 为功能测试定义一个新的 source set。
- 提供一个新的 task 来执行功能测试，并且在专门的输出目录中生成测试结果 / 报告。
- 集成功能测试作为验证生命周期的一部分。

### 引入和使用自定义的功能测试配置

Geb 有自己的一套依赖需要被声明。对于编译测试代码，你需要指定 Geb JUnit 实现和 Selenium 的 API（如果测试需要的话）。为了运行测试，你还需要提供 Selenium 驱动来远程控制 Firefox 浏览器。

你可以轻松地将这些依赖指定给现有的 testCompile 和 testRuntime 配置。不足之处是会混合功能测试与单元测试和集成测试的 classpath，这可能导致版本冲突。为了尽可能地保持功能测试的 classpath 清晰，应将它与其他的测试类型分离，我们来看两个新的配置：functTestCompile 和 functTestRuntime。图 7.14 显示了如何将它们应用到由 Java 插件引入的现有的配置层次结构中。



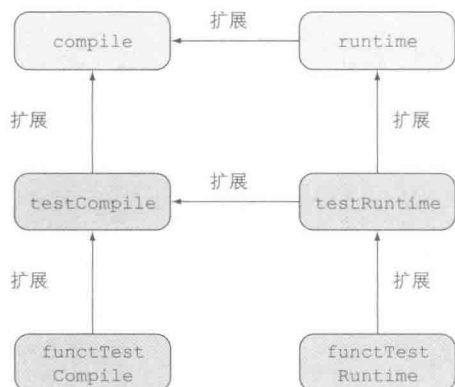


图 7.14 为指定功能测试依赖引入的配置

Geb 针对 UI 来测试应用程序工作是否正常。因此，在 Web 项目中添加测试定义就显得非常重要。下面的清单显示了为功能测试依赖定义所需配置的基本设置。

#### 清单 7.18 声明功能测试配置和依赖

```
project(':web') {
    apply plugin: 'war'
    apply plugin: 'jetty'
    apply plugin: 'groovy'

    repositories {
        mavenCentral()
    }

    configurations {
        functTestCompile.extendsFrom testCompile
        functTestRuntime.extendsFrom testRuntime
    }

    ext.seleniumGroup = 'org.seleniumhq.selenium'
    ext.seleniumVer = '2.32.0'

    dependencies {
        compile project(':repository')
        providedCompile 'javax.servlet:servlet-api:2.5'
        runtime 'javax.servlet:jstl:1.1.2'
        testCompile 'org.codehaus.groovy:groovy:2.0.6'
        testCompile 'junit:junit:4.11'
        functTestCompile 'org.codehaus.gib:geb-junit4:0.7.2'
        functTestCompile "$seleniumGroup:selenium-api:$seleniumVersion"
        functTestRuntime "$seleniumGroup:selenium-firefox-
            ➡ driver:$seleniumVersion"
    }
    ...
}
```

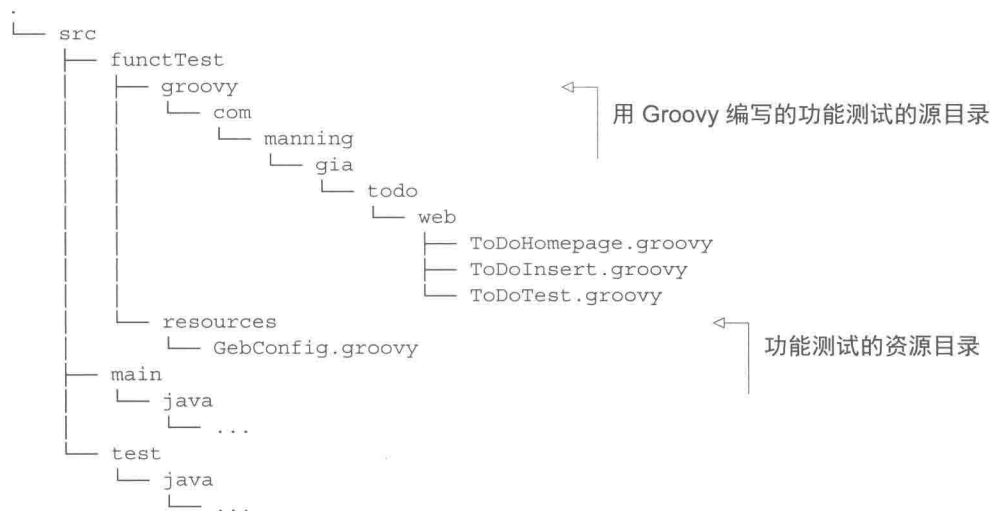
功能测试的编译和运行时配置

声明编译  
依赖  
Geb JUnit  
支持和  
Selenium  
的 API

声明运行时依赖  
Selenium 驱动来  
远程控制 Firefox

## 定义 source set 和测试 task

你可能已经注意到在清单 7.18 中使用了 Groovy 插件。Geb 测试是用 Groovy 编写的，而且需要为编译和运行时测试指定外部类库。下面的目录树显示了在示例项目中测试的存放位置：



你需要为功能测试建立一个新的 source set，指向 `src/functTest/groovy` 和 `src/functTest/resources` 目录。实现方式类似于集成测试 source set。最大的区别在于必须将自定义配置指定给相关的 classpath 属性，如下面的清单所示。

清单 7.19 功能测试 source set

```
sourceSets {
    functionalTest {
        groovy.srcDir file('src/functTest/groovy')
        resources.srcDir file('src/functTest/resources')

        compileClasspath = sourceSets.main.output +
            ➡ configurations.functTestCompile <-- 将自定义的功能
        runtimeClasspath = output + compileClasspath +
            ➡ configurations.functTestRuntime <-- 测试配置指定给
                                                    classpath 属性
    }
}
```

定义新的 source set 之后，你可以在一个新的增强的 Test 类类型的 task 中使用它的类输出目录和运行时 classpath。下面的清单显示了 functionalTest task，它将测试结果和报告写入到自定义目录中。

清单 7.20 在 functionalTest task 中使用 source set

```

task functionalTest(type: Test) {
    testClassesDir = sourceSets.functionalTest.output.classesDir
    classpath = sourceSets.functionalTest.runtimeClasspath
    reports.html.destination = file("$reports.html.destination/functional")
    reports.junitXml.destination = file("$reports.junitXml.destination/functional")
    systemProperty 'geb.env', 'firefox'
    systemProperty 'geb.build.reportsDir', reporting.file("$name/geb")
}

```

指定自定义的测试结果和报告目录

强制性的 Geb 系统属性

Geb 还需要设置一些强制性的系统属性。其中一个属性就是运行测试针对的浏览器名称。如果你想要针对多个浏览器运行测试，则需要单独创建测试 task 和传入合适的值。请参阅 Geb 文档来了解更多信息。

有了这些构建脚本，你可以针对通过网络可访问的 Web 应用程序的一个实例运行功能测试。接下来，我们将更进一步，提供一种专门在本地机器上运行功能测试的方式。

### 针对嵌入式 Jetty 运行功能测试

在本地机器上运行功能测试需要你在嵌入式 Servlet 容器中部署 Web 应用程序。它将提供测试页面。好处在于你不需要依赖服务器来运行测试。

你已经知道了如何使用 Jetty 插件来部署应用程序。在默认情况下，jettyRun task 将阻塞构建执行，直到用户按“Ctrl+C”停止进程。但是这并不能帮助你运行功能测试。幸运的是，Jetty 插件可以被配置在后台线程中通过 daemon 属性来执行嵌入式容器。为此，你需要创建一个增强的 task 来启动和停止 Servlet 容器，如下面的清单所示。

清单 7.21 为功能测试声明增强的 Jetty task

```

ext {
    functionalJettyStopPort = 8081
    functionalJettyStopKey = 'stopKey'
}

task functionalJettyRun(type: org.gradle.api.plugins.jetty.JettyRun) {
    stopPort = functionalJettyStopPort
    stopKey = functionalJettyStopKey
    contextPath = 'todo'
    daemon = true
}

task functionalJettyStop(type: org.gradle.api.plugins.jetty.JettyStop) {
    stopPort = functionalJettyStopPort
    stopKey = functionalJettyStopKey
}

```

增强的 task 用于启动嵌入式 Jetty 容器

确定在后台运行容器 (task 不会被阻塞)

增强的 task 用于停止嵌入式 Jetty 容器

现在你有了两个专门的 task 用来控制 Web 应用程序运行时环境，可以在它们之间加入 functionalTest task。图 7.15 展示了需要建模的 Gradle task 的顺序，将功能测试集成到构建中。



图 7.15 浏览器的自动化测试 task

task 依赖链可以帮助你实现这个目标。链中最后一个 task 应该是验证 task check，如清单 7.22 所示。check task 是由 Java 插件提供的生命周期 task，它依赖于任何验证 task，如 test。如果你想要自动执行整个测试 task 链，这个 task 是最便捷的。

#### 清单 7.22 将功能测试集成到构建生命周期中

```
functionalTest.dependsOn functionalJettyRun
functionalTest.finalizedBy functionalJettyStop
check.dependsOn functionalTest
```

← 在执行功能测试之前启动  
Jetty 容器

← 在执行功能测试之后停止  
Jetty 容器

对于 web 项目执行 gradle build 命令会开始以下动作：首先编译功能测试类，在后台启动嵌入式 Jetty 容器，自动打开 Firefox 浏览器，然后按照测试定义进行远程控制。所有测试运行后，Jetty 关闭，命令的控制台输出如下：

```
$ gradle :web:build
...
:web:compileFunctionalTestJava UP-TO-DATE
:web:compileFunctionalTestGroovy
:web:processFunctionalTestResources
:web:functionalTestClasses
:web:functionalJettyRun
:web:functionalTest
:web:functionalJettyStop
...
```

## 7.7 总结

自动化测试是确保应用程序的功能正确性的一个重要工具，而且是有效重构的直接推动者。单元测试、集成测试和功能测试的范围、实现方式以及执行时间都不相同。你看到了由 Mike Cohn 提出的自动化测试金字塔是如何显示与项目的 ROI 相

关的这些标准的。越容易实现的测试，执行速度就越快，ROI 也就越高。测试类型的 ROI 越高，满足这种类型的测试用例就越多。

Gradle 的 Java 插件提供了广泛的开箱即用的测试支持。通过应用插件，你的项目就会自动知道到哪里去搜索测试类，编译并执行它们，暴露配置用来指定所需的测试依赖，并生成具有吸引力的 HTML 报告。

在本章中，你了解了在 JUnit、TestNG 和 Spock 三种流行的测试框架的帮助下如何实现单元测试。Gradle 的 Test API 在配置测试执行时发挥了重要作用。我们详细讨论的两个例子可以被直接运用到真实项目中。当试图确定测试失败的根本原因时，细粒度地控制测试日志有很大的好处。测试类作为庞大的测试套件的一部分可以被并行运行，以最小化它们的执行时间，并且利用硬件的处理能力来充分发挥其能力。

集成测试和功能测试比单元测试更难以编写和维护。集成测试通常涉及调用其他组件、子系统或外部服务。我们讨论了如何结合一个运行的 SQL 数据库来测试应用程序的数据持久层。功能测试从用户的角度来验证应用程序的正确性。在自动化测试框架的帮助下，可以远程控制浏览器和模拟用户交互。配置构建为不同类型的测试提供 source set，提供新的 task，把它们集成到构建生命周期中，甚至在需要时引入测试环境。

下一章将讨论如何使用插件来扩展构建脚本。我们不仅会实现一个具有完整功能的真实插件，而且会在构建中使用它，还可以通过验证其功能来进一步扩展测试的主题。



# 扩展Gradle 8

## 本章涵盖

- 通过例子学习 Gradle 的扩展机制
- 编写、使用脚本和对象插件
- 测试定制化任务和插件
- 通过扩展对象从构建脚本中获取配置

在前面的章节中，我们讨论过如何通过 Gradle 构建一个自包含的样例项目。通过在构建脚本中声明简单 `task` 和定制化 `task` 类来添加定制逻辑。通常，你会想要将一个非常有用的 `task` 共享到多个项目中使用。Gradle 提供了多种不同的方法来重用代码，但每种方法都有各自的优缺点。插件让重用和扩展的概念走得更远。它们通过引入某个特定的问题领域的约定和模式为项目添加新的能力。

在本书前面，你已经看到插件的强大。在第 3 章中，使用 `Java`、`War` 和 `Jetty` 插件实现了一个任务来管理 Web 应用。使用这些插件就像写一行代码一样简单，而且为构建添加了新的能力。`Java` 插件添加了编译、测试和打包 Java 代码的标准方法。`War` 插件能够构建一个 WAR 文件，而 `Jetty` 插件可以将 WAR 文件部署到一个嵌入式的 Servlet 容器中。

所有这些插件都是很小的、可选的框架，它们会引入一些默认约定和项目布局。然而，说到构建 Java 或者 Web 应用，你的看法也许会有些不同。Gradle 承认定制

化是一个必要的特性。所有的核心插件都允许修改默认约定，这样可以更容易地适应非标准项目。

在本章中，你要学习如何组织、实现、测试和构建自己的定制化插件。我们的讨论会涉及如何重用代码、编写灵活的任务和引入约定优于配置的概念等开发实践。让我们从一个实际应用开始吧。

## 8.1 通过案例学习介绍插件

在本书中使用的例子是一个用 Java 实现的基于 Web 的 To Do 应用。通过 Gradle 的开箱即用插件的支持，你能够创建一个 WAR 文件，将它部署到一个嵌入式的 Servlet 容器中，并在浏览器中测试它的功能。我敢打赌你一定渴望通过一个网络可访问的 Web 容器部署应用，向终端用户炫耀你的开发成果。传统的方法是自己去管理 Web 服务器。虽然你对该设施有完全的控制权，但是购买和维护是非常昂贵的。还记得上一次你不得不让你的基础设施团队给你的应用提供一台服务器和兼容的运行环境吗？硬件和软件的供应会耽误软件进入市场的时间，最终，你甚至没有调整应用运行时参数的 root 权限。快速、简单地宿主一个应用程序的办法是使用 PaaS（平台即服务），部署平台和解决方案栈的结合在许多情况下是免费的。PaaS 将传统的应用服务器功能与可扩展性、负载均衡和高可用性的支持相结合。让我们一起将 Gradle 带入到这个等式中。

### 8.1.1 在云中使用 Grade 管理应用

手动地部署应用到服务器是一个重复而且易犯错的任務。幸运的是，许多 PaaS 服务商都提供了 API，可以程式管理他们的平台服务和资源。在 Gradle 的帮助下，你可以自动地部署应用到远程容器，让它成为项目构建生命周期中的一部分。每个 Web 应用程序都需要在开发过程中的某个点在运行时环境中启动，这样以可重用方式编写代码才变得有意义。遗憾的是，你找不到一个实现了该功能的 Gradle 核心插件，所以你需要自己去实现。这是练习构建工具扩展机制的好机会。在开始编写代码之前，让我们来选择一个满足需要的 PaaS 提供商。

在过去的几年里，许多 JVM PaaS 提供商都迅速发展起来。有些供应商提供了某种编程模型需要你去遵从他们指定的软件栈和 API，例如一个专有的数据存储实现。你不想把自己困在里面，因为你希望以后能够将 Web 应用转移到一个不同的服务器环境中。你可以使用 CloudBees 的 RUN 服务，也叫作 RUN@cloud，一个基础设施不可知的部署平台和应用运行时环境。CloudBees 提供了一个基于 Java 的客户端类库，可以通过 HTTP 与运行时服务通信。在 Gradle 脚本中使用该类库非常直接：将它定义为一个 classpath 的依赖，然后写 task 使用该 API 即可。



图 8.1 展示了在 Gradle 构建脚本缩略图中如何与 RUN@cloud 交互。CloudBees API 提供了一个基于 HTTP 的编程接口来管理在 RUN@cloud 平台上的服务和应用。但是如果你的同事也想要在他们的项目中使用相同的 task，该怎么办？为了避免催促就简单地复制代码！真正的好朋友是不会让你复制和粘贴代码的——对于构建逻辑也是一样。正确的方式是将代码格式化成 Gradle 插件。你将通过在 CloudBees 上创建账户来开始你的旅程。

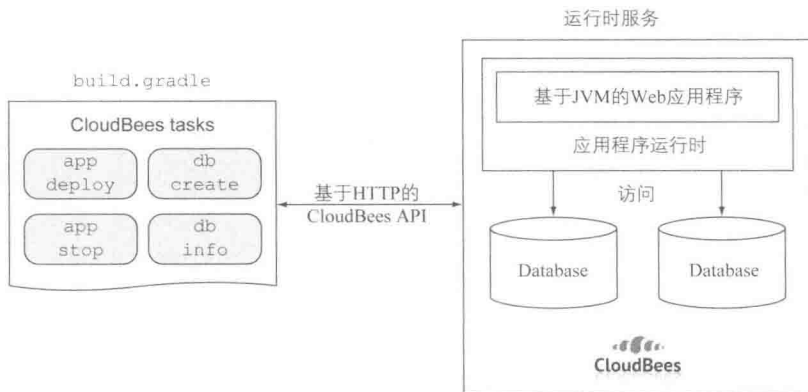


图 8.1 在 Gradle 构建脚本中通过 HTTP 管理 CloudBees 运行时服务

### 8.1.2 设置云环境

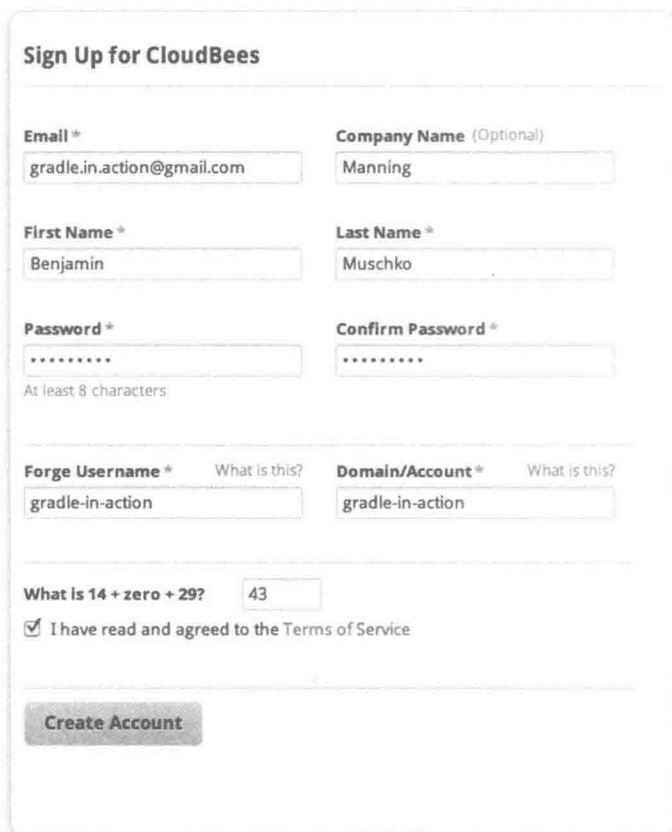
在通过 API 与 CloudBees 的 PaaS 交互之前，你需要准备一个账号。在本节中，我会带领你完成注册和应用程序安装过程。

#### 注册一个 CloudBees 账号

注册一个 CloudBees 账号很简单，不到 30 秒就能完成。打开浏览器，输入注册页面 URL：<https://www.cloudbees.com/signup>。图 8.2 展示了注册表格。需要你填写 E-mail 地址、姓名、密码，以及用户名和域名。请注意，你输入的域名/账户名将会被用来构建应用对应的服务 URL：[https://\[app-name\].\[account-name\].cloudbees.net](https://[app-name].[account-name].cloudbees.net)。

注册成功后，你会收到一封电子邮件来确认注册。现在你可以登录了。在 CloudBees 的页面上，你可以找到一些可用的服务。在这一章中，我们只集中在应用这些服务上。

在开始访问任何应用的运行时服务之前，你需要选择一个订阅计划。点击如图 8.3 所示的应用链接，点击订阅按钮，选择免费的 RUN@cloud 计划。在接下来的页面中，将对应的应用服务添加到你的账号中。



**Sign Up for CloudBees**

**Email \***  
gradle.in.action@gmail.com

**Company Name (Optional)**  
Manning

**First Name \***  
Benjamin

**Last Name \***  
Muschko

**Password \***  
\*\*\*\*\*

**Confirm Password \***  
\*\*\*\*\*

At least 8 characters

**Forge Username \*** What is this?  
gradle-in-action

**Domain/Account \*** What is this?  
gradle-in-action

What is 14 + zero + 29? 43

☒ I have read and agreed to the Terms of Service

**Create Account**

图 8.2 注册一个 CloudBees 账号

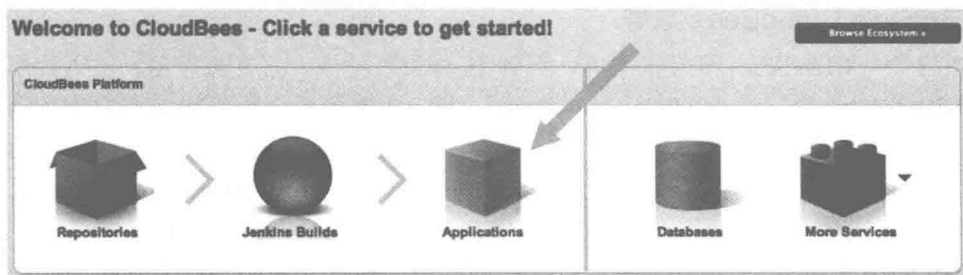


图 8.3 CloudBees 授权中心主页

## 提供应用

设置好应用服务之后，你就可以开始准备自己的应用程序了。要创建一个新的应用，选择页面顶部的应用菜单项或者页面中的应用链接。两个链接都会把你带

到应用管理页面。在部署任何应用之前，你需要提供该应用。点击创建应用按钮会打开一个对话框，它看上去和图 8.4 相似，它会让你输入应用名字和支持的运行环境。因为你想要部署一个 WAR 文件，所以需要从下拉列表中选择“JVM Web Application (WAR)”，输入 todo 作为应用的名字。点击完成按钮来初始化应用的创建。



图 8.4 在 RUN@cloud 上提供 To Do 应用程序

这样就可以了——开始使用这个应用吧。你只需要在浏览器中输入合适的 URL。因为我在注册时用的账户名是 `gradle-in-action`，所以我的应用 URL 是 `http://todo.gradle-in-action.cloudbees.net`。试一把吧！应用的 URL 已经被解析，即便你还没有部署 WAR 文件。

在管理页面中，你可以配置应用程序、部署新的版本、总览请求、监控内存使用和服务负载，以及查看日志文件，这些都可以在一个中央控制面板上访问。通过切换查看不同的标签，可以快速地熟悉管理功能。

虽然应用管理面板非常容易使用，但是你可能还是希望尽量避免手动操作。要达到这个目的，你需要使用 CloudBees 的 API，它能够让你完全自动化地与服务器后端通信。

### 设置 API 密钥

每个对 CloudBees API 的请求都需要调用者提供一个 API 密钥和一个私钥。这个 API 密钥对于每个账户都是唯一的，可以明确地确认 API 的调用者。私钥是为了安全地向 CloudBees 服务发送 HTTP 的 Web 请求。你可以在 `Account Settings > API Keys` 下查到这两个 key。因为这两个 key 的私有特性，保存它们的一个好办法是存在 `gradle.properties` 文件中。你应该避免将它们提交到版本控制中。如果公开这个文件，就等于自动将访问账号的权限赋予给所有可以访问你的源代码的用户。如果你还没有创建该文件，现在是时候创建了。下面的命令显示了如何在 \*nix 系统中创建该文件：

```
$ cd $HOME/.gradle
$ vi gradle.properties
```

在 `properties` 文件中，添加下面的 `key`，用你账号中的实际值替换下面的占位符：

```
cloudbeesApiKey = Your-CloudBees-API-key  
cloudbeesApiSecret = Your-CloudBees-API-secret
```

设置好 `CloudBees` 账号和提供应用是非常简单的。可能也就花费不到 5 分钟的时间就完成了整个过程。想象一下要在自己维护的服务器上设置类似的运行时环境要花多少精力。接下来，我们将讨论如何一步步构建这个插件。

## 8.2 从零起步构建插件

在 `Gradle` 中开发插件并不难。在应用前面章节中学习过的技术的同时，你还需要知道一些新的概念。`Gradle` 将插件分为两类：脚本插件和对象插件。一个脚本插件只不过是一个普通的 `Gradle` 构建脚本，它可以被导入到其他的构建脚本中。使用脚本插件，你可以做你学到的任何事情。对象插件需要实现 `org.gradle.api.Plugin` 接口。对象插件的源代码通常存放在 `buildSrc` 目录下，要么和项目在一起，要么是一个独立的项目，并且作为独立的 `JAR` 文件发布。在本章中，你会学习如何使用这两种方法来编写插件。

在敏捷开发的思想下，你需要以可用片段的方式迭代式地构建功能。目的是为了快速得到第一个版本并运行起来，以获得早期反馈。从一个高层次的角度看，构建 `CloudBees` 插件需要三个主要步骤，如图 8.5 所示。在下面的每个迭代中，我们要在开发过程中发现之前开发代码的缺点，并讨论如何提高它。

在第一步中，你需要熟悉 `CloudBees` 的 `API` 并体验其功能。你需要在一个脚本插件中写两个简单的 `task`：一个用来获取 `CloudBees` 账号中已创建应用的信息，另一个用来将 `WAR` 文件部署到云端。

在第二步中，你需要将在 `task action` 闭包中写的逻辑转移包装到定制的 `task` 类中。通过暴露属性，`task` 类的行为将会变得高度可配置和可重用。属性的值将由一个增强型 `task` 提供，它是一个 `task` 类的消费者。

在最后一步中，你会学习如何创建一个完备的对象插件。你会建立一个独立的 `Groovy` 工程来构建插件的 `JAR` 文件。

总的来说，你需要写一些 `task` 和 `CloudBees` 客户端 `SDK` 进行交互。

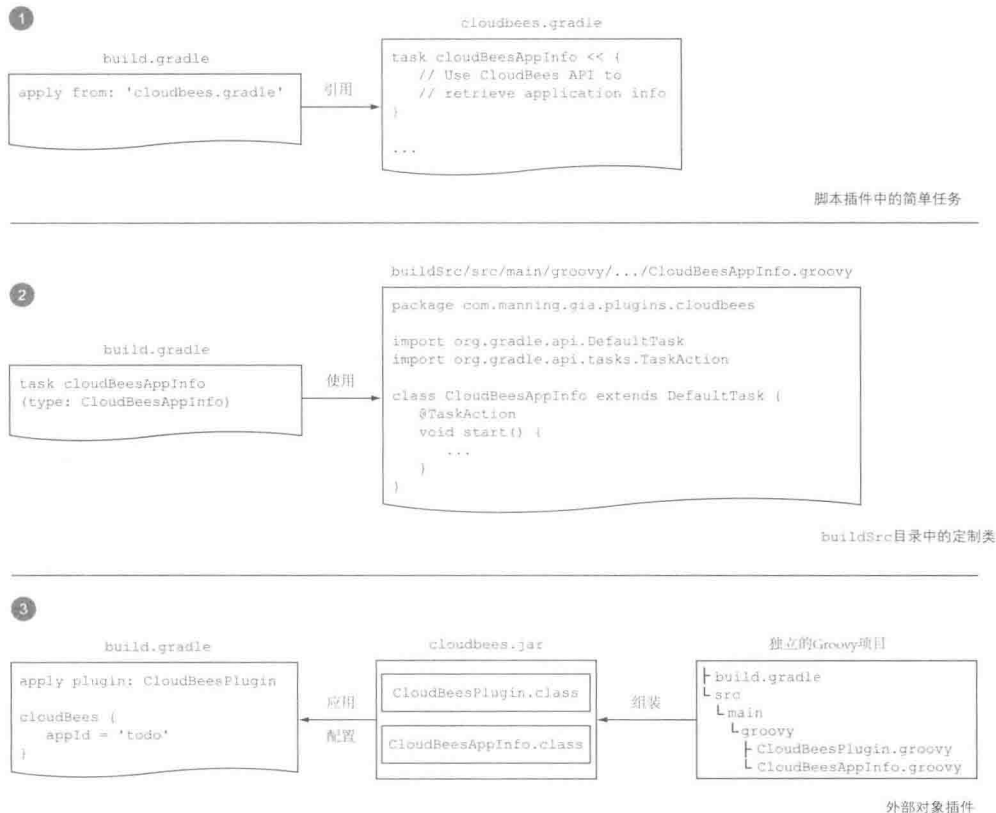


图 8.5 三步实现 CloudBees 插件

## 8.3 写一个脚本插件

一个脚本插件和一个普通的 build.gradle 文件没什么区别。你可以使用同样的 Gradle 构建语言。新建一个名字为 cloudbees.gradle 的脚本，里面会包含 CloudBees 的 task。因为构建脚本的名字不遵从默认的命名约定，所以你需要使用 -b 命令行选项来显式指定使用该脚本。执行 gradle -b cloudbees.gradle tasks 应该会显示默认的帮助任务。在你开始写第一个 task 之前，构建脚本需要知道 CloudBees 的 API 客户端类库在哪里。

### 8.3.1 添加 CloudBees 的 API 类库

要在构建脚本中直接使用一个外部类库，你需要在 classpath 下声明它。为此，

Gradle 的 API 类 `org.gradle.api.Project` 暴露了一个名为 `buildscript` 的方法。该方法期待一个参数，一个闭包，里面定义了你想要解析的依赖，该依赖定义在 `classpath` 配置分组中。CloudBees 的 API 客户端类库可以从 Maven Central 获得。你可以通过调用 `mavenCentral()` 方法指定仓库。下面的清单展示了如何将最新版本的类库添加到构建脚本的 `classpath` 中。

清单 8.1 将 CloudBees 的 API 类库添加到构建脚本的 `classpath` 中

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.cloudbees:cloudbees-api-client:1.4.0'
    }
}
```

使用 Maven Central 来解析已声明的依赖

将 CloudBees 客户端 API 类库添加到构建脚本的 `classpath` 中

当脚本第一次被执行时，CloudBees 类库会被下载并保存到本地依赖缓存中。你现在就可以在构建脚本中直接引入和使用任何 CloudBees 的 SDK 类。接下来，你就可以写第一个 task 与 CloudBees 账户进行交互了。

### 8.3.2 在 task 中使用 CloudBees 的 API

CloudBees API 的核心类是客户端的实现：`com.cloudbees.api.BeesClient`。每一个暴露的方法都允许你访问 RUN@cloud 平台上的一个指定服务。在实例化时，该类希望你可以提供账户信息、API 的 URL、格式以及版本。例如：

```
BeesClient client = new BeesClient('https://api.cloudbees.com/api',
    '24HE9X5DFF743671', '24QSXAHS1LAAVWDFAZS3TUFE6FZHK1DBYA=',
    'xml', '1.0')
```

#### 准备客户端属性配置

和你在不久的将来改变 API 参数不同。现在，你需要将它们定义为外部属性，就像下面的代码块那样：

```
ext {
    apiUrl = 'https://api.cloudbees.com/api'
    apiFormat = 'xml'
    apiVersion = '1.0'
}
```

你应该不会想要将 API 密钥和私钥分享出去或提交到版本控制中。在上一节中，你已经在本地 `gradle.properties` 文件中保存了这两个值。读取这两个值，然

后把它们存储在属性 `apiKey` 和 `secret` 中：

```
if (project.hasProperty('cloudbeesApiKey')) {
    ext.apiKey = project.property('cloudbeesApiKey')
}

if (project.hasProperty('cloudbeesApiSecret')) {
    ext.secret = project.property('cloudbeesApiSecret')
}
```

## 获取应用信息

当设置 CloudBees 账号时，你配置了一个名为 `todo` 的应用。CloudBees 的 API 能够在不登录到控制面板的情况下远程获取关于该应用的信息，代码如下：

**清单 8.2 编写任务列出 CloudBees 账号上可用的应用**

```
import com.cloudbees.api.ApplicationInfo
import com.cloudbees.api.BeesClient

task cloudBeesAppInfo(description: 'Returns the basic information about an
    ➡ application.', group: 'CloudBees') {

    inputs.property('apiKey', apiKey)
    inputs.property('secret', secret)
    inputs.property('appId', appId)

    doLast {
        BeesClient client = new BeesClient(apiUrl, apiKey, secret, apiFormat,
            ➡ apiVersion)

        ApplicationInfo info

        try {
            info = client.applicationInfo(appId)
        } catch (Exception e) {
            throw new GradleException(e.message)
        }

        logger.quiet "Application id : $info.id"
        logger.quiet "                title : $info.title"
        logger.quiet "                created : $info.created"
        logger.quiet "                urls : $info.urls"
        logger.quiet "                status : $info.status"
    }
}
```

给 task 声明输入属性；如果不提供任何属性，则 task 执行失败

CloudBees 的 SDK 客户端实现暴露对所有服务的访问

任何执行失败都当作异常（比如，客户端的授权异常），并作为特殊的 Gradle 异常捕获和重抛

通过指定 `appId` 属性获取关于应用的信息

使用 Gradle 日志来打印响应信息到控制台

在执行该任务之前，你需要创建另一个 Gradle 脚本文件：`build.gradle`。下面的代码片段展示了如何重用外部脚本：

```
apply from: 'cloudbees.gradle'
```

请注意 `apply` 方法调用时传入的 `from` 属性，它的值可以是任何类型的 URL，比如 HTTP 地址 `http://my.scripts.com/shared/cloudbees.gradle`。以基于 HTTP(S) 的方式暴露脚本插件是一个组织的部门间共享的较好办法。现在是时候让 `task` 跑起来了。下面的控制台输出显示了如何获取 ID 是 `gradle-in-action/todo` 的应用信息：

```
$ gradle -PappId=gradle-in-action/todo cloudBeesAppInfo
:cloudBeesAppInfo
...
Application id : gradle-in-action/todo
    title : todo

    created : Sun Sep 16 10:17:11 EDT 2012
        urls : [todo.gradle-in-action.cloudbees.net]
    status : hibernate
```

输出给出了应用的标题、何时被创建、什么 URL 可以访问它，以及当前的状态。在这个例子中，应用处于休眠状态。如果应用被闲置得太久，它们会被置为休眠状态，以为其他应用节省资源。在下次请求时，应用程序会被自动地重新激活。这可能会花上几秒钟的时间。通过成功获取应用信息，你知道它是存在于给定的 ID 账户下的。现在，你可以实际部署一个 WAR 文件，享受一下辛勤努力的成果。

### 部署一个 WAR 文件

清单 8.3 展示了如何编写一个 `task` 通过 CloudBees 的客户端 API 部署 WAR 文件，它和获取应用信息非常相似。唯一的区别是你需要提供其他的输入参数，比如 WAR 文件本身和一些可选信息。

#### 清单 8.3 编写任务来部署 WAR 文件

```
import com.cloudbees.api.ApplicationDeployArchiveResponse
import com.cloudbees.api.BeesClient

task cloudBeesDeployWar(description: 'Deploys a new version of an application
    ➤ using a WAR archive file.',
    ➤ group: 'CloudBees') {
    inputs.property('apiKey', apiKey)
    inputs.property('secret', secret)
    inputs.property('appId', appId)
    inputs.file file(warFile)
    ext.message = project.hasProperty('message') ? project.message : null
    inputs.property('message', message)
```

除了前面  
task 鉴  
别出来的  
task 输入  
属性，还  
要确保提  
供了 WAR  
文件



方法提供了可扩展的参数列表，其中大部分参数现在都不重要，所以将值设为 null

```
doLast {
    logger.quiet "Deploying WAR '$warFile' to application ID '$appId'
        with message '$message'"
    BeesClient client = new BeesClient(apiUrl, apiKey, secret, apiFormat,
        apiVersion)
    ApplicationDeployArchiveResponse response

    try {
        response = client.applicationDeployWar(appId, null, message,
            file(warFile), null, null)
    }
    catch(Exception e) {
        throw new GradleException("Error: $e.message")
    }

    logger.quiet "Application uploaded successfully to: '$response.url'"
}
```

我们开始吧——关键时刻到了。将 To Do 应用部署到云端：

```
$ gradle -PappId=gradle-in-action/todo -PwarFile=todo.war
-Pmessage=v0.1 cloudBeesDeployWar
:cloudBeesDeployWar
...
Deploying WAR 'todo.war' to application ID 'gradle-in-action/todo' with
message 'v0.1'
Application uploaded successfully to: 'http://todo.gradle-in-
action.cloudbees.net'
```

正如控制台输出显示的那样，部署是成功的。在 CloudBees 的应用控制面板上，你应该可以看到更新的部署版本，如图 8.6 所示。


Deployments		
Controls	Date	Message
	2012 September 21 06:59:06 UTC-4	v0.1
	2012 September 16 10:18:11 UTC-4	Initial create
<div>Upload new version</div>		

图 8.6 将 To Do 应用的 0.1 版本部署到 CloudBees

当然，你不会想要错过尝试一把你的应用的机会。在浏览器中打开 URL：<http://todo.gradle-in-action.cloudbees.net/>，你就可以看到该应用程序了。你已经看到写 task 和 CloudBees 的 API 交互是多么的简单。因为你把代码写在了一个可共享的脚本中，任何应用了该脚本的项目都可以这样管理。让我们更进一步，看看如何提高你的设计，将简单的 task 编写成定制的 task 类。

## 8.4 编写定制的task类

在最后一节中，你会看到如何创建一个可以和 PaaS 服务商交互的可共享脚本。通过使用一个脚本插件，你给项目提供了可以在云环境中管理和部署 Web 应用的 task。让我们一起来看看这个方法有什么优缺点。

优点：

- task 都是可复用的，并且可以导入到其他项目中。
- task 是可配置的。消费脚本仅仅需要知道必要的输入。
- 最新状态检查功能可以通过 task 的输入和输出属性使用。

缺点：

- task 的逻辑是通过 action 的闭包定义的，因此不能够放入类和包中。
- 添加的 task 越多，构建脚本就越长，越难维护。
- 不能进行单元测试和集成测试。

简单的 task 是开发一次性实现的最佳方法。即便你做到极致，为 task 提供了可配置属性，代码的可维护性和可测试性也依然差得很远。如果你想要更进一步，最好的办法是在一个定制的 task 中实现逻辑。行为和属性都定义在一个 task 类的实现中。当使用该定制的 task 时，可以通过给对应属性提供相应的值来定义 task 的行为。如果 task 代码增加了，定制的 task 可以帮助组织和封装构建逻辑。

### 8.4.1 定制 task 的实现选项

Gradle 提供了一个可以继承的默认实现：`org.gradle.api.DefaultTask`。实际上，许多 Gradle 标准插件的 task 都继承自 `DefaultTask`。

Gradle 有多种定义定制化 task 类的方式。最简单的方式就是把它和构建代码一起放在构建脚本里。当触发脚本中的一个 task 时，定制的 task 会自动编译并放到 classpath 中。

另一种方式就是将定制的 task 放到项目根目录下的 `buildSrc` 目录中。要确保遵从语言插件定义的源代码目录约定。例如，如果你用 Java 语言编写定制的 task，就要将它放到 `buildSrc/src/main/java` 目录下。Gradle 会将该目录当作默认的源代码目录，当运行构建时，自动尝试编译所有的源代码文件。记住，Gradle 的增量构建特性在这里也是支持的。位于 `buildSrc` 目录下的定制的 task 类被所有的项目构建脚本共享，并且在 classpath 中自动可用。

为了使定制的 task 可以在多个项目中共享，你可以将它们打成 JAR 文件，然后在构建脚本的 classpath 中定义。图 8.7 展示了定制任务的不同实现方式。

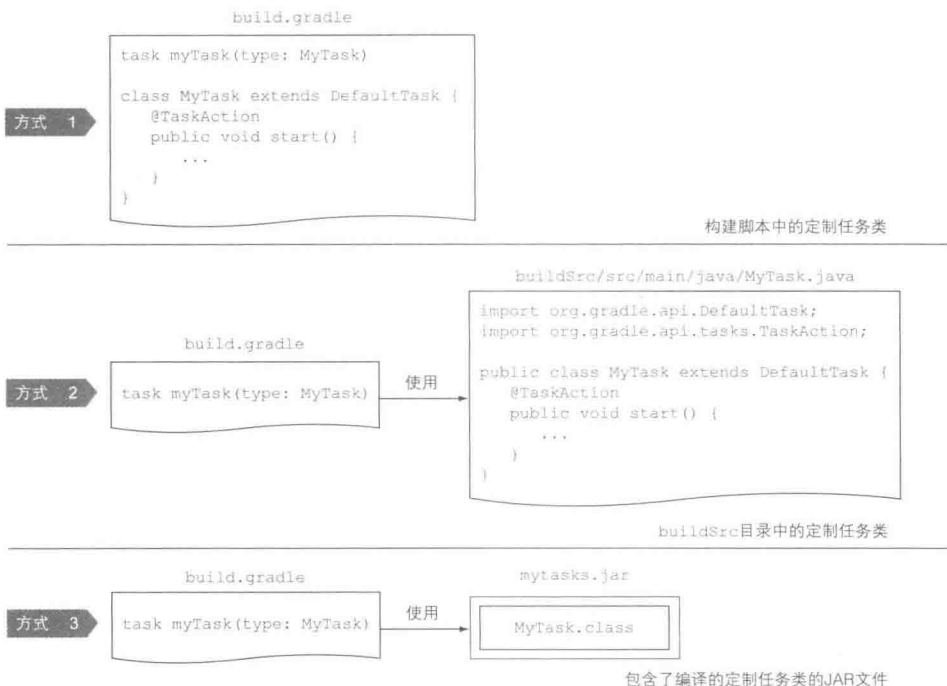
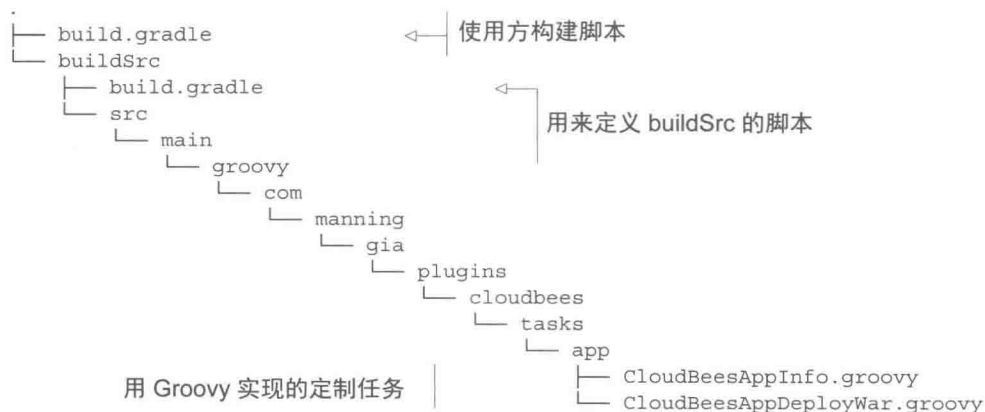


图 8.7 定制任务的实现方式

### 8.4.2 在 buildSrc 下定义定制任务

现在，来试试方式 2。你需要在 `buildSrc` 目录下创建定制 task 源文件，这是后期通过对象插件使用它们最好的方法。所有的定制 task 类都放在 `com.manning.gia.plugins.cloudbees.tasks.app` 包下。`buildSrc` 目录中的构建脚本里声明了 `CloudBees` 类库中的依赖。项目最终的目录结构如下：



如果你想要看看这些定制的任务类什么样，你可以重写从 CloudBees 获取应用信息的简单 task。你写出来的代码应该和下面的类似：

清单 8.4 获取应用信息的定制任务

```
package com.manning.gia.plugins.cloudbees.tasks.app
```

```
import com.cloudbees.api.ApplicationInfo
import com.cloudbees.api.BeesClient
import org.gradle.api.*
import org.gradle.api.tasks.*
```

```
class CloudBeesAppInfo extends DefaultTask {
    @Input String apiUrl
    @Input String apiKey
    @Input String secret
    @Input String apiFormat
    @Input String apiVersion
    @Input String appId
```

可配置的 task 属性

指定 task  
的 group  
和描述信  
息

```
CloudBeesAppInfo() {
    description = Returns the basic information about an application.'
    group = 'CloudBees'
}

@TaskAction
void start() {
    BeesClient client = new BeesClient(apiUrl, apiKey, secret,
                                     apiFormat, apiVersion)

    ApplicationInfo info

    try {
        info = client.applicationInfo(appId)
    }
}
```

由注解指定的  
执行方法

```

        catch(Exception e) {
            throw new GradleException(e.message)
        }

        logger.quiet "Application id : $info.id"
        logger.quiet "           title : $info.title"
        logger.quiet "           created : $info.created"
        logger.quiet "           urls : $info.urls"
        logger.quiet "           status : $info.status"
    }
}

```

task 的行为封装在 task 的 action 中。要指定哪个 action 执行，需要在 start() 方法上加上注解 @TaskAction。task 执行方法的名字可以任意取，只要不覆盖父类的 void execute() 方法就行。task 的 action 行为可以通过属性配置——比如，用 appId 作为应用标识符。

在定制 task 的实现中，你需要引入 CloudBees 类库中的类。为了确保类文件可以被正确编译，你需要给 buildSrc 工程创建一个构建脚本和声明 CloudBees 类库，如下面的清单所示。

#### 清单 8.5 buildSrc 目录中的构建脚本

```

repositories {
    mavenCentral()
}

dependencies {
    compile 'com.cloudbees:cloudbees-api-client:1.4.0'
}

```

### 使用定制的 task

定制的 task 类不能由自己直接执行。要使用和配置由定制 task 定义的行为，你需要创建一个增强型 task。该 task 定义了它要使用的 task 类型，在这个例子中是 CloudBeesAppInfo，如下面的清单所示。

#### 清单 8.6 使用定制任务

```

import com.manning.gia.plugins.cloudbees.tasks.app.CloudBeesAppInfo
task cloudBeesAppInfo(type: CloudBeesAppInfo) {
    apiUrl = project.apiUrl
    apiKey = project.apiKey
    secret = project.secret
    apiFormat = project.apiFormat
    apiVersion = project.apiVersion
    appId = project.hasProperty('appId') ? project.appId : null
}

```

导入的定制任务类  
 定制任务的使用

在 task 的闭包里面，指定了应用标识符。该应用标识符由命令传入并解析。其他 API 选项配置和认证信息在脚本的其他部分被定义为了外部属性。执行增强型 task 会先编译 buildSrc 工程中的所有定制类，然后从 CloudBees 中获取应用信息：

```
$ gradle -PappId=gradle-in-action/todo cloudBeesAppInfo
:buildSrc:compileJava UP-TO-DATE
:buildSrc:compileGroovy
:buildSrc:processResources UP-TO-DATE
:buildSrc:classes
:buildSrc:jar
:buildSrc:assemble
:buildSrc:compileTestJava UP-TO-DATE
:buildSrc:compileTestGroovy UP-TO-DATE
:buildSrc:processTestResources UP-TO-DATE
:buildSrc:testClasses UP-TO-DATE
:buildSrc:test
:buildSrc:check
:buildSrc:build
:cloudBeesAppInfo
...
```

编译在 buildSrc/src/main/groovy 中以 Groovy 编写的定制任务

增强型 task 会从 CloudBees 中获取应用信息

我们讨论了如何实现和使用一个独立的定制 task 与 CloudBees 后端交互。CloudBeesAppDeployWar 用来部署一个 WAR 文件，看起来应该比较熟悉。你可以在本书提供的样例代码中找到它。

### 通过重构提高可重用性

当比较两个 CloudBees 的定制任务时，你会发现它们的实现在结构上非常相似的。我们可以确定它们有下面这些共同点：

- 两个类都创建了 CloudBees 的 API 客户端实例 BeesClient。
- 它们都需要提供 CloudBees 的 API 选项和认证信息。
- 当 CloudBees 的 API 交互时，你需要捕获一个异常并适当处理该异常。
- 所有的 CloudBees 定制 task 都需要分配给 group CloudBees。

当与实际的类打交道时，其中一个好处就是，你有效地利用了面向对象编程的原则。通过创建一个父类，你可以大大地简化当前的 CloudBees 定制 task 的代码。下面的清单显示了将上面提到的所有共同部分放在父类中实现。

#### 清单 8.7 通过引入父类来简化与 CloudBees 的交互

```
package com.manning.gia.plugins.cloudbees.tasks

import com.cloudbees.api.BeesClient
import org.gradle.api.*
import org.gradle.api.tasks.*
```

```

abstract class CloudBeesTask extends DefaultTask {
    @Input String apiFormat = 'xml'
    @Input String apiVersion = '1.0'
    @Input String apiUrl = 'https://api.cloudbees.com/api'
    @Input String apiKey
    @Input String secret

    CloudBeesTask(String description) {
        this.description = description
        group = 'CloudBees'
    }

    @TaskAction
    void start() {
        withExceptionHandling {
            BeesClient client = new BeesClient(apiUrl, apiKey, secret,
                apiFormat, apiVersion)
            executeAction(client)
        }
    }

    private void withExceptionHandling(Closure c) {
        try {
            c()
        }
        catch (Exception e) {
            throw new GradleException(e.message)
        }
    }

    abstract void executeAction(BeesClient client)
}

```

为 API 认证信息暴露属性

指定默认的 task 的 group 名字

创建 CloudBees 的 API 客户端实例

捕获异常并处理

由子类实现的抽象方法

使用父类的 CloudBees 的 task 作为其中一个定制的 task。清单 8.8 说明了和 CloudBees 的 API 打交道是多么的简单。task 的基础架构已经给你搭建好了。API 客户端的创建或异常处理不再重复。你只需专注于实现业务逻辑就行了。

### 清单 8.8 简化定制任务

```

package com.manning.gia.plugins.cloudbees.tasks.app

import com.cloudbees.api.ApplicationInfo
import com.cloudbees.api.BeesClient
import org.gradle.api.tasks.Input
import com.manning.gia.plugins.cloudbees.tasks.CloudBeesTask

class CloudBeesAppInfo extends CloudBeesTask {
    @Input String appId

    CloudBeesAppInfo() {

```

扩展 CloudBees 的父 task

针对这个 task 的功能暴露指定的属性

```

    super('Returns the basic information about an application.')
}

@Override
void executeAction(BeesClient client) {
    ApplicationInfo info = client.applicationInfo(appId)
    logger.quiet "Application title : $info.title"
    logger.quiet "                created : $info.created"
    logger.quiet "                urls : $info.urls"
    logger.quiet "                status : $info.status"
}
}

```

提供任务描述

实现任务的 action ; 提供已经创建的 CloudBees 的 API 客户端实例

你已经执行过这个任务，也知道它可以工作。你加入到项目中的定制 task 越多，你就越不想每次改变代码后都手动重新运行来确认它们可以工作。接下来，你可以通过编写一些测试代码，让自己对之后的重构更有信心。

### 测试定制的 task

Gradle 的 API 提供的测试工具允许你在真实的工作环境中测试定制的 task 和插件。它的思想是提供一个假的 Gradle Project 实例，它暴露了和你在构建脚本中使用的 Project 对象一样的方法和属性。该 Project 实例是通过 `org.gradle.testfixtures.ProjectBuilder` 类的 `build()` 方法提供的，而且可以被任何测试类使用。

在 Spock 框架的帮助下，通过为定制任务 `CloudBeesAppInfo` 编写测试代码，你可以实际看到 `ProjectBuilder` 是怎么使用的，如清单 8.9 所示。你首先需要在 `buildSrc/src/test/groovy` 目录下创建 `CloudBeesAppInfoSpec.groovy` 类。正如你在清单中所看到的，你使用的是和测试类相同的包。无论何时运行构建，该类都会自动被编译，并且测试用例被执行。

#### 清单 8.9 通过 ProjectBuilder 测试定制任务 CloudBeesAppInfo

```

package com.manning.gia.plugins.cloudbees.tasks.app

import spock.lang.Specification
import org.gradle.api.*
import org.gradle.api.plugins.*
import org.gradle.testfixtures.ProjectBuilder

class CloudBeesAppInfoSpec extends Specification {
    static final TASK_NAME = 'cloudBeesAppInfo'
    Project project

    def setup() {
        project = ProjectBuilder.builder().build()
    }
}

```

创建一个假的 Project 实例



创建一个  
CloudBees  
AppInfo 类  
型的增强型  
task 并给  
task 属性赋  
值

```
def "Adds app info task"() {
    expect:
        project.tasks.findByName(TASK_NAME) == null
    when:
        project.task(TASK_NAME, type: CloudBeesAppInfo) {
            appId = 'gradle-in-action/todo'
            apiKey = 'myKey'
            secret = 'mySecret'
        }
    then:
        Task task = project.tasks.findByName(TASK_NAME)
        task != null
        task.description == 'Returns the basic information about an
            application.'
        task.group == 'CloudBees'
        task.apiFormat == 'xml'
        task.apiVersion == '1.0'
        task.apiUrl == 'https://api.cloudbees.com/api'
        task.appId == 'gradle-in-action/todo'
        task.apiKey == 'myKey'
        task.secret == 'mySecret'
}

def "Executes app info task with wrong credentials"() {
    expect:
        project.tasks.findByName(TASK_NAME) == null
    when:
        Task task = project.task(TASK_NAME, type: CloudBeesAppInfo) {
            appId = 'gradle-in-action/todo'
            apiKey = 'myKey'
            secret = 'mySecret'
        }

        task.start()
    then:
        project.tasks.findByName(TASK_NAME) != null
        thrown(GradleException)
}
...
}
```

验证 task  
已经添加到  
Project 中

ProjectBuilder 为使用测试驱动方式开发构建代码开启了一道门，虽然在功能上有些限制。Project 实例是由 ProjectBuilder 产生的，和真实世界的行为并不是 100% 一样。比如，对输入 / 输出注解的最新代码检查或从主目录真实载入 Gradle 属性，都没有实现。在大多数情况下，通过在测试类中编写额外的代码，可以绕过这些缺点。和复杂工具深入集成测试是 Gradle 路线图上的项，在未来的 Gradle 版本中会出现。在下一节中，我们将讨论如何将现有的代码变成对象插件并且在不同的项目中使用。

## 8.5 使用和构建对象插件

通过定制的 task 来实现逻辑是一种可维护、可测试的解决方案。通过打包成 JAR 文件, task 可以在独立的项目间被完全重用。然而, 该方法仍存在一些限制。让我们来看看打包的定制 task 实现有哪些优缺点。

优点:

- 定制逻辑在类中是自包含的, 并可以通过增强型 task 配置。
- 通过将 task 属性用注解标志可以支持声明式增量构建。
- 定制的 task 可以测试。

缺点:

- 定制的 task 仅仅暴露独立的工作单元。所提供的额外的公式化代码、约定和生命周期的整合并不是很直接。
- 定制的 task 仅仅能通过增强型 task 来配置。通过自定义的 DSL, 缺乏有表达性的扩展机制。
- 其他插件的功能不容易使用或扩展。

对象插件给你最大的灵活性去封装高复杂度的逻辑, 并且提供一种强大的扩展机制可以在构建脚本中定制它的行为。和定制 task 类一样, 你可以完全访问 Gradle 的公共 API 和工程模型。Gradle 自带了开箱即用的插件, 称作标准插件, 但是也可以通过第三方插件扩展。许多插件都是自包含的。这意味着它们要么依赖于 Gradle 的核心 API, 要么通过包装代码提供功能。更复杂的插件也许依赖于其他类库、工具或插件提供的特性。图 8.8 展示了插件在 Gradle 架构中的位置。

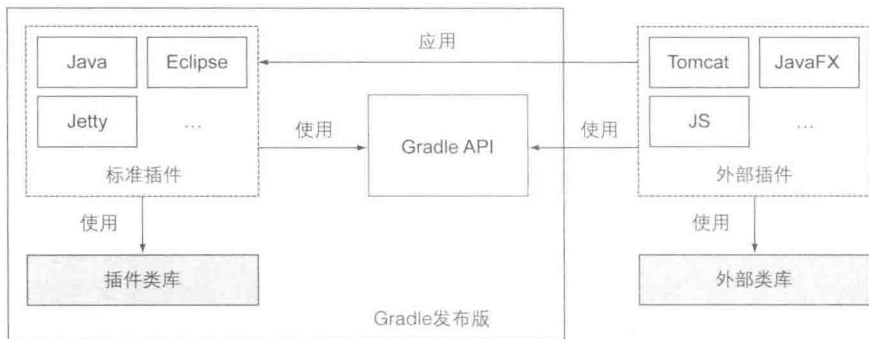


图 8.8 插件架构

在前面的章节中, 使用了不同的标准插件, 包括支持编程语言和与软件开发工具的平滑集成。回想一下第 3 章, 还记得如何使用 Java 插件来扩展项目功能吗? 如图 8.9 所示, 插件可以提供一个 task 集合, 并且整合到执行生命周期中, 引入新的

项目布局并提供有意义的默认值，添加属性来定制化它的行为，给依赖管理暴露对应的配置。

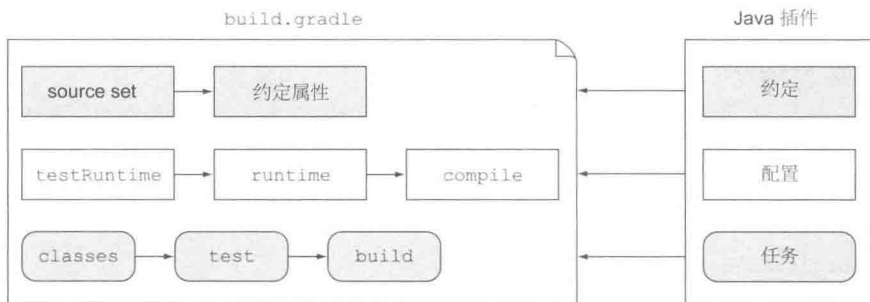


图 8.9 Java 插件特性

只需要额外的一行代码，你的项目就可以编译源代码、运行单元测试、生成报告，并且将项目打包成 JAR 文件。所有的这些功能都只需要你做一个最小的配置。

标准插件提供了非常棒的通用功能。真实世界的项目几乎不会被这些现有的功能所限制。由 Gradle 社区提供的或者由企业开发和共享的第三方插件，可以被用来给构建脚本增强非标准功能。你可能习惯于使用复杂的插件搜索系统，来搜索现有的插件，查看插件文档，甚至给它们评分。在写这本书的时候，Gradle 还没有为社区插件提供一个中心化的仓库。你可能会问，那现在怎么样？Gradle 提供了一个持续维护的 wiki 页面：<http://wiki.gradle.org/display/GRADLE/Plugins>，提供了所有可用的社区插件列表。是不是感觉很笨重？Gradleware 认识到插件仓库是共享和发布插件的重要先决条件，已经把它放在了 Gradle 发展的路线图上。更多关于 Gradle 未来发展路线图的内容，请参考 <http://www.gradle.org/roadmap>。

在本节中，我们重新了解了如何在构建脚本中使用标准插件和第三方插件。接下来，我们会学习插件的内部，来深入理解构建块和机制。最后，你会应用所学到的知识来编写自己的带有附加功能的对象插件。

### 8.5.1 使用对象插件

让我们重温一下如何在项目中使用对象插件。你已经看到可以通过 `apply` 方法来配置项目使用标准插件。我隐式使用方法这个词，是想强调你在使用 Gradle `project` 对象提供的 `apply` 方法，它是 `org.gradle.api.Project` 类的实例。该方法定义了一个参数 `options`，类型是 `java.util.Map`。

#### 通过名字使用插件

插件的标识符，一个很短的名字，通过插件的元信息提供，要在项目中使用

Java 插件，传入键值对 `plugin: 'java'`，如下：

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

### 通过类型使用插件

你也可以使用插件实现类的名字。如果插件没有暴露名字，或者两个不同插件命名冲突，那么这种方法是非常有用的，通过类型来使用插件很明确，但也显得比较笨重：

```
apply plugin: org.gradle.api.plugins.JavaPlugin
```

使用标准插件的便利是它们属于 Gradle 运行时的一部分。在大多数情况下，用户不需要知道插件所依赖的类库或者版本。Gradle 发布方会确保所有的标准插件都是兼容的。如果你比较好奇在哪里去找这些类库，请查看 Gradle 安装路径下的 `lib/plugins` 目录。

### 使用外部插件

构建脚本并不知道外部插件的存在，直到你将它放入到 `classpath` 下。你可以通过使用 `buildscript` 方法来做这件事，它定义了插件的位置、仓库和插件依赖。`buildscript` 和 `apply` 方法的定义顺序是不相关的。在配置阶段，Gradle 会构建项目模型，连接插件和构建逻辑。外部插件和 Gradle 中的其他依赖相等对待。一旦下载完成，就会被放置到本地依赖缓存中，以便后续的运行可以使用。下面的清单显示了如何使用外部插件 `tomcat` 将 Web 应用部署到嵌入式的 Tomcat 容器中。

#### 清单 8.10 使用 Maven Central 中的 `tomcat` 插件

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'org.gradle.api.plugins:gradle-tomcat-plugin:0.9.7'
    }
}

apply plugin: 'tomcat'
```

使用外部插件是出奇的简单。构建脚本仅仅需要定义插件依赖和它的原始仓库。在下一小节中，我们会仔细分析插件的内部，以进一步理解它的结构。

## 8.5.2 解析对象插件

图 8.10 从高层次的角度展示了实现一个对象插件时的几种选择。

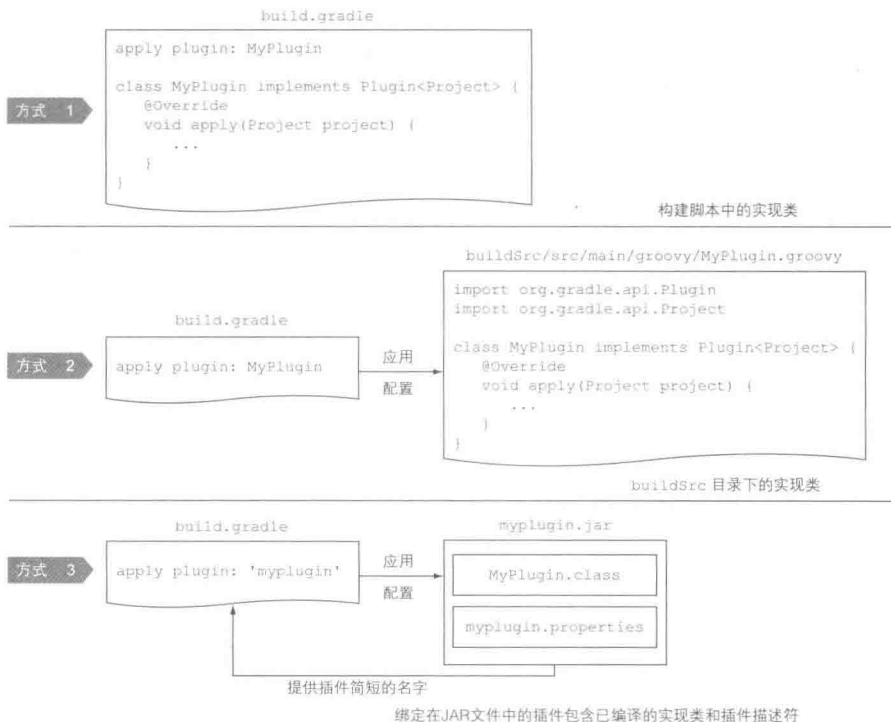


图 8.10 对象插件的实现方式

对于实现一个对象插件，有 4 个基本元素是非常重要的：

- 在放置插件实现的位置方面 Gradle 给你完全的灵活性。代码可以放在构建脚本中或者 `buildSrc` 目录下，也可以作为一个独立的工程被开发并且以 JAR 文件方式发布。
- 每个插件都需要提供一个实现类，它代表着插件的入口点。插件可以用任何 JVM 语言编写并编译成字节码。我倾向于 Groovy，因为能有效利用动态语言的特性和简洁性。然而，你也可以用 Java 或者 Scala 语言去实现构建逻辑。
- 应用到项目中的插件可以通过暴露出来的扩展对象进行定制。如果用户想要在构建脚本中覆盖插件的默认配置时，这一点特别有用。
- 插件描述符是一个属性文件，它包含了关于插件的元信息。通常，它包含有插件的简短名字和插件实现类的映射。

理论聊够了——是时候开始实现一个对象插件了。你要创建的插件将会使用到之前实现的定制 task。

### 8.5.3 编写对象插件

编写一个插件最低的要求是提供 `org.gradle.api.Plugin<Project>` 接口的一个实现类。该接口仅仅定义了一个简单的方法：`apply(Project)`。

首先，你需要在 `buildSrc` 工程的 `com.manning.gia.plugins.cloudbees` 包下创建一个插件的实现类。这样做有几个好处。在早期的开发插件阶段，你会希望得到一个快速的反馈环。因为你不需要打包插件代码，你只需要专心地使用 Gradle 的 API 实现业务逻辑就行。为了表明插件的意图，将类命名为 `CloudBeesPlugin`，如下面的清单所示。

清单 8.11 实现插件接口

```
package com.manning.gia.plugins.cloudbees

import org.gradle.api.Plugin
import org.gradle.api.Project
import org.gradle.api.plugins.WarPlugin
import com.manning.gia.plugins.cloudbees.tasks.*

class CloudBeesPlugin implements Plugin<Project> {
    @Override
    void apply(Project project) {
        project.plugins.apply(WarPlugin)
        addTasks(project)
    }

    private void addTasks(Project project) {
        project.tasks.withType(CloudBeesTask) {
            apiUrl = 'https://api.cloudbees.com/api'
            apiKey = project.property('cloudbeesApiKey')
            secret = project.property('cloudbeesApiSecret')
        }

        addAppTasks(project)
    }

    private void addAppTasks(Project project) {
        project.task('cloudBeesAppDeployWar', type: CloudBeesAppDeployWar) {
            appId = project.hasProperty('appId') ? project.appId : null
            message = project.hasProperty('message') ? project.message : null
            warFile = project.hasProperty('warFile') ?
                ➤ new File(project.getProperty('warFile')) :
                ➤ project.tasks.getByName(WarPlugin.WAR_TASK_NAME)
                ➤ .archivePath
        }
        ...
    }
}
```

应用 WAR 插件

对于所有的 CloudBees 任务，我们自动给 API URL、key 和 secret 赋值

如果没有提供 warFile 属性，则将 War 插件产生的路径赋值给 WAR 文件

正如清单所示，集成定制的 task 并给它们预配置默认值。不止如此，插件会对关于使用该插件的项目本质做出某种假设。例如，自动应用 War 插件，将产生的工件作为增强型 task `cloudBeesAppDeployWar` 的输入。

### 插件能力 vs 约定

作为一个插件的开发者，你常常会徘徊于插件提供的能力和约定两条线上。一方面，你可能想要加强另一个项目的功能；比如，通过 task。另一方面，你想要引入约定帮助用户做出有意义的决定；比如，标准化的项目布局。如果约定强制和武断地对使用该插件的项目做出了结构上的要求，那么通过创建两个不同的插件，将基本功能和约定分离就是一种合理的做法：一个基础插件包含了所提供的能力，而另一个插件则应用该基础插件并根据约定预配置这些能力。这种方法被 Java 插件所使用，Java 插件继承自 Java 基础插件。关于它们特性的更多信息，请参考附录 B 或者在线文档。

在项目中使用该插件，你要做的是在 `build.gradle` 中使用插件的实现类型，代码片段如下：

```
apply plugin: com.manning.gia.plugins.cloudbees.CloudBeesPlugin
```

为了验证 task 已经创建，运行 `gradle tasks` 命令。你应该能够看到 `cloudBeesAppDeployWar` 显示在列表中。目前，你通过命令行方式给定制 task 提供输入的。你也可以修改插件，让它从构建脚本中获取这些配置。

## 8.5.4 插件扩展机制

通过解析命令行参数来给 task 提供输入并不总是可取的。你可以通过暴露一个带有唯一命名空间的 DSL 来建立自己的构建语言。看下面的清单，它展示了一个名为 `cloudBees` 的闭包，允许从构建脚本中给 task 所需要的属性设值。

### 清单 8.12 提供一个插件的 DSL 来捕获用户输入

```
cloudBees {  
    apiUrl = 'https://api.cloudbees.com/api'  
    apiKey = project.apiKey  
    secret = project.secret  
    appId = 'gradle-in-action/todo'  
}
```

Gradle 将语言结构模型化作为扩展。扩展可以被添加到许多 Gradle 对象中，比如 `Project` 或者 `Task`，只要它们是扩展可知的。

如果一个对象实现了 `org.gradle.api.plugins.ExtensionAware` 接口，就认为它是扩展可知的。每个扩展都是一种数据模型，它是扩展的基础。这个模型可以是一个 POJO 或者 Groovy Bean。下面的清单展示了 CloudBees 插件的扩展模型。

#### 清单 8.13 插件扩展 POGO

```
package com.manning.gia.plugins.cloudbees

class CloudBeesPluginExtension {
    String apiUrl
    String apiKey
    String secret
    String appId
}
```

如清单 8.12 所示，你需要扩展应用了 CloudBees 插件的构建脚本背后的 Project。扩展可知对象暴露了方法 `extensions()`，它会返回一个容器对象，通过一个名字来注册扩展模型。该容器的实现接口是 `org.gradle.api.plugins.ExtensionContainer`。新的扩展通过方法 `create` 来注册。该方法接受一个名字和模型类型作为参数。一旦扩展被注册，你就可以获取模型的值并将它们赋值给定制 task 的属性。

#### 扩展对象 vs 额外属性

被用来扩展一个对象的 DSL 的扩展是扩展可知的。一个已注册的扩展模型会暴露一些属性和方法，用来给构建脚本建立新的构建语言结构。扩展模型的典型用例是插件。额外属性，另一方面，是一些通过 `ext` 命名空间创建的简单变量。它们一般提供给用户空间也就是构建脚本使用。请尽量避免在插件实现中使用额外属性。

使用扩展值来给定制 task 提供输入是有些取巧的。记住，定制 task 属性可以在构建生命周期配置阶段设置。在运行时阶段，是不可以赋值的。你可以通过使用约定映射概念来解决赋值顺序问题。下面的清单说明了如何在插件实现类中注册和使用 `CloudBeesPluginExtension` 类型的扩展。

#### 清单 8.14 注册和使用扩展

```
class CloudBeesPlugin implements Plugin<Project> {
    static final String EXTENSION_NAME = 'cloudBees'

    @Override
    void apply(Project project) {
```



```

用名字 cloudBees 注册扩展容器
    project.plugins.apply(WarPlugin)
    project.extensions.create(EXTENSION_NAME, CloudBeesPluginExtension)
    addTasks(project)
}

private void addTasks(Project project) {
    project.tasks.withType(CloudBeesTask) {
        def extension = project.extensions.findByName(EXTENSION_NAME)
        conventionMapping.apiUrl = { extension.apiUrl }
        conventionMapping.apiKey = { extension.apiKey }
        conventionMapping.secret = { extension.secret }
    }
    addAppTasks(project)
}

private void addAppTasks(Project project) {
    project.task('cloudBeesAppInfo', type: CloudBeesAppInfo) {
        conventionMapping.appId = { getAppId(project) }
    }
    ...
}
}

```

在 project 传入后, 给 project 添加 task, 确保扩展值已设置

查找扩展容器查询已配置属性

将包装在闭包中的扩展属性值赋给 task 的约定映射

插件中的每个 task 都有一个名字是 conventionMapping 的属性。更准确地说, 每个从 DefaultTask 继承而来的 task 都拥有这个属性。你使用这个属性将扩展模型的值赋给 task 的输入或者输出字段。通过将扩展模型值包装成一个闭包, 实现惰性赋值。这意味着这个值只有当 task 执行时才会被计算。为了获取存储在约定映射中的一个属性值, 你需要显示使用 getter 方法, 如下面的清单所示。记住, 如果尝试直接获取一个值, 只会返回一个 null 值。

清单 8.15 通过约定映射使用属性集

```

class CloudBeesAppInfo extends CloudBeesTask {
    @Input String appId

    CloudBeesAppInfo() {
        super('Returns the basic information about an application.')
    }

    @Override
    void executeAction(BeesClient client) {
        ApplicationInfo info = client.applicationInfo(getAppId())
        logger.quiet "Application title : $info.title"
        logger.quiet "          created : $info.created"
        logger.quiet "          urls : $info.urls"
        logger.quiet "          status : $info.status"
    }
}

```

通过约定映射设置的属性需要显式使用 getter 方法

约定映射是一个强大的概念，许多 Gradle 核心插件都使用它来确保扩展属性可以在运行时赋值。虽然 `conventionMapping` 属性并不是 Task 的公共 API 的一部分，但是结合扩展给 task 设置输入/输出属性是最好的办法。

### 设置配置属性的其他办法

处理这种情况还有别的办法，但它们都有优缺点。通常，它们极度地依赖于具体的情况以及用来实现插件的语言。这些办法有惰性 GStrings, `Project#afterEvaluate` 等。这些话题常常在 Gradle 的在线论坛上被讨论。

接下来，你需要给插件赋予一个更具描述性的名字。

## 8.5.5 给插件一个有意义的名字

在默认情况下，插件的名字从实现了 `org.gradle.api.Plugin` 接口的全限定类名继承而来。虽然命名空间不太容易和其他插件产生命名冲突，但是要取一个更短、更具表达性的插件名字还是很难。

对于对象插件，你可以在 `META-INF/gradle-plugins` 目录下的一个属性文件中配置名字。该属性文件的名字自动决定了插件的名字。比如，文件 `META-INF/gradle-plugins/cloudbees.properties` 暴露了插件名字是 `cloudbees`。在这个文件中，将该类的全限定类名赋值给键 `implementation-class`，如下面的清单所示。

### 清单 8.16 给插件添加一个短名字

```
implementation-class=com.manning.gia.plugins.cloudbees.CloudBeesPlugin
```

下面的清单展示了如何在构建脚本中使用短名字的插件。

### 清单 8.17 使用插件的短名字

```
apply plugin: 'cloudbees'
```

从现在开始，当你想要使用 `CloudBees` 插件时只在构建脚本中使用这个短名字。一个插件也可以在实现类层面进行测试。接下来，我们通过给 `CloudBeesPlugin.groovy` 写一个 `Spock` 测试类来实现 100% 测试覆盖率。

## 8.5.6 测试对象插件

测试插件代码就和测试定制 task 一样简单。Project 实例由

ProjectBuilder 产生，提供了测试插件功能的最好配置。在下面的清单中，展示了如何使用该插件，设置扩展值，并测试已创建 task 的正确行为。

**清单 8.18 为插件实现类编写测试代码**

```
package com.manning.gia.plugins.cloudbees

import spock.lang.Specification
import org.gradle.api.*
import org.gradle.api.plugins.*
import org.gradle.testfixtures.ProjectBuilder

class CloudBeesPluginSpec extends Specification {
    static final APP_INFO_TASK_NAME = 'cloudBeesAppInfo'
    static final APP_DEPLOY_WAR_TASK_NAME = 'cloudBeesAppDeployWar'
    Project project

    def setup() {
        project = ProjectBuilder.builder().build()
    }

    def "Applies plugin and sets extension values"() {
        expect:
        project.tasks.findByName(APP_INFO_TASK_NAME) == null
        project.tasks.findByName(APP_DEPLOY_WAR_TASK_NAME) == null
        when:
        project.apply plugin: 'cloudbees'
        project.cloudbees {
            apiKey = 'myKey'
            secret = 'mySecret'
            appId = 'todo'
        }
        then:
        project.plugins.hasPlugin(WarPlugin)
        project.extensions.findByName(CloudBeesPlugin.EXTENSION_NAME) != null
        Task appInfoTask = project.tasks.findByName(APP_INFO_TASK_NAME)
        appInfoTask != null
        appInfoTask.description == 'Returns the basic information about an
        application.'
        appInfoTask.group == 'CloudBees'
        appInfoTask.apiKey == 'myKey'
        appInfoTask.secret == 'mySecret'
        appInfoTask.appId == 'todo'
        ...
    }
    ...
}
```

通过短名字使用插件

设置扩展中对应的值

检查 WAR 插件已经自动应用到项目中

下一步，你需要为插件创建一个独立的项目，以便你可以构建一个 JAR 发布文件，并在不同的项目中使用。

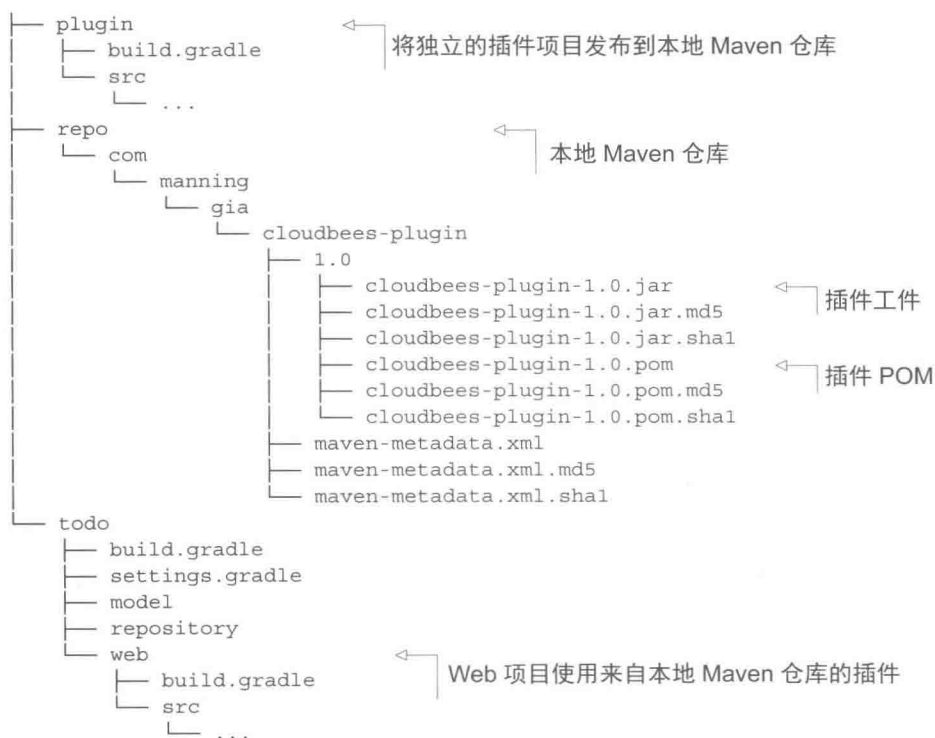
### 8.5.7 开发和使用独立的对象插件

如果代码会在主构建的构建脚本中使用，那么在 `buildSrc` 项目中实现一个插件是非常方便的；例如，多项目构建场景。如果你想要在多个构建中共享插件，那么需要将插件作为独立项目开发，然后将工件发布到仓库中。

#### 项目和仓库配置

在本小节中，你需要将现有的插件代码移动到一个独立的项目中。每一次你想要发布一个新的插件版本时，所产生的 JAR 文件就会被发布到一个名字为 `repo` 的本地 Maven 仓库中。这个仓库会和插件工程放在相同的目录层级。To Do Web 应用会作为插件的消费者。它的构建脚本会定义本地仓库，声明插件作为依赖，并使用插件中的 `task` 与 CloudBees 的后端服务进行交互。

下面的目录树展示了最后的配置效果：



首先，你需要在 `plugin` 目录下给插件创建一个新的项目。然后，将现有的结构从 `buildSrc` 目录拷贝到新的项目中。`todo` 项目是从你的 To Do 应用多项目构建中——对应拷贝过来的。你不需要为本地仓库创建目录——它会在发布时自动生成。

## 构建插件项目

为插件编写构建代码很直接。项目不需要再访问 buildSrc 基础设施，所以你需要的是定义对 Groovy 和 Gradle API 类库的依赖。通过 Maven 插件，可以非常容易地实现为插件生成 POM 文件和将工件发布到 Maven 仓库。你需要配置 Maven 部署器将 POM 文件和工件上传到本地目录。为了清晰地识别工件，需要指定插件的 group、name 和 version。下面的清单展示了完整的插件构建脚本。

清单 8.19 独立插件项目的构建脚本

```

apply plugin: 'groovy'
apply plugin: 'maven'

archivesBaseName = 'cloudbees-plugin'
group = 'com.manning.gia'
version = '1.0'

repositories {
    mavenCentral()
}

dependencies {
    compile localGroovy()
    compile gradleApi()
    compile 'com.cloudbees:cloudbees-api-client:1.4.0'
    testCompile 'org.spockframework:spock-core:0.6-groovy-1.8'
}

uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: "file://$projectDir/../repo")
        }
    }
}

```

应用 Maven 插件来发布插件工件

定义工件的 group、name 和 version

定义对 Gradle API 类的依赖

配置 Maven 部署器  
播件工件上传到本地目录

在插件可以被 To Do Web 应用使用之前，你需要在一个名为 uploadArchives 的 Maven 插件的 task 帮助下上传它。执行该 task 应该产生类似于下面的输出：

```

$ gradle uploadArchives
:compileJava UP-TO-DATE
:compileGroovy
:processResources
:classes
:jar
:uploadArchives
Uploading: com/manning/gia/cloudbees-plugin/1.0/cloudbees-plugin-
➡ 1.0.jar to repository remote at file:///Users/ben/gradle-in-
➡ action/code/plugin/../repo
Transferring 32K from remote
Uploaded 32K

```

在发布了工件之后，你会发现一个名字是 `repo` 的新目录。它包含了插件的 JAR 文件和 POM 文件。在第 14 章中，我们会讨论 Maven 插件的更多内容，以及如何将工件发布到公共仓库。

使用 Maven 仓库是准备一个对象插件最方便的方式。工件会自动找到它定义在 POM 文件中的依赖。另一种方式是，使用由文件直接定义的依赖。如果这么做，你需要自己处理传递依赖。

### 在项目中使用插件

是时候在你的 Web 项目中使用插件了。下面的清单说明了让构建脚本依赖于本地 Maven 仓库中可用的插件是多么的简单。

清单 8.20 在 Web 项目中使用对象插件

```
buildscript {
    repositories {
        maven { url "file://$projectDir/../../repo" }
        mavenCentral()
    }

    dependencies {
        classpath 'com.manning.gia:cloudbees-plugin:1.0'
    }
}

project(':web') {
    apply plugin: 'war'
    apply plugin: 'jetty'
    apply plugin: 'cloudbees'

    ...

    cloudBees {
        apiUrl = 'https://api.cloudbees.com/api'
        apiKey = project.apiKey
        secret = project.secret
        appId = 'gradle-in-action/todo'
    }
}
```

指定本地 Maven 仓库中包含该插件

指定 Maven Central 获取插件依赖 (即, CloudBees 的 API 客户端类库)

将插件添加到构建脚本的 classpath 中

使用插件

就是这样——你已经知道所有重要的开发实践了，为写好自己的 Gradle 插件做好最佳准备。

## 8.6 总结

Gradle 提供了一个丰富的插件生态系统来重用那些开箱即用的标准插件和由社区提供的第三方插件。有两种类型的插件：脚本插件和对象插件。

脚本插件是一种普通的 Gradle 构建脚本，它可以完全访问 Gradle 的 API。编写一个脚本插件是非常简单的，降低了分享代码的难度，可以通过 URL 被另一个项目使用。

对象插件通常包含更为复杂的逻辑，需要适当的包和类。每个对象插件的入口点都是 Plugin 接口，它提供了一种直接的方式来访问 Gradle 的 Project 模型。通过将对象插件添加到构建脚本的 classpath 下，许多对象插件都可以在多个独立的项目中使用，可以被打包成 JAR 文件，发布到仓库中。

在本章中，你创建了一个 Gradle 插件，它通过 API 类库与 CloudBees 的后端系统交互。我们讨论了两个非常有用的功能：将 WAR 文件部署到 CloudBees Web 容器和获取关于该应用的运行时信息。你一步步实现了插件的功能。你在脚本插件中编写简单的 task，将这些 task 转化成定制的 task，并放置到 buildSrc 项目中，之后又将它变成一个成熟的对象插件。

一个插件可以为配置功能暴露它自己的 DSL。扩展是强有力的 API 元素，在插件中引入了约定优于配置的思想。你体验了一个典型的场景，注册一个扩展，它作为模型来捕获用户的输入，覆盖默认的配置值。为插件编写测试代码和为应用编写一样重要。Gradle 的 ProjectBuilder 允许创建一个假的 Project 代表对象，它可以用于测试定制组件。这种工具可以排除编写测试的障碍，也鼓励开发人员完成更高的测试覆盖率。

下一章的内容对于使用 Ant 或者 Maven 并希望将它迁移到 Gradle 的用户特别有用。我们会了解如何在项目中升级 Gradle 版本，通过比较迁移之前和之后的输出，验证迁移成功与否。





# 集成与迁移

## 本章涵盖

- 导入已有 Ant 项目
- 在 Gradle 中使用 Ant 的 task
- 将 Maven 的 pom.xml 转换为 Gradle 项目
- 从 Ant 和 Maven 到 Gradle 的迁移策略
- 比较构建输出和升级测试

长期开发项目通常都会在建立构建基础设施和逻辑方面花费较多的投入。作为首选的构建工具之一，Gradle 知道迁移到一个新的系统需要策略规划、知识的传递、学习新的东西，同时还要确保不阻碍当前的构建和交付过程。Gradle 提供了强大的工具来集成现有的构建逻辑和减轻迁移所需要的投入。

如果你是一个 Java 开发者，你可能至少有一些使用另一种构建工具的经验。许多人都曾使用过 Ant 和 Maven，要么是自己选择的，要么就是工作在一个遗留系统中。如果你打算将 Gradle 作为主要的构建工具，那么就没有必要丢弃已经拥有的知识或者重写已有的构建逻辑。在本章中，我们会学习如何将 Gradle 与 Ant 和 Maven 集成。如果你打算长期使用 Gradle，我们也会一起探索迁移策略。

与 Gradle 集成，Gradle 给 Ant 用户提供了极佳的方案。Gradle 运行时包含标准的 Ant 类库。通过帮助类 `AntBuilder`，任何标准的 Ant 任务都可以通过具有 Groovy 构建风格的标记来使用，和你在 XML 中使用的方式类似，而且所有的构建脚本都可以获取到该 `AntBuilder` 类。Gradle 也可以直接指向一个 Ant 构建脚本，并重用里面的 `target`、`property` 和 `path`。这允许你进行平滑的迁移，以小步前进的方式选择你想重用或者重写的 Ant 构建逻辑。

从 Maven 到 Gradle 的迁移就没有那么简单了。在写这本书的时候，Gradle 还不能和 Maven 的 POM 文件深入集成。但是为了能够让你进行迁移，Gradle 提供了一个转换工具，用来将 Maven 的 `pom.xml` 文件转换成 `build.gradle` 文件。无论你是从一个已有的 Maven 构建迁移还是从头开始，Maven 仓库在构建工具的世界里一直存在。而 Gradle 的 Maven 插件也允许你发布一个工件到本地和远程 Maven 仓库。

将构建流程从一个构建工具迁移到另一个工具是需要策略和使命般付出的。迁移的结果应该是一个更好的、可工作的和可靠的构建，而且不会破坏软件的交付过程。即便对于相对较小规模的迁移，比如，将 Gradle 从一个版本迁移到另一个版本也一样重要。Gradle 提供了一个插件，用来比较两个构建输出的构建结果——升级之前和升级之后——而且可以在产品代码改变和提交之前，给出一个具有决策性的评价。在本章中，你会学习如何使用插件将 To Do 应用从 Gradle 的一个版本升级到另一个版本。让我们先来进一步看看 Gradle 给 Ant 迁移所提供的能力。

## 9.1 Ant与Gradle

Gradle 从两个层次去理解 Ant 的语法。一方面，它能够导入已有的 Ant 脚本，并且假如是原生的 Gradle 语言元素，就可以从一个 Gradle 的构建脚本中直接使用 Ant。Gradle 和 Ant 的这种集成，不需要添加额外的逻辑到 Ant 构建中。另一方面，常用的 Ant 任务（比如 Copy、FTP 等）可以直接在 Gradle 构建脚本中使用，而不需要引入任何 Ant 脚本或额外依赖。Ant 的长期使用者会有一种宾至如归的感觉，可以通过约定和简单易学的 Groovy DSL 重用熟悉的 Ant 功能。第 2 章已经向你展示过如何在 Gradle 构建中使用 Ant 的 Echo 任务。

这两种方式的核心都是 Groovy 的运行时类 `groovy.util.AntBuilder`。它允许在 Groovy 中以一种简洁的方式直接使用 Ant 提供的能力。而 Gradle 则通过添加新的方法来增强 Groovy 的 `AntBuilder` 实现。它是通过类 `org.gradle.api.AntBuilder` 做到的，它继承 Groovy 的 `AntBuilder` 实现，如图 9.1 所示。

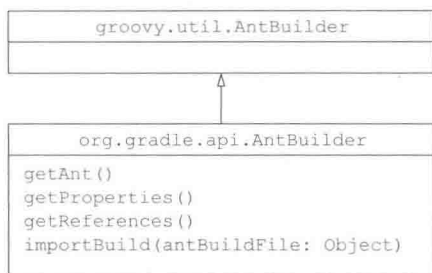


图 9.1 通过 Gradle 提供的 AntBuilder 类访问 Ant 功能

`org.gradle.api.AntBuilder` 类的实例对所有的 Gradle 项目都是隐式可用的，也包括通过属性 `ant` 继承 `DefaultTask` 的每个类。在一般的类文件中，并没有对 `project` 或者 `task` 对象访问的权限，你需要创建一个新的 `AntBuilder` 实例。下面的代码片段展示了如何在 Groovy 类中创建一个 `AntBuilder` 实例：

```
def ant = new org.gradle.api.AntBuilder()
```

让我们来看一些 Gradle 的 `AntBuilder` 用例。你需要使用第 1 章中的 Ant 构建脚本，导入到 Gradle 构建脚本中，重用它的功能，并学习怎么操控它。

### 9.1.1 在 Gradle 中使用 Ant 脚本功能

在 Gradle 中导入 Ant 脚本和重用它的功能是相当简单的。你所需要做的是使用 Gradle 的 `AntBuilder` 类所提供的方法 `importBuild`，并且提供给目标 Ant 构建脚本，如图 9.2 所示。

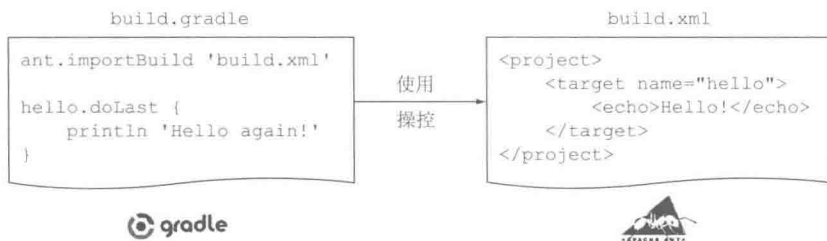
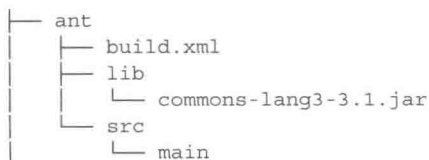
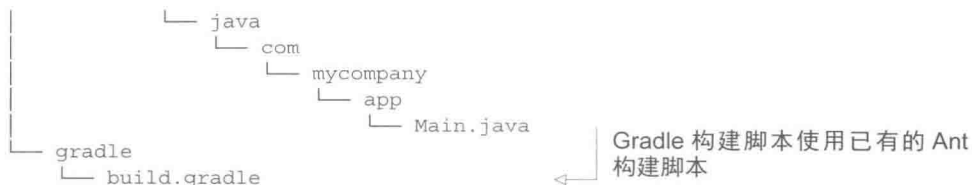


图 9.2 将已有的 Ant 脚本导入到 Gradle 中

为了验证导入功能，你需要将第 1 章中 Ant 构建的目录结构放置到 `ant` 目录下。与该目录位于同一层次，创建另一个目录 `gradle`，该目录中存放负责导入 Ant 脚本的 Gradle 构建脚本。最后的结果应该和下面的目录树结构相似：



原始的 Ant 构建脚本包含多个属性和任务



让我们来看看清单 9.1 中的 Ant 构建脚本。这是一个很简单的脚本，用于编译 Java 源代码和创建 JAR 文件。所需要的外部类库存放在 lib 目录下。该目录仅仅包含一个单一类库，Apache Commons 语言 API。该脚本还定义了一个 target 来初始化构建输出目录。另一个名字为 clean 的 target 会确保目录中已有的 class 文件和 JAR 文件被清除。

清单 9.1 原始的 Ant 构建脚本

```

<project name="my-app" default="dist" basedir=".">
  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="lib" location="lib"/>
  <property name="dist" location="dist"/>
  <property name="version" value="1.0"/>

  <target name="init">
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init" description="compile the source">
    <javac srcdir="${src}" destdir="${build}"
      classpath="${lib}/commons-lang3-3.1.jar"
      includeantruntime="false"/>
  </target>

  <target name="dist" depends="compile"
    description="generate the distribution">
    <mkdir dir="${dist}"/>
    <jar jarfile="${dist}/my-app-${version}.jar" basedir="${build}"/>
  </target>

  <target name="clean" description="clean up">
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>
</project>

```

用于定义目录和 JAR 文件版本的 Ant 属性

用于编译 Java 源代码和创建 JAR 文件的 Ant 任务

现在，让我们来看一下 Gradle 构建文件。这里是 AntBuilder 发挥作用的地方。使用隐式属性 ant，调用方法 importBuild，提供 Ant 构建脚本的路径，如下面的代码片段所示：

```
ant.importBuild '../ant/build.xml'
```

显示了对 Gradle 构建脚本可用的所有 task，可以看到 Ant 的 target 被看作 Gradle 的 task：

```
$ gradle tasks --all
:tasks
```

```
-----
All tasks runnable from root project
-----
```

```
Help tasks
-----
...
```

```
Other tasks
-----
```

```
clean - clean up
dist - generate the distribution
compile - compile the source
init
```

列出 Ant 的顶级 target

Ant 中存在依赖的 target 以缩进形式显示

正如命令行输出显示的那样，Gradle 会察觉到所有可用的 Ant 的 target，将它们包装成 Gradle 的 task，重用它们的描述信息，并保持它们的依赖原封不动。现在，你可以像在执行 Gradle 的 task 一样，执行转化过来的 Ant 的 target：

```
$ gradle dist
:init
:compile
:dist
```

当执行完 dist 这个 target 之后，你会看到源代码被编译、JAR 文件被创建——和在 Ant 中直接运行脚本完全一样。下面的目录树展示了最终的运行结果：

```
.
├── ant
│   ├── build
│   │   ├── com
│   │   │   └── mycompany
│   │   │       └── app
│   │   │           └── Main.class
│   ├── build.xml
│   ├── dist
│   │   └── my-app-1.0.jar
│   ├── lib
│   │   └── commons-lang3-3.1.1.jar
│   ├── src
│   │   ├── main
│   │   │   ├── java
│   │   │   │   ├── com
│   │   │   │   │   ├── mycompany
│   │   │   │   │       └── app
│   │   │   │           └── Main.java
│   └── gradle
│       └── build.gradle
```

Java 类文件

创建的 JAR 文件

### 从 Gradle 中访问导入的 Ant 的 property 和 path

Ant 的 target 会被一对一的转换成 Gradle 的 task，并且可以用完全一样的名字从 Gradle 中触发。这样就在两个构建工具之间实现了一个非常平滑的接口。但是关于 Ant 的 property 或 path 就不是这样了。要访问它们，需要使用 Gradle 的 AntBuilder 对象中的方法 `getProperties()` 和 `getReferences()`。这里要注意的是，Gradle 的 task properties 不会显示任何关于 Ant 的 property 信息。

将 Ant 脚本导入到 Gradle 中只是彻底迁移到 Gradle 的第一步。在 9.1.3 节中，我们会深入讨论更多细节。Gradle task 所包装的 Ant target 在命令行中的输出是非常少的。作为一个 Ant 用户，你也许想要看到 target 被执行时的信息。接下来，我们来看如何减轻看不到信息所带来的痛苦。

### Ant 任务输出日志

在任何时候，你都可以通过将日志级别改变为 INFO（`-i` 命令行参数）来渲染 Ant 的任务输出。正如以往那样，这个命令行参数会给出比你实际想要的更多信息。与其使用这个命令行参数，你可以直接改变 Gradle task 所包装的 Ant target 的日志级别。下面的操作可以改变所有 Ant target 的日志级别到 INFO：

```
[init, compile, dist, clean]*.logging*.level = LogLevel.INFO
```

手动列出导入的 Ant target 是乏味和易犯错的。遗憾的是，没有简单的办法可以绕过它，因为你没有办法区分 task 的起源。运行 `dist` 再次看看 Ant 中的输出：

```
$ gradle dist
:init
[ant:mkdir] Created dir: /Users/Ben/books/gradle-in-action/code/
➤ ant-import/ant/build
:compile
[ant:javac] Compiling 1 source file to /Users/Ben/books/
➤ gradle-in-action/code/ant-import/ant/build
:dist
[ant:mkdir] Created dir: /Users/Ben/books/gradle-in-action/code/
➤ ant-import/ant/dist
[ant:jar] Building jar: /Users/Ben/books/gradle-in-action/code/
➤ ant-import/ant/dist/my-app-1.0.jar
```

你会看到所熟悉的 Ant 的输出。Gradle 前置信息 `[ant:<ant_task_name>]` 会帮助区分出哪些输出来源于 Ant。

Gradle 和 Ant 的集成到这里并没有结束。通常，你会想要进一步修改原始的 Ant 脚本功能甚至扩展它。如果你打算从现有的 Ant 脚本逐步地转移到 Gradle，情

况可能就是这样的。让我们看几个选择。

### 修改 Ant 的 target 行为

当导入已有的 Ant 脚本时，它的 target 会被有效地当作 Gradle 的 task。相应地，你也可以有效地利用它们的特性。还记得我们在第 4 章讨论的将 action 添加到已有 Gradle 的 task 吗？可以通过声明 `doFirst` 和 `doLast` action 将同样的行为应用到已导入的 Ant 的 target 中。下面的清单展示了如何应用该概念，在 Ant 的 target 逻辑执行之前和之后将日志信息添加到 target `init` 中。

清单 9.2 给现有的 Ant 任务添加功能

```
init {
    doFirst {
        logger.quiet "Deleting the directory '${ant.properties.build}'."
    }
    doLast {
        logger.quiet "Starting from a clean slate."
    }
}
```

在 Ant 任务运行之前，执行 Gradle 的 action

在任何 Ant 的 Target 代码执行之后，添加一个 Gradle 的 action

告诉用户即将要删除 Ant 的属性 `build` 所指定的目录

现在当你执行 `init` 时，对应的信息会打印到终端上：

```
$ gradle init
:init
Deleting the directory '/Users/Ben/Dev/books/gradle-in-action/
➡ code/ant-import/ant/build'.
[ant:mkdir] Created dir: /Users/Ben/Dev/books/gradle-in-action/
➡ code/ant-import/ant/build
Starting from a clean slate.
```

将 Ant 的 target 导入到 Gradle 脚本中通常只是在一个联合构建上工作的开始。你也许还想要通过在 Gradle 构建脚本中定义功能来扩展已有的构建模型。在清单 9.2 中，你将看到如何通过 `AntBuilder` 的 `getProperties()` 方法来访问 Ant 的属性 `build`。导入的 Ant 属性并不是静态实体。你甚至可以改变一个 Ant 属性的值以满足自己的需求。你还可以通过挂入一些新的 task 来改变 task 依赖图。通过常规的 Gradle API 工具，可以在 Ant 的 target 和 Gradle 的 task 之间定义依赖，反之亦然。

让我们看一个将这些概念结合在一起使用的具体例子。在下面的清单中，你会大量使用已有的 Ant 属性，改变一个 Ant 属性的值，并让一个导入的 Ant 的 target 依赖于一个新的 Gradle 的 task。

清单 9.3 Ant 和 Gradle 构建的无缝交互

```
ext.antBuildDir = '../ant/build'
ant.properties.build = "${antBuildDir}/classes"
ant.properties.dist = "${antBuildDir}/libs"
```

改变 Ant 属性的值

```

task sourcesJar(type: Jar) {
    baseName = 'my-app'
    classifier = 'sources'
    version = ant.properties.version
    destinationDir = file(ant.properties.dist)
    from new File(ant.properties.src, 'main/java')
}
dist.dependsOn sourcesJar

```

创建包含源文件的 JAR 文件

在 Ant 的 target 上添加一个 task 依赖

这个新的 task `sourcesJar` 看起来并不陌生。它简单地在 `ant/build/libs` 目录下创建了一个包含 Java 源文件的 JAR 文件。因为该 JAR 文件应该是发布内容的一部分，所以将它定义为 `dist` 的依赖。执行构建自动执行 task 图中的 task。

```

$ gradle clean dist
:clean
:init
:compile
:sourcesJar
:dist

```

任务 `sourcesJar` 作为任务 `dist` 的依赖被执行

构建结果的 JAR 文件可以在新的发布目录中找到：

```

└─ ant
    └─ build
        ├── classes
        │   └─ ...
        └─ libs
            ├── my-app-1.0-sources.jar
            └── my-app-1.0.jar
    └─ ...

```

重新定义 classes 输出目录

重新定义发布目录

生成包含 Java 源文件的 JAR 文件

包含产品 class 文件的 JAR 文件

到目前为止，你已经学到了如何应用 Gradle 的特性集来简化和已有的 Ant 构建脚本的集成。接下来，你将学习 Gradle 独特的强大特性：增量构建。

### 给 Ant 的 target 添加增量构建能力

因为 Ant 的 target 并不能判断自己的状态，所以构建工具 Ant 并不支持增量构建。当然，你也可以使用国产技术实现它（有可能导致错误）来防止不必要的 target 执行（比如，在时间戳文件的帮助下）。如果 Gradle 为该功能提供了内建机制，为什么还要自己去实现呢？下面的清单展示了如何为从 Ant 脚本中导入的编译 target 定义输入和输出。

#### 清单 9.4 为导入的 Ant 的 target 定义输入和输出

```

compile {
    inputs.dir file(ant.properties.src)
    outputs.dir file(ant.properties.build)
}

```

定义编译输入目录 `../ant/src`

定义编译输出目录 `../ant/build/classes`



看起来很直接，对吧？试一把。首先，清理已有的 class 和 JAR 文件并运行整个生成过程：

```
$ gradle clean dist
:init
:compile
:sourcesJar
:dist
```

正如所期望的，Java 源代码被编译，JAR 文件被创建。如果不删除这些文件，再次运行 dist 任务时，Gradle 就会意识到源文件没有改变，输出文件已经存在。编译任务就自动标记为 UP-TO-DATE，正如下面命令行输出的那样：

```
$ gradle dist
:init
:compile UP-TO-DATE
:sourcesJar UP-TO-DATE
:dist
```

← Gradle 将编译任务标记为 UP-TO-DATE 并跳过执行

能够给导入的 Ant 的 target 添加增量构建功能已经是 Ant 用户转向 Gradle 的巨大胜利了。结果证明，这么做节省了大量时间，特别是含有许多依赖和源文件的企业级构建。

即使你不导入已有的 Ant 构建脚本，Gradle 也允许从构建脚本中直接执行 Ant 任务。在本书中，你已经看到了一些相关例子。为了完善所看到的用例，我们来讨论如何将一个 Ant 标准任务添加到构建中。

### 9.1.2 在 Gradle 中使用标准的 Ant 任务

Gradle 的 AntBuilder 提供了在构建脚本中直接访问所有的标准 Ant 任务的能力——不需要任何额外配置。在运行时中，Gradle 会检查其绑定的 Ant JAR 文件在 classpath 上对相应的 Ant 任务都是可用的。图 9.3 展示了在 Gradle 构建脚本中 Ant 任务的使用、Gradle 运行时以及其包含的 Ant 任务之间是如何交互的。

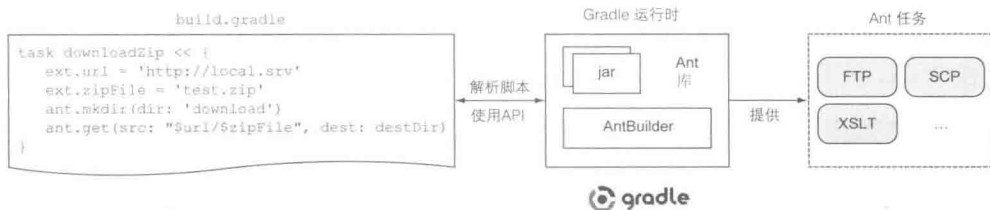


图 9.3 在 Gradle 中使用 Ant 任务

在 Gradle 构建脚本中使用 Ant 任务并不难，你只要记住一些基本规则：

- 使用隐式的 AntBuilder 变量 `ant` 来定义一个 Ant 任务。
- 在 AntBuilder 实例中使用的 Ant 任务名和 Ant 中的标签名是一样的。
- `task` 属性包装在圆括号中。
- 使用这种方式定义一个 `task` 属性名和属性值：`<name>:<value>`。另外，`task` 属性可以以 Map 方式提供；比如 `[<name1>: <value1>, <name2>: <value2>]`。
- 嵌套的 `task` 元素不需要使用 `ant` 隐式变量。父级 `task` 元素会用大括号包住它们。

让我们看一个具体的例子：Ant 的 Get 任务。这个任务的目的是通过 HTTP(S) 将远程文件下载到本地磁盘上。你可以在 Ant 的在线手册上看到相关文档 (<https://ant.apache.org/manual/Tasks/get.html>)。假设你想要下载两个文件到 `downloads` 目录中。图 9.4 展示了如何使用 AntBuilder 的 DSL 来表达这段逻辑。

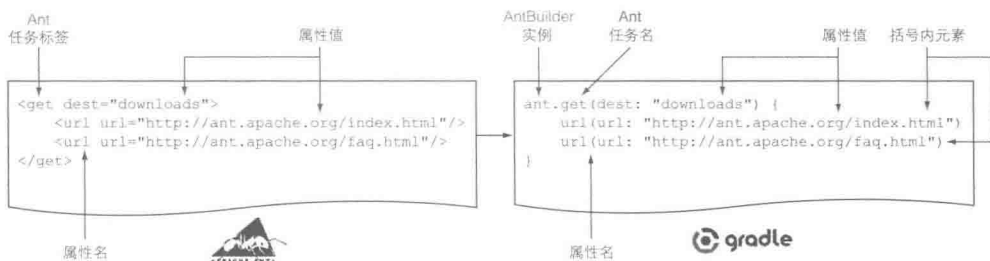


图 9.4 与 Gradle 相关的 Ant 任务元素

如果你比较 Ant 和 Gradle 中的 `task` 定义，就会看到它们有一点点不同。你丢弃了尖括号，换来了更可读的 `task` 定义。重要的一点是，你不需要重写 Ant 的功能，而且它和 Gradle 无缝地集成在一起。

接下来，你会使用之前代码样例中的 `Get` 任务。作为发布的一部分，你需要绑定项目描述文件和版本注释。每个文件放在不同的服务器上，并且不是版本控制中源代码的一部分。这是使用 Ant 任务 `Get` 的最佳用例。下面的清单展示了如何使用 Ant 任务 `Get` 来下载这两个文件，并将它们作为生成的 JAR 文件的一部分。

#### 清单 9.5 使用标准的 Ant 任务 Get

```
task downloadReleaseDocumentation {
    logging.level = LogLevel.INFO
    ext.repoUrl = 'https://repository-gradle-in-action.forge.cloudbees.com/
        release'

    doLast {
        ant.get(dest: ant.properties.build) {
            url(url: "$repoUrl/files/README.txt")
            url(url: "$repoUrl/files/RELEASE_NOTES.txt")
        }
    }
}
```

使用隐式的 AntBuilder 变量来访问 Ant 任务 Get

定义嵌套在 Get 任务中的 URL 元素

```

    }
}
}
dist.dependsOn downloadReleaseDocumentation

```

所有的标准 Ant 任务都可以通过这种技术来使用，因为它们都和 Gradle 运行时绑定在一起。当写逻辑时，要确保 Ant 文档在手边。可选的或者第三方 Ant 任务通常需要你将额外的 JAR 文件添加到构建脚本的 classpath 下。在第 5 章中，当你使用外部的 Ant 任务 Cargo 将 To Do 应用部署到 Web 容器时，你已经知道怎么做了。关于如何使用可选的 Ant 任务更多信息，请参考第 5 章中的代码样例。

至此，你已经学习了许多方法和已有的 Ant 构建脚本或者 Gradle 的 task 进行交互。但是如果你打算长期使用 Gradle 怎么办？如何一步步地完成迁移呢？

### 9.1.3 迁移策略

Gradle 并不强制你将已有的 Ant 脚本一次性完全迁移到 Gradle 上。导入已有的 Ant 构建脚本，使用现有逻辑，然后慢慢熟悉 Gradle，是一个好的开始。第一步，你只需要投入最小的努力。让我们来看看你也许想要尝试的其他方法。

#### 将独立的 Ant 的 target 迁移到 Gradle 的 task

你需要将 Ant 的 target 的逻辑转换成 Gradle 的 task，但是应该从简单的逻辑开始。尝试用“Gradle 方式”去实现 target 的逻辑，而不是回到之前使用 AntBuilder 的方式。让我们通过一个例子来讨论一下。假设你有下面的 Ant 的 target：

```

<target name="create-manual">
  <zip destfile="dist/manual.zip">
    <fileset dir="docs/manual"/>
    <fileset dir="." includes="README.txt"/>
  </zip>
</target>

```

在 Gradle 中，最好在一个增强型 org.gradle.api.tasks.bundling.Zip 的 task 的帮助下实现，正如下面这段代码所示：

```

task createManual(type: Zip) {
    baseName = 'manual'
    destinationDir = file('dist')
    from 'docs/manual'
    from('.') {
        include 'README.txt'
    }
}

```

这种方式可以在不需要显式的定义输入和输出的情况下，自动地提供增量构建功能。对于想要转换的逻辑，如果没有直接的 Gradle 的 task 类型，则仍然可以使用

AntBuilder。随着时间的推移，你会发现 Ant 构建脚本变得越来越少，而 Gradle 构建中的会逐渐增加。

## 引入依赖管理

Gradle 的众多有用特性之一就是依赖管理。如果你是 Ant 用户而且没有使用 Ivy 进行依赖管理，那么它会帮你减轻手动管理外部类库所带来的负担。当迁移到 Gradle 时，即便你在 Ant 的 target 内编译源代码，也可以使用依赖管理。你所需要做的是将依赖声明放到 Gradle 构建脚本中，并给 Ant 脚本提供一个新的属性。下面的清单展示了这样做的一个简单的代码样例。

清单 9.6 定义编译依赖

```
configurations {  
    antCompile  
}  
  
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    antCompile 'org.apache.commons:commons-lang3:3.1'  
}  
  
ant.properties.antCompileClasspath = configurations.antCompile.asPath
```

给 Ant 编译依赖定义对应的 configuration

设置一个新的 Ant 属性，它将会在 Ant 构建脚本的编译过程中被使用

定义指定给对应 configuration 的 Ant 依赖

有了这段代码，你可以使用一个名字为 antCompileClasspath 的 Ant 属性，在 Ant 构建脚本中设置 classpath：

```
<target name="compile" depends="init" description="compile the source">  
    <javac srcdir="${src}" destdir="${build}"  
        classpath="${antCompileClasspath}" includeantruntime="false"/>  
</target>
```

使用在 Gradle 构建脚本中定义的编译 classpath

对 Ant 构建脚本的改变是非常少的。现在你可以去掉 Ant 构建中的 lib 目录了，因为 Gradle 的依赖管理器会自动下载依赖。当然，你也可以将 Javac task 移动到 Gradle 中。但是当你可以简单地使用 Gradle 的 Java 插件时，为什么还要这么麻烦呢？引入插件会消除对 Ant 构建逻辑的需要。

## 解决 task 命名冲突

迟早有一天，在迁移过程中，你会遇到想要加入一个或多个 Gradle 插件的情况。你的样例 Ant 项目就类似于 Java 项目所需要的典型任务：编译源代码，组装 JAR 文件，清除已有工件。只要 Ant 脚本不会定义任何 target 和插件所暴露出来的 task 名字一样，

使用 Java 插件就有神效。试一下，修改 Gradle 脚本成下面的样子：

```
ant.importBuild '../ant/build.xml'
apply plugin: 'java'
```

执行任意的 Gradle 的 task，Gradle 都会指出你有一个 task 命名空间冲突了，如下面命令行输出所示：

```
$ gradle tasks

FAILURE: Build failed with an exception.

* Where:
Build file '/Users/Ben/Dev/books/gradle-in-action/code/migrating-ant-
build/gradle/build.gradle' line: 2

* What went wrong:
A problem occurred evaluating root project 'gradle'.
> Cannot add task ':clean' as a task with that name already exists.
```

遇到这种情况，你有两种选择。要么排除已有的 Ant 的 target，要么用带有不同名字的 Gradle 的 task 来包装导入的 Ant 的 target。使用什么方法取决于具体的情况。下面的代码片段展示了如何通过 AntBuilder 来解决 task 已经存在的问题：

```
ant.project.addTarget('clean', new org.apache.tools.ant.Target())
ant.importBuild '../ant/build.xml'
apply plugin: 'java'
```

通过名字  
clean 排  
除 Ant 的  
target

结果就是，原先 Ant 的 target 被排除掉；使用 Java 插件提供的 clean 任务来代替。

排除一些不太复杂的 Ant 的 target 可能有用，但是有些时候，你想要保留已有的逻辑，因为重写它会花费很多时间。在这种情况下，你可以以另一种间接的方式来构建它，如图 9.5 所示。

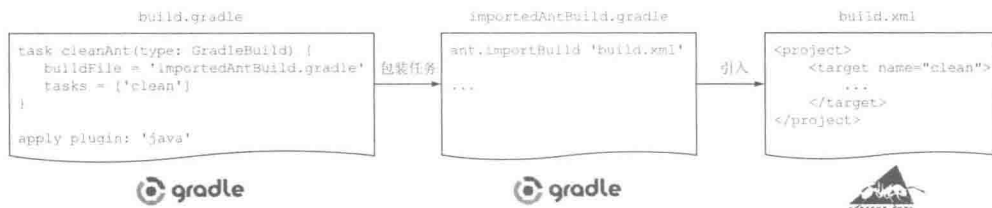


图 9.5 通过暴露带有新名字的 Gradle 任务包装导入的 Ant 任务

导入的 Ant 构建脚本可以放在另一个名字是 importAntBuild.gradle 的 Gradle 构建脚本中：

```
ant.importBuild '../ant/build.xml'
```

作为消费者的 Gradle 构建脚本会声明一个增强型 GradleBuild 的 task，它会用一个新名字来定义你想要使用的 Ant 的 target。你可以将这种技术看作是重命名一个 Ant 的 target。下面的代码展示了这种用法：

```
task cleanAnt(type: GradleBuild) {
    buildFile = 'importedAntBuild.gradle'
    tasks = ['clean']
}
apply plugin: 'java'
```

GradleBuild 类型的 task 的名字

构建脚本引入的 task 在被包装的 task 触发时执行

引入 Ant 的 Gradle 构建文件

有了这段代码，暴露的 Gradle 任务名字就是 cleanAnt：

```
$ gradle tasks
:tasks

-----
All tasks runnable from root project
-----

...
Other tasks
-----
cleanAnt
...

```

名字是 cleanAnt 的 task 会直接调用引入的 Ant 任务

如果你想要让标准的 Gradle 的 clean 任务依赖于 Ant 的 clean-up 逻辑，则可以在它们之间定义一个任务依赖：

```
clean.dependsOn cleanAnt
```

我们讨论了在不阻碍构建和交付过程的前提下，如何一步步地将 Ant 迁移到 Gradle 上。在下一小节中，我们会对比 Maven 和 Gradle 概念之间的共同点和不同点。正如在这一节所做的，我们也会讨论构建迁移策略。

## 9.2 Maven和Gradle

Gradle 与 Ant 能够很好地集成。允许导入已有的 Ant 构建脚本，将 target 转换成 Gradle 的 task，让 Gradle 毫无阻碍地执行 Ant 的 target，并提供通过额外的 Gradle 功能使用 target 的能力（比如，增量构建支持）。有了这些特性的支持，你可以采取各种策略将 Ant 迁移到 Gradle。

遗憾的是，对于 Maven 的集成却没有这么方便。撰写本书时，Gradle 还没有支持对于 Maven 构建脚本的导入。也就是说，你不能给 Gradle 构建指定一个 POM 文件，然后希望在运行时获得相关构建信息并执行 Maven 构建。但是也有相应的策略来解决这个问题，我们会在本节中讨论。在讨论将 Maven 迁移到 Gradle 之前，我

们来比较一下这两个构建系统的异同之处。然后，我们来看看 Maven 的核心概念和 Gradle 功能的对应关系。

9.2.1 异同之处

直接比较 Maven 和 Gradle 时，你会发现有很多相同的地方。这两种构建工具使用了相同的概念（比如，依赖管理和约定优于配置），尽管它们使用了不同的方式来实现，采用了不同的词汇，或者不同的使用方法。我们来讨论一下在 Gradle 论坛中经常被问到的一些重要的区别。

provided 作用域

Maven 用户在 Gradle 中声明依赖时会感到很方便。Java 插件或者 War 插件提供的很多依赖作用域都直接等同于 Maven 中的定义。请参考表 9.1 进行快速的复习。

表 9-1 Maven 依赖作用域以及对应的 Gradle 配置

Maven 作用域	Gradle Java 插件配置	Gradle War 插件配置
compile	compile	N/A
provided	N/A	providedCompile, providedRuntime
runtime	runtime	N/A
test	testCompile, testRuntime	N/A

War 插件中仅有的一个有代表性的 Maven 作用域是 provided。使用 provided 作用域定义的依赖在编译时会用到，但是不会被导出（与运行时分发包绑定）。这个作用域假设运行时环境提供了这个依赖。一个典型的例子就是 Servlet API 库。如果你不是在构建 Web 应用程序，那么不会在 Gradle 中有相应的配置。这很好解决——你可以在构建脚本中将作用域的行为定义成自定义配置。下面的清单显示了如何创建一个用于编译目的的 provided 作用域，以及一个专用于单元测试的 provided 作用域。

清单 9.7 自定义的 provided 配置

```
configurations {
    provided
    testProvided.extendsFrom provided
}

sourceSets {
    main {
        compileClasspath += configurations.provided
    }
    test {
        compileClasspath += configurations.testProvided
    }
}
```

声明 provided 配置

将 provided 配置添加到编译 classpath 下

## 部署到 Maven 仓库的工件

在构建工具的外部依赖库中，Maven 仓库无所不在。尤其是 Maven Central，它是在互联网上搜寻开源库的首选位置。

至此，你已经学到了如何使用 Maven 仓库中的库。能够将工件发布到 Maven 仓库也是非常重要的，因为在企业中可能会设置一个 Maven 仓库用于在不同团队或者部门之间共享可重用的库。Maven 本身就支持将工件发布到本地或者远程的 Maven 仓库。作为这个过程的一部分，会生成一个 pom.xml 文件，它包含了关于工件的元信息。

Gradle 提供了一个 100% 兼容的插件，和 Maven 相似，用来完成上传工件到仓库的功能，这就是 Gradle Maven Publishing 插件。在本章中我们不会进一步讨论这个插件。如果你迫不及待地想要学到关于它的更多知识，请直接跳到第 14 章吧。

## 对 Maven profile 的支持

Maven 2.0 引入了构建 *profile* 的概念。一个 profile 通过 POM（项目 pom.xml、settings.xml 或者 profiles.xml 文件）中的元素定义了一些与环境相关的参数集合。profile 的一个典型用例就是在特定的部署环境下定义属性。在 Gradle 中，你会想要重用已经存在的 profile 定义，或者模拟这个功能。

很抱歉让你失望了，Gradle 并不支持 profile 的概念。请不要因此沮丧。有两种办法可以解决这个问题。我们先来看看如何读取一个已经存在的 profile 文件。假设在 Maven 家目录（~/m2）下有一个 settings.xml 文件，如下面的清单所示。

清单 9.8 Maven profile 文件定义应用程序服务器家目录

```
<?xml version="1.0" encoding="UTF-8"?>
<settings>
  <profiles>
    <profile>
      <id>appserverConfig-dev</id>
      <activation>
        <property>
          <name>env</name>
          <value>dev</value>
        </property>
      </activation>
      <properties>
        <appserver.home>/path/to/dev/appserver</appserver.home>
      </properties>
    </profile>
    <profile>
      <id>appserverConfig-test</id>
      <activation>
        <property>
          <name>env</name>
          <value>test</value>
        </property>
      </activation>
      <properties>
```

开发环境属性

用于开发环境的应用程序服务器家目录

测试环境属性



```

        <appserver.home>/path/to/test/appserver</appserver.home>
    </properties>
</profile>
</profiles>
...
</settings>

```

用于测试环境的应用程序服务器家目录

这个 settings 文件声明了两个 profile 用来确定用于部署目的的应用服务器家目录。根据 env 指定的环境信息，Maven 会选择对应的 profile。

通过 Gradle 实现同样的功能非常简单。下面的清单显示了如何读取 settings 文件，遍历 XML 元素，并根据提供的 env 选择所要求的 profile 文件。

### 清单 9.9 从 settings 文件中读取环境相关的属性

```

def getMavenSettingsCredentials = {
    String userHome = System.getProperty('user.home')
    File mavenSettings = new File(userHome, '.m2/settings.xml')
    XmlSlurper xmlSlurper = new XmlSlurper()
    xmlSlurper.parse(mavenSettings)
}

task printAppServerHome << {
    def env = project.hasProperty('env') ? project.getProperty('env')
    : 'dev'
    logger.quiet "Using environment '$env'"
    def settings = getMavenSettingsCredentials()
    def allProfiles = settings.profiles.profile
    def profile = allProfiles.find {
        it.activation.property.name == 'env' &&
        it.activation.property.value == env
    }
    def appServerHome = profile.properties.'appserver.home'
    println "The $env server's home directory: $appServerHome"
}

```

使用 Groovy 的 XmlSlurper 解析 settings 文件

解析特定环境提供的属性

遍历 XML 元素找到应用程序服务器属性的值

运行这个示例，调用任务名称并且提供属性作为命令行参数：

```

$ gradle printAppServerHome -Penv=test
:printAppServerHome
Using environment 'test'
The test server's home directory: /path/to/test/appserver

```

如果你坚持使用这个 settings 文件，这可以很好地工作。然而，有时候你可能想要摆脱这个工件，从而完全摆脱对 Maven 概念的依赖。定义特定的 profile，有很多选择：

- 把它们放在 gradle.properties 文件中。遗憾的是，属性名本身是扁平的，所以需要指定一个命名格式；比如，env.test.app.server.path。
- 直接在构建脚本中定义。这种方法的好处就是你能够使用 Groovy 支持的任意数据类型来声明属性。

- 使用 Groovy 的 ConfigSlurper 工具类读取 Groovy 脚本形式的配置文件。第 14 章使用了这种方法并且提供了完整的示例。

## 生成站点

在使用 Gradle 的时候, Maven 用户最感兴趣的一个功能就是为项目创建一个站点。Maven 的 site 插件允许通过运行一个目标来创建一系列的 HTML 文件, 网站通常公开了一个从 POM 中提取的信息的统一视图, 也包含了关于测试和静态代码分析结果的整合报告。

在撰写本书的时候, Gradle 并没提供这个功能。用来实现这一功能的标准的 Gradle 插件就是 build-dashboard。这个插件在 Gradle 1.5 的时候被引入, 暴露了一个用来生成 HTML 格式的报告面板的 task, 其中包含了所有生成报告的引用。请参考 Gradle 在线文档来获取更多信息。

## 9.2.2 迁移策略

在前面一节中, 我们讨论了 Maven 和 Gradle 这两种构建工具的主要区别。前面演示的解决差异的方法会让你的成功迁移有个大概的开始。因为 Gradle 并不支持在运行时将 Maven 目标导入到 Gradle 模型中, 对于已经存在的 Maven 设置的迁移就必须并行进行。这个过程保证在流畅迁移的同时不会破坏构建和交付流程。

我们来看看第 1 章中的 Maven POM 文件。快速回顾一下代码, 见如下清单。

清单 9.10 : 原始的 Maven POM 文件

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0</version>
  <dependencies>
    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-lang3</artifactId>
      <version>3.1</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
</project>
```

← Maven 的 artifactId 元素值映射到 Gradle 项目名称属性

幸运的是, 不需要手动将 Maven 构建逻辑转换成 Gradle 脚本。如果你是 Gradle 1.6 及以后版本的用户, 那么能获得一点帮助: build setup 插件。这个插件支持通过分析 Maven 的 pom.xml 生成 build.gradle 和 settings.gradle 文件, 如图 9.6 所示。

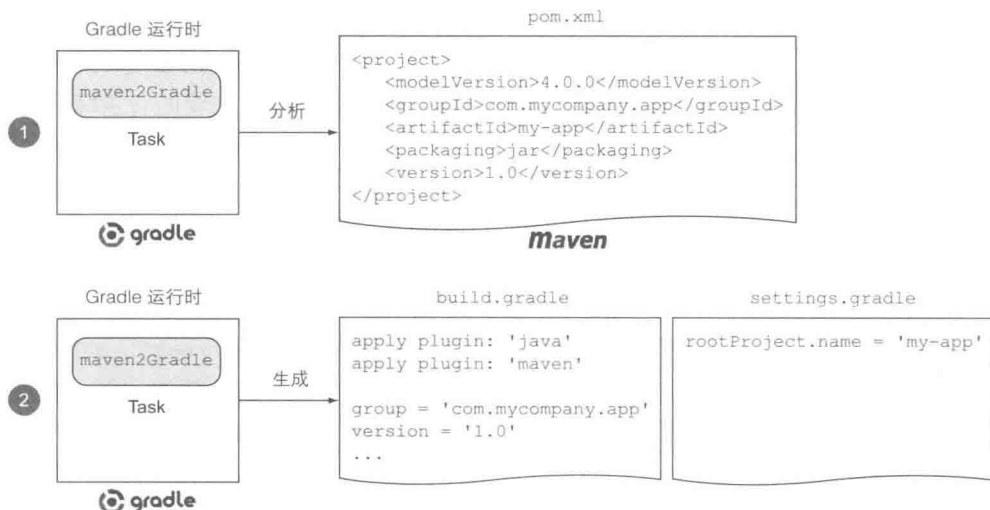


图 9.6 从 Maven POM 生成 Gradle 构建脚本

我们来仔细看看如何使用这个插件。

### 使用 maven2Gradle task

build setup 插件对于所有独立于构建脚本的配置的 Gradle 构建是自动生效的。在第 2 章和第 3 章中我们已经学到了这个插件能够被用来生成新的 Gradle 构建脚本或者为项目添加 wrapper 文件。这个插件提供的另一个 task 我们还没有仔细讨论过，那就是 maven2Gradle。这个 task 只有在当前目录中包含一个 pom.xml 文件的时候才可见。创建一个新目录，创建一个 pom.xml 文件，然后把清单 9.11 的内容复制进去。这个目录应该如下：

```
.
└─ pom.xml
```

现在，在命令行中切换到这个目录，列出所有可用的 Gradle task：

```
$ gradle tasks --all
:tasks

-----
All tasks runnable from root project
-----

Build Setup tasks
-----
setupBuild - Initializes a new Gradle build. [incubating]
maven2Gradle - Generates a Gradle build from a Maven POM.
               [incubating]
setupWrapper - Generates Gradle wrapper files. [incubating]
...
```

Maven POM  
转换器 task

执行 `maven2Gradle` task 会分析有效的 Maven POM 配置。这里说有效的 POM，是指 `pom.xml` 文件中的配置、任何父 POM，以及活跃的 profile 提供的任何设置。即使这个 task 还处于早期的开发阶段，你也可以获得如下主要的转换特性：

- 单模块和多模块 Maven 项目的转换
- 对定义的 Maven 依赖和仓库的转换
- 支持将 Maven 项目转换为构建纯 Java 项目以及 Web 项目
- 分析项目基本信息如 ID、描述、版本和编译器设置，并将其转换成 Gradle 配置

在大多数情况下，这个转换 task 在将 Maven 构建逻辑转换为 Gradle 的时候，都做得很好。需要注意的是，这个转换器不能够理解任何第三方插件的配置，因此不能对其创建任何 Gradle 代码。这个逻辑只能手动实现。你将在 `pom.xml` 文件中执行这个 task：

```
$ gradle maven2Gradle
:maven2Gradle
Maven to Gradle conversion is an incubating feature. Enjoy it and let
us know how it works for you.
Working path: /Users/Ben/Dev/books/gradle-in-action/code/maven2gradle

This is single module project.
Configuring Maven repositories... Done.
Configuring Dependencies... Done.
Adding tests packaging...Generating settings.gradle if needed...
Done.
Generating main build.gradle... Done.
```

执行完这个 task 后，你会在与 POM 文件相同的目录下找到 Gradle 文件：

```
.
├── build.gradle
├── pom.xml
└── settings.gradle
```



生成的 Gradle 文件

我们来仔细看看这个生成的 `build.gradle` 文件，如下面清单所示。这个文件包含了所有必需的 DSL 元素：插件、项目基本信息、仓库和依赖。

#### 清单 9.11 生成的 Gradle 构建脚本

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

group = 'com.mycompany.app'
version = '1.0'

description = ""
```

```
sourceCompatibility = 1.5
targetCompatibility = 1.5

repositories {
    mavenRepo url: "http://repo.maven.apache.org/maven2"
}

dependencies {
    compile group: 'org.apache.commons', name: 'commons-lang3', version: '3.1'
}
```

为 Maven Central 生成 URL，而不是使用 `mavenCentral()` 快捷方式

项目名只能在 Gradle 初始化阶段设置。因为这个原因，`maven2Gradle task` 只生成了如下清单所示的 `settings` 文件。

#### 清单 9.12 生成的 Gradle settings 文件

```
rootProject.name = 'my-app'
```

生成的 Gradle 文件为你的迁移开了个好头。在将所有功能完全迁移之前，你不需要破坏原有的构建，你现在可以在生成的 Gradle 逻辑上工作了。此时，你可以继续使用 Gradle 作为主要的构建工具，并且开始树立起信心了。如果遇到了任何阻碍，你可以随时切换回 Maven 构建。

如果你能够自动确定 Maven 和 Gradle 构建之间的构建工件是否相同，是不是很好呢？那就是 Gradle 的 `build comparison` 插件的主要目标。

## 9.3 比较构建

`Build comparison` 插件是在 Gradle 1.2 版本时引入的。其最重要的目标就是比较两个构建的输出。当我说输出的时候，意思是指构建产生的二进制工件——比如，JAR、WAR 或 EAR 文件。这个插件的目标是支持如下比较：

- 在构建工具迁移的过程中比较 Gradle 构建和 Ant 或者 Maven 构建
- 在升级的情况下，比较两个不同 Gradle 版本的构建
- 在改变构建逻辑后比较同一版本的 Gradle 构建

我知道你很兴奋，因为这个插件对你从 Maven 或者 Ant 迁移到 Gradle 后比较构建输出有非常大的帮助。遗憾的是，我不得不扫你的兴。在撰写本书的时候，这个功能还没有被实现。你所能做的就是，比较 Gradle 版本升级前后的构建。图 9.7 演示了在 To Do 应用程序上下文中的一个用例。

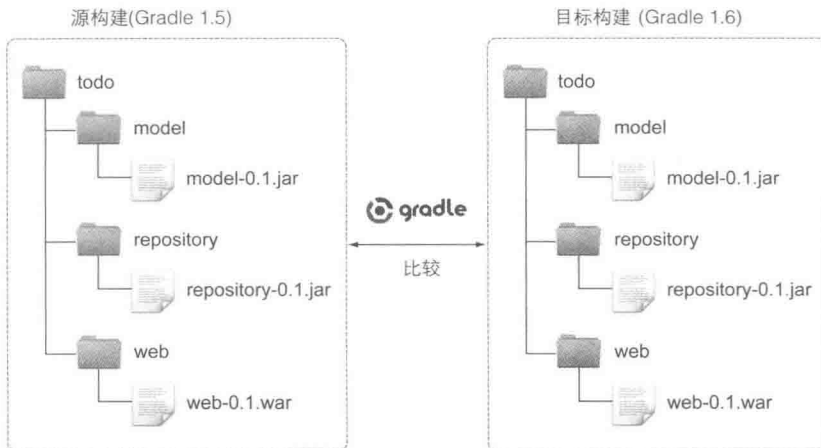


图 9.7 比较两个 Gradle 版本构建的输出

你的示例项目创建了 3 个二进制工件：model 和 repository 产生的两个 JAR 文件，以及 web 项目产生的一个 WAR 文件。这个 WAR 文件包含了两个其他的 JAR 文件。两个构建之间的比较必须要考虑这些文件。我们假设你想要从 Gradle 1.5 升级到 Gradle 1.6。如下清单显示了必需的设置。

#### 清单 9.13 升级 Gradle 运行时

```

apply plugin: 'compare-gradle-builds'

compareGradleBuilds {
    sourceBuild {
        projectDir = rootProject.projectDir
        gradleVersion = '1.5'
    }

    targetBuild {
        projectDir = sourceBuild.projectDir
        gradleVersion = '1.6'
    }
}

```

指向根项目的源构建定义

指向源构建定义的根项目的目标构建定义

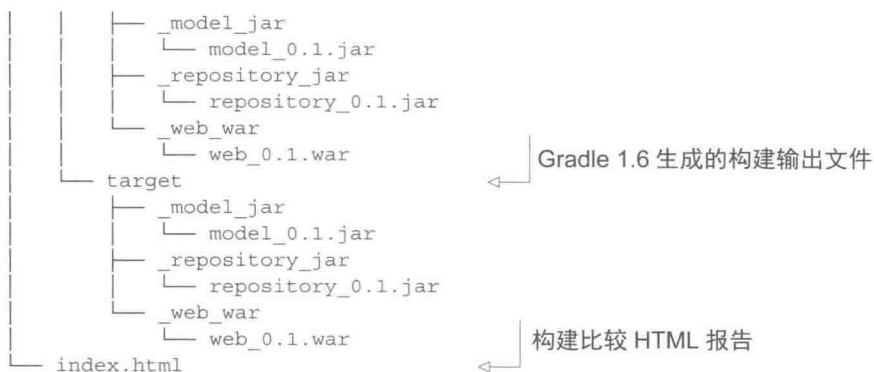
如果你想要开始构建比较，你需要做的就是执行所提供的 `compareGradleBuilds` task。如果两个构建的输出不同，那么这个 task 就会执行失败。这个 task 的命令行输出很冗长，这里就不演示了。更有趣的是这个 task 产生的报告结构。如下目录树显示了比较的构建输出文件和 HTML 报告：

```

├── build
│   └── reports
│       └── compareGradleBuilds
│           ├── files
│           └── source

```

Gradle 1.5 生成的构建输出文件



报告文件 `index.html` 给出了关于比较构建的详细信息、它们的版本、涉及的二进制工件，以及比较结果。如果 Gradle 认为比较的构建不是相同的，那么会在报告中反映出来，如图 9.8 所示。

## Gradle Build Comparison

The build outcomes were not found to be identical.



### Comparison host details

Project: /Users/Ben/Dev/books/gradle-in-action/code/chapter09/todo-build-comparison  
 Task: :compareGradleBuilds  
 Gradle version: 1.5  
 Executed at: 5/11/13 7:45 PM

### Compared builds

	Source Build	Target Build
Project	/Users/Ben/Dev/books/gradle-in-action/code/chapter09/todo-build-comparison	/Users/Ben/Dev/books/gradle-in-action/code/chapter09/todo-build-comparison
Gradle version	1.5	1.6
Tasks	clean assemble	clean assemble
Arguments		

图 9.8 升级 Gradle 版本的构建比较 HTML 报告示例

我想这个插件在判断升级是否有副作用的时候帮助是非常明显的。这个插件将来有很多潜在的价值，尤其是在构建迁移过程中想要通过其他构建工具来比较已有的构建的时候。

## 9.4 总结

在本章中，我们讨论了如何实现传统的基于 Java 的构建工具 Ant 和 Maven 与 Gradle 的集成和迁移。Ant 提供了最丰富的选项和最强大的工具，而 Gradle 则允许通过将 Ant 的 `target` 转换为 Gradle 的 `task` 来实现 Ant 脚本的深层引入。即使不考虑导入已有的 Ant 构建，重用标准的以及第三方的 Ant `task` 也能让使用者受益良多。

本章中还介绍了可以通过小步方式完成 Ant 到 Gradle 的迁移：首先导入现有的构建，然后引入依赖管理，最后使用 Gradle API 将 target 转换为 task。最后要强调的是，好好利用 Gradle 插件。

Maven 和 Gradle 拥有相似的概念和约定。如果你之前使用过 Maven，就会发现 Gradle 与 Maven 在基础项目结构和依赖管理使用模式方面几乎相同。虽然 Gradle 并未提供 Maven 所具有的某些特性，如 provided 作用域以及 profile 的概念，但 Gradle（及其最终实现语言 Groovy）的灵活性足以提供方法来弥补这种缺失。遗憾的是，Gradle 并不支持在运行时对 POM 中 Maven 目标的导入，这也使得 Maven 到 Gradle 的迁移不如 Ant 那么顺畅。作为 Maven 到 Gradle 成功迁移的第一步，Gradle 提供了 maven2Gradle 转换 task，可以根据有效的 POM 生成 build.gradle 文件。

Gradle 版本的升级不会产生任何副作用或者影响代码的编译、组装以及产品代码的部署。Build comparison 插件通过比较不同版本 Gradle 的构建结果对版本升级进行自动化测试，从而降低升级失败带来的影响。

恭喜你，你已经学到了 Gradle 最核心的特性！这一章也是本书第 2 部分的结尾。在第 3 部分中，我们将重点关注持续交付过程中 Gradle 的用法。首先，我们将讨论 Gradle 在 IDE 中的使用



## 第3部分

# 从构建到部署

在开发人员的开发机器上构建应用程序只是整个软件构建的一部分。在快速和频繁交付软件压力增加的时代，自动化部署和发布过程显得极其重要。在第3部分中，你会学到如何在持续交付的场景中将 Gradle 运用到极致。

很多开发人员运用 IDE 就像生存和呼吸一样自如，IDE 是使得这些程序员高效工作的关键。第10章通过从头开始构建一个 Gradle 项目，并使用 IDE 来管理和执行构建，从而详细地讲述如何使用 Gradle 的核心插件来生成 IDE 项目文件。

现如今很少有软件项目仅仅集中使用一种编程语言来完成所有的功能。相反，开发人员会引入合适的编程语言来使他们的工作更快捷高效。在第11章中，我们会讨论如何只利用 Gradle 来构建多种编程语言混合的软件项目。

持续交付描述了一个构建管道，这个精心设计的构建管道能够将软件从开发人员的开发机器上一步步地部署到产品环境中。本书最后 4 章用 Gradle 和第三方工具实现了一个这样的构建管道。我们会从每个构建管道的核心——持续集成开始。在第 13 章中，你会学到如何使用 Jenkins，一个开源的持续集成工具来创建一连串的构建步骤。在这些构建步骤之间，第 12 章中所描述的代码分析工具将会为所构建的代码质量把关。第 14 章和第 15 章将讨论如何将软件项目打包和发布成可部署的工件，你会将这个工件部署到不同的环境当中。

学习完第 3 部分后，你将能够使用 Gradle 技巧将项目提升一个等级。全自动化的交付过程会为你所服务的公司或组织省下一大笔钱，并大大地缩短软件的上线周期。

# 10

## IDE支持和工具

---

### 本章涵盖

- 使用 Gradle 的 IDE 插件生成项目文件
- 在流行的 IDE 中管理 Gradle 项目
- 使用工具 API 内嵌 Gradle

Gradle 的 target 运行时环境是命令行，通常使用文本编辑器来编辑构建脚本。开发人员可选择的文本编辑器有从简单的面向终端的编辑器如 vi 或 Emacs，到成熟的集成开发环境（IDE）如 Eclipse 和 IntelliJ IDEA。IDE 可以极大地提高开发人员的效率。IDE 提供的语法高亮显示、快捷键快速跳转、重构、代码自动补全和自动生成等功能为开发人员在开发过程中节省了大量宝贵的时间，并且使得开发过程更加有趣。开发 Gradle 代码也不例外。

IDE 将项目的配置数据存放在项目文件中。这些项目文件通常都是 XML 文件，不同的 IDE 有不同的项目文件。很早以前流行的 IDE 并不支持与 Gradle 的集成，所以 Gradle 不得不提供插件来生成这些文件。在本章中，我们会讲解 Gradle 所提供的插件如何为流行的 Java IDE 生成项目文件。这些插件作为至关重要的工具，使得 IDE 能够从构建脚本中获得几乎所有的基本信息。虽然自动生成项目文件是一个非

常好的开始，但是你通常需要更细粒度的控制来自定义配置。我们会讨论如何使用这些插件所暴露的 DSL 来调整这些配置，从而满足项目需要。

随着 Gradle 越来越流行，IDE 供应商提供了很多一流的管理 Gradle 项目的内置支持。我们会探讨如何通过 Gradle 构建脚本导入已有的项目、切换项目，并直接在 IDE 中执行 Gradle 的 task。这项功能因 IDE 而异。在本章中，我们会比较 Eclipse、IntelliJ IDEA、NetBeans 和 Sublime Text 所提供的特性集。

大部分的 IDE 供应商都利用工具 API 将 Gradle 嵌入到他们的产品中。工具 API 是 Gradle 对外发布的公共 API 的一部分，用来执行和监控构建。虽然这些 API 主要关注 Gradle 与第三方应用的集成，但它也可被应用在其他场景中。你将会了解到这些用例并且实践这些 API。最开始，你将会用 Gradle 的 IDE 插件来为 To Do 应用生成项目文件。

## 10.1 使用IDE插件生成项目文件

IDE 以项目为基本单位，与 Gradle 项目类似。每个项目都定义了类型（如 Web 应用与桌面应用）、外部依赖（如 JAR、源代码和 Javadoc 文件）和个人设置。有了项目文件，一个项目就可以在 IDE 中打开，并且在不同的开发人员之间共享配置数据。遗憾的是，不同的 IDE 产品之间的项目文件格式并不统一。最常被使用的两种格式是 XML 和 JSON。

Gradle 允许使用插件为 IDE 生成项目文件。标准的 Gradle 分发包提供了两个开箱即用的插件：Eclipse 和 IDEA。每个插件都知道如何生成特定的 IDE 项目文件。这两个插件也暴露了强大的 DSL 来自定义所生成的项目文件。为了生成项目文件，你需要将 IDE 插件应用到构建脚本中，执行插件所提供的 task，然后将生成的项目文件导入到 IDE 中，如图 10.1 所示。

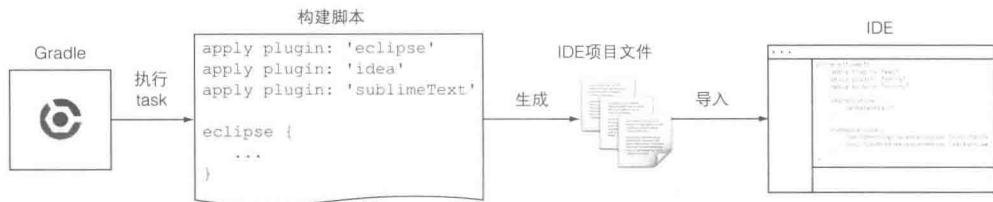


图 10.1 使用 Gradle 生成 IDE 项目

为了能够共享项目文件，你通常会将项目文件与源代码文件一起放到 VCS 中。当其他开发人员从 VCS 中获取项目时，他们能够直接在 IDE 中打开项目并开始工作。使用 Gradle 的 IDE 插件使得这一步骤显得多余，因为你已经使用插件提供的 DSL

描述了项目设置。与将源代码编译成 class 文件的过程类似，项目文件可以在任何时候被重新生成。

在本章中，你会生成项目文件使 To Do 应用能够被加载到 Eclipse、IntelliJ IDEA 和 Sublime Text 中。因此，Gradle 的 IDE 插件需要将独立的需求翻译成 IDE 的配置数据。在本书中，你会为一个简单的 Gradle 项目添加一些非常具体的配置元素：

- 多项目构建定义，包括编译时项目依赖
- 自定义配置用来声明外部依赖
- 为集成测试和功能测试自定义 source set

你会学到如何利用 Gradle 的插件所提供的 DSL 来自定义这些配置。你会从学习生成 Eclipse 项目文件开始。

### 10.1.1 使用 Eclipse 插件

Eclipse (<http://www.eclipse.org/>) 可能是 Java 世界最流行、最被广泛使用的 IDE。Eclipse 完全开源，并且能够以扩展插件的方式来添加特殊功能，比如支持 Groovy 项目和版本控制系统的使用如 Git。在开始生成 Eclipse 项目文件之前，你需要对 Eclipse 项目文件的格式有一个好的理解，因为插件的 DSL 元素直接与具体的文件相关。

#### 项目格式和文件

Eclipse 为每个项目都保存了一份配置数据。这意味着多项目构建中的每个子项目都包含其自己的 Eclipse 项目文件和目录。所有的配置数据都是 XML 格式的。文件或目录有如下 3 种类型：

- `.project`：这个文件的文件名不言自明——它保存了项目的基本信息。包括名字、描述、其他项目或资源的引用和项目类型。
- `.classpath`：这个文件描述了所引用的外部依赖库和其他项目的清单。
- `.settings`：这个目录是可选的。包含了特定工作空间的相关设置。这个目录下的文件保存了像 Java 编译器版本和源代码版本合规等设置。

对于 To Do 应用，所生成的项目文件与图 10.2 类似。

图中的每一个项目，包括根项目和子项目都有其自己的项目文件。接下来，你将会为项目应用相应的 Gradle 插件并生成项目文件。

Eclipse 项目文件

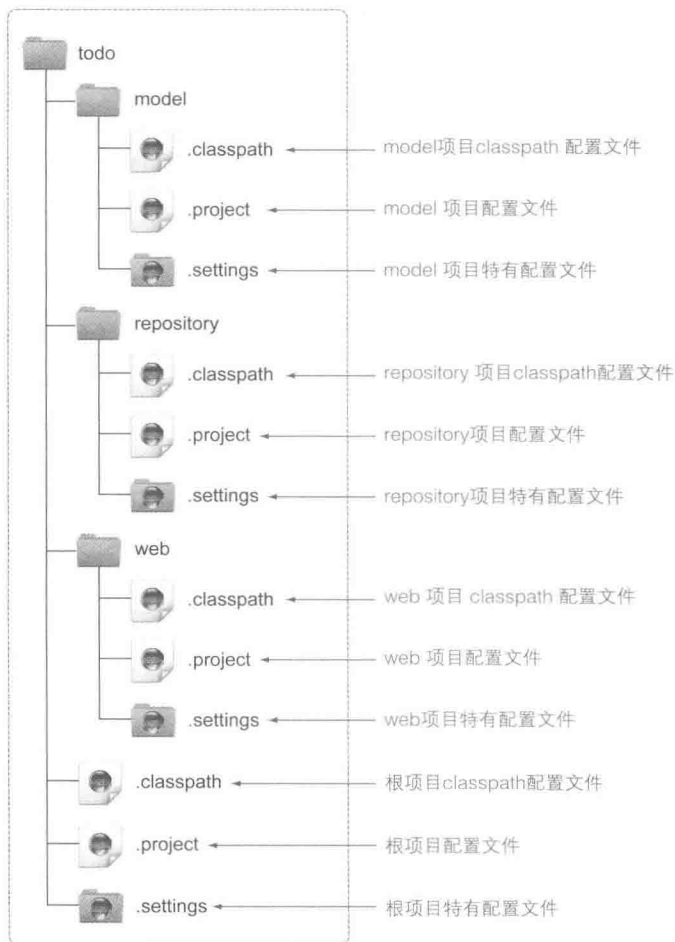


图 10.2 Eclipse 项目文件

### 应用并执行插件

Gradle 分发包自带两个 Eclipse 插件：`eclipse` 和 `eclipse-wtp`。`eclipse` 插件负责生成标准的 ECLIPSE 配置数据。`eclipse-wtp` 插件依赖于 `eclipse` 插件并负责生成 Eclipse 的 Web 工具平台（Web Tool Platform，WTP）使用的相关配置文件。WTP 提供开发 Java EE 程序的相关工具，可以作为可选的 Eclipse 插件被安装。WTP 插件使生成 Web 工件的过程变得简单，如 Web 描述符、Servlet 和 JSP 文件。它也支持将 WAR 文件部署到不同的 Web 容器中，这使得你可以方便地在 IDE 中调试应用程序。

To Do 程序的核心是一个 web 程序，所以其非常适合同时代应用这两个 Gradle 的 Eclipse 插件。首先，将 `eclipse` 插件应用到程序的所有 Gradle 项目中：

```
allprojects {
    apply plugin: 'eclipse'
}
```

将插件应用到 `allprojects` 配置块会为根项目和所有的子项目创建项目文件。你只需要为 `web` 项目生成 `WTP` 配置文件。可以如下面代码片段所示来应用这个插件：

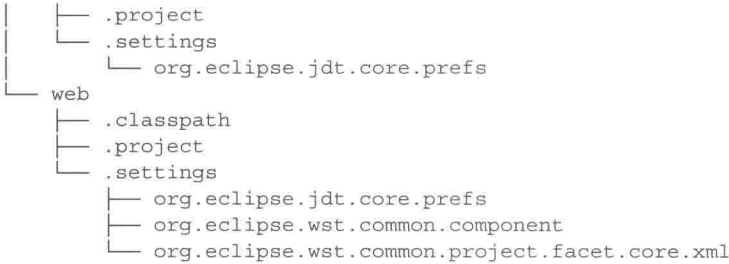
```
project(':web') {
    apply plugin: 'eclipse-wtp'
}
```

有了这两个插件后，你可以执行它们所提供的 `task` 来生成 Eclipse 项目文件了。两个非常重要的 `task` 是：`eclipse` 和 `cleaneclipse`。`eclipse` `task` 负责生成所有的 Eclipse 项目文件，包括 `.project`、`classpath` 和 `.settings` 目录下的设置文件。`cleaneclipse` `task` 负责清除所有已存在的 Eclipse 项目文件。试着在多个项目结构的根项目目录下执行 `eclipse` `task`：

<code>\$ gradle eclipse</code>	
<code>:eclipseProject</code>	为根项目创建 <code>.project</code> 文件
<code>:eclipse</code>	
<code>:model:eclipseClasspath</code>	为 model 项目创建 <code>.project</code> 、 <code>classpath</code> 和 JDT 设置文件
<code>:model:eclipseJdt</code>	
<code>:model:eclipseProject</code>	
<code>:model:eclipse</code>	
<code>:repository:eclipseClasspath</code>	为 repository 项目创建 <code>.project</code> 、 <code>classpath</code> 和 JDT 设置文件
<code>:repository:eclipseJdt</code>	
<code>:repository:eclipseProject</code>	
<code>:repository:eclipse</code>	
<code>:web:eclipseClasspath</code>	为 web 项目创建 <code>.project</code> 、 <code>classpath</code> 和 JDT 设置文件
<code>:web:eclipseJdt</code>	
<code>:web:eclipseProject</code>	
<code>:web:eclipseWtpComponent</code>	为 web 项目创建 WTP 设置文件
<code>:web:eclipseWtpFacet</code>	
<code>:web:eclipseWtp</code>	
<code>:web:eclipse</code>	

`eclipse` `task` 自动执行了很多依赖的 `task`。每个依赖的 `task` 都负责生成一种特定类型的项目文件。例如，`eclipseclasspath` `task` 生成 `.classpath` 文件的内容。下面的目录树显示了最终结果。

```
├─ .project
├─ model
│   ├── .classpath
│   ├── .project
│   └── .settings
│       └─ org.eclipse.jdt.core.prefs
├─ repository
│   └─ .classpath
```



我们已经讨论了 `.project` 和 `.classpath` 配置文件的作用。我们来仔细看一下其中一个生成的设置文件。每个子项目都有一个 `org.eclipse.jdt.core.prefs` 文件，这个文件包含了由 Java 开发工具集 (JDT) 所提供的 Java 项目相关配置信息。JDT 设置的一个例子就是 Java 编译器的版本。

现在用 Eclipse 打开生成的项目文件，默认的 Gradle 插件配置已经为你做了哪些呢？所有的项目都会被识别成 Java 或 Groovy 项目（在 Eclipse 中叫作项目性质），设置好了正确的源路径，Gradle 的 Java 插件默认的依赖也会被加入到项目中。至此，使用 Eclipse 的 WTP 工具的准备工作就完成了。Gradle 插件提供了很多默认的配置，这些配置不需要你再做过多的自定义，就能直接使用。这样，还缺什么呢？缺所有不能被 Gradle 自动识别的需要自定义的配置。在这个例子中就是 `functtestcompile` 和 `functtestruntime`，这两个配置是为了描述功能测试依赖所定义的。你可以自定义项目文件的生成过程来达到这个目的。

自定义配置

Gradle 提供的 Eclipse 插件暴露的强扩展性的 DSL 几乎可以自定义项目文件生成过程的每一个方面。表 10.1 显示了一些重要的关键属性来访问 Eclipse 的生成模型。

表 10.1 Eclipse 插件配置属性

属性名	Gradle API 类	插件	描述
project	Eclipseproject	eclipse	配置项目信息
classpath	EclipseClasspath	eclipse	配置 classpath 信息
jdt	EclipseJdt	eclipse	配置 JDT 信息
wtp.component	EclipsewtpComponent	eclipse-wtp	配置 WTP 信息
wtp.facet	EclipsewtpFacet	eclipse-wtp	配置 WTP facet 信息

所有的关注点都被很好地分离了，所以每一个特有的属性都可以单独配置。在接下来的一节中，我们会依次使用每一个属性。项目环境准备工作可能因具体项目而异。接下来的例子很有可能与你的真实案例不同。如果你想要有更深入的理解，Gradle DSL 指南就是最好的开始。请记住，任何时候改变配置想要重新生成项目文件时，清除已有的项目文件是一个好习惯。通过执行 `cleanclipse` task 可以很



容易地清除已有的项目文件：

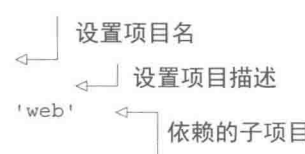
```
$ gradle cleanEclipse eclipse
```

Gradle 的 Eclipse 插件从显式的或者隐式的构建脚本配置中继承了很多 Eclipse 项目文件的配置值。例如，Eclipse 的项目名就是从 Gradle 的 `project.name` 属性继承而来的。所有的这些预配置的值都是可以重新配置的。

你可以从调整根项目细节开始。在下面的清单中，通过给 `eclipse.project.name` 属性赋一个新值可以很容易地覆盖 Eclipse 项目的默认值。你还设置了几个其他的属性。

#### 清单 10.1 设置根项目属性

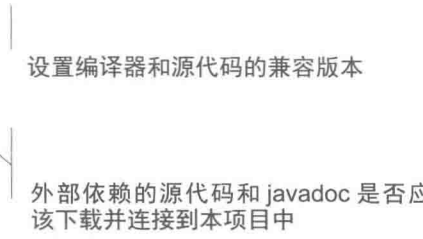
```
eclipse {  
    project {  
        name = 'todo'  
        comment = 'A task management application'  
        referencedProjects 'model', 'repository', 'web'  
    }  
}
```



如果你只是想设置子项目的属性，则可以把设置语句放在 `subprojects` 配置块中。下面的清单显示了如何为 JDT 属性设置值，以及是如何影响 `classpath` 生成的。

#### 清单 10.2 为所有的子项目设置 JDT 和 classpath 属性

```
subprojects {  
    apply plugin: 'java'  
  
    eclipse {  
        jdt {  
            sourceCompatibility = 1.6  
            targetCompatibility = 1.6  
        }  
  
        classpath {  
            downloadSources = true  
            downloadJavadoc = false  
        }  
    }  
}
```



Eclipse 单个子项目的配置是建立在已经存在的配置信息基础之上的。你所需要做的就是在这个子项目的构建脚本中添加另一个 `eclipse` 配置块。你也可以用同样的方法修改一个已经存在的配置值。如下清单演示了如何设置项目描述，添加自

定义的 Gradle 配置，以及个性化 WTP 配置信息。

### 清单 10.3 调整 web 子项目的 Eclipse 属性

```
project(':web') {  
    eclipse {  
        project {  
            comment = 'Web components for managing To Do items in the browser'  设置项目描述  
        }  
        classpath {  
            plusConfigurations << configurations.functestCompile  
            plusConfigurations << configurations.functestRuntime  添加自定义配置到 Eclipse classpath  
        }  
        wtp {  
            component {  
                contextPath = 'todo'  在 WTP 中为 Web 应用程序设置 contextPath  
            }  
        }  
    }  
}
```

这些例子告诉我们使用 Gradle 插件的 DSL 所暴露的属性自定义已生成的配置非常简单。所暴露的属性包含了大多数常用的配置选项。然而，DSL 不可能考虑到第三方 Eclipse 插件的配置选项或者用户的所有个性化设置。因此，DSL 提供了 hook 让你能够操作模型的生成过程，从而控制最后生成的 XML 格式的项目文件。

### 操作已生成的配置

有两种方法可以自定义已经生成的项目文件。一种是使用 withXml hook 直接操作 XML 领域对象模型（DOM）。另外一种注册 merge hook beforeMerged 或者 whenMerged，从而能够直接操作 Eclipse 元数据模型的领域对象。我们通过实现一个实例来看看这些 hook 的使用。

当将 Gradle 的 eclipse-wtp 插件应用到 web 项目后，你可以为 Eclipse WTP 的基本功能生成项目文件。遗憾的是，eclipse-wtp 插件默认并没有预配置支持编辑 JavaScript 文件（包括语法高亮显示和代码补全等）。

在 Eclipse 中，一个功能单元叫作一个 facet，如 JavaScript 的编辑功能就是一个 facet。为了配置 JavaScript facet，你需要指定 wst.jsdt.web 值作为 facet key。下面的清单演示了如何追加一个 XML DOM 节点来代表添加 1.0 版本的 JavaScript facet。

清单 10.4 使用 XML hook 添加 WTP JavaScript facet

```

project(':web') {
    eclipse {
        wtp {
            facet {
                file {
                    withXml { xml ->
                        def node = xml.asNode()
                        node.appendNode('installed', [facet: 'wst.jsdt.web',
                            version: '1.0'])
                    }
                }
            }
        }
    }
}

```

为操作已生成的 XML 文件  
添加 hook

为添加的 JavaScript facet 追加新的节点

添加完这些配置并重新生成项目文件后，你就能够看到 `web/.settings/org.eclipse.wst.common.project.facet.core.xml` 文件了，它包含了 JavaScript facet 节点。

使用 `merge hook` 也可以达到同样的配置作用，如 `whenMerged`。在 WTP facet 元素中应用 `merge hook` 时，能够获得一个具有 `WtpComponent` 实例的闭包。在下面的清单中，你会实例化一个新的 `Facet` 类型的实例，将需要的数据放在构造函数中，并将新生成的实例添加到 facet 列表中。

清单 10.5 使用 merge hook 添加 WTP JavaScript facet

```

import org.gradle.plugins.ide.eclipse.model.Facet

project(':web') {
    eclipse {
        wtp {
            facet {
                file {
                    whenMerged { wtpComponent ->
                        wtpComponent.facets << new Facet('wst.jsdt.web', '1.0')
                    }
                }
            }
        }
    }
}

```

在 Gradle 获得了构建信息后  
添加 hook 用于管理 WTP 组件文件

向 WTP 组件的 facet 列表中  
添加 JavaScript facet

### 选择“正确的”hook 类型

你可能会问自己，在什么样的情况下应该用什么类型的 hook，或者是不是有更好的选择？一句话，没有。Gradle 提供了大量的灵活选项。通常，在项目中你都能随意选择配置选项。

我个人的观点是用 mergehook，因为我可以直接操作表示配置信息的领域对象。这些领域对象通常描述了能够使其正常工作的数据。如果你不确定要配置哪些选项，则可以根据 Eclipse 插件所提供的 Javadocs 来判断。

如果你需要添加一个配置，但是这个配置在 Gradle API 中不是通过领域对象来维护的，那么你随时可以使用 XML hook 来实现。XML hook 不会为你提供配置数据做任何校验，所以很可能提供一些不规范的配置数据。对于 Eclipse XML 配置文件的结构，请参考 Eclipse 官方文档。

是时候看看你努力的成果了。接下来，你会将生成的项目文件导入到 Eclipse 中。

## 导入项目

Eclipse 针对不同的项目类型有不同的安装包。因为你工作在 Java Enterprise 项目下，所以最合适的就是专门为 JavaEE 开发人员定制的 Eclipse IDE。安装包包含了很多有用的插件。如果你还没有安装 Eclipse，那么从 Eclipse 官方网站下载这个安装包并安装吧。

标准的 Eclipse 安装包没有提供导入项目结构的方便方法，所以你需要单独导入每个项目。导入单独的一个项目的方法是，选择菜单项 File → Import... → General → Existing Projects into Workspace。单击 Next 按钮并浏览选择项目所在的根目录。Eclipse 会将找到的有效的项目文件显示在准备导入的项目列表中。单击 Finish 按钮后，你就能够在 Package Explorer 标签中找到刚导入的项目。如果针对根项目和所有的子项目重复做如上导入操作，那么 Eclipse 工作空间应该类似于图 10.3。

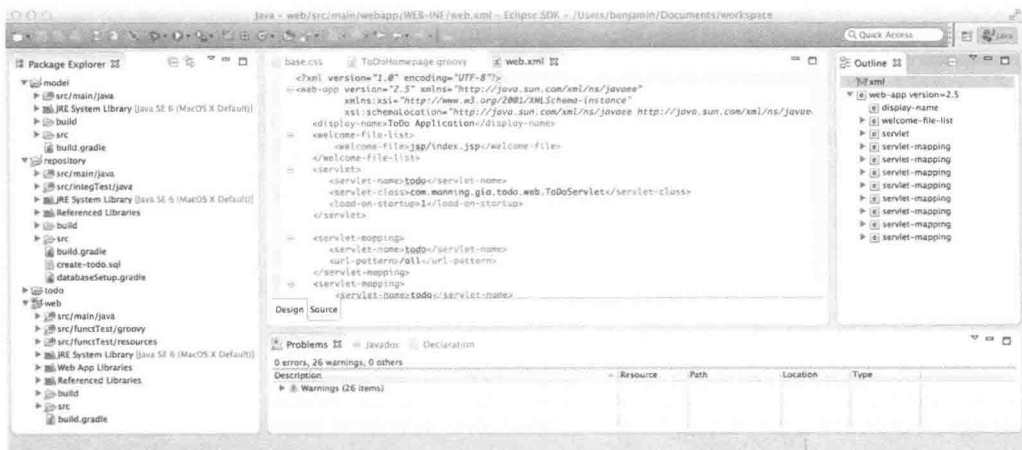


图 10.3 在 Eclipse 中导入项目

祝贺你——如果你是 Eclipse 用户，你已经成功地以重复的方式为你和你的同行们设置好了项目。任何时候，你都可以重新生成项目文件。在后面的章节中，我

们还会接触到一个更加复杂的 Eclipse 安装包，叫作 SpringSource Tool Suite (STS)，它提供了现成的导入 Gradle 项目结构的支持。在下一节中，你会学到如何为 IntelliJ IDEA 用户实现同样的效果。

### 10.1.2 使用 IDEA 插件

IntelliJ IDEA (<http://www.jetbrains.com/idea/>) 是基于 JVM 项目的一个商业 IDE。它为很多流行的框架提供扩展并且被一套开发者集成工具所支持。尽管很多功能都是其核心产品预安装的，但可以通过插件的形式进行扩展。让我们来快速地浏览一下 IntelliJ 的项目文件。

#### 项目格式和文件

IntelliJ 将项目文件保存在项目的根目录下。实际数据是 XML 格式的。我们根据下面的文件扩展名来区分项目文件：

- `.ipr`：存储了核心项目信息。在多项目环境下，对于子项目的引用就保存在这里。
- `.iml`：在 IntelliJ 中，一个功能单元（又名 module，模块）存储在一个 `.iml` 的模块描述文件中。一个模块配置文件中包含了源代码相关的配置信息、构建脚本和一些相关的描述信息。在多项目构建，每个 Gradle 项目对应一个模块配置文件。
- `.iws`：包含了特定的工作空间设置。不管单项目还是多项目都只有一个 settings 文件。

To Do 应用程序基于的项目文件如图 10.4 所示。

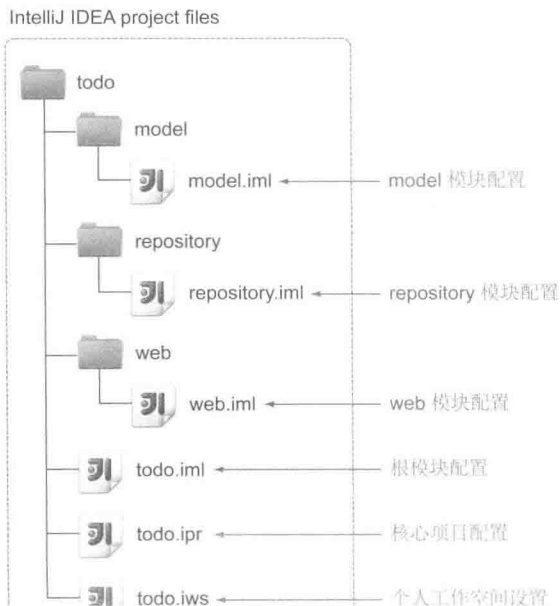


图 10.4 IntelliJ IDEA 项目文件

每个项目——根项目和子项目——至少包含了模块配置文件。`.ipr` 和 `.iws` 文件保存在根项目级别的路径下。接下来，你会应用并执行 Gradle IDEA 插件来生成这些项目文件。

应用并执行插件

Gradle 的标准分发包包含了 `idea` 核心插件。你可以用这个插件来生成上述讨论的所有的 INTELLI 项目文件。为了给 To Do 应用程序的所有项目生成项目文件，需要将 `idea` 插件应用到根项目构建脚本的 `allprojects` 配置块中：

```
allprojects {
    apply plugin: 'idea'
}
```

这个插件提供的主要 task 就是 `idea`。执行这个 task，会生成如图 10.4 所示的所有项目文件。这个 task 执行后的命令行输出如下：

```
$ gradle idea
:ideaModule
:ideaProject
:ideaWorkspace
:idea
:model:ideaModule
:model:idea
:repository:ideaModule
:repository:idea
:web:ideaModule
:web:idea
```

为根项目创建的模块、项目和工作空间相关文件

对于子项目，只有生成模块文件的 task 被执行

你已经能够使用生成的项目文件打开整个 IntelliJ 项目了。现在，先暂停一下，因为 IDEA 插件还不能识别自定义的 `source set` 和配置。只能通过自定义配置来实现。接下来，你会学到如何调整所生成的配置数据。

自定义配置

Gradle IDEA 插件为自定义项目文件、`module` 文件和工作空间文件的生成设置属性暴露了丰富的 DSL。表 10.2 显示了访问内建模型的相关配置。

表 10.2 IDEA 插件配置属性

属性名	Gradle API 类	插 件	描 述
<code>project</code>	<code>IdeaProject</code>	<code>idea</code>	配置项目信息
<code>module</code>	<code>IdeaModule</code>	<code>idea</code>	配置模块信息
<code>workspace</code>	<code>IdeaWorkspace</code>	<code>idea</code>	配置工作空间信息

我们将在 To Do 应用程序上下文中研究每一个配置元素。如果在任何时候你想

要设置这个例子中没有提到的属性，那么请确保你手边有 Gradle DSL 指南。修改了配置后，最好用 `cleanIdea` 命令将已有的项目文件清理掉。

```
$ gradle cleanIdea idea
```

主要的 IDE task 的用法在 Eclipse 和 IDEA 插件之间是相同的。这使得用户可以方便地切换 IDE，或者同时支持两种 IDE。你将从自定义项目文件开始。如下清单演示了如何在多项目构建的根项目级别设置项目属性。

#### 清单 10.6 设置根项目属性

```
idea {  
    project {  
        jdkName = '1.6'  
        languageLevel = '1.6'  
    }  
}
```

设置特定的 JDK 和 Java 语言版本

如果你想要在多项目构建中为所有的子项目做自定义配置，则可以使用 `subprojects` 配置块。接下来，你会提供指令来下载 Java 项目外部依赖的源文件。如下清单显示了如何为每个 `module` 文件做充分的配置。

#### 清单 10.7 为所有的子项目设置模块属性

```
subprojects {  
    apply plugin: 'java'  
  
    idea {  
        module {  
            downloadSources = true  
            downloadJavadoc = false  
        }  
    }  
}
```

为所有子项目的外部依赖下载源文件

避免为所有的子项目都下载 Javadoc

正如之前所提到的，IDEA 插件并不能立刻就识别到自定义的 `source set`。你可以通过修改 `repository` 和 `web` 子项目的构建脚本来很容易地修改默认的 `source set` 设置。从 `repository` 子项目定义的 `source set integrationTest` 开始。幸运的是，你不用再次重复目录路径。Gradle 允许通过其 API 直接访问文件路径。如下清单演示了一个关于如何使用 Gradle 的 Java 插件提供的 DSL 来遍历所要求的信息，从而应用 IDEA 插件的完美例子。

清单 10.8 为 repository 子项目添加自定义的 source set

```

project(':repository') {
    idea {
        module {
            sourceSets.integrationTest.allSource.srcDirs.each {
                testSourceDirs += it
            }
        }
    }
}

```

遍历 integration Test source set 的所有目录

把每个目录添加到 IntelliJ 所识别的 test 源目录中

你需要为 web 模块应用相同的自定义配置。清单 10.9 显示了如何添加 functionalTest 这个 test source set。除此之外，清单中还演示了如何将自定义配置定义的依赖添加到 IDE 的 test 编译阶段。

清单 10.9 为 web 子项目添加自定义 source set 和配置

```

project(':web') {
    idea {
        module {
            sourceSets.functionalTest.allSource.srcDirs.each {
                testSourceDirs += it
            }

            scopes.TEST.plus += configurations.functestCompile
            scopes.TEST.plus += configurations.functestRuntime
        }
    }
}

```

将每个目录添加到 IntelliJ 能识别的 test 源目录中

遍历 functionalTest source set 的所有目录

添加自定义配置到模块 classpath

创建完这些配置后，你可以重新生成 IntelliJ 项目文件并在 IDE 中打开项目。在这样做之前，我们来看看这个插件所提供的 XML hook 和 merge hook。

### 操作已生成的配置

IDEA 插件提供了与 Eclipse 插件类似的底层的自定义 hook。事实上，它们甚至遵循相同的命名规范。使用 withXml hook 可以修改 XML 配置信息，使用 beforeMerge 和 afterMerge 可以合并修改。

因为 IDEA 插件很好地继承了构建脚本的一些默认配置，所以作为一个终端用户，插件不可能立即就为你提供所需要的所有配置。我们来看一个例子。Gradle 预安装了 JetGradle 插件来支持运行构建脚本。为了能够使用这个插件的所有功能，必须先选中一个构建脚本。IDEA 插件在生成项目文件时并没有为你做这个选择。幸运的是，你可以使用 XML hook 和 merge hook 来生成这些配置。

我们来看看如何使用 XML hook 实现这个目标。在下面的清单中，你首先要获得访问 XML 根节点，然后新增一个 XML 节点作为根节点的子节点，最后将所需要



的配置指向根项目的构建脚本。

清单 10.10 使用 XML hook 预配置项目设置

```
idea {
  project {
    ipr.withXml { provider ->
      def node = provider.asNode()
      def gradleSettings = node.appendNode('component',
        [name: 'GradleSettings'])
      gradleSettings.appendNode('option', [name: 'linkedProjectPath',
        value: '$PROJECT_DIR$/build.gradle'])
    }
  }
}
```

创建新的 XML 组件指向根项目构建脚本

添加 hook 来管理生成的 XML 文件

遗憾的是，使用 merge hook 不能够达到相同的效果。相比 Eclipse 插件所提供的领域模型，IDEA 插件提供的更少。最好的选择就是使用 XML hook 实现所需要的功能，同时提供了最大的灵活性。

merge hook 同样也是有其用处的，我们来看一个例子。除了使用 `idea.project` 提供的属性以外，你还可以使用插件提供的项目领域模型。下面的清单演示了如何使用 `withMerged` hook 来设置 JDK 名字和 IntelliJ 项目的语言级别。

清单 10.11 使用 merge hook 预配置项目设置

```
import org.gradle.plugins.ide.idea.model.IdeaLanguageLevel
import org.gradle.plugins.ide.idea.model.Jdk

idea {
  project {
    ipr.whenMerged { project ->
      project.jdk = new Jdk('1.6', new IdeaLanguageLevel('1.6'))
    }
  }
}
```

添加 hook 来管理项目领域模型

使用领域类配置 JDK 名字和语言级别

## 导入项目

为了充分利用 IntelliJ 所提供的性能，下载旗舰版本。你有 30 天的免费试用期。在本机安装 IntelliJ 后，你就可以导入项目结构了。在 IntelliJ 中，通过选择根项目级别的项目文件可以导入整个多项目结构。在菜单栏，选择 `File → Open...` 并选择生成的 `.ipr` 文件。最终导入后如图 10.5 所示。



- `.sublime-workspace`：保存用户相关的数据，如当前打开的文件和修改的设置。

在 To Do 应用程序上下文中，对于每个 Gradle 项目都会有一个 `.sublime-project` 文件。在编辑器中打开项目之后，Sublime Text 会自动创建 `workspace` 文件。一个 `workspace` 设置文件可以被任意多个项目所共享，而不仅仅是根项目。图 10.6 以目录树的形式显示了项目文件。

Sublime Text 项目文件

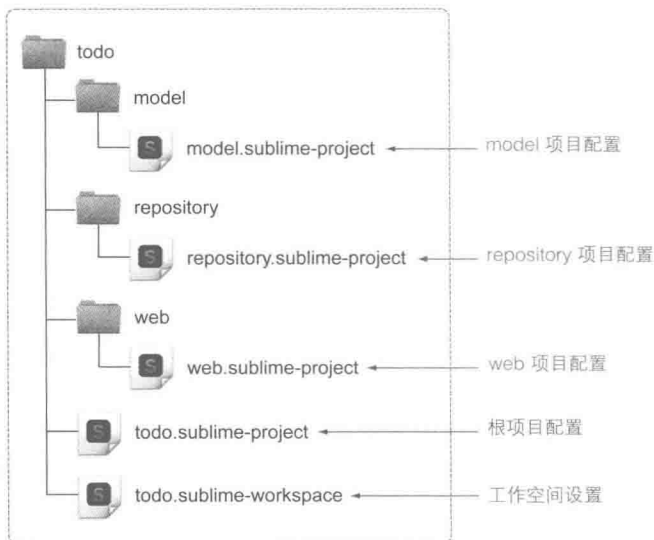


图 10.6 Sublime Text 项目文件

与 Gradle 自带的 Eclipse 和 IDEA 插件类似，你可以在第三方插件的帮助下生成 Sublime Text 项目文件。接下来，你会通过实践来熟悉其功能。

### 应用并执行插件

在撰写本书的时候，Gradle 插件 Sublime Text (<https://github.com/phidopus/gradle-sublimetext-plugin>) 还处于开发的前期阶段，只提供了很有限的一些功能。其核心功能是生成并且自定义项目文件。作为一个附加组件，它为在 IDE 中运行 Java 项目的 Gradle task 做准备。

下面的清单显示了根项目构建脚本的内容，其中声明了插件并将其添加到 `classpath` 中。因为要为所有的项目生成 Sublime Text 项目文件，所以需要将插件应用到 `allprojects` 配置块中。

#### 清单 10.12 将 Sublime Text 插件应用到构建的所有项目中

```

buildscript {
    repositories {
        maven { url 'http://phildop.us/m2repo' } }
    }
}

```

声明自定义仓库

```
dependencies {
    classpath 'us.phildop:gradle-sublimetext-plugin:0.5'
}

allprojects {
    apply plugin: 'sublimeText'
}
```

定义依赖的插件，使其在脚本的 classpath 下面可用

应用插件到所有项目

是时候生成项目文件了。如果你在根项目下运行 `sublimeText` task，则会得到如下命令行输出：

```
$ gradle sublimeText
:sublimeText
:model:sublimeText
:repository:sublimeText
:web:sublimeText
```

为根项目生成 Sublime Text 项目文件

为子项目生成 Sublime Text 项目文件

在插件的默认设置下，生成的项目文件仅仅包含指向目录路径的指针配置。你可以打开项目并且编辑文件，但是你不能用 **Gradle** 编译代码。接下来，你会做更多的自定义配置。

### 自定义配置

自定义的选项很有限。清单 10.13 显示了如何去除不需要的目录以及设置项目的源目录和 `classpath`。在默认情况下，插件并不做这样的选择，即使是与项目完全不相关的目录。

**清单 10.13 调整 Sublime Text 项目文件的配置**

```
allprojects {
    sublimeText {
        defaultFolderExcludePatterns = ['.gradle', 'build']
        addGradleCompile = true
    }
}

subprojects {
    apply plugin: 'java'

    sublimeText {
        generateSublimeJavaClasspath = true
        generateSublimeJavaSrcpath = true
    }
}
```

去除不想要的文件和目录

添加 compileJava task 到 Sublime 的构建工具支持中

为 Java 项目生成源路径和 classpath

**Gradle** 插件没有定义 `clean` task 去清理已经存在的项目文件。如果你想要清理已经存在的项目文件，则需要再次运行 `sublimeText` task。它会覆盖已经存在的项目文件。现在你可以在 **Sublime Text** 中打开项目了。

## 导入项目

Sublime Text 非常容易下载和安装。因为你正工作在多项目构建中，所以你会期望在用户界面看到整个项目的结构。选择 **Project → Open Project...**，并选择根项目的 `.sublime-project` 文件来导入项目。To Do 应用程序导入后项目结构如图 10.7 所示。选择 **Tools → Build System → Gradle** 来触发 Gradle 构建。在撰写本书的时候，只有一个安装好的 Gradle 运行时环境可用；而不能使用 Gradle Wrapper。

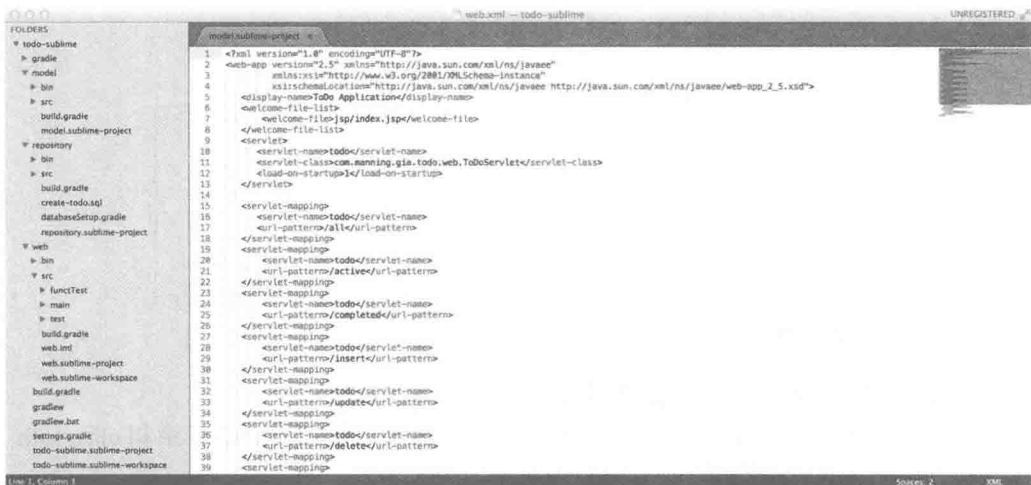


图 10.7 Sublime Text 中导入的项目

为流行的 IDE 和文本编辑器生成项目文件的讨论先告一段落。在下一节中，你会反过来让 IDE 去分析构建脚本，从而生成项目文件。

## 10.2 在流行的IDE中管理Gradle项目

几年前，Gradle 相比于 Ant 和 Maven 来说还是一个新来者。几乎没有 IDE 对 Gradle 项目提供支持。为了能够导入一个 Gradle 项目，你必须用 Eclipse 和 IDEA 插件来生成项目文件。如果你使用的是其他的 IDE（如 NetBeans），几乎就没有工具支持了。

随着 Gradle 的影响力越来越大，这种状况发生了改变。工具提供者认识到对 Gradle 支持的重要性。图 10.8 显示了在构建脚本和 Gradle 运行时环境之间 IDE 如何扮演中间人角色。

你可能会问自己哪一种生成项目文件的方法更好：通过构建生成项目文件，或者让 IDE 分析构建代码。我个人的喜好是让 IDE 去做那些繁重的任务。这通常让

你可以快速开始。如果你需要对配置有更细粒度的控制,则可以引入 Gradle 相应的 IDE 插件支持。当这样做的时候,你应该注意的是不要将 IDE 产生的配置给覆盖了。在实践中,往往要折腾一会儿才能达到期望的结果。

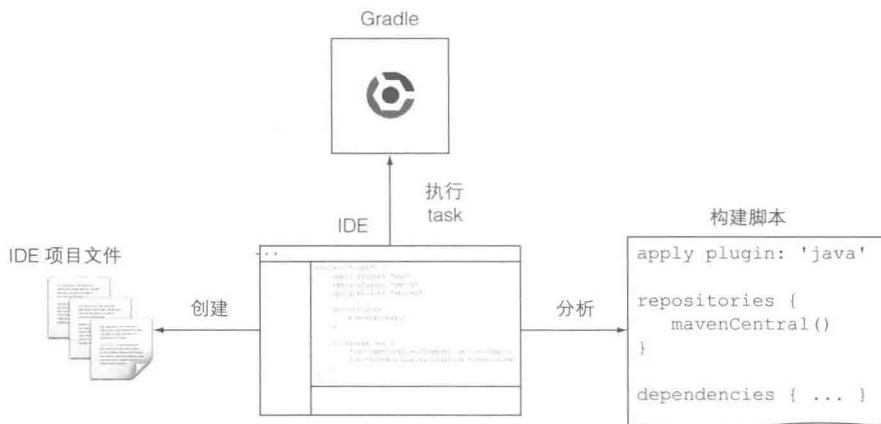


图 10.8 IDE 中内建 Gradle 支持

以下特性对 IDE 用户来说极其重要:

- 通过分析 Gradle 构建脚本打开项目,并且自动设置正确的源目录和 classpath
- 根据已有的构建脚本生成项目文件
- 当构建脚本发生改变的时候同步项目设置(比如添加一个新的外部依赖)
- 在 IDE 用户界面上执行 Gradle task
- 对 Gradle DSL 配置元素提供代码补全和语法高亮显示

在本节中,我们会比较三个不同产品对 Gradle 的支持: SpringSource STS、IntelliJ IDEA 和 NetBeans。你会了解到,它们所支持的性能集和效率会有细小的区别。我们首先来看看 SpringSource STS。

### 10.2.1 SpringSource STS 对 Gradle 的支持

在本章前面你学到了如何为 Eclipse 生成项目文件。利用生成的项目文件,能够将项目导入到 IDE 中。到目前为止,我们还没有讨论过将项目导入到 IDE 中后如何与 Gradle 项目打交道。标准的 Eclipse 分发版并没有对 Gradle 进行集成。你需要手动安装 Groovy 和 Gradle 插件。更多的详细信息,请参考相关的 Eclipse marketplace 页面(如 <http://marketplace.eclipse.org/content/gradle-integration-eclipse>)来了解 Gradle 集成。

如果你可以很容易且方便地使用 Spring Tool Suite (STS) (<http://www.springsource.org/sts>)来管理项目,为什么还要手动安装这些 Gradle 插件呢? STS

是一个基于 Eclipse 的开发环境，主要致力于使用 Spring 框架构建应用程序。其对 Gradle 项目的导入和管理也有非常好的支持。

### 安装带 Gradle 支持的 STS

安装 STS 只需要从项目主页上面下载安装包并运行安装文件。你也可以修改已经存在的 Eclipse 实例来使用 STS 的功能。在 SpringSource 的网页上有相应的安装向导。以下的描述都是基于 STS 3.2.0 版本的。

STS 提供了一个集中的 Dashboard。Dashboard 包括创建新项目、访问教程和文档、安装扩展程序等功能。扩展程序的主要设计思想是为一些常用的语言、框架或者工具提供预配置的 Eclipse 插件。我们主要关注的是安装 Groovy 和 Gradle 的扩展程序。

第一次打开 STS 时，Dashboard 会自动呈现在主面板中。任何时候，你都可以通过菜单 Help → Dashboard 来访问 Dashboard。点击 Extensions 面板，滚动到 Language and Framework Tooling 小节，勾选 Gradle Support and Groovy-Eclipse 旁边的复选框，如图 10.9 所示。然后点击 Dashboard 底部的 Install 按钮开始安装。扩展程序安装成功后，必须要重启 STS 才能生效。重启后，你就可以使用所有的 Gradle 支持了，比如导入多项目构建、管理依赖、DSL 代码补全以及执行 task 等。我们来实际试用下这些特性。

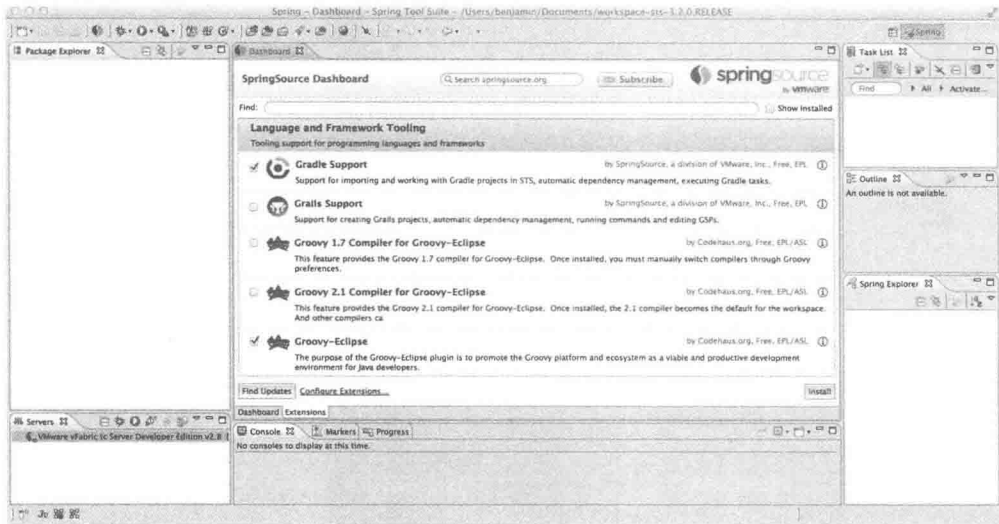


图 10.9 从 Dashboard 安装 Groovy 和 Gradle 插件

### 导入 Gradle 项目

STS 为导入 Gradle 项目提供了向导。这个向导不要求被导入的应用程序包含已

有的项目文件。在导入的过程中，STS 会分析 Gradle 构建脚本，并生成项目文件。你可以通过选择 File 菜单下的 Import... 进入向导，如图 10.10 所示。在过滤输入域中输入“gradle”表明你想要导入一个 Gradle 项目。



图 10.10 Gradle 项目导入向导

点击 Next 按钮，将显示如图 10.11 所示的对话框。在这个对话框中，你可以选择 To Do 项目的根目录，并点击 Build Model 按钮，从而生成 STS 项目结构。



图 10.11 导入多项目 Gradle 构建





## 10.2.2 IntelliJ IDEA 对 Gradle 的支持

从 IntelliJ 12.1 开始,就有了对 Gradle 的支持。在预安装好的 Gradle 插件的帮助下,你可以通过指向 Gradle 构建脚本来导入 Gradle 项目,生成 IDE 元数据,直接在 IDE 中运行 Gradle task,以及在编辑器中使用 DSL 代码补全功能。我们通过导入 To Do 应用程序来看看 IntelliJ 对 Gradle 的支持。

### 导入 Gradle 项目

IntelliJ 知道如何显示和管理多项目应用程序的结构。这相对于 Eclipse 来说是一个很大的优势,因为项目的物理结构可以在 IDE 中直接反映出来。你所需要做的就是打开 IntelliJ,在菜单栏中选择 File → Import Project...,并选择根项目的 Gradle 构建脚本,就可以导入一个多项目应用程序,如图 10.13 所示。

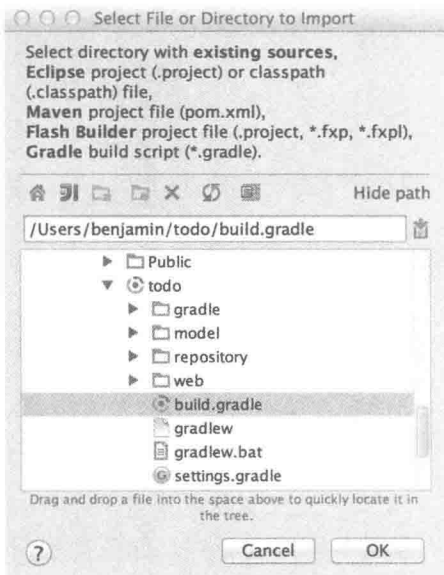


图 10.13 选择 Gradle 构建脚本导入

点击 OK 按钮会弹出下一个对话框,如图 10.14 所示。在这个对话框中,你可以选择 Gradle 运行时环境和一些其他的配置参数,比如 Gradle 安装目录。如果被导入的项目使用了 Gradle wrapper,那么 IntelliJ 会自动识别到。你不但可以在命令行下使用 Gradle wrapper,而且在 IDE 中也能使用 Gradle wrapper,从而确保项目组所有人使用一致的运行时环境。点击 Next 按钮继续。

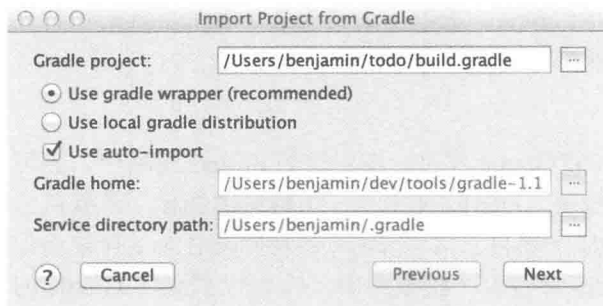


图 10.14 Gradle 运行时配置

图 10.15 显示的是关于生成项目文件的一些细节设置。在对话框的左边，你可以看到 IntelliJ 将要生成的项目结构。在对话框的右边，你可以改变项目的重要设置。对话框中的初始值都是根据构建脚本的配置生成的。如果你决定修改项目文件的位置或者改变 JDK，则可以在这里修改。保持默认设置，点击 Finish 按钮让 IntelliJ 为你生成项目文件。

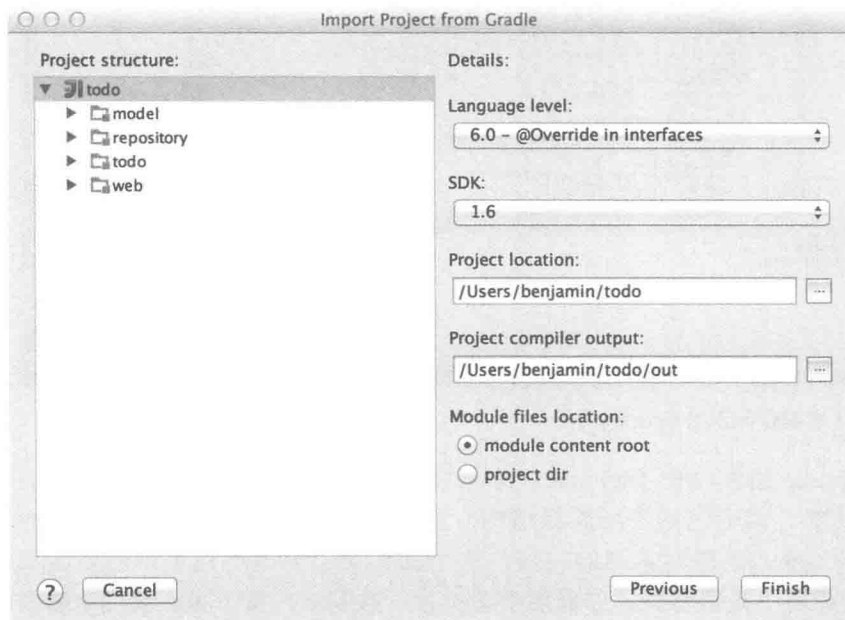


图 10.15 设置项目详细信息

生成的项目文件的格式可能会让你感到惊讶。仔细看看项目结构，你会发现 IntelliJ 创建了一个叫作 `.idea` 的目录，用来存储 IDE 元数据。这个项目文件格式与我们在 10.1.1 节中提到的项目文件格式不同。较新版本的 IntelliJ 相对于基于文件

格式的项目文件（.ipr、.iml、.iws）而言，更倾向于使用基于目录格式的项目文件（.idea）。导入应用程序后，你就可以尽情地写代码了。

### 使用 Gradle 支持

图 10.16 显示了正在操作所导入的 Gradle 项目。在左边的 Project 窗格中，你可以看到项目的层次结构。为了表明模块，IntelliJ 使用了一个特殊的图标（带蓝色方块的文件夹图标）来标识每个 Gradle 子项目。以 .gradle 为扩展名的文件被当作 Gradle 构建脚本，并且用 Gradle 图标来标识。在截图中，你可以看到对 Gradle 的 DSL 代码补全功能。在输入过程中，如果 IntelliJ 能识别出 DSL 关键字，一个上下文菜单就会弹出来并且提供一些可选的配置元素供选择。你也可以使用 Ctrl + 空格键来激活代码补全功能。

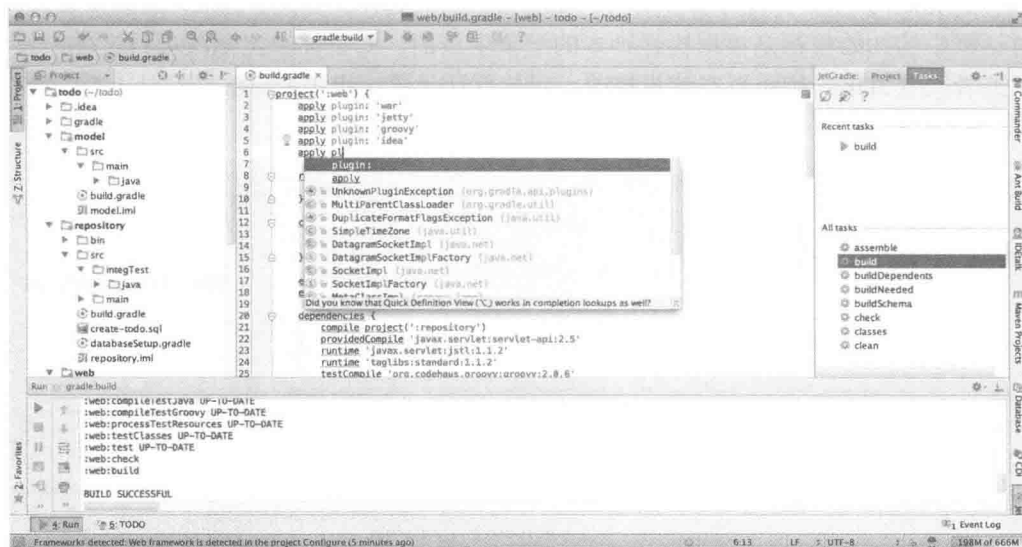


图 10.16 在 IntelliJ 中操作导入的 Gradle 项目

IntelliJ 的 Gradle 插件叫作 JetGradle。你可以通过编辑器右边的标签面板来打开 JetGradle 的功能。其中包括分析项目结构，显示依赖，下载依赖，在 IDE 中直接执行 Gradle 的 task。在撰写本书的时候，这个插件对于修改构建脚本后新定义的依赖关系的处理还不是很智能。你需要手动执行一次刷新。接下来，我们来看看 NetBeans 的 Gradle 支持。

### 10.2.3 NetBeans 对 Gradle 的支持

NetBeans IDE (<https://netbeans.org/>) 是三个最流行的 Java 世界 IDE 其中之一。

NetBeans 本身就支持对 Java 和 Groovy 应用程序的实现，它具有一个优秀的 IDE 应该拥有的功能，比如重构、代码补全、对流行框架或工具的集成。NetBeans 的功能可以以插件的形式被扩展。

### 安装带 Gradle 支持的 NetBeans

安装 NetBeans 非常简单。下载对应你的操作系统的分发包（撰写此书时，版本号是 7.3）并且运行安装文件。几分钟之后，IDE 就被安装好了，你就可以添加 Gradle 支持了。

要安装 Gradle 支持，你需要从 <http://plugins.netbeans.org/plugin/44510/gradle-support> 下载一个第三方插件。你可以把插件放在文件系统的任何位置。在 NetBeans 中，从菜单栏中选择 Tools → Plugins，会弹出一个对话框来管理插件。在 Downloaded 标签中，点击 Add Plugins... 按钮，选择你所下载的 Gradle 插件文件，如图 10.17 所示。

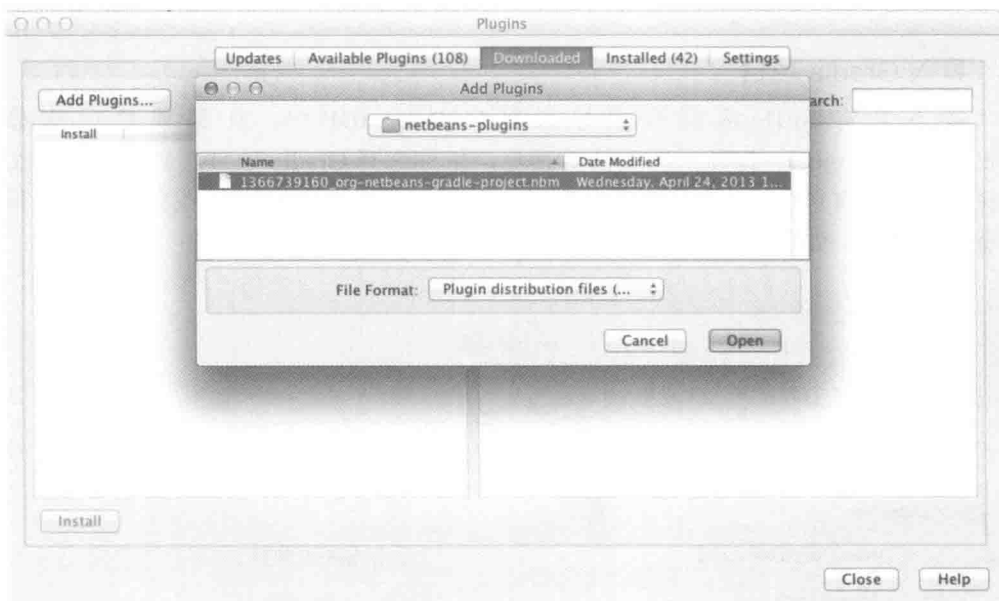


图 10.17 添加所下载的 Gradle 插件

点击 Open 按钮后，插件的详细信息就展示出来了。在图 10.18 中，你可以看到将要安装的插件版本、插件的来源和一段对其功能的描述。勾选插件名字前面的复选框，然后点击 Install 按钮开始安装。成功安装后，NetBeans 需要重启才能生效。

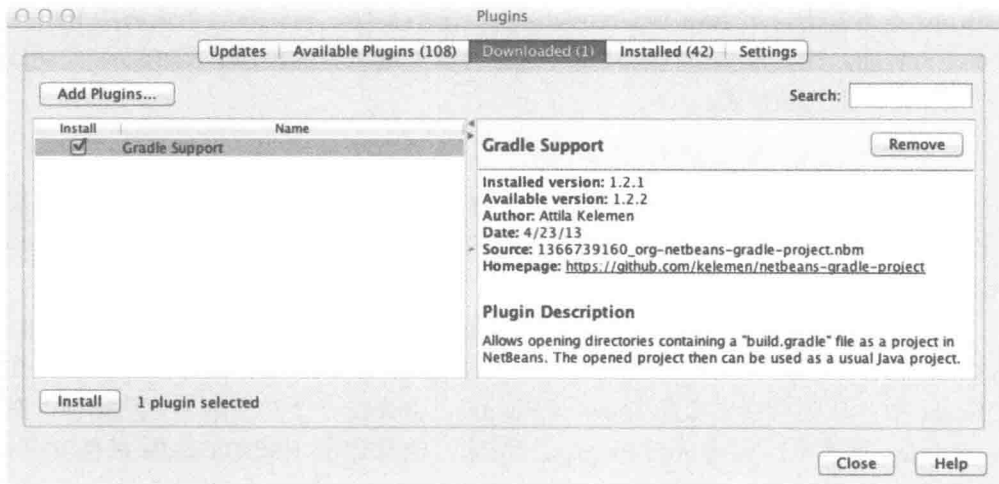


图 10.18 安装 Gradle 插件

### 导入 Gradle 项目

接下来将 To Do 应用程序导入到 NetBeans IDE 中。在菜单栏中选择 File → Open Project，打开一个新的对话框，供你选择 Gradle 项目。显示出来的文件浏览器为所有包含 Gradle 构建脚本的文件夹显示了 Gradle 图标。找到 To Do 项目并选择根目录，如图 10.19 所示。

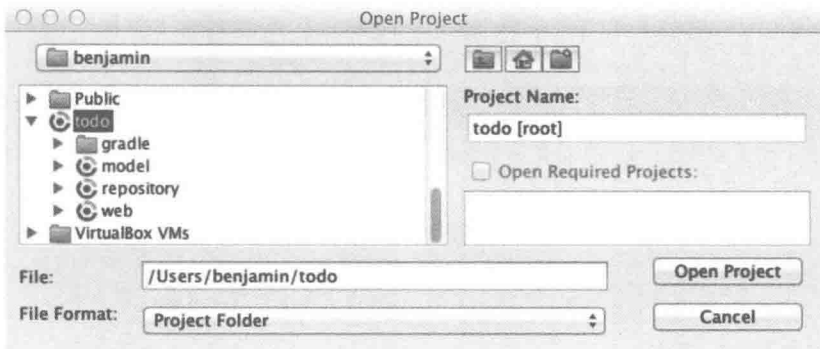


图 10.19 导入 Gradle 项目

点击 Open Project 按钮后，项目就被导入了。接下来你就可以在 IDE 中工作了。

### 使用 Gradle 支持

项目刚被打开的时候显示在左边的 Projects 视图中。你只能看到根项目。所有的子项目都排列在树节点下。为了方便，展开子项目节点并且点击每个子项目的名字。这样子项目会被自动添加到 Projects 视图的顶层节点，如图 10.20 所示。

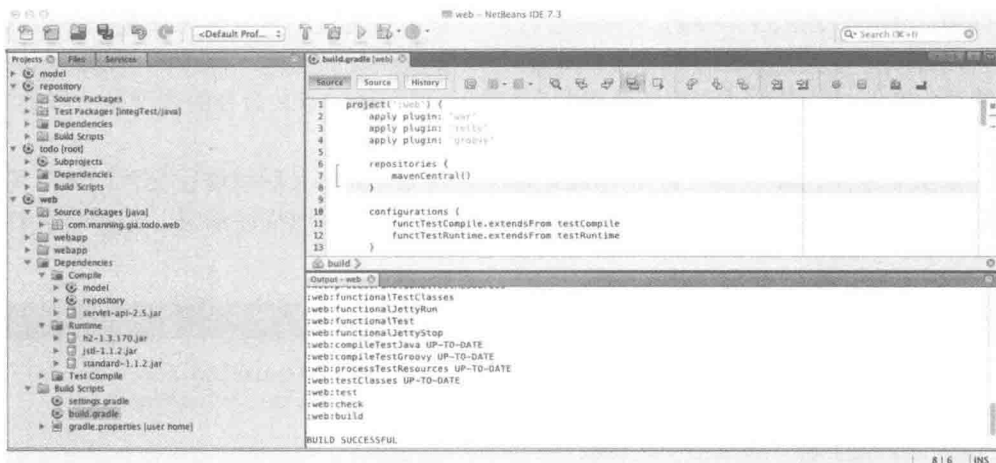


图 10.20 在 NetBeans IDE 中操作导入的 Gradle 项目

点击项目节点，会弹出一个上下文菜单，从列表中选择一个 task，从而执行 Gradle task。每个项目节点都分成 Source 和 Test 包、依赖和构建脚本。如果希望将 Gradle 元素进行逻辑分组的话，这个功能就非常棒了。如果需要修改一个单独的文件，你可以在任何时候切换到 Files 标签。在截图中，你可以在编辑器中看到打开的 Gradle 构建脚本。遗憾的是，Gradle 插件不支持对 DSL 代码的补全功能。

以上就是在 IDE 中对 Gradle 支持的相关讨论。接下来，我们会看看 Gradle 的工具 API，这个 API 使得对于所有的 IDE 提供流畅的 Gradle 集成成为可能。

## 10.3 使用工具API集成Gradle

工具 API 的主要作用是使应用程序集成内嵌的 Gradle。虽然你不一定知道工具 API 的存在，但是你却已经在使用它了，因为在 IDE 中对 Gradle 的支持都离不开它。使用工具 API 有利于实现如下三个目标：

- 使用特定版本的 Gradle 执行构建（使用 wrapper 或者安装的 Gradle 运行时环境）。
- 查询构建的运行时信息和构建内建数据模型（比如 task、依赖和命令行输出）。
- 监听构建事件，并做出响应。比如依赖配置在 IDE 中发生改变时，能够自动下载外部依赖。

大多数程序员并不是在开发工具——我们通常是在开发企业级软件或者开源应用程序的时候使用 Gradle 作为构建系统。我们为什么要关心工具 API 呢？工具 API 的作用非常大，并且可以被应用到很多其他的应用场景中。我们通过例子来理解一下。在下面内容中，你会使用工具 API 来为构建脚本做集成测试。为了验证其正确

的行为，你可以通过工具 API 来查询运行时信息。当使用工具 API 来做集成测试的时候，你没必要使用 ProjectBuilder 类来模拟项目的创建。但你需要从终端用户的角度来测试构建脚本，它应该跟你在控制台运行 gradle 命令的表现是一样的。我们来看个例子。

在第 8 章中，你开发了很多 Gradle task 来与 CloudBees 后端进行交互。如下清单显示的 task 能够获取一个部署在 RUN@cloud 上面的应用程序信息，然后将其打印在命令行终端。

#### 清单 10.14 CloudBees task 获取应用程序信息

```
task cloudBeesAppInfo(description: 'Returns the basic information about an
                                application.', group: 'CloudBees') {
    inputs.property('apiKey', apiKey)
    inputs.property('secret', secret)
    inputs.property('appId', appId)

    doLast {
        BeesClient client = new BeesClient(apiUrl, apiKey, secret, 'xml',
                                           '1.0')

        ApplicationInfo info

        try {
            info = client.applicationInfo(appId)
        }
        catch(Exception e) {
            throw new GradleException(e.message)
        }

        logger.quiet "Application id : $info.id"
        logger.quiet "                title : $info.title"
        logger.quiet "                created : $info.created"
        logger.quiet "                urls : $info.urls"
        logger.quiet "                status : $info.status"
    }
}
```

命令行终端打出的应用程序信息

在真实环境下，这个 task 很难被全面测试，因为它用到了 gradle.properties 文件中提供的在命令行中使用的一些属性。我们来看看通过使用构建脚本工具 API 是如何验证这个 task 的正确行为的。

首先，设置好目录结构。将 CloudBees task 放到一个普通的 Gradle 构建脚本中，并将这个构建脚本放到 script-under-test 目录下。将测试代码放在同一层级的 tooling-api-integ-test 目录下。这个目录下还包含一个构建脚本，并且已经配置好了 Spock 作为测试框架，并且引入了工具 API 的相应依赖。测试类实现叫作 CloudBeesSpec.groovy。最后的结构如下：

```
- script-under-test
  └─ build.gradle
- tooling-api-integ-test
  └─ build.gradle
    └─ src
      └─ test
```

包含从 CloudBees 获取应用信息 task 的构建脚本

设置集成测试依赖





我们来快速浏览一下清单 10.5 中所示的集成测试的构建脚本。因为你正在使用 Spock 来编写测试，所以需要应用 Groovy 插件。因为你只是在编写测试，所以所有的依赖都使用 testCompile 配置。

### 清单 10.15 准备集成测试

```

apply plugin: 'groovy'

repositories {
    mavenCentral()
}

dependencies {
    testCompile localGroovy()
    testCompile gradleApi()
    testCompile 'org.spockframework:spock-core:0.7-groovy-1.8'
}
  
```

添加对 Gradle API 的依赖

定义集成测试要用到的 Spock

清单 10.6 显示了测试类 CloudBeesSpec.groovy 的内容。里面只定义了一个测试方法，使用 Gradle 的工具 API 来执行 CloudBees 的 cloudBeesAppsInfo task。你不只是想要运行这个 task，而且还想要确保这个 task 如预期一样的工作。这个 task 只有命令行输出，所以你需要去解析标准的输出流的内容，从而确认打印出了正确的信息。因为 Gradle 的 task 不允许为 logger 实例创建一个 mock 对象，所以我们能在单元测试中对这个功能进行测试。

### 清单 10.16 使用工具 API 进行集成测试

```

package com.manning.gia

import org.gradle.tooling.*
import spock.lang.Specification

class CloudBeesSpec extends Specification {
    static final String GRADLE_VERSION = '1.5'
    static final File PROJECT_DIR = new File('../script-under-test')

    def "CloudBees application information is rendered on the command line"() {
        given:
            String[] tasks = ['cloudBeesAppInfo'] as String[]
            String arguments = '-PappId=gradle-in-action/todo'
        when:
            ByteArrayOutputStream stream = executeWithGradleConnector(
                GRADLE_VERSION, PROJECT_DIR, tasks, arguments)
            String output = stream.toString('UTF-8')
    }
}
  
```

指向测试脚本的项目目录

使用 Gradle  
connector  
执行测试脚  
本并解析其  
命令行输出

```

        then:
            output != null
            output.contains('Application id : gradle-in-action/todo')
            output.contains('title : todo')
            output.contains('urls : [todo.gradle-in-action.cloudbees.net]')
        }

private ByteArrayOutputStream executeWithGradleConnector(String
    gradleVersion, File projectDir,
    String[] tasks, String arguments) {
    GradleConnector connector = GradleConnector.newConnector()
    ProjectConnection connection

    try {
        connection = connector.useGradleVersion(gradleVersion)
            .forProjectDirectory(projectDir).connect()
        BuildLauncher buildLauncher = connection.newBuild()
        buildLauncher.forTasks(tasks).withArguments(arguments)
        ByteArrayOutputStream stream = new ByteArrayOutputStream()
        buildLauncher.setStandardOutput(stream).run()
        return stream
    } finally {
        connection.close()
    }
}

```

断言 task 输出

创建 Gradle-Connector 实例

为构建脚本设置目标 task、参数和输出流

设置 Gradle 版本和项目目录

创建新的 BuildLauncher 实例

关闭 GradleConnector 连接

这个测试方法的目的应该是很明确的。我们来分析如何在测试类 `CloudBeesSpec` 中使用 `Gradle` 的工具 API。如清单所示，你定义了一个 `withGradleConnector` 方法，繁重的工作都在这个方法里面做了。工具 API 的使用分为 5 步：

- 1 创建一个 `GradleConnector` 实例——调用构建脚本的主要入口。
- 2 配置 `GradleConnector` 实例，包括设置所期望的 `Gradle` 版本、安装路径、包含构建脚本的项目目录。在默认情况下，会自动下载并使用 `wrapper`。
- 3 调用 `connect()` 方法来连接目标构建脚本。
- 4 提供构建参数，如 `task`、命令行参数，以及你想要写到的输出流等。
- 5 构建脚本中的 `task` 被执行后，应该调用 `close()` 方法来关闭连接，从而释放资源。

上述就是对工具 API 的简单描述。还有更多的选项可以去尝试下，但是这个例子应该可以给你一个关于工具 API 的强大功能的大概印象。你可以在 <http://www.gradle.org/docs/current/userguide/embedding.html> 中找到更多的关于使用模式和配置选项信息。

## 10.4 总结

Gradle 的主要运行时环境是命令行。这个特性使得构建便捷，很多开发员在可视化编辑器或者 IDE 中是最高效的。在命令行工具和编辑器之间切换可能会使开发过程变慢。

Gradle 不会让你束手无策。借助一些插件，你可以为很多 IDE 和编辑器生成项目文件。所生成的项目文件包含一些基础信息，从而使你能够方便地在正确设置的 IDE 中直接打开项目。我们在 To Do 应用程序上下文中讨论了如何使用这些插件来为 Eclipse、IntelliJ IDEA 和 Sublime Text 生成项目文件。这些基础信息通常是根据默认的构建脚本配置生成的，但是也可以根据不同的需要被自定义。

流行的 Java IDE 供应商认识到了对 Gradle 支持的必要性。IDE 如 SpringSource STS、IntelliJ IDEA 和 NetBeans 提供了实用的工具，使你能够通过指向一个构建脚本就能轻松地打开 Gradle 项目。项目被打开后，开发人员就能够管理依赖，使用 DSL 代码补全功能，并且在 IDE 中执行 Gradle task 了。

工具 API 为 IDE 集成 Gradle 提供了基础。它允许执行 Gradle task，同时监控或者查询正在运行的构建。在真实的环境中，工具 API 可以被用来为 Grsdle 构建脚本做集成测试。



# 11 构建多语言项目

---

## 本章涵盖

- 使用 Gradle 管理 JavaScript
- 构建一个包含 Java、Groovy 和 Scala 的 Gradle 项目
- 探讨支持其他语言的插件

近年来，人们对多语言编程本身及其在现实世界中的应用，以及在任务主导型软件领域应用等兴趣十足。多语言编程（polyglot programming）这个术语被用来描述那些包含多种编程语言的项目。这种现象在 Java 世界中尤其常见。对于 Web 开发人员，没有 JavaScript、相关的框架、类库和 CSS 世界将不能正常运转。后端代码或者桌面应用程序不再仅仅使用 Java 一种编程语言，其他基于 JVM 的编程语言如 Groovy 或者 Scala 也被混合进来了。只需要为特定的工作选择合适的工具即可。在本章中，我们会以 To Do 应用程序为例，讨论 Gradle 是如何面对组织和构建多语言项目的挑战的。

作为 Java 程序员，我们被扩展性工具和一些自动化构建支持给宠坏了。像外部类库的构建时依赖管理和对于项目结构约定优于配置等概念已经存在数年，并且已经是我们的日常词汇中的一部分了。然而，在 JavaScript 世界中，被广泛接受的

模式和自动化构建技术还处于比较早的开发时期。Gradle 并没有提供对 JavaScript 和 CSS 的完美支持，但是我们会讨论如何利用 API 来实现典型的应用场景，比如 JavaScript 文件的组合和压缩。

在前面的章节中，我们主要关注于对 Java 项目的构建。包含有多种编程语言的软件栈在很多企业或组织中都存在。基于 JVM 的编程语言都能很好地被 Gradle 支持。我们会详细看看利用核心的语言插件构建 Java、Groovy 和 Scala 代码的过程。在这个过程中，我们会接触到像编译器守护进程和联合编译等话题。

此外，很多企业需要对使用非 JVM 编程语言编写的软件进行集成，如 C 或者 C++。本章会对如何将异构的软件基础设施融入到构建中进行概述。我们从 Web 项目构建开始。

## 11.1 使用Gradle管理JavaScript

丰富的用户体验成为了应用程序成功与否的关键。这同样适用于桌面、移动和 Web 应用程序。如今，用户要求 Web 应用程序与原生应用程序有相同的外观和体验。用户满足于静态页面和同步的服务器通信的时代已经过去了。

JavaScript 成为了能够满足这一需求的最流行的编程语言。程序员不再需要依赖纯 JavaScript 来实现功能。很多应用程序开发框架和类库（如 jQuery、Prototype 和 Backbone，仅列举了一些）都可用于简化 JavaScript 开发，让这种语言的引入边界更加平滑。

JavaScript 不止用于客户端运行时环境。像 Node.js (<http://nodejs.org>) 这样的平台也利用 JavaScript 来实现服务器端应用程序。这使得 JavaScript 相对来说几乎无处不在。

### 11.1.1 处理 JavaScript 的典型 task

说到处理 JavaScript，你可能想到的只是日常工作流中的几个 task。实际上，其中一些听起来应该很熟悉，因为我们在 Java Web 应用程序上下文中已经讨论过了：

- **压缩**：JavaScript 通常都作为外部文件被包含在 HTML 页面中。这意味着在它被用来渲染页面之前被下载到浏览器中。文件越小，页面加载速度就越快。压缩的主要目标就是减小原始 JavaScript 文件的大小，通常通过应用一些智能优化来达到，比如移除注释、空白字符和制表符等。Google Closure compiler (<https://developers.google.com/closure/compiler>) 就是这样一个工具来帮助你对 JavaScript 文件做压缩。

- 合并：JavaScript 文件的大小并不是影响页面加载性能的唯一因素。还有一个因素就是请求页面中的所有 JavaScript 文件需要向服务器发送的 HTTP 请求的数量。所请求的外部 JavaScript 文件越多，用户等待的时间就越长。将多个 JavaScript 文件合并为一个就能解决这个问题。
- 测试：与为服务器端代码编写测试代码类似，你同样想要验证 JavaScript 所实现的功能的行为是正确的。测试框架如 Jasmine (<http://pivotal.github.io/jasmine>) 支持使用 JavaScript 编写测试代码。
- 代码分析：通过分析 JavaScript 源代码可以找到一些潜在的 bug 和问题、结构性问题和错误的代码规范。JSLint (<http://www.jshint.com/lint.html>) 就是一个用 JavaScript 编写的，旨在以自动化的风格暴露这些问题的工具。
- 翻译型编译：一些客户端编程语言如 CoffeeScript 或 Dart 提供了它们自己的语法和语言构造。遗憾的是，这些语言并不能在浏览器中直接运行。翻译型编译，一种特殊的编译过程，将源代码翻译成 JavaScript，从而使其能够在目标环境中运行。

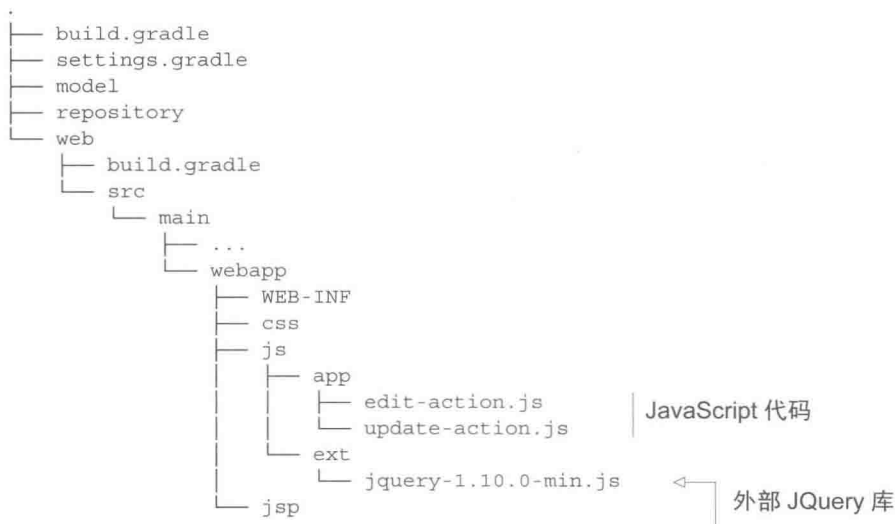
如果你尝试着在 To Do 程序中使用上述的 task，但是没有合适的自动化工具的话，这些步骤在生成 WAR 文件的时候需要手动并且重复地运行。与在第 1 章中提到的原因一样，你应该在所有场景下都避免发生这种情况。在本章中，我们不会覆盖到上述所有的 task，但是你会学到如何使用 Gradle 使其自动化。

### 11.1.2 在 To Do 应用程序中使用 JavaScript

假设你想升级 To Do 应用程序。替代之前的每个操作都要向服务器提交数据并且重新渲染页面的方案，你想要通过异步的 JavaScript 调用 (AJAX) 与服务器通信。如果 HTML 用户界面需要发生改变，你可以直接修改文档对象模型 (DOM)。结果就是，To Do 应用程序会像桌面应用程序一样，不需要每次都重新加载页面。

为了使 JavaScript 的使用变得简单，你的应用程序会使用一个外部的 JavaScript 库。在众多可用的库中，我们选择 jQuery (<http://jquery.com/>)，一个成熟的、功能丰富的、为处理 AJAX 调用和操作 HTML DOM 提供了简单易用的 API 的库。

在 JQuery 的帮助下，你会实现更新一个已有的 To Do 项的名字的功能。这个功能需要两个操作。双击 To Do 项的名字，会提供一个输入框使 To Do 项的名字变成可编辑状态。在编辑状态下按 Enter 键，会将修改后的 To Do 项的名字发送到服务器，并在数据存储里面更新对应的值，然后退出编辑模式。这两个操作是在 `edit-action.js` 和 `update-action.js` 这两个 JavaScript 文件中实现的。我们在这里不会讨论这两个文件的内容。感兴趣的话可以看看本书源码示例中的实现。下面的目录结构显示了这两个 JavaScript 文件，以及 web 项目中被压缩的 JQuery 库：



现在，你只需要手动下载 JQuery 库并将其放在 `src/main/webapp/js/ext` 目录下。很多项目都是这么做的。还记得我们在第 5 章中讨论的对于外部 Java 库的依赖管理吗？对于 JavaScript 文件的依赖，你可以应用同样的概念。我们来看看怎么做。

### 11.1.3 对 JavaScript 库依赖管理

在过去，JavaScript 程序员并不关心依赖管理。我们通常把外部库直接与程序源代码放在一块儿。随着代码库的增长，渐渐的我们就搞不清楚项目所依赖的 JavaScript 库的版本了。如果幸运的话，从 JavaScript 库的文件名上可能会看出来。

如果你单独使用 JavaScript，则可以使用 `node package manager (NPM)` 来解决问题。JavaScript 文件之间的依赖可以使用 `RequireJS` (<http://requirejs.org/>) 来管理。但是如果你想要使用 Gradle 来管理 JavaScript 依赖呢？没问题；你可以使用标准的 Gradle 依赖声明和管理方式。

然而，使用这种方式我们依然会面临一些挑战。因为作为 JVM 世界编程的程序员，我们通常对所有想要使用的库都拥有访问权限。我们只需要将构建脚本指向 `Maven Central` 就好了。对于 JavaScript 库，没有建立比较好的这样的托管服务。其中一个收集了一些常用的开源的 JavaScript 库的托管提供者是 `Google Hosted Libraries` (<https://developers.google.com/speed/libraries>)。替代方案是，如果这个托管仓库中提供了某个库的特定版本，你也可以直接使用。

因为 JavaScript 托管服务的不足，所以你不能期望所有的依赖库都有统一的描



述符格式，如 Maven 仓库的 pom.xml 文件。这意味着具有传递性的 JavaScript 库、CSS 甚至是图片等都需要分开声明。

我们来看看如何从 JQuery 的下载页面来获取库的压缩版本。在这里你会看到熟悉的模式。首先，需要自定义一个配置项，然后在 dependencies 配置块中声明带有具体版本的库。为了将库下载到源代码路径中，创建一个 Copy 类型的 task。下面的清单演示了如何将这个 task 与项目中其他重要的 task 配合使用，从而确保在部署到 Jetty 或者打包成 WAR 文件之前正确地获取了所需要的 JavaScript 文件。

清单 11.1 对 JavaScript 库的依赖管理

```
repositories {
    ivy {
        name 'JQuery'
        url 'http://code.jquery.com'
        layout 'pattern', {
            artifact '[module]-[revision]([classifier]).[ext]'
        }
    }
}

configurations {
    jquery
}

dependencies {
    jquery group: 'jquery', name: 'jquery', version: '1.10.0',
           classifier: 'min', ext: 'js'
}

task fetchExternalJs(type: Copy) {
    from configurations.jquery
    into "$webAppDir/js/ext"
}

[jettyRun, jettyRunWar, war]*.dependsOn fetchExternalJs
```

将 JQuery 的 URL 配置成 Ivy 仓库

自定义 JavaScript 依赖

将 JQuery 库声明为依赖

用于下载 JavaScript 依赖到 src/main/webapp/js/ext 目录的 task

需要 JavaScript 库依赖的 task 会先下载依赖

在 Gradle 项目的根目录下运行这个 task：

```
$ gradle :web:fetchExternalJs
:web:fetchExternalJs
Download http://code.jquery.com/jquery-1.10.0.min.js
```

与预期一样，JavaScript 库被下载到指定的 src/main/webapp/js/ext 目录中了。你现在可以直接在 JSP 页面或者 HTML 文件中引用这个 JavaScript 库了。在第一次下载后，fetchExternalJs task 就会自动知道它不需要再被运行了：

```
$ gradle :web:fetchExternalJs
:web:fetchExternalJs UP-TO-DATE
```

现在你的应用程序源目录下已经包含了所有所需要的 JavaScript 文件了。你可以通过运行 `gradle :web:jettyRun` 来检验一下，你会看到应用程序已经完全能工作了。

你可能会问，为什么所下载的 JavaScript 文件被直接放到了项目源路径下？问得好！通常，你不希望将有版本控制的文件和下载的依赖混到一起。这样做是为了支持 Jetty 插件通过使用 `jettyRun` task 对已有的 Web 应用程序的部署。遗憾的是，这个插件并不允许设置另外一个 Web 应用程序路径。因此，jQuery 的 JavaScript 文件需要被放到指定的 Web 应用程序位置。有很多方法可以克服这个不足。比如，你可以不对这个文件做版本控制（请参考你所使用的 VCS 文档）。

这些都是为 JavaScript 代码实现更多的自动化 task 做铺垫。在接下来的两节中，我们会聊聊如何利用标准的 Gradle 工具来实现自己的 JavaScript 自动化构建。通常，我们会看看如何实现压缩 JavaScript 文件和如何生成代码质量度量报告。

### 11.1.4 利用第三方 Ant task 合并和压缩 JavaScript

有两种很有效的优化方式，每个大 JavaScript 项目都应该推崇：将多个 JavaScript 文件合并成一个文件，并对其内容进行压缩。Google Closure Compiler ([https:// developers.google.com/closure/compiler/](https://developers.google.com/closure/compiler/)) 工作对这两种操作提供了支持。它的一个分发库是 Java 库，能被直接包含在 Gradle 构建中，如下面的清单所示。

清单 11.2 声明对 Google Closure Compiler 库的依赖

```
repositories {  
    mavenCentral()  
}  
  
configurations {  
    googleClosure  
}  
  
dependencies {  
    googleClosure 'com.google.javascript:closure-compiler:v20130603'  
}
```

自定义配置 Google Closure Compiler 库

声明 Google Closure Compiler 依赖

除了纯 Java API 类以外，这个库还包括一个封装好的 Ant task 来提供所支持的功能。在 Gradle 构建中采用哪种方式完全根据个人喜好——两种方式都可以。在接下来的例子中，你将使用 Ant task，因为其很方便地封装了 API。下面的清单演示了一个用 Groovy 编写的自定义 task，并放到项目的 `buildSrc/src/main/groovy` 目录下。如果你想要复习一下如何在 Gradle 中使用 Ant task，请参考第 9 章。

清单 11.3 调用 Google Closure Compiler 的 Ant task

```

package com.manning.gia.js

import org.gradle.api.DefaultTask
import org.gradle.api.file.FileCollection
import org.gradle.api.tasks.InputFiles
import org.gradle.api.tasks.OutputFile
import org.gradle.api.tasks.TaskAction

class GoogleClosureMinifier extends DefaultTask {
    @InputFiles
    FileCollection inputFiles

    @OutputFile
    File outputFile

    @TaskAction
    void minify() {
        ant.taskdef(name: 'jscomp', classname:
            ➡ 'com.google.javascript.jscomp.ant.CompileTask', classpath:
            ➡ project.configurations.googleClosure.asPath)

        ant.jscomp(compilationLevel: 'simple', warning: 'verbose', debug:
            ➡ 'false', output: outputFile.canonicalPath) {
            inputFiles.each { inputFile ->
                ant.sources(dir: inputFile.parent) {
                    ant.file(name: inputFile.name)
                }
            }
        }
    }
}

```

在依赖中声明的 Ant task

Ant task 的用法

声明输入文件

为了使这个自定义的 task 能工作，增强的 GoogleClosureMinifier 类型的 task 需要指定一系列 JavaScript 文件作为输入，并且指定目标输出文件，其中会包含被合并和压缩的 JavaScript 代码。下面的清单使用 fileTree 方法来获得应用程序的 JavaScript 文件作为上述 task 的输入。输出文件被定义为 all-min.js，并且被放在 build/js 目录下。

清单 11.4：使用自定义 task 来压缩 JavaScript

```

import com.manning.gia.js.GoogleClosureMinifier

ext {
    jsSourceDir = "$webAppDir/js/app"
    jsOutputDir = "$buildDir/js"
}

def jsSourceFiles = fileTree(jsSourceDir).include('*.js')

task minifyJs(type: GoogleClosureMinifier) {
    inputFiles = jsSourceFiles
    outputFile = file("$jsOutputDir/all-min.js")
}

```

声明给定目录下的所有 JavaScript 文件

设置输入 JavaScript 文件

设置压缩的 JavaScript 输出文件

执行 minifyJS task，首先会在 buildSrc 下编译压缩自定义 task，然后会生成一个优化的 JavaScript 文件。

```
$ gradle :web:minifyJs
:buildSrc:compileJava UP-TO-DATE
:buildSrc:compileGroovy
:buildSrc:processResources UP-TO-DATE
:buildSrc:classes
:buildSrc:jar
:buildSrc:assemble
:buildSrc:compileTestJava UP-TO-DATE
:buildSrc:compileTestGroovy UP-TO-DATE
:buildSrc:processTestResources UP-TO-DATE
:buildSrc:testClasses UP-TO-DATE
:buildSrc:test
:buildSrc:check
:buildSrc:build
:web:minifyJs
```

能够得到一个优化的 JavaScript 文件虽然很好，但是现在你必须要在动态或者静态的页面文件中修改原来对 JavaScript 文件的引用。当然，这个过程应该是完全自动化并且集成在开发过程中的。在下一节中，我们会讨论一种实现的方法。

### 11.1.5 将 JavaScript 优化作为开发工作流的一部分

在产品环境中，性能非常重要。这使得大多数组织采取部署经过优化的 JavaScript 文件。不好的地方就是这个只有一行，高度优化的 JavaScript 文件毫无可读性并且不利于调试或者诊断问题。因此，你不想打包和运行只有压缩文件的应用程序。在开发或测试环境中，你还是想要使用原始的 JavaScript 文件。很显然，在构建的时候，你需要能够控制和选择是否对 JavaScript 文件进行优化，以及是否使用优化的 JavaScript 文件。

有很多方法可以解决这个问题。在这里我们只讨论其中一种。对项目来说，我们会讨论一个新的项目属性叫作 jsOptimized。如果在命令行提供了这个属性，就表示需要做优化。如果没有提供这个属性，就使用原始的 JavaScript 文件。这里有个使用的例子：`gradle :web:war -PjsOptimize`。

Gradle 已经足够灵活地来支持这个功能。基于这个优化标识，你可以配置 war task 来触发压缩，并去除原始的 JavaScript 文件，引入压缩后的文件，并修改 JSP 文件中对它的引用。下面的清单显示了在 To Do 应用程序中如何做。

#### 清单 11.5 条件性地打包以及使用优化的 JavaScript 文件

```
ext.jsOptimize = project.hasProperty('jsOptimize')
```

```
war {
    if(jsOptimize) {
        dependsOn minifyJs
```

← 只在通过属性要求的时候才优化 JavaScript

```

exclude 'js/app/*'

from(jsOutputDir) {
    into 'js/app'
    include 'all-min.js'
}

exclude 'jsp/app-js.jsp'

from("$webAppDir/jsp") {
    include 'todo-list.jsp'
    into 'jsp'
    filter { String line ->
        if(line.contains('<c:import url="app-js.jsp"/>')) {
            return '<c:import url="app-js-min.jsp"/>'
        }
        line
    }
}
}
}

```

去除原来的 JavaScript 文件；  
包含优化后的

去除应用了原始的  
JavaScript 文件的  
JSP 文件

用 todo-list.jsp  
文件的压缩版  
本 替 换 JSP  
引用文件

这个配置最有趣的地方是 `todo-list.jsp` 文件中关于 JavaScript 引用的替换。为了简单起见，创建了两个新的 JSP 文件，分别引用原始的 JavaScript 文件和压缩文件。在构建的时候，如果你决定使用优化后的版本，那么只需要替换 JSTL 中的 `import` 语句即可。记住我们并没有接触原始的源文件。这个文件是在创建 WAR 文件时快速被修改的。

这个例子覆盖了 JavaScript 程序员最通用的用例。接下来我们来讨论 JavaScript 源文件的代码质量问题。

### 11.1.6 使用外部的 Java 库分析 JavaScript 代码

在代码被部署到产品环境中之前主动去检测漏洞和发现潜在的问题是每个 JavaScript 程序员都应该谨记于心的。JSHint 是一个非常流行的用 JavaScript 语言编写的工具，用来检测 JavaScript 错误和潜在的问题。为了运行 JSHint，你需要使用 Gradle 来执行。但是如何在 Gradle 中执行 JavaScript 呢？

Rhino 是一个用 Java 语言编写的开源的 JavaScript 实现。借助 Rhino，你就能够运行 JavaScript 文件了，并在构建中使用 JSHint。在下面的清单中，你会从 Maven Central 获取 Rhino，并通过一个托管仓库中的 URL 下载 JSHint 的 JavaScript 文件。

#### 清单 11.6 定义 Rhino 和 JSHint 依赖

```

repositories {
    mavenCentral()

    ivy {
        name 'JSHint'
        url 'http://www.jshint.com/get'
        layout 'pattern', {

```

声明 JSHint 下载 URL 为 Ivy 仓库

```

        artifact '[module]-[revision]([.][classifier]).[ext]'
    }
}

configurations {
    rhino
    jshint
}

dependencies {
    rhino 'org.mozilla:rhino:1.7R4'
    jshint 'jshint-rhino:jshint-rhino:2.1.4@js'
}

```

声明 JSHint 下载 URL 为 Ivy 仓库

Rhino 和 JSHint 的自定义配置

定义 Rhino 依赖

使用 @js 定义 JSHint 的 JavaScript 依赖

你将 Rhino 定义成一个依赖——现在我们实际来看看。为了不直接使用 Ant task，我们通过将其赋值给一个增强的 JavaExec 类型的 Gradle task 来调用其 main 方法。JSHint 要求你将需要检查的 JavaScript 文件以参数的形式提供。在后台，应用程序对 Rhino 的 Java 调用与下面的命令类似：

```
java -jar rhino-1.7RC4.jar jshint-rhino-2.1.4.js edit-action.js update-
  ➤ action.js
```

下面的清单演示了如何构造这个参数列表，以及为调用 Rhino 的 main 主类编写增强的 task。

### 清单 11.7 在 Gradle 中通过 Rhino 执行 JSHint JavaScript

```

ext.jsSourceDir = "$webAppDir/js/app"
def jsSourceFiles = fileTree(jsSourceDir).include('*.js')
def jsHintArgs = configurations.jshint + jsSourceFiles

task jsHint(type: JavaExec) {
    classpath configurations.rhino
    main 'org.mozilla.javascript.tools.shell.Main'
    args jsHintArgs
}

```

将 Rhino 和 JSHint 的参数放在一起，JavaScript 是第一个

增加 task 以 Java 进程执行 Rhino

在具有完整功能的应用程序下执行这个 task，应该不会报告任何问题：

```
$ gradle jsHint
:web:jsHint
```

然而，试着模拟一个 bug，将 edit-action.js 文件的最后一行的分号删除，你就会看到 JSHint 会准确地报出错误并且使构建失败：

```

$ gradle :web:jsHint
:web:jsHint
Missing semicolon. (/Users/Ben/gradle-in-action/code/chapter11/todo-
  ➤ js-code-quality/web/src/main/webapp/js/app/edit-action.js:2:46)
> $("#{todoItem_" + row).addClass("editing")
:web:jsHint FAILED

```

### 11.1.7 使用第三方 Gradle JavaScript 插件

我们看到通过引入第三方库可以很方便地在 Gradle 构建中创建高效的 JavaScript 工具。这可以通过重用 Ant task，使用 JavaExec 类型的 task 执行 Java 应用程序，或者通过调用外部的 shell 脚本来实现。只需要一点点的调查，并抱有一种使用最好的工具的渴望。

我们都知道好的程序员都很懒——懒是因为他们讨厌做单调重复的工作。大多数程序员倾向于重用已有的功能。Gradle 插件社区提供了你需要实现的大部分功能，甚至更多（如生成 JSDoc 格式的 API 文档）。进入 Gradle JavaScript 插件（<https://github.com/eriwen/gradle-js-plugin>）。这个插件对于需要使用到 JavaScript 的项目来说就是个最好的开始，并且不需要你实现自己的 task。

在本节中，我们会讨论如何使用这个插件，并使用它所提供的 DSL 来优化 JavaScript——也就是合并和压缩 JavaScript 文件。首先，从 Maven Central 下载这个插件，并将其应用到 web 项目中，如下面的清单所示。

清单 11.8 将 JavaScript 插件添加到构建脚本的 classpath 中

```
buildscript {
    repositories {
        mavenCentral()
    }

    dependencies {
        classpath 'com.eriwen:gradle-js-plugin:1.5.1'
    }
}

apply plugin: com.eriwen.gradle.js.JsPlugin
```

这个插件用一个专有的名字来描述 JavaScript 源目录。这个名字会被这个插件定义的一个 task 用作输入。下面的清单显示了这个插件更详细的 DSL。这段代码应该跟你前面所构建的非常相似，但是更加可读且更加容易理解了。

清单 11.9 配置 JavaScript 插件

```
ext {
    jsSourceDir = "$webAppDir/js/app"
    jsOutputDir = "$buildDir/js"
}

javascript.source {
    app {
        js {
            srcDir jsSourceDir
            include '*.js'
            exclude '*.min.js'
        }
    }
}
```

定义 src/main/webapp/jsp/app/  
目录为源目录

只包含未压缩的 JavaScript 文件

```
    }  
  }  
  
  combineJs {  
    source = javascript.source.dev.js.files  
    dest = file("$jsOutputDir/all.js")  
  }  
  
  minifyJs {  
    source = combineJs  
    dest = file("$jsOutputDir/all-min.js")  
  }  
}
```

将 JavaScript source set 作为合并 JavaScript 文件的输入

将 combineJs 的输出作为 minifyJs 的输入

这个 task 的用法跟你目前所体验的其他 task 的用法类似。需要对文件做压缩的话，执行 minifyJs task。这个 task 在压缩文件之前，首先会调用 combineJs task 来合并项目中用到的 JavaScript 文件。

### 关于 CSS 呢？

对于 CSS 优化的需求在很多情况下与 JavaScript 没有什么不同。可以实现类似的 task 来合并和压缩 CSS 文件，使用其他工具例外。你可以快速看一下 Gradle 的 CSS 插件：<https://github.com/eriwen/gradle-css-plugin>。

有第三方插件支持 Gradle 中的 JavaScript 非常棒。但是如果你的前端团队并不是使用 Gradle 实现的自动化构建呢？你需要重写全部已有的逻辑吗？谢天谢地，不需要，你可以在 Gradle 中直接调用其他的自动化构建工具。我们来看一个使用 Grunt 的例子。

## 11.1.8 在 Gradle 中使用 Grunt

如果对 JavaScript 社区有深入的了解，你应该知道构建工具 Grunt (<http://gruntjs.com>)。Grunt 主要致力于实现 JavaScript 代码的常用 task，如压缩、合并、自动化测试等。利用其完善的插件系统和快速成长的社区，你需要考虑将已有的 JavaScript 工作流自动化。

在开始使用 Grunt 之前，首先需要通过 NPM (Node.js 包管理器) 安装可执行的程序。详细的安装步骤，请参考 Grunt 官方文档。基于你的操作系统，最后你会得到一个叫作 grunt.cmd (Windows) 的批处理脚本，或者一个叫作 grunt (\*nix) 的 shell 脚本。这个可执行的脚本被用来运行 Grunt 构建。一个 Grunt 构建需要两个必不可少的文件：Gruntfile.js 定义了需要被执行的 task 和相应的配置，package.json 定义了项目的基本数据和所需要的外部依赖（如插件）。

在本节中，我们会讨论如何在 Gradle 中调用 Grunt 的 task。这是一个很有价值的方法，主要有两个原因：可能你已经有了一个自动化的构建过程，并且不想在 Gradle 中重写这部分代码，或者你想要用一个单独的自动化构建工具来覆盖应用程



序的各个方面。图 11.1 演示了在实践中如何执行 Grunt 可执行命令。

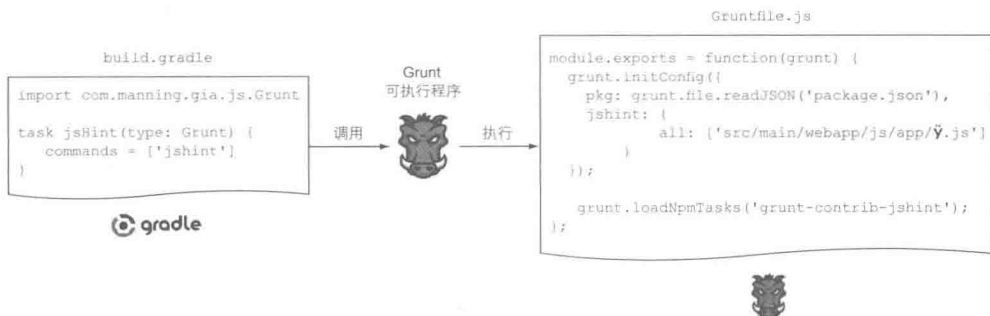


图 11.1 在 Gradle 中调用 Grunt 可执行程序

我们通过例子来看看。Grunt 并不会自动安装 `package.json` 中定义的依赖。如果一个用户忘记了这个步骤，Grunt 构建就会失败。为了安装这些依赖，请跳转到 `package.json` 所在的目录，运行 `npm install` 命令。NPM 会将依赖安装到 `node_modules` 目录下。

当然，我们，作为自动化专家，想要使这件事情自动发生。在运行 Grunt 的 task 之前，运行 Node 的 `install` 命令。如下清单演示了如何把这个调用封装成一个 Gradle 的 task。

#### 清单 11.10 通过 npm 安装 Grunt 依赖的 task

```

task installGruntDependencies(type: Exec) {
    inputs.file 'package.json'
    outputs.dir 'node_modules'

    executable 'npm'
    args 'install'
}

```

定义必须依赖的 Grunt 包文件

npm 用来存储其所下载的依赖的输出目录

执行 npm install 命令

仔细看图 11.1，你就会注意到有一个 Grunt 类型的 task。清单 11.11 演示了 Groovy 的 Grunt task，这个类被定义在 `buildSrc/src/main/groovy` 下，作为一个简单的调用 Grunt 可执行程序的 wrapper。你只需要为这个增强的 task 提供定义在 Grunt 构建文件中的命令（Grunt task）。

#### 清单 11.11 为调用 Grunt 可执行程序自定义 task

```

package com.manning.gia.js

import org.gradle.api.DefaultTask
import org.gradle.api.tasks.Input
import org.gradle.api.tasks.TaskAction

```

```

class Grunt extends DefaultTask {
    @Input
    List<String> commands

    @TaskAction
    void callGrunt() {
        project.exec {
            executable isWindows() ? 'grunt.cmd' : 'grunt'
            args commands
        }
    }

    boolean isWindows() {
        System.properties['os.name'].toLowerCase().contains('windows')
    }
}

```

使用 Project 的 exec 方法调用 Grunt 可执行程序

基于操作系统确定 Grunt 可执行程序

将 Grunt task 作为命令行参数

我们假设你有一个应用了 JSHint 插件的 Grunt 构建。JSHint 的 task 名字叫作 jshint。为了调用这个 Grunt task，创建一个新的 Grunt 类型的 task，并将 Grunt 的 task 名字作为命令，如下面的清单所示。

#### 清单 11.12 使用增强的 task 执行 Grunt JSHint 插件

```

import com.manning.gia.js.Grunt

task jsHint(type: Grunt, dependsOn: installGruntDependencies) {
    commands = ['jshint']
}

```

执行 task 前安装 Grunt 项目依赖

执行一系列 GruntJS task

运行 To Do 应用程序的 jsHint task。下面的命令行输出显示 Grunt 发现了一个有错的 JavaScript 文件，它缺少了一个分号：

```

$ gradle :web:jsHint
:buildSrc:compileJava UP-TO-DATE
:buildSrc:compileGroovy
...
:buildSrc:build
:web:installGruntDependencies
npm WARN package.json my-project-name@0.1.0 No README.md file found!
npm http GET https://registry.npmjs.org/grunt
...
grunt@0.4.1 node_modules/grunt
├─ dateFormat@1.0.2-1.2.3
├─ ...
...
grunt-contrib-jshint@0.6.0 node_modules/grunt-contrib-jshint
├─ jshint@2.1.4 (console-browserify@0.1.6, underscore@1.4.4, shelljs@0.1.4,
  └─ minimatch@0.2.12, cli@0.4.4-2)
:web:jsHint
Running "jshint:all" (jshint) task
Linting src/main/webapp/js/app/edit-action.js...ERROR
[L2:C46] W033: Missing semicolon.
    $("#toDoItem_" + row).addClass("editing")

```

```
Warning: Task "jshint:all" failed. Use --force to continue.  
Aborted due to warnings.  
:web:jsHint FAILED
```

讨论完 Gradle 与 JavaScript 的集成后，我们来看看 Gradle 对除 Java 外其他 JVM 语言的支持。

## 11.2 构建基于JVM的多语言项目

Java 一枝独秀的时代已经过去了。现如今，当想到 Java 的时候，我们通常将其作为一个成熟的开发平台：Java 虚拟机（JVM）。一系列可以在 JVM 上运行的语言被用来开发企业级应用和服务器端程序。流行的语言有 Groovy、Scala、Clojure 和 JRuby 等。你为什么想要使用不同于 Java 的语言或者在项目中混合使用它们呢？再强调一遍，我们需要为工作选择正确的工具。你可能在实现业务逻辑的时候更喜欢 Java 的静态类型特性和类库支持。然而，Java 的语法并不能很自然地生成 DSL。这正是其他语言如 Groovy 的用武之地。

到目前为止，我们讨论了如何构建 Java 应用程序，但是 Gradle 的作用不仅仅如此。它对于其他 JVM 语言也有成熟的支持。在本节中，我们将会看到如何组织和构建 Groovy 和 Scala 项目，以及如何将这些项目放到一个项目中。在进一步讨论之前，我们回顾一下 Java 插件的内部工作原理。这是把构建扩展到其他语言插件的基础。

### 11.2.1 JVM 语言插件的基本功能

在第 8 章中，我们讨论了在插件中对于职责分离的一个指导性原则：功能和约定。我们来快速回顾一下这个概念。一个提供功能的插件通常会引入新的概念或者新的 task，一个提供约定的插件会为这些概念强加一些默认值。一些听起来很抽象的东西，通过剖析 Java 插件的内部工作原理可以变得很清晰。

#### Java 项目的功能和约定

作为一个 Gradle 用户，当你将 Java 插件通过 apply 语句应用到项目中的时候，你看到的是这个插件所提供的所有功能。你的项目就自动具备了 source set 和 Java 的特定配置，并将其 task 和 property 暴露出来。这个基本的 Java 支持是所有的基于 JVM 语言的项目的构建基础，这些功能有：

- 配置：compile, runtime
- 每个 source set 的 task：compileJava, processResources, classes
- 与生命周期相关的 task：check, build, buildNeeded, buildDependents
- 属性：sourceCompatibility, targetCompatibility

Gradle 还为项目预先做了一些默认配置，也就是通常所说的约定——例如，产品源代码被放置在 `src/main/java` 目录下，测试源代码被放置在 `src/test/java` 目录下。这些约定并不是无法改变的，可以被重新配置，但是它们给了项目一个最初的结构。Java 项目中的一些 task 是基于预配置的 source set 的。下面列出了 Java 插件中的一些约定：

- 配置：testCompile, testRuntime
- source sets：main, test
- task：test, javadoc

### Java 基础插件

Java 插件将功能和约定分离开了。怎么做到的呢？功能被提取到另一个插件中，即 Java 基础插件。Java 插件在内部应用了 Java 基础插件，并在其上添加了一些约定，如图 11.2 所示。

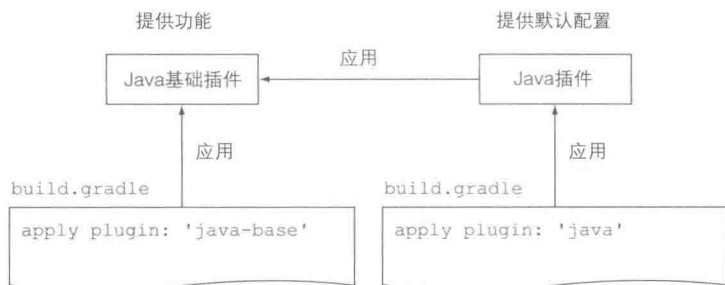


图 11.2 Java 插件自动应用 Java 基础插件

这样的关注点分离有很多好处。首先，如果 Java 项目并不适合 Java 插件所引入的默认约定，你可以采用 Java 基础插件，并定义自己的约定。其次，Java 基础插件可以被用来作为构建其他 JVM 语言插件的基础，因为它们都拥有相同的概念。这是代码复用的一个非常好的例子，并且被运用到了构建 Groovy 和 Scala 项目的核心插件中。在我们详细了解如何应用这些基本功能之前，先来研究一下在编译 Java 源代码时另一个很有用的优化。

### 使用编译器守护进程加速 Java 编译过程

在默认情况下，Gradle 会为每个编译 task 创建一个新的 JVM 实例。尤其是在大型的基于 Java 的多项目构建中，这导致了一些不必要的开销。你可以通过重用已经创建好的 JVM 实例来加速编译过程。在下面的清单中，配置每个 JavaCompile 类型的 task 运行在所创建的子进程里面，同时集成 Gradle 自己的 Java 编译器。

清单 11.13 激活编译器守护进程

```
subprojects {
    apply plugin: 'java'
    tasks.withType(JavaCompile) {
        options.useAnt = false
        options.fork = true
    }
}
```

直接使用 Gradle 集成的编译器，  
绕过 Ant 的 javac

在独立的进程中编译

当在根目录下通过 `gradle clean build` 执行多项目构建的时候，你会注意到构建的总时间会比没有上述修改的时候少。在内部，Gradle 重用了编译器守护进程，即使 task 属于不同的子项目。在构建完成后，编译器守护进程就被停止了。这意味着这些进程的生命周期不会超过一次构建。接下来，我们将集中讨论对 Groovy 项目的构建和配置。

### 11.2.2 构建 Groovy 项目

Groovy 和 Java 配合得很好。Groovy 是基于 Java 的，但是它有更简洁的语法，并且添加了动态语言特性。在 Java 项目中引入 Groovy，只需要简单地将 Groovy 的 JAR 文件放置到 classpath 下即可。

在本书中，你已经在使用 Groovy 编写项目的基础设施代码了。你使用过 Spock 框架来为产品程序代码编写单元测试。我们还看过如何使用 Groovy 来实现自定义的 Gradle task。对于一些保守的组织，这些就是集成 Groovy 的很好的用例，因为代码还没有被部署到产品环境中。可能有各种原因导致你会想要使用 Groovy 而不是 Java 来实现产品代码。通过使用 Groovy 的动态语言特性，可以很方便地编写 DSL。这就是 Gradle 做的事情，其使用 Java 实现核心逻辑，但是使用 Groovy 封装来暴露强大的 DSL，也就是你现在所熟悉的 DSL。Gradle 中对 Groovy 的支持是通过 Groovy 插件来提供的。我们来看看这个特性，并看看 Java 基础插件是如何起作用的。

#### Groovy 基础插件和 Groovy 插件

Groovy 插件系统跟 Java 插件一样，引入了类似的分离功能和约定的概念。为了它它们如何合作有个大概的了解，请看图 11.3。

图中间是 Groovy 基础插件和 Groovy 插件。Groovy 基础插件继承了 Java 基础插件的所有功能，并引入了自己的约定。如下是它的特性清单：

- `source set` : `groovy`
- 每个 `source set` 的 `task` : `compileGroovy`, `compileTestGroovy`

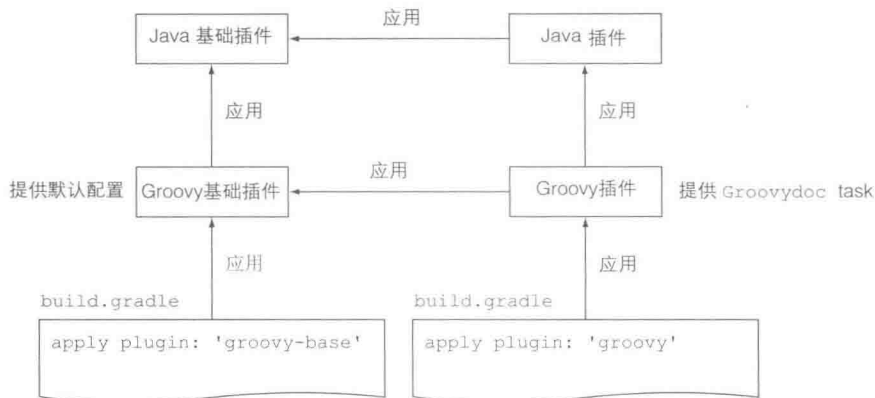


图 11.3 Groovy 插件继承了 Java 基础插件的功能

Groovy 插件是基于 Groovy 基础插件和 Java 插件的。这个插件带来的另一个好处是引入了 Groovydoc 类型的 task。执行这个 task 会为 Groovy 源代码生成 HTML API 文档。接下来,你会将 To Do 应用程序的一个程序从 Java 项目转换成 Groovy 项目。

### 准备工作

将一个独立的 Java 项目转换成 Groovy 项目很容易。我们会以 model 项目为例来讲解转换过程。目前,这个项目只定义了一个 Java 类文件:ToDoItem.java。在将这个类转换成 Groovy 类之前,需要在 model/build.gradle 脚本中添加构建基础设施代码,如下面的清单所示。

#### 清单 11.4 使用 Groovy 管理产品源代码

```

apply plugin: 'groovy'
repositories {
    mavenCentral()
}
dependencies {
    compile 'org.codehaus.groovy:groovy-all:2.1.5'
}

```

应用 Groovy 插件

声明 Groovy JAR 文件为编译依赖

通过应用 Groovy 插件,你就拥有了编译 Groovy 源代码的能力,并且定义默认的 source set,这样编译器就知道去哪寻找源代码文件了。

Groovy 插件也自动应用了 Java 插件。这就意味着你的项目现在既拥有了 Java 的 source set,也拥有了 Groovy 的 source set。图 11.4 显示了应用 Groovy 插件后默认的项目结构。

默认的项目结构

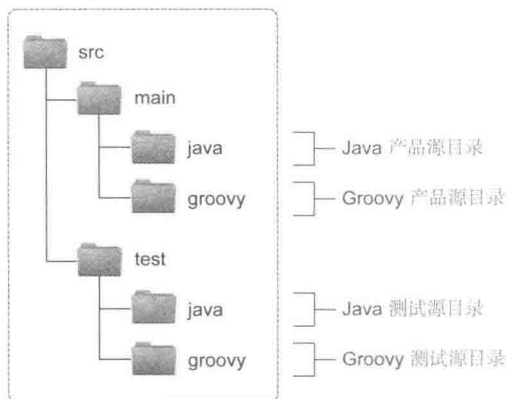


图 11.4 应用了 Groovy 插件的项目默认源目录

### 我应该使用哪个 Groovy 分发包？

你可能知道 Groovy 库有多个分发包。groovy-all 文件将所有的外部依赖（如 Antlr 和 Apache Commons CLI）包含在了一个 JAR 文件中，从而保证它们之间的兼容性。groovy 文件并不包含这些依赖。你应该在构建脚本中声明这些依赖。不管使用哪个分发包，你都能够编译项目中的 Groovy 源代码。

记住，这并不代表你必须要在项目中包含 Java 源文件。然而，它能够使你在一个项目中混合使用 Java 和 Groovy 文件。现在，将已经存在的 Java 类重命名为 `ToDoItem.groovy`，并将这个文件移动到 `src/main/groovy` 目录下，并将这个文件移动到 `src/main/groovy` 目录下的 `com.manning.gia.todo.model` 包下面。是时候将这个笨重的 POJO 类实现转换到 Groovy 了。下面的清单演示了压缩代码的极端例子。

#### 清单 11.15 使用 Groovy 编写 `ToDoItem` 模型类

```

package com.manning.gia.todo.model

import groovy.transform.Canonical

@Canonical
class ToDoItem implements Comparable<ToDoItem> {
    Long id
    String name
    boolean completed

    @Override
    int compareTo(ToDoItem toDoItem) {
        this.id.compareTo(toDoItem.id)
    }
}

```

← 自动生成 `equal`、`hashCode` 和 `toString` 方法的 Groovy 注解

尝试着构建 Groovy 项目。你会看到 `compileJava` 被标记成了 `UP-TO-DATE`，这是因为在 `src/main/java` 下没有 Java 源文件了。`compileGroovy` 找到了 `ToDoItem.groovy` 文件并对其进行编译。如下命令行输出演示了这一行为：

```
$ gradle :model:build
:model:compileJava UP-TO-DATE
:model:compileGroovy
Download http://repo1.maven.org/maven2/org/codehaus/groovy/groovy-
➡ all/2.1.5/groovy-all-2.1.5.pom
Download http://repo1.maven.org/maven2/org/codehaus/groovy/groovy-
➡ all/2.1.5/groovy-all-2.1.5.jar
:model:processResources UP-TO-DATE
:model:classes
:model:jar
:model:assemble
:model:compileTestJava UP-TO-DATE
:model:compileTestGroovy UP-TO-DATE
:model:processTestResources UP-TO-DATE
:model:testClasses UP-TO-DATE
:model:test
:model:check
:model:build
```

### 自定义项目结构

如果通过 Groovy 基础插件引入的默认约定并不符合要求，那么你可以像在 Java 项目中一样重新对其进行定义。`SourceSet` 基本概念能够使你重新配置默认的源目录。在下面的清单中，你会将 Groovy 产品源文件放在 `src/groovy` 下，Groovy 测试源文件放在 `test/groovy` 下。

清单 11.16 自定义默认的 Groovy source set 目录

```
sourceSets {
    main {
        groovy {
            srcDirs = ['src/groovy']
        }
    }
    test {
        groovy {
            srcDirs = ['test/groovy']
        }
    }
}
```

### Java 和 Groovy 联合编译

前面我提到过一个 Gradle 项目能够同时包含 Java 和 Groovy 文件。对于开发人员，在 Java 类中使用 Groovy 类是一件很自然的事，反之亦然。最后，这两种类型



的文件都应该被编译成字节码不是吗？这里有个比较棘手的问题，Groovy 可以依赖 Java，但是 Java 不能依赖 Groovy。

为了演示这个行为，将 Groovy 插件应用到 repository 项目。你会将 `ToDoRepository` 类从 Java 接口转换到 Groovy 接口，如下面的清单所示。

### 清单 11.17 使用 Groovy 编写的 repository 接口

```
package com.manning.gia.todo.repository

import com.manning.gia.todo.model.ToDoItem

interface ToDoRepository {
    List<ToDoItem> findAll()
    List<ToDoItem> findAllActive()
    List<ToDoItem> findAllCompleted()
    ToDoItem findById(Long id)
    Long insert(ToDoItem toDoItem)
    void update(ToDoItem toDoItem)
    void delete(ToDoItem toDoItem)
}
```

这个接口拥有两个实现，它们还都是 Java 类：`InMemoryToDoRepository.java` 和 `H2ToDoRepository.java`。现在，编译 repository 项目，你会得到编译错误：

```
$ gradle :repository:classes
:model:compileJava UP-TO-DATE
:model:processResources UP-TO-DATE
:model:classes UP-TO-DATE
:model:jar UP-TO-DATE
:repository:compileJava
/Users/Ben/gradle-in-action/chapter11/todo-mixed-java-groovy/
➤ repository/src/main/java/com/manning/gia/todo/repository/
➤ H2ToDoRepository.java:9: cannot find symbol
symbol: class ToDoRepository
public class H2ToDoRepository implements ToDoRepository {
                                         ^

/Users/Ben/gradle-in-action/code/chapter11/todo-mixed-java-groovy/
➤ repository/src/main/java/com/manning/gia/todo/repository/
➤ InMemoryToDoRepository.java:12: cannot find symbol
symbol: class ToDoRepository
public class InMemoryToDoRepository implements ToDoRepository {
                                         ^

...
16 errors
:repository:compileJava FAILED
```

这是不是意味着你永远都不能在 Java 类中对 Groovy 类有依赖？不是。让这个行为起作用的关键就是联合编译，这使得你能够自由地混合使用 Java 和 Groovy 源代码，并且有双向依赖。解决这个问题的一种方法是将 Java 源代码与 Groovy 源代码都放在 `src/main/groovy` 目录下。或者，你可以配置 Groovy 编译器，使其能

够联合编译。下面的清单显示了如果要使用这个功能所需要做的配置。

清单 11.18 为联合编译重新配置 source set

```
sourceSets.main.java.srcDirs = []  
sourceSets.main.groovy.srcDirs = ['src/main/java', 'src/main/groovy']
```

移除 Java source set 的源目录

指定 Groovy 编译器包含 Java 和 Groovy 源目录

在 repository 项目的 build.gradle 文件中添加了如上代码片段后，联合编译就像预期一样起作用了：

```
$ gradle :repository:classes  
:model:compileJava UP-TO-DATE  
:model:processResources UP-TO-DATE  
:model:classes UP-TO-DATE  
:model:jar UP-TO-DATE  
:repository:compileJava UP-TO-DATE  
:repository:compileGroovy  
:repository:processResources UP-TO-DATE  
:repository:classes
```

作为编译器迷，我们来看看后端发生了什么：

- 1 编译器解析 Groovy 源文件，并为其生成存根。
- 2 Groovy 编译器调用 Java 编译器，并为 Java 源文件生成存根。
- 3 有了 Groovy 源路径下的 Java 存根，Groovy 编译器就可以编译两种语言了。

使用本章中介绍的这些技巧，你应该能够为自己的项目集成 Groovy 了，不管是一个单独的 Groovy 项目还是一个混合的 Java 和 Groovy 项目。接下来，我们来看看 Gradle 对 Scala 项目的支持。

### 11.2.3 构建 Scala 项目

Scala 是 JVM 下面的另一种编程语言，在过去的几年里变得越来越流行。Scala 有静态的类型，集合了面向对象编程和函数式编程的支持，并且其设计是以优雅简洁的表达逻辑为目标的。与 Groovy 一样，Scala 可以使用所有 Java 开发者已知的 Java 库。Scala 已经在现实世界中被采用了。Twitter 用 Scala 重新实现了它们的后端服务，Foursquare 也迁移到了 Scala。

Gradle 对 Scala 的支持与其对 Groovy 的支持一样成熟。在本节中，我们会研究相应的 Scala 语言插件。你会采用与使用 Groovy 的时候类似的做法，将 To Do 应用程序转换成 Scala 并用 Gradle 来构建。

#### Scala 基础插件和 Scala 插件

Scala 插件与 Groovy 插件拥有相同的概念。基本思想都是将功能和约定分离开

来。在图 11.5 中，你可以看到 Gradle 提供了两个 Scala 插件。

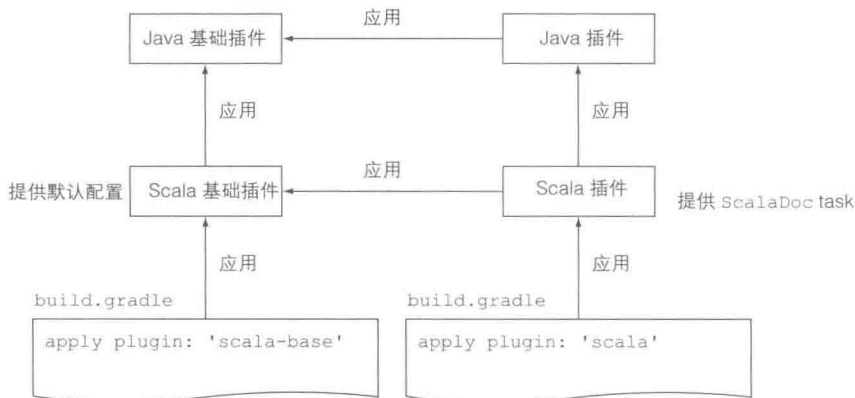


图 11.5 Scala 插件继承了 Java 基础插件的功能

Scala 基础插件自动应用了 Java 基础插件。前面你已经学到了 Java 插件拥有构建基于 Java 项目的功能。Scala 基础插件使用这些功能并且为 Scala 项目预配置了默认的约定，如下所示：

- `source set` : `scala`
- 每个 `source set` 的 `task` : `compileScala`, `compileTestScala`

Scala 插件内部不仅应用了 Scala 基础插件，还应用了 Java 插件。这使得你能够在在一个项目中构建 Java 和 Scala 源代码。Scala 插件在 Scala 基础插件基础上新增的功能是为 Scala 源代码生成 HTML 格式的 API 文档。如果你正在构建一个成熟的 Scala 项目，应用 Scala 插件就是最好的选择。我们在实际例子中来看看。

## 准备工作

为了在 To Do 应用程序中演示如何使用 Scala，你会将一个 Java 类转换成 Scala 类。在你的构建中源代码最简单的项目就是 `model` 项目。目前，这个项目的 `build.gradle` 并没有定义任何逻辑。下面的清单显示了应用 Scala 插件并声明 Scala 库，从而获得 Scala 编译器。

### 清单 11.19 使用 Scala 来管理产品源代码

```

apply plugin: 'scala'
repositories {
    mavenCentral()
}
dependencies {
    compile 'org.scala-lang:scala-library:2.10.1'
}

```

应用 Scala 插件

声明 Scala 库 JAR 文件为编译依赖

在应用了 Scala 插件后，你的项目就能够编译 `src/main/scala` 和 `src/test/scala` 目录下的 Scala 源代码了。图 11.6 显示了与 Java 插件源代码目录结构约定相统一的源目录结构。

默认的项目结构

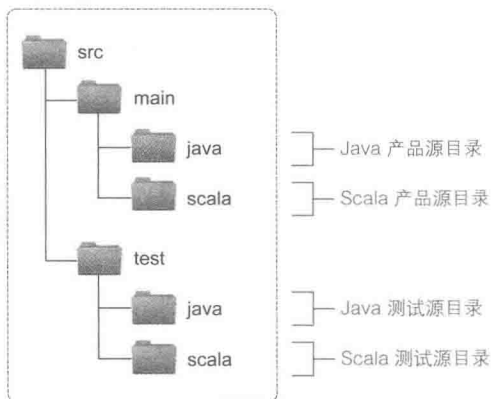


图 11.6 应用了 Scala 插件的项目默认源目录

接下来，你会将已经存在的 `ToDoItem.java` 转换成 Scala 文件。每一个 Scala 源代码文件的扩展名都是 `.scala`。将其重命名为 `ToDoItem.scala` 并移动到 `src/main/scala` 目录下。现在我们将源代码改成 Scala 语言，如下面的清单所示。

#### 清单 11.20 使用 Scala 编写 `ToDoItem` 模型类

```
package com.manning.gia.todo.model

class ToDoItem extends Ordered[ToDoItem] {
  var id: Long = _
  var name: String = _
  var completed: Boolean = _

  def getId: Long = {
    id
  }

  def setId(id: Long) = {
    this.id = id
  }

  ...

  override def compare(that: ToDoItem) = this.id compare that.id

  override def equals(that: Any) = {
    that match {
      case t: ToDoItem => t.id == id && t.name == name
                        && t.completed == completed
      case _ => false
    }
  }
}
```

即使你对 Scala 不是很熟悉，其语法也是很容易理解的。实际上，这个类并没

有引入任何其他的逻辑，并且跟 Java 代码做的工作完全一样。执行 model 项目的构建会得到如下输出：

```
$ gradle :model:build
:model:compileJava UP-TO-DATE
:model:compileScala
Download http://repo1.maven.org/maven2/org/scala-lang/scala-library/
➡ 2.10.1/scala-library-2.10.1.pom
Download http://repo1.maven.org/maven2/org/scala-lang/scala-library/
➡ 2.10.1/scala-library-2.10.1.jar
Download http://repo1.maven.org/maven2/org/scala-lang/scala-compiler/
➡ 2.10.1/scala-compiler-2.10.1.pom
Download http://repo1.maven.org/maven2/org/scala-lang/scala-reflect/
➡ 2.10.1/scala-reflect-2.10.1.pom
Download http://repo1.maven.org/maven2/org/scala-lang/scala-compiler/
➡ 2.10.1/scala-compiler-2.10.1.jar
Download http://repo1.maven.org/maven2/org/scala-lang/scala-reflect/
➡ 2.10.1/scala-reflect-2.10.1.jar
:model:processResources UP-TO-DATE
:model:classes
:model:jar
:model:assemble
:model:compileTestJava UP-TO-DATE
:model:compileTestScala UP-TO-DATE
:model:processTestResources UP-TO-DATE
:model:testClasses UP-TO-DATE
:model:test
:model:check
:model:build
```

我们可以看到 compileScala 在初始化的时候下载了 Scala 库，所以你才能够调用 Scala 编译器。因此，你的项目编译的 Scala 代码就可以被作为依赖用在其他项目中了。

### 自定义项目结构

如果你不喜欢默认的 Scala 源目录结构，则可以重新定义。到现在为止，你已经运用过这个概念很多次了，所以对下面清单中的代码应该很熟悉了。在这个例子中，你将产品源目录配置为 src/scala，将测试源目录配置为 test/scala。

#### 清单 11.21 自定义默认的 Scala source set 目录

```
sourceSets {
    main {
        scala {
            srcDirs = ['src/scala']
        }
    }
    test {
        scala {
```

```
        srcDirs = ['test/scala']
    }
}
```

## Java 和 Scala 联合编译

在一个 Gradle 项目中混合使用 Java 和 Scala 源代码的规则跟混合使用 Java 和 Groovy 的规则是一样的。不方便的地方就是 Scala 可以依赖 Java，但是 Java 不能依赖 Scala。双向依赖只能通过联合编译来实现。你可以将 Java 代码放到 Scala 源目录（默认是 `src/main/scala`）中，或者重新配置源目录，如下面的清单所示。

清单 11.22 为联合编译重新配置 source set

```
sourceSets.main.scala.srcDirs
sourceSets.main.groovy.srcDirs = ['src/main/java', 'src/main/scala']
```

移除 Java source set 的源目录

指定 Scala 编译器包含 Java 和 Scala 源目录

当你配置了 Scala 联合编译后发生了什么呢？对 Scala 代码的联合编译与 Groovy 代码有点不同。Scala 编译器并不直接调用 Java 编译器。如下步骤会解释得更清楚：

- 1 Scala 编译器解析并分析 Scala 源代码，从而知道其与 Java 类之间的依赖关系。Scala 编译器能够理解 Java 语法，但是并不调用 Java 编译器。
- 2 Scala 源代码被编译后，编译器为每个 Java 源文件生成一个类文件。然而，这些文件并不包含二进制代码。它们只是在 Java 和 Scala 之间做类型检查的时候被用到。
- 3 Java 编译器编译 Java 源代码。

我们谈到了构建 Scala 项目需要的一些最基本的技巧。对编译过程的优化如增量编译可以减少编译时间。请参考在线文档了解更多信息。在下一节中，我们会浏览一下 Gradle 对其他语言的编译和打包支持。

## 11.3 其他语言

编程语言有很多种，本书中不可能一一尽述。其中一些语言可以被核心插件直接支持；其他的可以通过 Gradle 社区编写的插件来进行集成。表 11.1 列出了一些比较流行的语言插件。

表 11.1 流行的编程语言的 Gradle 插件

语 言	主 页	Gradle 插件
C++	<a href="http://www.cplusplus.com">http://www.cplusplus.com</a>	Gradle 核心插件
Clojure	<a href="http://clojure.org">http://clojure.org</a>	<a href="http://bitbucket.org/clojuresque/clojuresque">http://bitbucket.org/clojuresque/clojuresque</a>
Golo	<a href="http://golo-lang.org">http://golo-lang.org</a>	<a href="http://github.com/golo-lang/gradle-golo-plugin">http://github.com/golo-lang/gradle-golo-plugin</a>
Kotlin	<a href="http://kotlin.jetbrains.org">http://kotlin.jetbrains.org</a>	<a href="http://repository.jetbrains.com/kotlin/org/jetbrains/kotlin/kotlin-gradle-plugin">http://repository.jetbrains.com/kotlin/org/jetbrains/kotlin/kotlin-gradle-plugin</a>
R	<a href="http://www.programmingr.com">http://www.programmingr.com</a>	<a href="http://github.com/jamiefolson/gradle-plugin-r">http://github.com/jamiefolson/gradle-plugin-r</a>

即使为你所使用的语言找不到一个适当的插件集成到构件中，你也还是有希望的。你已经学过了 Ant 的 task 和 JavaAPI 可以轻易的被 Gradle 的 task 封装。Groovy 插件就用了类似的技术，比如，其内部调用了 `groovyc` Ant task。或者，任何时候你都可以通过创建一个增强的 `Exec` 类型的 task 在命令行执行编译器。

## 11.4 总结

今天的软件项目更倾向于使用多种编程语言、多种技术、多种库来解决问题。总之，就是为工作选择合适的工具。无论怎样，只要做得最好、能提高效率，或者能够更方便地解决问题的都应该被选择。这一准则在所谓的多语言项目中显得尤其重要。在本章中，我们讨论了使用 Gradle 如何配置、管理和构建三种不同的语言：JavaScript 以及基于 JVM 的 Groovy 和 Scala。

JavaScript 被作为创建动态 Web 应用程序的主导语言已经有数十年了。显然，页面加载时间受 JavaScript 文件的大小影响。JavaScript 文件可以被合并和压缩以提高页面渲染效率。你学到了 Gradle 可以帮助将这些本来需要手动执行的步骤自动化了，并且将它们集成到了 To Do 应用程序的开发生命周期中。我们还探讨了通过使用社区的 JavaScript 插件如何来简化 JavaScript 构建代码的配置。后来，你用 Gradle 封装了已经存在的 Grunt 构建代码，从而提供了一个统一的自动化应用程序所有组件的控制单元。

基于 JVM 的 Groovy 和 Scala 语言在 Gradle 支持。Gradle 对这两种语言提供了一流的插件支持。每种语言插件都基于 Java 基础插件将功能和约定分离开来。你将 To Do 应用程序的部分 Java 类转换成 Groovy 和 Scala 语言的，从而演示了这些插件的使用。source set 约定可以被改变。你学到了如何重新配置它们以适应一个遗留项目布局的需要。Groovy 和 Scala 源代码可以和 Java 源代码在一个项目中同时存在。Java 和 Groovy 或 Scala 的双向依赖需要靠联合编译来达到。我们讨论了如何配置编译器，从而使其能够处理这样的场景。在本章的最后一部分，我们还接触到了 Gradle 所支持的其他语言。

下一章中我们会专注于通过将外部工具集成到构建过程中来监测项目的代码质量。





# 12

## 代码质量管理和监测

---

### 本章涵盖

- 将代码分析集成到构建中
- 衡量代码测试覆盖率
- 执行静态代码分析
- 集成 Sonar 来衡量软件质量

大多数商业项目的最终产品都是二进制程序。除非你所交付的软件并没有源代码或者是源代码本身，用户通常并不关心代码的质量。他们只需要软件能够满足功能需求并且没有缺陷。那么为什么软件开发人员和交付团队需要关心呢？简而言之，高质量代码的结果就是更少的 bug，以及会影响一些非功能性需求，如可维护性、可扩展性和可读性，这些会直接影响到业务的投资回报率（ROI）。在本章中，我们会关注能够监测代码质量，并把结果可视化的一些工具，这些工具能够帮助找出代码中存在的问题域。当你学完本章后，你会知道如何将代码质量工具集成到构建中。

之前，你学到了如何编写单元测试、集成测试和功能测试来验证 To Do 应用程序代码的正确性。代码覆盖分析（也叫作测试覆盖分析）就是找出代码中没有被测试用例覆盖的区域的过程。经验表明合理的代码覆盖率会对代码质量有间接的影响。

监测代码质量并不只有代码覆盖分析。每个团队或者组织都有自己的代码规范标准，这个标准包括从简单的代码格式方面，如空格的使用和缩进，到编程最佳实践。遵循这些规范，会使代码对于其他团队成员来说更加可读，从而提高其可维护性，并且防止潜在的 bug。尽管有这些好处，但是代码分析也取代不了一个有经验的同事所做的代码审查；当然，这是一种补充。

在一个大型并且一直在变化的项目中，专门用一个人来手动跟踪这些指标的做法是不切实际的。因此，以全局的观点来识别问题域并且跟踪进展很有必要，代码质量工具帮助你自动分析软件质量，并且提供有效的报告。在 Java 世界中，你可以选择使用广泛的开源工具或者商业解决方案，比如 Checkstyle、PMD、Cobertura、FindBugs 和 Sonar。大多数这些工具都以 Gralde 核心插件或者第三方插件的形式使用，并且被无缝地集成到构建中在本章中，你会在 To Do 应用程序中使用这些插件来监测代码质量。

## 12.1 将代码分析集成到构建中

我们先来回想一下在第 2 章中介绍的构建管道阶段。现在的情况如何？目前，你已经学到了如何编译代码，以及实现并执行不同类型的测试。这些 task 覆盖了提交阶段的前两个阶段。如果这些 task 失败了，构建就会自动失败。

尽管代码编译和测试的结果给出了关项目健康状况的一个基本观点，但是它并没有提供关于代码质量的任何反馈，如可维护性和代码覆盖率等。代码分析工具帮助你产生这些度量值。包括：

- 代码覆盖率
- 代码标准的遵守情况
- 不好的编码实践和设计问题
- 过度复杂、重复、强耦合的代码

在构建管道的上下文中，代码分析在提交阶段的前两个阶段之后执行，如图 12.1 所示。

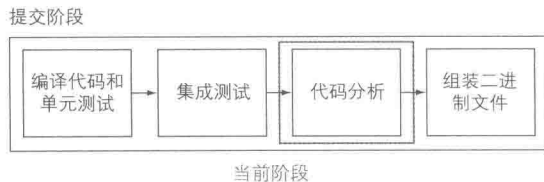


图 12.1 在部署管道上下文中的代码分析阶段

与运行集成测试和功能测试类似，执行代码分析的过程可能会花很长的时间，

这取决于代码库的大小和质量工具所审查的代码行数。因此，为此创建一个 Gradle 的 task 专用集是非常有帮助的，通过它来调用代码质量工具。这些 task 通常都由插件来提供，所以你自己不用创建。在实践中，你会想要单独运行某个代码质量分析的 task。比如，在开发的时候，你可能想要知道当前重构的类是否提高了代码测试覆盖率，而不需要运行其他漫长的代码质量检查过程。

代码质量检查的 task 不应该相互依赖，从而可以让它们并行运行。在 Java 项目的整个构建生命周期中，创建一个 check task 依赖所有代码质量检查的 task 是非常有帮助的，如图 12.2 所示。请记住图中的 task 名字不是构建中的真正 task 的名字，只是代表名称。

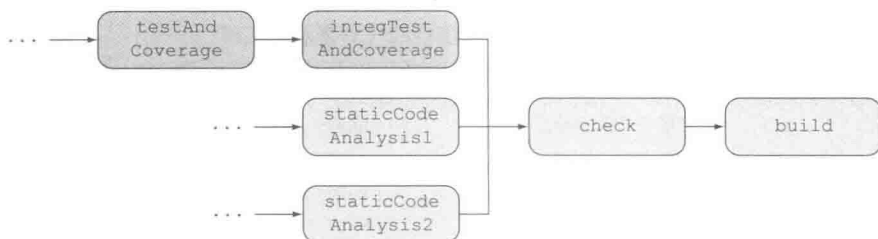


图 12.2 代码分析的 task 与标准的 Java 插件的 task 的关系

为了分离具有不同关注的代码，项目可能包括多个 source set。在第 7 章中已经学到了如何做，当为集成测试单独定义一个 source set 的时候。为了使其更清楚，针对某一个 source set 或者所有的 source set，你能够有选择性地执行代码分析。下面开始了解如何在 To Do 应用程序中衡量代码覆盖率。

## 12.2 衡量代码覆盖率

代码覆盖率并不能决定代码质量。它将代码中没有被测试到的执行分支暴露出来。讨论最多的度量值是产品代码的整体测试覆盖百分比。虽然 100% 的覆盖率是一个非常体面的数字，但是很少能达到，也不会让你满怀信心地认为代码逻辑是完全正确或者没有 bug 的。采用有意义的关键点验证，代码覆盖率的经验值是 70%~80%，你不应该盲目武断地追随。

当我在谈论代码覆盖率度量的时候，到底是什么意思呢？下面的列表给出了基本的覆盖率标准：

- 分支覆盖率：衡量可能被执行的分支（如 if/else 分支逻辑）的执行情况
- 语句覆盖率：衡量代码块中哪些语句被执行了
- 方法覆盖率：衡量执行测试的时候进入了哪些方法
- 复杂度度量：包、类和方法的圈复杂度（代码块中独立的分支数）

我们来看看一些已有的代码覆盖率工具以及相应的 Gradle 插件。

12.2.1 探索代码覆盖率工具

Java 系统创建了很多免费的和商业的代码覆盖率工具。它们所提供的功能通常根据生成度量值、产生的度量类型、覆盖质量以及在运行时生成报表（称作注入，*instrumentation*）的性能而有所不同。大多数这些工具都以 Ant 的 task 实现，使它们可以很容易地在构建中封装成 Gradle 的 task。其中一些工具有可用的插件。在撰写本书时，没有任何一个此类插件被集成到 Gradle 的核心代码库中。表 12.1 给出了 Gradle 对一些比较流行的代码覆盖率工具的支持。

表 12.1 对于 Java 和 Groovy 项目流行的代码覆盖率工具

名字	版权	注入类型	Gradle 支持
Cobertura	免费	离线字节码注入	<a href="https://github.com/eriwen/gradle-cobertura-plugin">https://github.com/eriwen/gradle-cobertura-plugin</a>
Clover	商业，但针对开源项目免费	源代码注入	<a href="https://github.com/bmuschko/gradle-clover-plugin">https://github.com/bmuschko/gradle-clover-plugin</a>
Emma	免费	离线字节码注入	没有复杂的插件支持；Ant task 可用
JaCoCo	免费	即时字节码注入	<a href="https://github.com/ajoberstar/grable-jacoco">https://github.com/ajoberstar/grable-jacoco</a>

比较代码覆盖率工具的功能

在这些工具中，如何选择适合自己的工具呢？首先，你需要一个一直在维护的工具以防工具本身出现 bug 需要被修复。对于 Clover，你可以很方便地就得到技术支持，因为这是一个收费工具。在免费的工具中，JaCoCo 是最活跃的，而 Emma 和 Cobertura 已经多年没有被更新了。

另外两个影响选择的因素是工具所能提供的度量和其质量。通常，覆盖率百分比用不同工具检查出来的结果不会有太大的偏差（最大约 3%~5%）。更重要的是其所能提供的度量维度。比如说 Emma，并不会产生分支覆盖方面的数据。如果考虑性能，JaCoCo 无疑是性能最好的。你可以在 SonarSource 主页（<http://www.sonarsource.org/pick-your-code-coverage-tool-in-sonar-2-2/>）上面找到更详细的关于这些工具的比较信息。

理解注入的方法

各工具之间提供代码覆盖率度量的方法不同。注入的主要工作就是在代码中插入一些指令，用于检测某个特定的代码行在测试的时候被执行过。我们来看看如下场景。一个名为 `ToDoItem` 的类代表一个 To Do 项：

```

public class ToDoItem {
    public final static int MAX_PRIORITY = 3;
    private int priority = 1;
    public int getPriority() {
        return priority;
    }

    public void setMaxPriority() {
        priority = MAX_PRIORITY;
    }

    public void bumpUpPriority() {
        if(priority < MAX_PRIORITY) {
            priority++;
        }
    }
}

```

priority 初始值为 1

只有在 item 的 priority 不是最大值的时候才能增加其 priority

针对 `ToDoItem` 类，你想要检查其测试覆盖率。为此，创建一个单元测试类，名为 `ToDoItemTest`。这个类中定义了一个测试方法，用于验证所设置的 `priority` 属性不能超过最大的 `priority` 值：

```

public class ToDoItemTest {
    @Test
    public void testBumpUpPriorityIfAlreadyMaxPriority() {
        ToDoItem toDoItem = new ToDoItem();
        toDoItem.setMaxPriority();
        assertEquals(toDoItem.getPriority(), ToDoItem.MAX_PRIORITY);
        toDoItem.bumpUpPriority();
        assertEquals(toDoItem.getPriority(), ToDoItem.MAX_PRIORITY);
    }
}

```

item 的 priority 已经是 3 了，因此没有增加

设置 item 的 priority 为 3

尝试增加 item 的 priority

尝试检查这个类的测试覆盖率，你会看到所有的方法都被覆盖到了，因为你在测试类中调用了它们中的每一个方法。只有一行代码没有被覆盖到，那就是 `priority++`。这是因为在测试方法中，在尝试增加 `To Do` 项的 `priority` 值之前，`priority` 的初始值是 3。对于 `ToDoItem` 类，为了达到 100% 的代码覆盖率，你不得增加另外一个测试方法，并且使用 `priority` 的初始值。

如表 12.1 所示，通过将指令添加到源代码、离线字节码和运行时二进制代码中等方法可以实现检测效果。这些方法有什么区别呢？源代码检测在编译前将指令添加到源代码中，从而跟踪哪部分代码被执行了。二进制代码检测是直接将指令添加到编译后的二进制代码中。运行时检测是将同样的指令添加到二进制代码中，但是是在代码被 JVM 类加载器加载的时候做这件事情的。

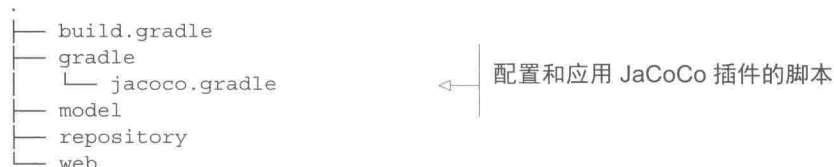
既然这些方法都能正常工作，并且产生的度量也不会有太大的差异，为什么还要纠结使用哪一种方法呢？在持续交付管道的上下文中，绑定可交付产品发生在代码分析阶段之后，你需要确保源代码或者二进制代码在编译过程中没有被改变，以免在产品环境中产生不一样的行为。因此，运行时检测应该更合适。在下一章中，我会介绍使用其他检测方法时如何避免问题。

接下来，你会学到如何使用 JaCoCo 和 Cobertura 两个插件来生成代码覆盖率报告。我们先来看看 JaCoCo 插件。

## 12.2.2 使用 JaCoCo 插件

JaCoCo (<http://www.eclemma.org/jacoco/>) 是 Java Code Coverage 的简称，是一个开源的工具，通过运行时二进制代码检测方法来测量代码覆盖率。为了实现这个目标，JaCoCo 在 JVM 类加载器上附加了一个 Java 代理，这个代理收集指令执行信息，并将这些信息写到文件中。JaCoCo 产生代码行和分支覆盖率报告。此外，其完全支持分析 Java 7 项目。

将软件功能拆分成不同的方面是一个最佳实践。在构建代码中也是如此。为了将主要的构建逻辑和代码覆盖逻辑分开，你可以在项目结构的根级别的 gradle 目录下创建一个新的构建脚本，叫作 `jacoco.gradle`。后面，你还会在这个目录下添加其他的 Gradle 构建文件。建立这个脚本之后，项目目录树如下：



这个脚本将充当容器来声明和配置 JaCoCo 插件。还记得在第 8 章中所介绍的将自己的插件添加到构建脚本的 `classpath` 中吗？现在我们就通过指定仓库和插件依赖来做同样的事情。如下清单展示了基本设置。

清单 12.1 将 JaCoCo 插件定义为脚本插件

```
buildscript {
    repositories {
        mavenCentral()
    }
}
```

将从 Maven Central 获取的 JaCoCo 插件添加到构建脚本的 classpath 中

```
dependencies {
    classpath 'org.ajoberstar:gradle-jacoco:0.3.0'
}

apply plugin: org.ajoberstar.gradle.jacoco.plugins.JacocoPlugin

jacoco {
    integrationTestTaskName = 'integrationTest'
}
```

← 将从 Maven Central 获取的 JaCoCo 插件添加到构建脚本的 classpath 中

← 按类型应用插件

← 定义集成测试的 task 名字，用于为集成测试 source set 生成代码覆盖率报告

现在你可以在其他项目中任意引用这个 `jacoco.gradle` 脚本插件了。在多项目构建中，可以通过在根项目构建脚本的 `subprojects` 配置块中定义一个引用来实现，如下面的清单所示。除了应用脚本插件外，你还需要告诉 JaCoCo 插件到什么地方去获得传递性依赖。目前，这个地方就是 Maven Central。

### 清单 12.2 在所有的子项目中应用 JaCoCo 脚本插件

```
subprojects {
    apply plugin: 'java'
    apply from: "$rootDir/gradle/jacoco.gradle"

    repositories {
        mavenCentral()
    }
}
```

← 应用 JaCoCo 脚本插件

← JaCoCo 插件所需要的传递性依赖是从 Maven Central 获取的

你的项目现在已经可以使用 JaCoCo 来生成代码覆盖率报告了。执行 `test` 类型的 task 的时候，JaCoCo 的代理会针对运行的测试类来收集相关的运行时信息。记住，你看不到任何额外的 task 来表示代码覆盖率数据被创建了。

试一试。执行完整的构建后，在每个子项目的 `build/jacoco` 目录下，会产生扩展名为 `exec` 的执行数据文件。下面的目录树显示了为 `repository` 子项目所产生的执行数据文件：

```

-
├── build.gradle
├── gradle
│   └── jacoco.gradle
├── model
├── repository
│   └── build
│       └── jacoco
│           ├── integrationTest.exec
│           └── test.exec
└── web
```

← JaCoCo 针对单元测试和集成测试的执行数据

JaCoCo 执行数据存储在二进制文件中。为了能够将代码覆盖率可视化，你需要生成一个 HTML 报告。JaCoCo 插件自动为每个子项目的每个 source set 都注册

了一个 task。在 repository 子项目中，这些 task 叫作 jacocoTestReport 和 jacocoIntegrationTestReport。图 12.3 显示了 jacocoTestReport 执行后产生的 HTML 报告。

repository > com.manning.gia.todo.repository > InMemoryToDoRepository Sessions

### InMemoryToDoRepository

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods
findAllActive()	<div></div>	87%	<div></div>	75%	1 3	0 7	0 1
findAllCompleted()	<div></div>	87%	<div></div>	100%	0 3	0 7	0 1
insert(ToDoItem)	<div></div>	100%	<div></div>	n/a	0 1	0 4	0 1
inMemoryToDoRepository()	<div></div>	100%	<div></div>	n/a	0 1	0 3	0 1
findAll()	<div></div>	100%	<div></div>	n/a	0 1	0 3	0 1
update(ToDoItem)	<div></div>	100%	<div></div>	n/a	0 1	0 2	0 1
delete(ToDoItem)	<div></div>	100%	<div></div>	n/a	0 1	0 2	0 1
findById(Long)	<div></div>	100%	<div></div>	n/a	0 1	0 1	0 1
Total	10 of 139	93%	1 of 8	88%	1 12	0 29	0 8

Created with JaCoCo 0.8.2.201302030002

图 12.3 JaCoCo 的 HTML 报告示例

点击 HTML 报告中的任意一个方法，都可以进入一个新页面，看到这个方法中对代码每一行标记的覆盖率信息。标记为绿色的行代表被测试覆盖到，红色代表未被覆盖到。

使用插件的默认设置，针对单元测试和集成测试会分别有一个覆盖率报告。如果你想要通过测试 source set 或多项目构建的所有项目得到一个关于项目整体覆盖的全局报告，则可以编写一个新的类型为 JaCoCoMerge 的 task。我们不会详细讨论这个 task。更多的细节，请看这个插件的文档。

很多企业使用的是 Cobertura 工具。为了比较 Cobertura 和 JaCoCo 所生成的报告，你需要在构建中配置好 Cobertura。

### 12.2.3 使用 Cobertura 插件

Cobertura (<http://cobertura.sourceforge.net/>) 是一个测量 Java 代码覆盖率工具采用对编译后的二进制代码添加指令的方式来工作。Cobertura 报告包括代码行和分支覆盖的百分比，也包括每个包的圈复杂度。与 JaCoCo 不同的是，Cobertura 不支持 Java7 项目。

你需要做与集成 JaCoCo 插件相同的基本设置。首先，创建一个新的构建脚本 cobertura.gradle，用来应用和配置 Cobertura 插件。然后，在构建的所有子项目中应用这个脚本插件。下面的项目结构显示了 gradle 目录下的新脚本：

```

-
├─ build.gradle
├─ gradle
│   └─ cobertura.gradle
├─ model
├─ repository
└─ web

```

← 配置和应用 Cobertura 插件的脚本



下面的清单显示了脚本插件 `cobertura.gradle` 中的内容。你会从 Maven Central 获取到这个插件并按类型应用它。

清单 12.3 定义 Cobertura 插件为脚本插件

```
buildscript {
    repositories {
        mavenCentral()
    }

    dependencies {
        classpath 'com.eriwen:gradle-cobertura-plugin:1.0'
    }
}

apply plugin: org.gradle.api.plugins.cobertura.CoberturaPlugin
```

将 Cobertura 插件添加到构建脚本的 classpath 中

按类型应用插件

与 JaCoCo 脚本插件类似，你现在可以在 To Do 应用程序的所有子项目中应用 Cobertura 脚本插件了。下面的清单显示了相关的代码修改。

清单 12.4 将 Cobertura 脚本插件应用到所有子项目中

```
subprojects {
    apply plugin: 'java'
    apply from: "$rootDir/gradle/cobertura.gradle"

    repositories {
        mavenCentral()
    }
}
```

应用 Cobertura 脚本插件

Cobertura 所需要的传递性依赖将从 Maven Central 获取

这个插件引入了两个新的 task，其中一个用来将检测指令插入到编译后的类文件中；另一个用来生成代码覆盖率报告。这两个 task 已经被很好地集成到了 Java 项目的构建生命周期中。构建时执行 check task 将会做相应的操作，从而生成代码覆盖率报告。为了做这些操作，插件从 `build/classes` 目录下将编译好的类文件复制到 `build/cobertura` 目录下，并往这些文件中插入指令，然后将覆盖率信息序列化到 `cobertura.ser` 文件中。结果报告被写到 `build/reports/cobertura` 目录下。如下目录树显示了执行构建后的相关文件：

```

├── build.gradle
├── gradle
│   └── cobertura.gradle
├── model
├── repository
│   └── build
│       ├── cobertura
│       │   └── main
│       │       ├── classes
│       │       ├── cobertura-test.ser
│       │       └── cobertura.ser
└── reports
    └── cobertura
```

注入后文件存放的目录

包含序列化后 Java 类的基本信息的数据文件



`testCoberturaReporttask` 负责生成代码覆盖率报告。在默认情况下, 这个 task 生成 HTML 报告, 但是也支持通过重新配置产生 XML 格式的代码覆盖率报告。图 12.4 显示了为 `repository` 项目的单元测试生成的 HTML 报告示例。



图 12.4 Cobertura 的 HTML 报告示例

关于代码覆盖率检查的工具, 我们就讨论这么多。本章中之后的内容, 你会在 Sonar 中重用这些生成的报告来跟踪代码覆盖质量。JaCoCo 和 Cobertura 是目前被广泛使用的开源的覆盖率检查工具。如果你打算使用另外的工具, 请参考表 12.1 中提供的链接。接下来, 我们会研究一些静态代码分析工具。

## 12.3 执行静态代码分析

软件开发团队成员通过代码审查来发现架构方面的问题、安全方面的缺陷和一些潜在的 bug。尽管 code review 对减少“技术债”的风险非常有帮助, 但是在大型的软件项目中, 这是一个代价很高且难于管理的过程。

我们当中有多少人厌倦了做 code review? 当然, 其很有用处——发现潜在的问题, 预防安全缺陷和软件漏洞——但是它很容易使成本过高并笨拙。一种低成本且能自动化地快速找出代码中问题的方法就是做静态代码分析。

静态代码分析就是仅仅分析源代码, 而没有必要真正地执行软件得到质量度量结果。其涵盖的范围可以从潜在的问题和必须严格遵守的代码规范, 到没必要的代码复杂度和不好的编码实践。我们来看一些这方面的工具。

### 12.3.1 探讨静态代码分析工具

用于检查低代码质量的开源工具有很多。在本节中, 我们会着重看看通过标准插件 Gradle 直接支持的工具。因为 Gradle 的运行时环境本身就包含这些工具, 所以很容易将它们集成到构建中并作为构建管道的一部分。表 12.2 显示了这些代码分析工具、它们所产生的分析结果, 以及将它们应用到构建中的 Gradle 插件名。

表 12.2 标准的 Gradle 静态代码分析工具

工具名	Gradle 插件	报告格式	描述
Checkstyle	checkstyle	仅 XML 格式	强制执行代码标准; 发现不好的设计问题、重复代码和漏洞模式
PMD	pmd	XML 和 HTML 格式	找到未被使用的、过度复杂的、低效的代码
CodeNarc	codenarc	文本、XML、HTML 格式	等同于 Groovy 项目中的 PMD
FindBugs	findbugs	XML、HTML 格式	发现潜在的漏洞、性能问题和不好的编码实践
JDepend	jdepend	文本、XML 格式	度量设计质量如扩展性、重用性和可维护性

这些插件及其特性刚开始看起来可能很空洞。为了区分它们的分析结果, 你会应用并配置它们来确定 To Do 应用程序的代码质量。在接下来的几节中, 你会尝试使用表 12.2 中的每一个工具, 除了 CodeNarc。

#### 准备子项目

与前面所用的代码覆盖率检查工具一样, 你会为每一个静态代码分析工具编写一个脚本插件, 以实现分离关注点。大多数这些 Gradle 插件都需要一个配置文件位于 `config/<工具名>` 目录下, 用来定义分析规则。为了做好充分的准备, 你需要在多项目构建的根级别创建 `config` 目录, 并定义一个可以被所有子项目使用的属性:

```
.
├── build.gradle
├── config
│   ├── checkstyle
│   └── ...
```

◀ 代码质量分析插件使用的默认配置目录



每个工具的依赖并没有被包含在 Gradle 分发包中。在运行时环境中，它们会自动地从仓库中获取。为此，你需要将 Maven Central 配置好。如下清单显示了对根项目的构建脚本你需要做的修改。

#### 清单 12.5 准备将静态代码分析插件集成到子项目中

```

subprojects {
    ext.configDir = new File(rootDir, 'config')
    // Apply static code analysis script plugin
    repositories {
        mavenCentral()
    }
}
  
```

用于指向配置目录的扩展属性（配置目录可以起任意名字）

配置 Maven Central 来获取插件依赖

#### 生成 HTML 报告

这些插件生成的默认报告格式是 XML 的。虽然 XML 对于后续的工具如 Sonar 做处理非常方便，但是其不利于阅读。更方便阅读的格式是 HTML。遗憾的是，并不是你使用的所有插件都支持 HTML 格式的输出。但是，你可以通过 XML 到 HTML 的转换轻松地达到这个效果。为此，你需要在 BUILDSRC 目录下创建一个可重用的 task，它使用的是 Ant 的 XSLT task，如下面的清单所示。

#### 清单 12.6 通过 XSLT 生成 HTML 报告的 task

```

package com.manning.gia

import org.gradle.api.DefaultTask
import org.gradle.api.tasks.*

class XsltReport extends DefaultTask {
    @InputFile @Optional File inputFile
    @InputFile File xslStyleFile
    @Input @Optional Map<String, String> params = [:]
    @OutputFile File outputFile

    XsltReport() {
        onlyIf {
            inputFile.exists()
        }
    }
  }
  
```

只在输入文件存在是才生成 HTML 报告，否则跳过

```

    }
}

@TaskAction
void start() {
    ant.xslt(in: inputFile, style: xslStyleFile, out: outputFile) {
        params.each { key, value ->
            ant.param(name: key, expression: value)
        }
    }
}
}
}

```

使用 Ant XSLT task 生成 HTML 报告

现在你已经为在构建中应用静态代码分析工具做好了准备。首先来使用 Checkstyle 插件。

### 12.3.2 使用 Checkstyle 插件

在企业级项目中，引入代码标准来定义统一的源代码格式、结构和注解是非常有帮助的。作为一个产品，这会帮你产生更多的可读的、可维护的代码。这就是 Checkstyle (<http://checkstyle.sourceforge.net/>) 的作用。这个项目最开始是一个检查没有遵循代码规范的源代码的工具。渐渐的，其所提供的功能已经扩展到检查设计问题、重复代码以及一些常见的 bug 模式。

首先在 gradle 目录下创建一个新的脚本文件 `checkstyle.gradle`，用来充当 Checkstyle 脚本插件。在这个脚本插件中，你会应用标准的 Gradle 的 Checkstyle 插件，并且对其版本和运行时行为做一些配置。下面的清单演示了如何使用 Sun 公司的代码规范，通过规则集在 `config/sun_checks.xml` 文件中定义。

清单 12.7 应用和配置 Checkstyle 插件为脚本插件

```

apply plugin: 'checkstyle'

ext.checkstyleConfigDir = "$configDir/checkstyle"

checkstyle {
    toolVersion = '5.6'
    configFile = new File(checkstyleConfigDir, 'sun_checks.xml')
    ignoreFailures = true
    showViolations = false
}

```

应用 Gradle 标准的 Checkstyle 插件

定义 Checkstyle 5.6 版本依赖

定义使用的规则集

阻止 Gradle 把所有违规的都打印在命令行

只要发现有违规的就改变构建失败的默认行为

有了 Checkstyle 的这个基本设置，你现在就可以将这个脚本插件应用到多项目构建的所有子项目中了，如下面的清单所示。

## 清单 12.8 将 Checkstyle 插件应用到所有子项目中

```

subprojects {
    apply plugin: 'java'

    ext.configDir = new File(rootDir, 'config')
    apply from: "$rootDir/gradle/checkstyle.gradle"

    repositories {
        mavenCentral()
    }
}

```

应用 Checkstyle 脚本插件

现在执行一个完整的构建就能够在每个子项目的 build/reports/checkstyle 目录下为项目中所有的 source set 生成 XML 报告了。下面的命令行输出显示了 repository 子项目相应的信息。

```

$ gradle build
...
:repository:checkstyleIntegrationTest
Checkstyle rule violations were found. See the report at:
  file:///Users/Ben/checkstyle/repository/build/reports/checkstyle/
  integrationTest.xml
:repository:checkstyleMain
Checkstyle rule violations were found. See the report at:
  file:///Users/Ben/checkstyle/repository/build/reports/checkstyle/main.xml
:repository:checkstyleTest
Checkstyle rule violations were found. See the report at:
  file:///Users/Ben/checkstyle/repository/build/reports/checkstyle/test.xml
...

```

主 source set 的 Checkstyle task 和违规提醒

集成测试 source set 的 Checkstyle task 和违规提醒

测试 source set 的 Checkstyle task 和违规提醒

当然，默认的报告位置是可以重新配置的。关于这个插件的更多信息及其他配置项，请参考 Checkstyle 插件的 DSL 文档。

Checkstyle 插件当前没有提供 HTML 报告的生成功能。清单 12.9 演示了如何为每个 source set 创建 XsltReport 类型的 task。为默认的 Checkstyletask 定义一个依赖，这样就能保证初始的 XML 报告生成之后，运行 HTML 报告的 task。你可以将这段代码直接添加到 checkstyle.gradle 中。

## 清单 12.9 为所有的 source set 生成 Checkstyle 的 HTML 报告

```

import com.manning.gia.XsltReport

afterEvaluate {
    plugins.withType(CheckstylePlugin) {
        sourceSets.each { sourceSet ->
            String capitalizedSourceSetName = sourceSet.name.capitalize()
            String reportTaskName = "checkstyle${capitalizedSourceSetName}Report"
        }
    }
}

```

只为应用了 Checkstyle 插件的项目执行逻辑

确保项目中的所有 source set 都被评估

遍历项目中的所有 source set

为每个  
source set  
定义并配置  
一个增强的  
生成 HTML  
报告 task

```
String reportDir = "$reporting.baseDir/checkstyle"
XsltReport reportTask = tasks.create(reportTaskName, XsltReport)

reportTask.with {
    description = "Generates a Checkstyle HTML report for
        ↳ ${sourceSet.name} classes."
    dependsOn tasks."checkstyle${capitalizedSourceSetName}"
    inputFile = new File(reportDir, "${sourceSet.name}.xml")
    xslStyleFile = new File(checkstyleConfigDir,
        ↳ 'checkstyle-noframes.xsl')
    outputFile = new File(reportDir,
        ↳ "checkstyle_${sourceSet.name}.html")
}

check.dependsOn reportTaskName
```

让验证的 task 依赖于产生报告的 task

在 XML 到 HTML 转换的过程中，你使用了一个 Checkstyle 分发包提供的 XSL 文件。它生成一个独立的 HTML 报告。执行 `gradle build` 生成的报告类似于图 12.5。

CheckStyle Audit	
Designed for use with <a href="#">CheckStyle</a> and <a href="#">Ant</a> .	
Summary	
Files	Errors
3	132
Files	
Name	Errors
/Users/Ben/Dev/books/gradle-in-action/code/chapter12/checkstyle/repository/src/main/java/com/manning/gia/todo/repository/H2ToDoRepository.java	98
/Users/Ben/Dev/books/gradle-in-action/code/chapter12/checkstyle/repository/src/main/java/com/manning/gia/todo/repository/InMemoryToDoRepository.java	24
/Users/Ben/Dev/books/gradle-in-action/code/chapter12/checkstyle/repository/src/main/java/com/manning/gia/todo/repository/ToDoRepository.java	10
File /Users/Ben/Dev/books/gradle-in-action/code/chapter12/checkstyle/repository/src/main/java/com/manning/gia/todo/repository/H2ToDoRepository.java	
Error Description	Line
File does not end with a newline.	0
Missing package-info.java file.	0
Using the '*' form of import should be avoided - java.sql.*.	5
Missing a Javadoc comment.	9
File contains tab characters (this is the first instance).	10
Method 'findAll' is not designed for extension - needs to be abstract, final or empty.	10
Line has trailing spaces.	16

图 12.5 Checkstyle 的 HTML 报告示例

### 12.3.3 使用 PMD 插件

PMD (<http://pmd.sourceforge.net/>) 与 Checkstyle 类似，但是它专门关注代码闻

题如无用代码或者重复代码、过于复杂的代码以及可能的 bug。PMD 分包中包含了很多专用的规则集——例如，JEE 组件、Web 框架如 JSF、移动技术如 Android。

与设置 Checkstyle 插件类似。首先，创建一个脚本插件，名字为 pmd.gradle，用来应用 PMD 插件。然后，在 To Do 应用程序的所有子项目中配置和应用这个脚本插件。下面的清单展示了如何使用 PMD 的默认版本和规则集。

清单 12.10 应用和配置 PMD 插件为脚本插件

```
apply plugin: 'pmd'
pmd {
    ignoreFailures = true
}
```

应用 Gradle 标准的 PMD 插件

只要发现有违规的就改变构建失败的默认行为

因为这个脚本插件与集成 Checkstyle 插件的方法非常类似，所以这里我们就不探讨如何将其应用到子项目中了。PMD 插件本身就支持生成 XML 和 HTML 报告，所以你不需要做其他额外的工作。如下命令行输出显示了 PMD 对 repository 子项目的所有 source set 生成的结果：

```
$ gradle build
...
:repository:pmdIntegrationTest
:repository:pmdMain
3 PMD rule violations were found. See the report at:
file:///Users/Ben/pmd/repository/build/reports/pmd/main.html
:repository:pmdTest
...
```

最后生成的 HTML 报告位于 build/reports/pmd 目录下，类似于图 12.6。

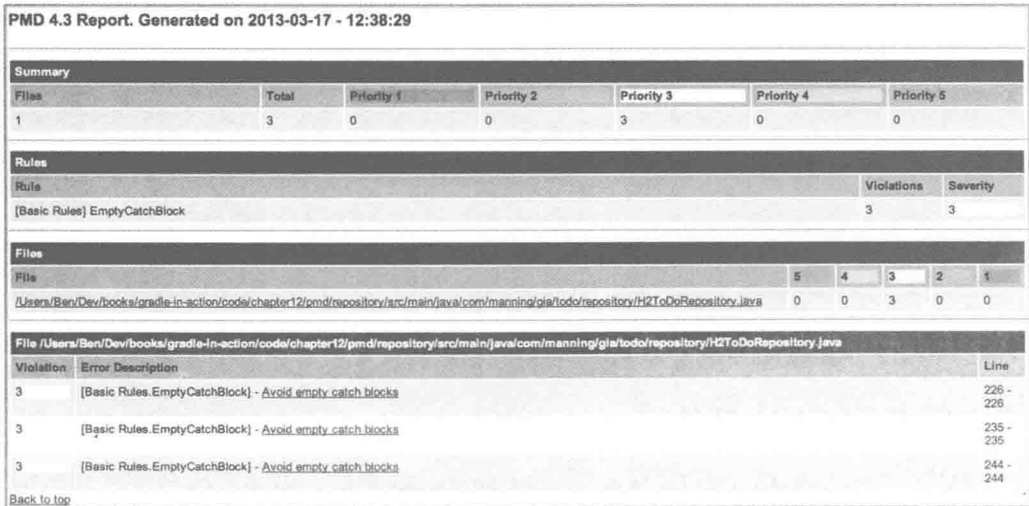


图 12.6 PMD 的 HTML 报告示例



有时候, 你可能只想要生成一种类型的报告。在开发期间需要验证开发进度时会出现这种情况。PMD 的 task 可以对使用或禁用特定报告类型进行配置, 如下面的清单所示。同样的方法可以被应用到与代码质量相关的 task 上面。

**清单 12.11 配置生成的 PMD 报告类型**

```
tasks.withType(Pmd) { ← 应用到所有 org.gradle.api.plugins.quality.Pmd 类型的 task
    reports {
        xml.enabled = false
        html.enabled = true
    }
}
```

禁用 XML 报告, 采用 HTML 报告

### 12.3.4 使用 FindBugs 插件

FindBugs (<http://findbugs.sourceforge.net/>) 是一个用于发现潜在的 bug 和不好的编码实践的静态代码分析工具。bug 包括如 equals/hashCode 实现问题、多余的 null 检查, 甚至是性能问题。与前面几种分析工具不同, FindBugs 操作的是 Java 二进制代码, 而不是源代码。你会发现操作二进制代码使得分析比源代码分析更耗时。对于比较大的项目, 对这个时间的花费做好心理准备吧。

清单 12.2 显示了 FindBugs 插件可以很容易地集成到构建中。欲了解更多的配置信息, 请参考 Gradle 的 DSL 指南。目前, FindBugs 插件只支持生成 XML 或 HTML 报告。

**清单 12.12 应用和配置 FindBugs 插件为脚本插件**

```
apply plugin: 'findbugs' ← 应用 Gradle 标准的 FindBugs 插件

findbugs {
    toolVersion = '2.0.1'
    ignoreFailures = true
    effort = 'max' ← 定义分析的 effort 级别, 级别越高, 分析越细致
}

tasks.withType(FindBugs) {
    reports {
        xml.enabled = false
        html.enabled = true
    }
}
```

定义 org.gradle.api.plugins.quality.FindBugs 类型的 task 来生成 HTML 报告

在所有的子项目下执行整个构建会为所有的 source set 生成预期的报告。如下命令行输出显示了对于 repository 子项目具有代表性的 task:

```
$ gradle build
...
:repository:findbugsIntegrationTest
:repository:findbugsMain
FindBugs rule violations were found. See the report at:
➡ file:///Users/Ben/findbugs/repository/build/reports/findbugs/main.html
:repository:findbugsTest
...
```

执行 FindBugs 的 task 后，你能在 `/build/reports/findbugs/` 目录下找到每个 source set 对应的 HTML 报告。图 12.7 展示了 main source set 的报告。

## FindBugs Report

### Project Information

Project:

FindBugs version: 2.0.1

Code analyzed:

- /Users/Ben/Dev/books/gradle-in-action/code/chapter12/findbugs/repository/build/classes/main/com/manning/gia/todo/repository/H2ToDoRepository.class
- /Users/Ben/Dev/books/gradle-in-action/code/chapter12/findbugs/repository/build/classes/main/com/manning/gia/todo/repository/InMemoryToDoRepository.class
- /Users/Ben/Dev/books/gradle-in-action/code/chapter12/findbugs/repository/build/classes/main/com/manning/gia/todo/repository/ToDoRepository.class

### Metrics

208 lines of code analyzed, in 3 classes, in 1 packages.

Metric	Total	Density*
High Priority Warnings		0.00
Medium Priority Warnings	3	14.42
<b>Total Warnings</b>	<b>3</b>	<b>14.42</b>

(\* Defects per Thousand lines of non-commenting source statements)

图 12.7 FindBugs 的 HTML 报告示例

## 12.3.5 使用 JDepend 插件

静态代码分析工具 JDepend (<http://clarkware.com/software/JDepend.html>) 为衡量代码的设计质量产生分析结果。它会扫描 Java 代码的所有包、计算类和接口的数量，并确定它们的依赖。这个信息帮助你识别不必要的组件或者强耦合部分。

清单 12.13 显示了如何应用和配置 Gradle 标准的 JDepend 插件。你可以选择 XML 或者纯文本格式的报告。默认会生成 XML 报告并且不需要任何额外的配置。

这个清单还显示了如何切换报告格式。

**清单 12.13 应用和配置 JDepend 插件为脚本插件**

```
apply plugin: 'jdepend'
```

应用 Gradle 标准的 JDepend 插件

```
def configDir = new File(rootDir, 'config')
ext.jdependConfigDir = "$configDir/jdepend"

jdepend {
    toolVersion = '2.9.1'
    ignoreFailures = true
}

tasks.withType(JDepend) {
    reports {
        text.enabled = false
        xml.enabled = true
    }
}
```

配置 org.gradle.api.plugins.quality.JDepend 类型的 task 来生成 XML 报告

这段代码被应用到 subprojects 配置块之后，就会为每个 source set 生成一个 XML 格式的 report。如下命令行输出显示了需要被执行的 JDepend 的 task 来生成报告：

```
$ gradle build
...
:repository:jdependIntegrationTest
:repository:jdependMain
:repository:jdependTest
...
```

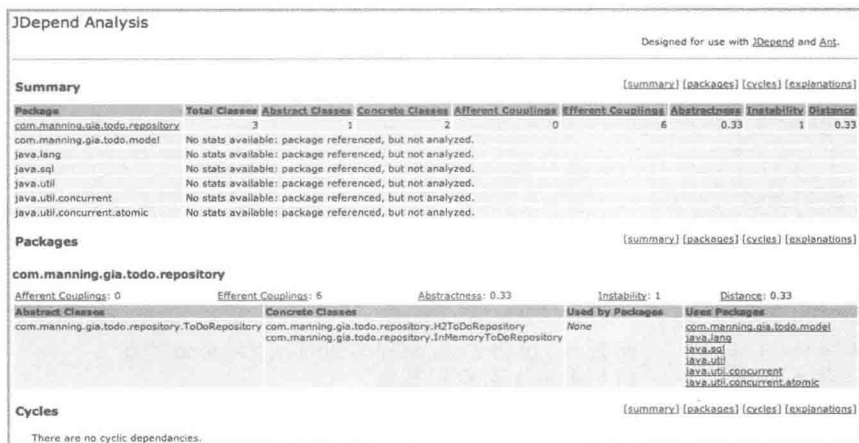
遗憾的是，你不能直接生成 HTML 格式的 report，但是可以利用 12.3.1 节中介绍的自定的 XSTL 的 task 来达到生成 HTML 报告的效果。JDepend 分发包中包含了一个 XSL 文件可以被使用。你可以在本书的源代码中找到详细的例子。生成 HTML 报告后，在 build/reports/jdepend 目录下会至少包含一个文件，效果与图 12.8 类似。

JDepend 的另一个很强大的功能就是能够可视化用图的形式包之间的依赖关系。JDepend 分发包中包含一个 XSL 文件，用来将 XML 报告转换成 Graphviz 的 DOT 文件。从本书的源代码中可以找到完整的例子。

## 12.4 集成Sonar

你已经看到了如何使用各种代码分析工具为项目做代码检查和分析。这些工具所提供的报告都需要单独做检查。在每个构建当中，已经存在的报告可能会被删除，并且有新的报告被创建，所以你很难看出来代码质量在一段时间内是提高了还是降

低了。为此，你需要一个工具能够集中监控、可视化和整合报告信息。Sonar (<http://www.sonarsource.org/>) 就是这样的一个工具。



JDepend Analysis									
Designed for use with JDepend and Ant.									
Summary									
Package	Total Classes	Abstract Classes	Concrete Classes	Afferent Couplings	Efferent Couplings	Abstractness	Instability	Distance	
com.manning.gia.todo.repository	3	1	2	0	6	0.33	1	0.33	
com.manning.gia.todo.model	No stats available: package referenced, but not analyzed.								
java.lang	No stats available: package referenced, but not analyzed.								
java.sql	No stats available: package referenced, but not analyzed.								
java.util	No stats available: package referenced, but not analyzed.								
java.util.concurrent	No stats available: package referenced, but not analyzed.								
java.util.concurrent.atomic	No stats available: package referenced, but not analyzed.								
Packages									
com.manning.gia.todo.repository									
Afferent Couplings: 0      Efferent Couplings: 6      Abstractness: 0.33      Instability: 1      Distance: 0.33									
Abstract Classes	Concrete Classes	Used by Packages	Uses Packages						
com.manning.gia.todo.repository.ToDoRepository	com.manning.gia.todo.repository.IToDoRepository com.manning.gia.todo.repository.InMemoryToDoRepository	None	com.manning.gia.todo.model java.lang java.sql java.util java.util.concurrent java.util.concurrent.atomic						
Cycles									
There are no cyclic dependencies.									

图 12.8 JDepend 的 HTML 报告示例

Sonar 是一个开源的、基于 Web 平台，用于管理和监控代码质量报告的工具，包括编码规则、单元测试覆盖率、源代码文档，以及架构方面如可维护性和“技术债”等。它和前面介绍的大多数工具都能很好地集成，包括 JaCoCo、Checkstyle 和 PMD。如果需要支持非常规的工具或者语言，Sonar 也可以通过插件的形式扩展。

Sonar 将代码质量相关的信息记录在一个中心数据库中。在其默认配置中，Sonar 有一个内嵌的 H2 实例，不需要做任何其他的设置。虽然 H2 是探讨 Sonar 功能的很好的开始，但是在产品环境中，它被推荐配置更具有扩展性的解决方案，如 MySQL 或者 Oracle。Gradle 通过 Sonar Runner 插件和 Sonar 很好地集成。Sonar Runner 能够分析项目，然后将收集到的分析结果通过 JDBC 的形式写到 Sonar 数据库中。你可以直接打开项目的 Dashboard，并查看整合的代码质量的实时和历史报告。图 12.9 显示了 Gradle 和 Sonar 的交互。

如图所示，Sonar 可以从外部获得这些报告信息。各种相关的规则被定义在 *quality profile* 里面。可以直接在 Sonar 上面配置一个 *quality profile*，其中定义了你想要运行的代码分析工具以及它们的验收标准。比如，你可以说，“一个类必须有超过 70% 的文档 API。”

这意味着 Sonar 不仅能让你选择在项目上应用哪些规则和相应的阈值，还允许你选择想要使用的分析工具。我们采用默认值，比如使用 Sonar 预先配置的 *quality profile*。它自动使用 Checkstyle 和 PMD 工具并采用 119 条规则来分析项目，不需要配置标准的 Gradle 插件。

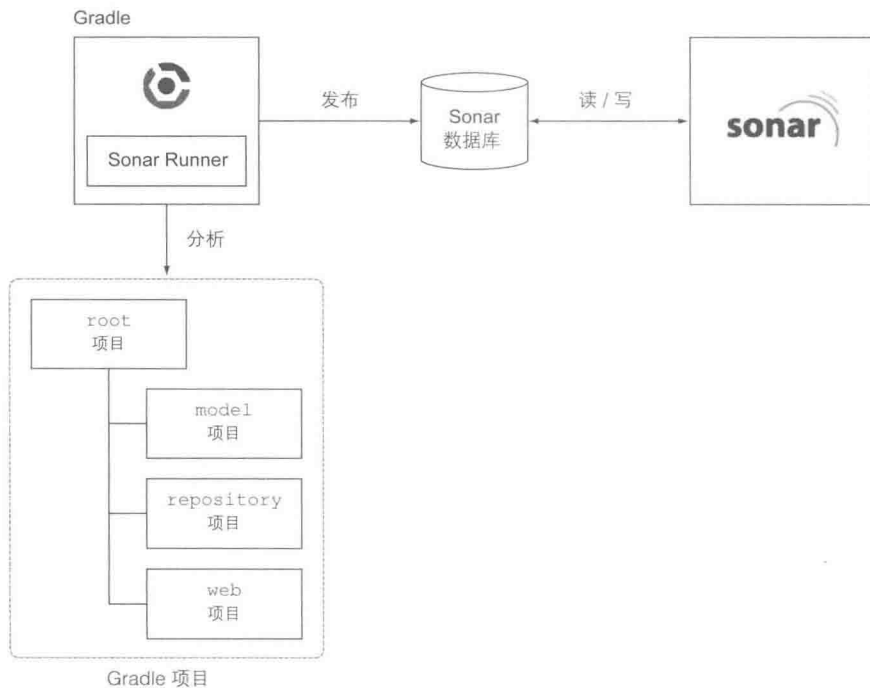


图 12.9 Gradle 和 Sonar 的交互

### 重用已存在的代码分析报告

之前，你使用 Gradle 的静态代码分析插件为代码做过静态分析并产生报告。你可能会问是否可以配置 SonarRunner 插件来重用这些报告。在撰写本书时，Sonar 并没有提供相应的机制来将这些报告导入到其数据库中。所以，你不得不依赖在 quality profile 中配置的 Checkstyle、PMD 和 FindBugs 插件。唯一不同的就是代码覆盖率规则，我们会在后面讨论。

## 12.4.1 安装并运行 Sonar

安装 Sonar 非常简单。到 Sonar 的主页，下载最新的版本（撰写本书时最新的版本为 3.5），然后解压缩 ZIP 文件。根据所使用的操作系统，你可以在 `$SONARHOME/bin` 目录下找到启动脚本来启动 SonarWeb 服务器。假设你想在 Mac OS X 64 位操作系统上面启动 Sonar。在命令行中切换到 Sonar 的 `bin` 目录并启动服务器：

```
$ cd $SONARHOME/bin/macosx-universal-64
$ ./sonar.sh start
Starting sonar...
Started sonar.
```

第一次启动 Sonar 需要大概 30 秒钟。你可以在 `<SONARHOME>/logs/sonar.log` 文件中查看到启动过程的日志。启动成功后，默认可以通过访问 `http://localhost:9000` 打开 Dashboard，如图 12.10 所示。

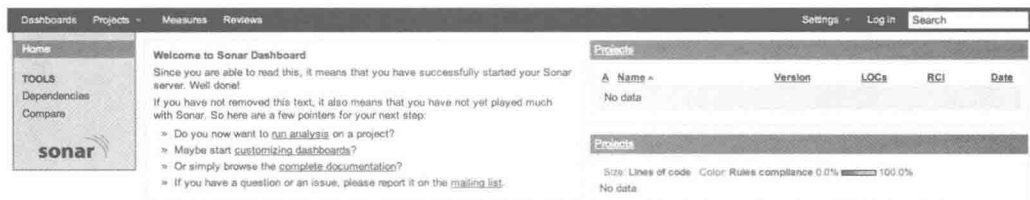


图 12.10 Sonar 的 Dashboard

在截图的右上角，有个 Projects 面板。因为你还没有发布任何报告，所以这个面板里面没有任何数据。接下来，你会通过改变 Gradle 构建的一些配置来使用 Gradle Runner 插件。

## 12.4.2 使用 Sonnar Runner 分析项目

在单项目或者多项目构建中，Sonnar Runner 插件都被推荐用来分析源代码。这个插件与 Sonar 3.4 及以上版本完全兼容，而且如果你使用的是 Sonar 默认配置，那么只需要做很少的设置。

### 使用低于 3.4 版本的 Sonar

如果你需要支持 3.4 版本之前的 Sonar 实例，则需要依赖 Gradle 标准的 Sonar 插件。因为本章我们只讨论 Sonnar Runner 插件，所以对于 Gradle 标准的 Sonar 插件信息请参考 Gradle 的官文档：[http://www.gradle.org/sonar\\_plugin](http://www.gradle.org/sonar_plugin)。

清单 12.14 显示了使用 Sonnar Runner 插件所需要改变的配置。将这个插件应用到根项目中，并且配置了一些基本的属性，比如项目的名字和描述、编码规范等。如果你没设置 `sonar.projectName` 和 `sonar.projectDescription` 属性，那么这些信息将从 Gradle 项目属性 `name` 和 `description` 中获得。

清单 12.14 应用和配置 Sonar Runner 插件

```

apply plugin: 'sonar-runner'
sonarRunner {
    sonarProperties {
        property 'sonar.projectName', 'todo'
        property 'sonar.projectDescription', 'A task management application'
    }
}

```

应用 Gradle 标准的 Sonar Runner 插件

设置  
Sonar  
项目和  
描述

```

    }
}

subprojects {
    ...

    sonarRunner {
        sonarProperties {
            property 'sonar.sourceEncoding', 'UTF-8'
        }
    }
}

```

← 改变源文件的系统默认编码为 UTF-8

这就是你开始使用 Sonar 的所有配置了。使用默认的 quality profile 分析项目，并将分析结果报告发布到 Sonar 数据库中。你可以通过执行插件所提供的 sonarRunner task 来开始这个过程：

```

$ gradle sonarRunner
...
:sonarRunner
07:01:25.468 INFO    .s.b.b.BatchSettings - Load batch settings
07:01:25.572 INFO    o.s.h.c.FileCache - User cache:
➡ /Users/Ben/.sonar/cache
07:01:25.577 INFO    atchPluginRepository - Install plugins
07:01:26.645 INFO    .s.b.b.TaskContainer - ----- Executing
➡ Project Scan
07:01:27.235 INFO    b.b.JdbcDriverHolder - Install JDBC driver
07:01:27.238 INFO    .b.ProjectExclusions - Apply project exclusions
07:01:27.242 WARN    .c.p.DefaultDatabase - H2 database should be used
➡ for evaluation purpose only
07:01:27.243 INFO    o.s.c.p.Database - Create JDBC datasource for
➡ jdbc:h2:tcp://localhost/sonar
...
07:01:38.122 INFO    .b.p.UpdateStatusJob - ANALYSIS SUCCESSFUL, you
➡ can browse http://localhost:9000
...

```

命令行输出给出了很多信息，使你能够更进一步地看到哪个项目和目录被分析过了，以及使用了什么样的代码质量分析工具等。执行成功后，刷新 Sonar 的 Dashboard，你会看到 todo 项目，包括一些基本的信息如代码行数、合规百分比、上一次分析日期。点击项目名，你就能看到更详细的视图，如图 12.11 所示。



图 12.11 Sonar 项目的 Dashboard

项目的 Dashboard 使你能够看到代码分析报告的一些信息，如代码复杂度、规则冲突和代码覆盖率。你可以点击这些选项查看更详细的信息。

这个插件允许你通过设置属性的方式来改变 Sonar 配置的每个方面。其中一组属性被用来设置 Sonar 所使用的数据库。这里我就不一一指出了，你可以通过阅读插件文档来使用相应的配置。我想讨论的另一组属性，与你的项目直接相关。为了将集成测试 source set 添加到 Sonar 中，你需要告诉 Sonar 这些信息。如下清单演示了如何将 source set 目录添加到适当的 Sonar 属性中。

#### 清单 12.15 为分析添加自定义的 source set

```
project(':repository') {
    ...
    sonarRunner {
        sonarProperties {
            properties['sonar.tests'] += sourceSets.integrationTest.
                                     allSource.srcDirs
        }
    }
}
```

将集成测试 source set 添加到默认的分析 source set 中

### 12.4.3 将代码覆盖率报告发布到 Sonar

如果仔细看 Sonar 的 Dashboard，在图 12.12 中你就会发现单元测试覆盖率为 100%，但是代码覆盖率却为 0%。代码覆盖率在 Sonar 的 Dashboard 中是特别有帮助的一个信息。如果可以的话，你可以直接浏览每个类来看看覆盖率的直观表示情况。



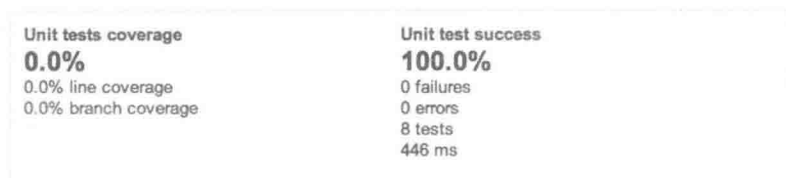


图 12.12 Sonar 中的单元测试覆盖率部件

在本章前面的部分，我们讨论了如何使用 JaCoCo 和 Cobertura 来生成代码覆盖率报告。你可以通过 Sonar Runner 插件来重用这些报告。Sonar 支持 JaCoCo、Emma、Cobertura 和 Clover 工具生成的遵从 JUnit 的 XML 格式的覆盖率报告。你所需要做的就是添加一些配置。下面先来看看重用 JaCoCo 报告。

### 重用 JaCoCo 报告

在 Sonar 中默认的代码覆盖率工具就是 JaCoCo。你只需要告诉 SonarRunner 在什么地方能找到 JaCoCo 报告文件。Sonar 要求你提供两个属性来设置报告文件的路径：一个针对单元测试 (`sonar.jacoco.reportPath`)，另一个针对集成测试 (`sonar.jacoco.itReportPath`)。幸运的是，你没必要手动设置这两个属性，JaCoCo 插件为你预先配置了 Sonar Runner。你会发现在命令行输出中，Sonar Runner 注册了 JaCoCo 报告文件：

```
$ gradle build sonarRunner
...
07:28:20.300 INFO o.s.p.j.JaCoCoPlugin - Analysing /Users/Ben/sonar-
jacoco/repository/build/jacoco/integrationTest.exec
07:28:23.812 INFO o.s.p.j.JaCoCoPlugin - Analysing /Users/Ben/sonar-
jacoco/repository/build/jacoco/test.exec
...
```

现在附加的信息也已经被发送到 Sonar 了，你可以刷新 Dashboard 来看看正确的单元测试代码覆盖率报告了。为了能看到集成测试覆盖率，你需要在 Dashboard 中添加集成测试部件。图 12.13 显示了这两个代码覆盖率部件。



图 12.13 Sonar 中的单元测试和集成测试代码覆盖率部件

在撰写本书时，Sonar 不支持添加功能测试覆盖率部件。为了弥补这个不足，你可以将功能测试的代码覆盖率注册为集成测试覆盖率添加到 Sonar 中。接下来，我们来看看如何重用 Cobertura 报告并集成到 Sonar 中去。

### 重用 Cobertura 报告

在使用 Cobertura 最开始生成代码覆盖率报告的时候，所选择的报告格式为 HTML 格式。Sonar 只能处理 XML 格式的报告文件，所以你需要重新配置 Cobertura 插件。在 `cobertura.gradle` 文件中，设置如下扩展属性：

```
cobertura {
    format = 'xml'
}
```

为了告诉 Sonar Runner 采用不同的覆盖率处理机制，你需要设置一个新的属性：`sonar.core.codeCoveragePlugin`。记住，只有当你想要重用非 JaCoCo 工具生成的报告时，才需要指定这个属性。除了这个属性之外，你还需要告诉 Sonar Runner 去哪里找 Cobertura 报告文件。如下清单显示了如何重用 Cobertura 的单元测试报告。

#### 清单 12.16 配置 Sonar Runner 插件来重用 Cobertura 报告

```
subprojects {
    ...

    sonarRunner {
        sonarProperties {
            property 'sonar.sourceEncoding', 'UTF-8'
            property 'sonar.core.codeCoveragePlugin', 'cobertura'
        }

        tasks.withType(SourceTask) { task ->
            if(task.name == 'testCoberturaReport') {
                property 'sonar.cobertura.reportPath',
                    new File(task.reportDir, 'coverage.xml')
            }
        }
    }
}
```

改变 Sonar 默认  
的测试覆盖率工  
具为 Cobertura

为 JaCoCo  
单元测试报  
告文件定义  
Sonar 属性

sonarRunner task 会告诉你被解析的报告文件的详细信息：

```
$ gradle build sonarRunner
...
12:10:42.895 INFO p.PhasesTimeProfiler - Sensor CoberturaSensor...
12:10:42.896 INFO .p.c.CoberturaSensor - parsing /Users/Ben/sonar-
  ➤ cobertura/repository/build/reports/cobertura/main/coverage.xml
12:10:42.949 INFO p.PhasesTimeProfiler - Sensor CoberturaSensor
  ➤ done: 54 ms
...
```

集成其他第三方工具如 Emma 或者 Clover 生成的代码覆盖率报告的过程基本相同。关键是到 Sonar 在线文档中查找相应的配置属性。

## 12.5 总结

不好的代码质量和过多的“技术债”不可避免地会导致开发者的效率降低、延期交付和更多的缺陷。除了通过同行做设计审查和 code review 外，在项目早起阶段就引入代码规范和借助于静态代码分析工具对项目的代码质量进行监控是非常重要的。Java 系统中有非常多的开源工具供选择来产生代码质量报告。Gradle 通过提供标准的或者第三方插件的方式，使集成这些工具到构建过程变得非常容易。

代码覆盖率衡量了那些被测试执行过的代码的百分比，并且发现那些没有被测试执行过的代码区域。高的代码覆盖率很明显能够帮助你更轻松的重构、维护代码、增强代码库。我们看过了如何应用和配置 JaCoCo 和 Cobertura 两个插件。虽然这两种工具都能够生成代码覆盖率报告，但是 JaCoCo 具有更好的灵活性、性能表现和开发者支持。

静态代码检查工具帮助你实施代码规范、发现不好的编码实践和潜在的软件缺陷。Gradle 提供了一系列的标准插件供你选择。我们讨论了如何应用、配置并执行这些插件。

使用 Sonar 可以跟踪、评估并且提高代码质量。Sonar 提供了一系列开箱即用的静态代码分析工具，定义了一些质量规则及其阈值。如果你需要管理更多的项目，并且需要一个集中的地方来整合质量报告，那么，Sonar 无疑是第一选择。你已经看到了将 Sonar 集成到构建中只需要做很少的配置。

在下一章中，我们将讨论如何安装和配置持续集成服务器，从而能够无论何时当代码变化被提交到 VCS 中时自动构建项目。



# 13

## 持续集成

---

### 本章涵盖

- 持续集成的好处
- 使用 Jenkins 构建 Gradle 项目
- 探索基于云的 CI 解决方案
- 通过 Jenkins 建模构建管道

如果你是一个软件开发团队中的一员，你不可避免地肯定会面对你的同事所编写的代码。在改变代码之前，程序员需要先从中央源代码仓库中复制一份代码，但是程序员本地复制的这份代码，很快就会与中央源代码仓库中的代码大相径庭。在你编写代码的时候，其他程序员可能已经将改变提交到已有的代码中，或者添加了新的工件如资源文件和依赖。提交时间越长，集成和合并代码就会变得越困难。

持续集成（Continuous Integration, CI）是一个软件开发实践，其提倡频繁地集成代码，最好一天当中有很多次。对于每个变化，源代码都会通过自动化构建被编译和测试，从而非常有效地减少集成的难度，并且让项目中的问题尽早暴露出来。

在本章中，我们会讨论持续集成的原理和架构。我们还会尝试一些使持续集成顺利进行的工具，叫作 CI 服务器或者平台。CI 服务器在中央源代码仓库中代码发

生变化的时候自动调度并执行一次构建。在了解 CI 基本知识以后，你会将这些知识应用到实践中。我们会讨论如何安装和使用开源的 CI 服务器 Jenkins 构建 To Do 应用程序。与很多的开发工具一样，CI 服务器也被迁移到了云端。我们会探讨一系列的解决方案，并比较它们的功能。

将一个大的构建分成几个小的、可执行的步骤，能够缩短反馈时间，增加灵活性。构建管道通过定义每个构建步骤的执行顺序和执行条件来协调它们。Jenkins 提供了很多有效的扩展，从而建模一个这样的管道。本章将会为配置本书中目前所接触到的构建管道步骤建立基础。在接下来的每章中，你都会添加新的步骤到构建管道中，直到将项目部署到产品环境中。我们首先来基本了解一下持续集成与开发过程的联系。

## 13.1 持续集成的好处

将不同开发人员的代码集成到一个中央 VCS 中应该是一个很正常的事件。持续集成就是通过明确定义的构建项目时间间隔（如，每 5 分钟）或者每次 VCS 中的变化来验证代码集成正确的过程。每一次提交，你都可以编译代码、运行各种类型的测试，甚至确定代码质量是提高了还是降低了。你到底得到了什么？除去最开始搭建和配置 CI 服务器的时间投资，持续集成提供了很多好处：

- **减少风险**：每次提交到 VCS 都会触发构建代码。因此，代码被频繁地集成了。这样就减少了在项目生命周期后期出现集成问题的风险；举个例子，每 2~4 周做一次新的发布。这样做产生的另一个效果就是，你还可以对构建过程保持更多的信心。
- **避免环境相关错误**：程序员通常只在一个操作系统上构建软件。虽然你可以通过 Gradle Wrapper 减少构建工具的运行时问题，但是需要依赖于机器的配置。在 CI 服务器上，你可以执行不依赖于特定的机器配置的构建。
- **提高效率**：虽然程序员每天要运行很多次构建，但是更合理的行为是让他们能够将注意力放在执行那些对工作重要的 task 上面：编译代码和运行某些测试。运行时间长的 task，如生成代码质量报告等，会降低他们的效率，所以这样的工作最好放到 CI 服务器上做。
- **快速反馈**：如果构建因为集成问题失败了，你会想要尽早地知道，从而解决问题。CI 服务器提供了很多通知错误的方法。最常见的通知方法就是发送包含失败构建的链接、错误消息和最近提交清单的 E-mail。
- **项目可视化**：持续集成能够让你看到当前项目的健康状况。许多 CI 服务器都提供一个基于 Web 的 Dashboard，能够显示成功的和失败的构建，并提供整合报告。

尽管持续集成有这些好处，但是将持续集成引入到一个团队或组织中需要持有透明的态度，在极端情况下可能需要一个彻底的文化氛围的转变。项目的健康状况通过 Dashboard 或者 E-mail 能够一直保持可见。这意味着一个失败的构建不再是一个秘密。为了提高项目的质量，尝试着培养一种对缺陷零容忍的氛围。从长期来看会有回报的。知道了这些好处，我们来看看持续集成在典型的场景中是怎么工作的。

在 CI 环境中，有三个组件是至关重要的：一个中央 VCS，供所有开发者提交代码，以及 CI 服务器和可执行的构建脚本。图 13.1 演示了这些组件之间的交互。

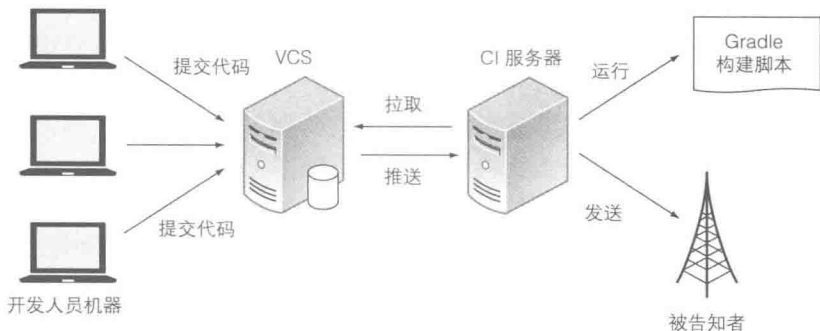


图 13.1 CI 环境剖析

我们来看看在一个拥有三个开发人员的团队中集成代码变化的典型场景。

- 1 提交代码：一个或多个开发人员在特定时间内将代码变化提交到 VCS 中。
- 2 触发构建：CI 服务器可以被配置成两种不同的模式，从而来判断 VCS 中是否有代码变化。CI 服务器可以被配置成每隔一段时间到 VCS 中检查是否有代码变化（拉模式，pull mode），或者可以被配置成监听 VCS 的回调（推模式，push mode）。如果确认有变化，构建就会被自动触发。或者，你也可以配置成每隔一段时间触发一次构建。
- 3 执行构建：一旦构建被触发，它就执行一个特定的动作。这个动作可以是任何事情，可以是调用一个 shell 脚本，执行一段代码，又或者是运行一个构建脚本，等等。在我们的讨论中，通常都是执行 Gradle 构建。
- 4 发送通知：CI 服务器可以被配置用来发出构建结果的通知，无论成功还是失败。通知方式可以是 E-mail、IM、IRC 消息、短信等。

根据 CI 服务器的配置，当一次提交或者多次提交发生时，以上步骤会被立即执行。配置的执行时间间隔越长，积累的变化就越多。

在过去 10 年里，很多开源的和商业化的 CI 服务器涌现出来。大多数都是能够被下载并且被安装到公司网络环境中的。最近，有很多关于在云环境中使用的 CI 服务器的宣传。基于云的解决方案将你从预先搭建基础设施的重担中解放出来，从

而降低了 CI 的门槛。它们通常都非常适合于开源的项目。最流行的 CI 服务器包括 Hudson/Jenkins、JetBrains TeamCity 和 Atlassian Bamboo。在本章中，主要使用 Jenkins 为 To Do 应用程序实现持续集成，因为其占有最大的市场份额。在你的本地机器上搭建 CI 服务器之前，你需要先安装一些组件。

## 13.2 安装 Git

最好地演示持续集成的方式就是实际动手做一遍。你所需要的是安装在本地的 CI 服务器、对中央 VCS 仓库的访问，以及一个用 Gradle 构建的项目。本节假设你的机器上已经安装了 Java 运行环境。

Jenkins 是持续集成入门的首选。其安装包可以从官方网站下载，并且理论上可以在 1 分钟内开始。为了方便起见，我将 To Do 应用程序上传到了 GitHub，一个在线托管服务网站。GitHub 后端使用的是免费的开源的 VCS 叫作 Git。即使你没有使用过这些工具，也不用担心。你会一步步地安装和配置这些工具。如果你没有 GitHub 账号，则从注册账号开始。

### 13.2.1 创建 GitHub 账号

创建一个免费的 GitHub (<https://github.com/>) 账号很容易，只需要在注册页面输入用户名、邮箱地址和密码即可，如图 13.2 所示。

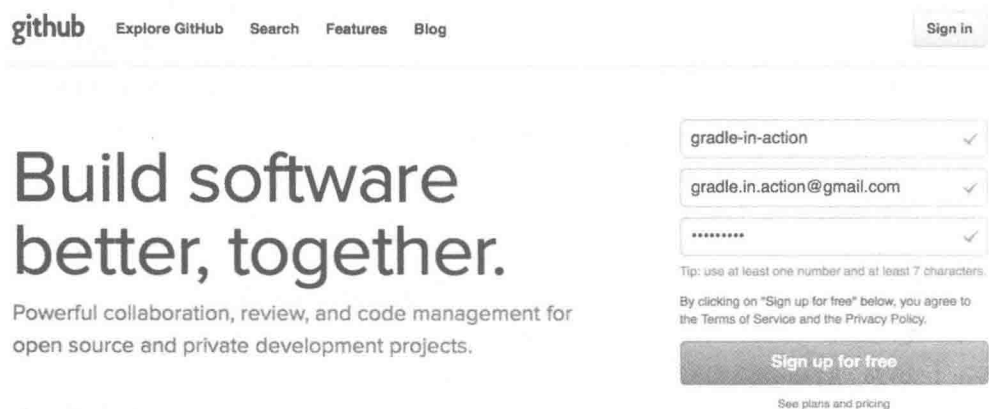


图 13.2 注册一个免费的 GitHub 账号

好了，你甚至不需要确认账户。成功注册后，你会进入到 Dashboard。不要犹豫去探索一下里面的功能，并且上传你的资料吧。为了能够在通信的时候使用加密的 SSH 链接，你需要生成 SSH 密钥并将公钥添加到 GitHub 账户中。GitHub 提供了



一个综合指南 (<https://help.github.com/articles/generating-ssh-keys>), 里面包括了一些具体细节。

## 13.2.2 forking GitHub 仓库

To Do 应用程序实例是 <https://github.com/bmuschko/todo> 下一个公共的 GitHub 仓库。因为你不是这个仓库的拥有人, 所以你不能对其提交代码变化。想要获得提交权限, 最简单的做法就是在自己的账户中 *fork* 这个仓库。*fork* 是对原始仓库的本地拷贝, 你可以对其做任意修改, 而不会改变原始仓库。要 *fork* 一个仓库, 请导航到实例仓库的 URL, 并单击导航栏中的 Fork 按钮, 如图 13.3 所示。



图 13.3 forking 实例仓库

几秒钟后, 项目就可以被使用了。为了与远程的 GitHub 仓库交互, 你需要在本地安装和配置 Git 客户端。

## 13.2.3 安装和配置 Git

你可以从 Git 主页 (<http://git-scm.com/>) 下载 Git 客户端安装包。这个页面提供了大多数常用的操作系统的安装文件。按照指示将 Git 安装到操作系统上。安装成功后, 你应该能够在命令行执行 Git。你可以通过如下命令来验证所安装的版本:

```
$ git --version
git version 1.8.2
```

针对远程仓库的提交可以被直接映射到你的 GitHub 账户。通过设置客户端的用户名和邮箱地址, GitHub 会自动将变化链接到账户上。如下两个命令显示了如何设置这两个配置值:

```
$ git config --global user.name "<username>"
$ git config --global user.email "<email>"
```

好了, 你已经配置好 Git 和实例仓库了。接下来, 你会安装 Jenkins 并配置一个构建任务来执行 To Do 应用程序的构建。

## 13.3 使用Jenkins构建项目

Jenkins (<http://jenkins-ci.org/>) 起源于 Hudson (<http://hudson-ci.org/>) 项目。2004 年 Sun Microsystems 开始了 Hudson 这个开源的项目。多年后, 它变成了最流行的 CI 服务器之一, 占据了很大的市场份额。2011 年 Oracle 收购 Sun 时, 社区决定在 GitHub 上 fork 这个项目, 并称之为 Jenkins。虽然现在 Hudson 仍然存在, 但是大多数项目转而使用 Jenkins 了, 因为其提供了更好的 bug 修复和功能扩展方面的支持。Jenkins, 这个完全用 Java 编写的工具, 能够方便地安装和升级, 提供了很好的脚本化能力, 并有超过 600 个的插件扩展。你将会在机器上安装 Jenkins。

### 13.3.1 开始使用 Jenkins

在 Jenkins 网站上, 你可以找到 Windows、Mac OS X 和各种 Linux 安装包。或者, 你也可以下载 Jenkins 的 WAR 文件, 放到你最喜欢的 Servlet 容器中或者直接使用 Java 命令启动它。下载 WAR 文件, 并使用 Java 命令启动内嵌的容器:

```
$ java -jar jenkins.war
```

启动成功后, 打开浏览器并输入 URL <http://localhost:8080/>。你应该能够看到 Jenkins 的 Dashboard。你现在可以安装插件和配置构建任务了。

### 13.3.2 安装 Git 和 Gradle 插件

Jenkins 最开始只有很少的功能。例如, 你只能从 CVS 或者 Subversion 托管的项目中拉取代码, 并且只能调用 Ant 脚本。如果你想要构建托管在 Git 仓库中的 Gradle 项目, 你需要安装相应的插件。这些插件可以通过 Plugin Manager 安装。通过点击 Dashboard 页面上的 ManagerJenkins, 可以访问 Plugin Manager。然后, 在接下来的页面中, 点击 Manage Plugins。你现在就在 Plugin Manager 页面上了, 如图 13.4 所示。

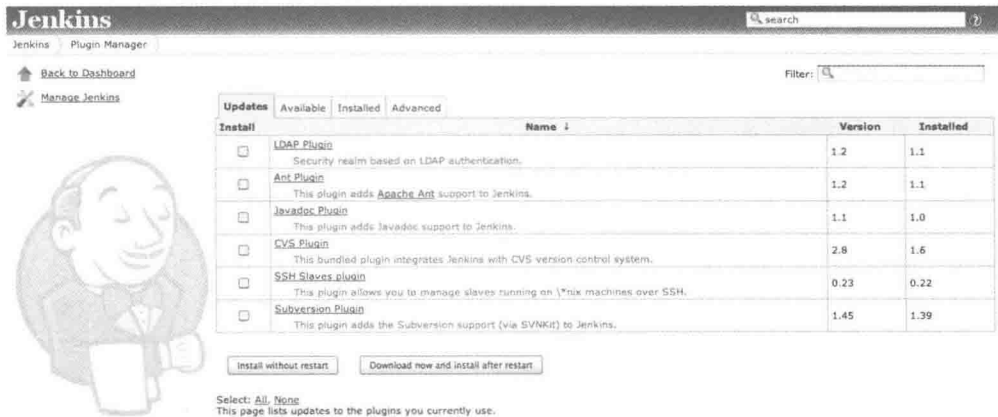


图 13.4 Jenkins 的 Plugin Manager 页面

Plugin Manager 页面显示了4个标签: Updates、Available、Installed 和 Advanced。跳转到 Available 标签来安装新插件。在右上角, 你会看到一个 Filter 搜索输入框。输入 “git plugin” 并选中 Git Plugin 旁边的复选框, 如图 13.5 所示。



图 13.5 安装 Git 插件

单击 Install without restart 按钮后, 插件就被下载和安装了。使用同样的方法, 你也会搜索到 Gradle 插件。在搜索框中输入 “gradle plugin”, 如图 13.6 所示。

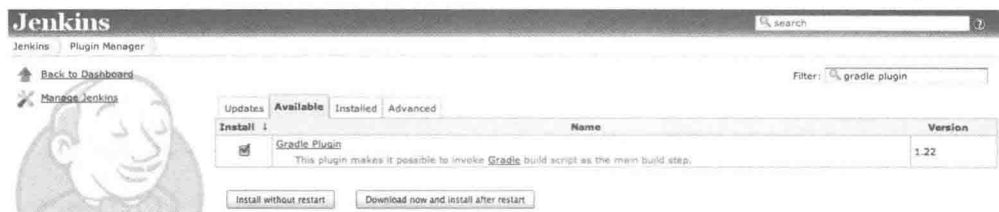


图 13.6 安装 Gradle 插件

选中插件复选框后, 单击 Download now and install after restart 按钮。你会看到类似于图 13.7 所示的页面显示下载和安装的插件。为了能够使用插件, 需要重启 Jenkins。选中 Restart Jenkins when installation is complete and no jobs are running 复选框后 Jenkins 会被重启。

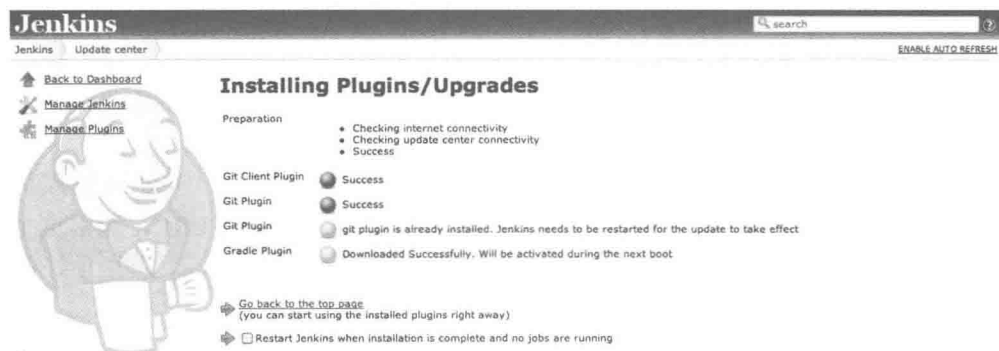


图 13.7 重启 Jenkins

一段时间后, Jenkins 被重启了并且插件能完全正常工作了。你已经可以开始定义第一个 build job 了。

### 13.3.3 定义 build job

Jenkins 用 *build job* 定义一个明确的工作步骤或者任务。一个 build job 通常定义了构建源代码的源, 如何获取源代码, 以及 job 运行的时候应该执行什么操作。举个例子, 一个 build job 可以简单到编译源代码, 运行单元测试。你会为 To Do 应用程序创建一个 build job 来做这些事情。

在 Jenkins 的 Dashboard 中, 点击 New Job 链接。新打开的页面要求你输入 job 名字, 并选择你想要构建的项目类型。对于 job 名字, 输入 “todo”, 然后选中 Build a free-style software project 单选钮。一个 free-style 类型的项目允许你控制 build job 的每个方面; 比如, VCS 和你想要使用的构建工具。图 13.8 显示了所选择的值。

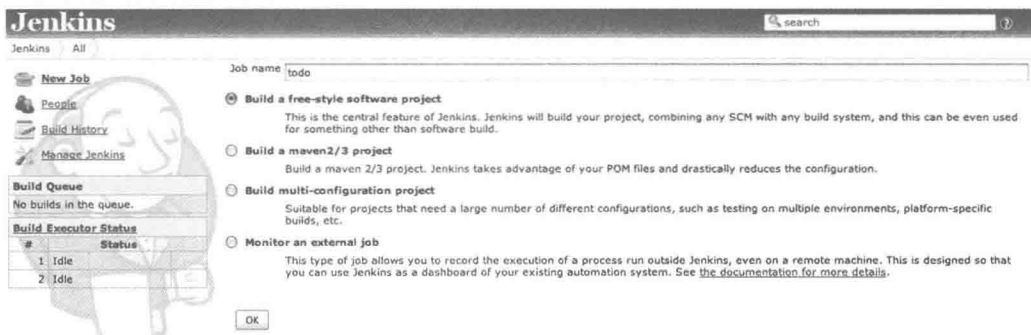


图 13.8 创建 free-style 类型的 build job

完成后, 单击 OK 按钮。build job 就创建成功了, job 配置界面就呈现在你的面前了。

#### 配置代码仓库

首先, 你会为 build job 配置 GitHub 代码仓库。配置好代码仓库后, Jenkins 就知道在执行 job 的时候去什么地方获取项目的源代码了。如果在配置界面上稍微往下滚动一下, 你就能看到 Source Code Management 部分。

你想要构建存储在 Git 代码仓库中的项目。选中 Git 单选钮, 并输入仓库的 URL, 也就是在你的账户下面 forked 的仓库中能够看到的 SSH 的 URL。通常这个 URL 是如下格式: `git@github.com:<username>/todoi.git`。图 13.9 演示了填写好的 Source Code Management 部分。



图 13.9 配置 Git 代码仓库

现在你已经告诉 Jenkins 到什么地方去获取源代码了，你还需要定义什么时候去拉取代码。接下来，你会设置一个构建触发器。

### 配置构建触发器

构建触发器是 Jenkins 的标准功能。它决定了一个构建什么时候会被执行或者触发。假设你想要每隔一段时间就轮询一次 GitHub 代码仓库，比如每分钟。滚动到 Build Triggers 配置部分，选中 Poll SCM 复选框，然后输入 Unix cron 表达式 “\* \* \* \* \*”，如图 13.10 所示。

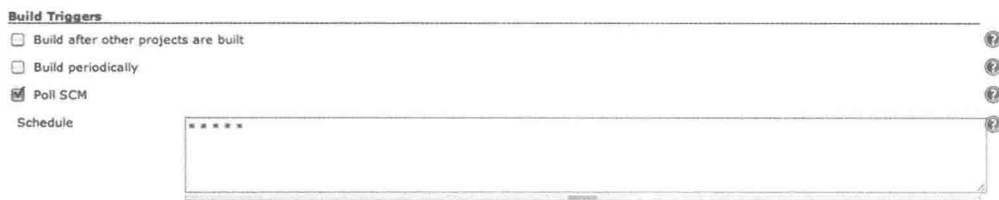


图 13.10 每分钟轮询一次代码仓库

表达式 “\* \* \* \* \*” 表示每分钟都会轮询代码仓库。轮询能够满足定期检查代码变化的要求。从另一方面来看，这种方法非常不高效。不仅仅是因为它为 VCS 和 Jenkins 服务器带来了许多不必要的工作量，而且当一个变化提交后根据你配置的 cron 表达式所定义的时间（这里配置的是每分钟），这个方法也给构建带来一定程度的延迟。

更好的方法是配置 Jenkins 的 job 去监听对代码仓库的推送通知。每当有变化提交到代码仓库的时候，VCS 都会调用 Jenkins 从而触发构建。因此，构建只在真正有变化发生的时候才执行。你会在网上找到更多的关于配置 VCS 的例子。接下来，我们来定义如果一个构建被触发了，这意味着什么。

### 配置构建步骤

一旦构建被触发，你就要执行 Gradle 构建脚本。每个需要执行的任务都被称作构建步骤。你可以在 Build 配置部分中添加构建步骤。在 Build 部分，点击 Add Build Step 下拉框，选择 InvokeGradleScript 选项。在图 13.11 中你看到的选项是由

你先前安装的 Gradle 插件提供的。选择 User Gradle Wrapper 单选钮，并且在 Tasks 输入框中输入 task “clean test”。

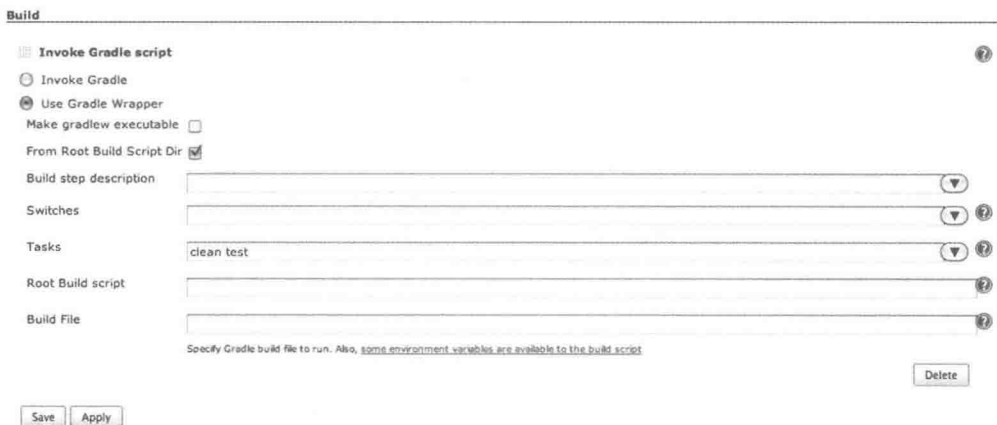


图 13.11 配置 Gradle 构建调用

这是 Gradle Wrapper 真正发挥作用的一个地方。你不需要安装 Gradle 运行时环境。你的构建提供了运行时环境，并且清楚地指明了所使用的 Gradle 版本。

如果你在开发机器上构建项目，你会想要好好地使用 Gradle 提供的增量构建功能，从而节省时间，提高构建效率。在 CI 设置中，构建需要从最干净的状态开始运行，从而确保所有的测试都被重新运行，并且被适当地记录。这就是你添加 “clean test” task 的原因。接下来，我们会接触到配置构建通知。

### 配置 E-mail 通知

E-mail 通知被配置成一个 post-build 动作，滚动到 Post-build Actions 部分，点击 Add Post-build Action 下拉框，选择 E-mail Notification 选项。只需要在 Recipients 输入框中输入邮箱地址，这样你就能在构建失败的时候收到邮件了，如图 13.12 所示。



图 13.12 设置一个 post-build 动作的 E-mail 通知

添加完所有配置后，应单击 Save 按钮保存这些设置。现在你就可以执行构建了。

## 13.3.4 执行 build job

保存 build job 后，你可以看到在 Jenkins 的 Dashboard 中列出来了。job 左边的

灰色球表示它还没有被构建过。成功的构建会使其变成蓝色，失败的构建会用红色来表示。你可以等一会让构建自动触发，或者通过点击那个时钟图标手动启动构建。图 13.13 显示了正在执行的第一次构建。

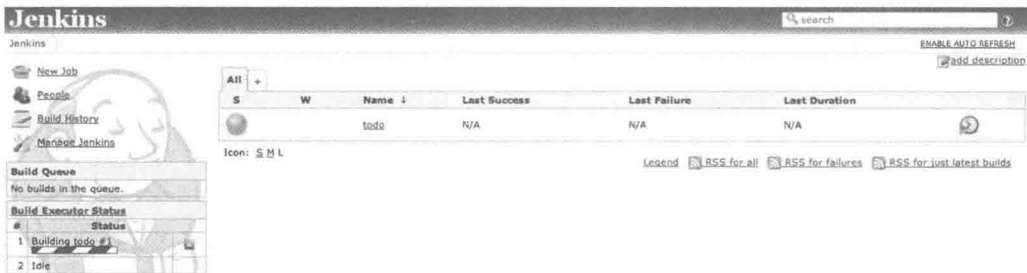


图 13.13 正在执行的构建

几分钟后，构建结束了。你应该能看到那个小球变成了蓝色，并且出现了一个太阳图标，用来表示项目的健康状况。这个 job 还报告了上次构建所花的时间，并显示了一个时间戳来告诉你上次成功构建的日期和时间。如图 13.14 所示在 Dashboard 中显示的一次成功构建。

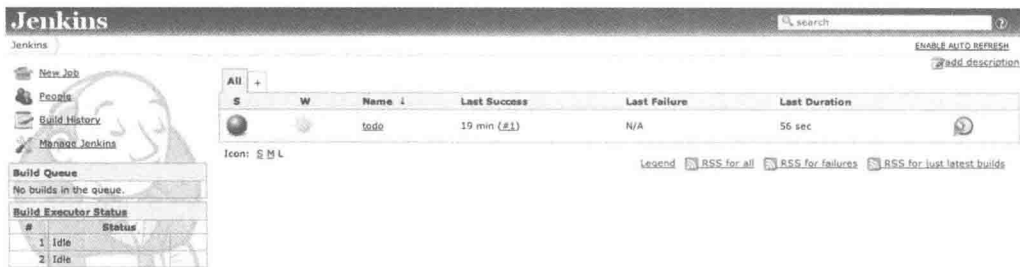


图 13.14 build job 执行成功

你可以点击 job 的名字进入到项目主页，从而得到关于具体构建的更多信息。这个页面允许你重新配置 job，手动触发构建，检查构建历史。你会在构建历史中找到第一次构建 #1。

点击 #1 来看看 job 被执行的时候背后发生了什么。左边的菜单项中有一个 Console Output。Console Output 记录了构建过程中执行的步骤信息。首先，Git 代码仓库的 master 分支被检出了。拉取完源代码后，为了执行所定义的 task，启动了 Gradle 构建。如果你仔细看的话，就会发现 Gradle Wrapper 和所有的依赖在执行 task 之前都被下载下来了。图 13.15 显示了 Console Output 的一部分。

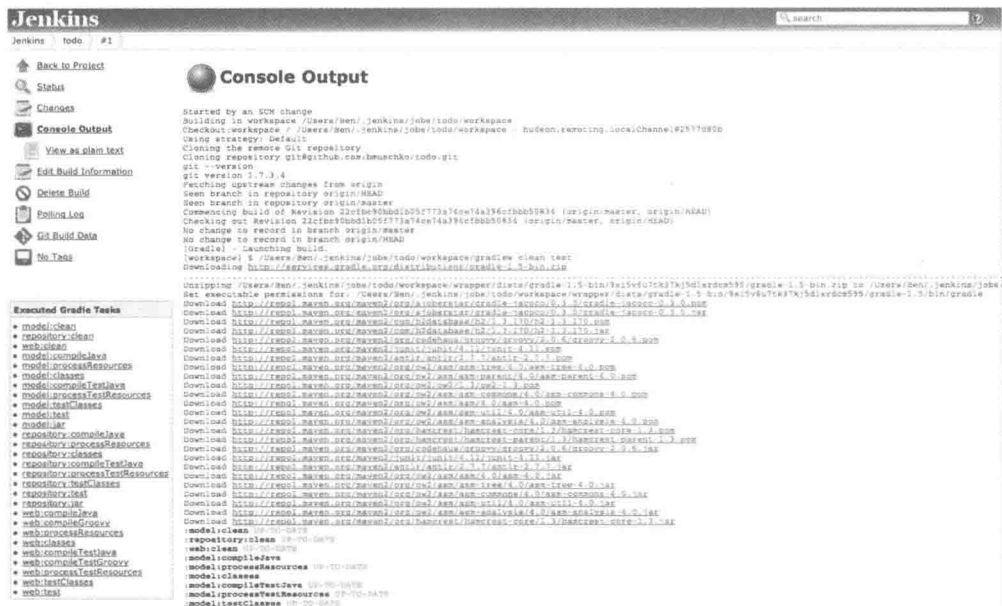


图 13.15 job 执行的控制台输出

Console Output 的内容是在构建执行的时候被实时显示上去的。这个功能在你想要追踪一个失败构建的根本原因的时候提供了非常有价值的信息。

恭喜，你为项目配置好了一个 CI job！你可以通过推送一个代码变化到代码仓库来触发接下来的构建，或者手动在项目 Dashboard 中启动它。接下来，你将提高项目的各方面报告能力。

### 13.3.5 添加测试报告

Jenkins 提供了可扩展的报告功能。只需要小小的努力，你就能配置项目来处理测试框架如 Junit、TestNG 和 Spock 所生成的 XML 测试结果了。相应地，Jenkins 生成了一个图表化测试结果趋势图，让你能够深入查看成功执行和测试失败的所有细节。虽然功能有限，但是它很容易设置，能够替代 Sonar 所提供的报告。

#### 发布单元测试结果

你可能还记得 Gradle 提供的 XML 格式的测试结果被放置在 build/test-results 目录下。为了将单元测试和集成测试的测试结果分开，你需要重新配置 GitHub 上的项目，将单元测试结果放在 unit 子目录下，将集成测试结果放在 integration 子目录下。

跳转回项目配置页面，滚动到 Post-build Actions 部分，点击 Add Post-build



Action 下拉框，并选择 Publish JUnit Test Result Report。通过在输入框中输入表达式 “\*\*/build/test-results/unit/\*.xml”，你可以告诉 Jenkins 解析所有子项目的测试结果，如图 13.16 所示。

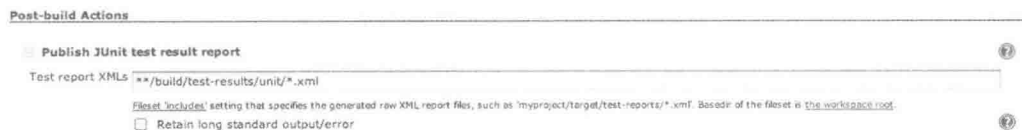


图 13.16 为所有的子项目配置测试报告

为了将测试结果显示在 Dashboard 上面，你至少需要执行一次构建。手动触发一次构建。你会发现一个新的 Latest Test Results 图标。点击它，你就能看到执行测试的统计信息。历史开发测试用多个数据点来记录测试结果趋势。在执行测试至少两次后，趋势图就被显示出来了。成功测试显示成蓝色，失败测试显示成红色。如图 13.17 所示为在项目的 Dashboard 中显示的测试结果趋势图。

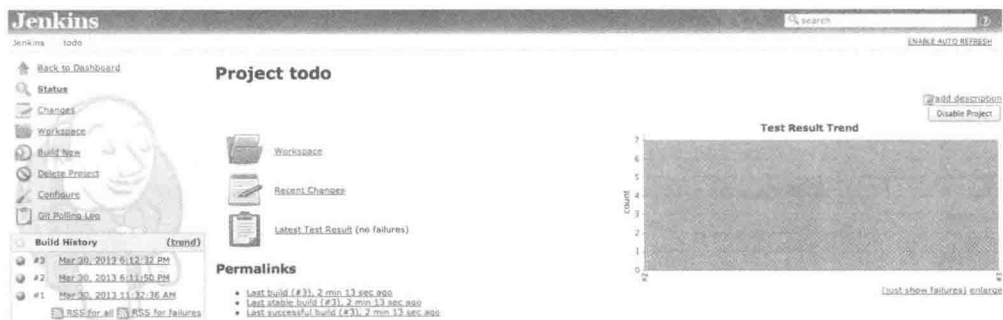


图 13.17 测试结果趋势图

单元测试配置使用 JaCoCo 来生成代码覆盖率度量。接下来，你会同时显示单元测试的测试覆盖率趋势。

### 发布代码覆盖率结果

发布 JaCoCo 代码覆盖率结果的功能是由第三方插件提供的。你已经知道了如何为 Jenkins 安装一个插件。进入 Plugin Manager 页面，搜索 jacoco 插件，并安装插件。在重启 Jenkins 后，你可以添加一个新的 post-build 动作叫作 Record JaCoCo 覆盖率报告。图 13.18 显示了如何配置这个插件使其指向正确的 exec 文件，以及包含 class 文件和源文件的目录。

Record JaCoCo coverage report						
Path to exec files (e.g.: **/target/**.exec, **/jacoco.exec)		Path to class directories (e.g.: **/target/classDir, **/classes)		Path to source directories (e.g.: **/mySourceFiles)		
**/build/jacoco/test.exec		**/build/classes		**/src/main/java		
Inclusions (e.g.: **/*.class)				Exclusions (e.g.: **/*Test*)		
	Instruction	% Branch	% Complexity	% Line	% Method	% Class
☀	100	70	70	70	70	70
☁	0	0	0	0	0	0

图 13.18 配置代码覆盖率报告

这个插件提供的另一个有用的功能是可以充当质量关。假设你想要确保类和方法的单元测试覆盖率在 70% 以上。为了防止项目的代码覆盖低于这个阈值，Jenkins 会将其反映成不健康的状态。

在图 13.19 中你可以看到在测试结果趋势图下面出现了代码覆盖率趋势图，当然至少要创建两个数据点。通过点击趋势图标或者左边的 Converage Trend 菜单项，可以深入查看覆盖率结果的详细信息。

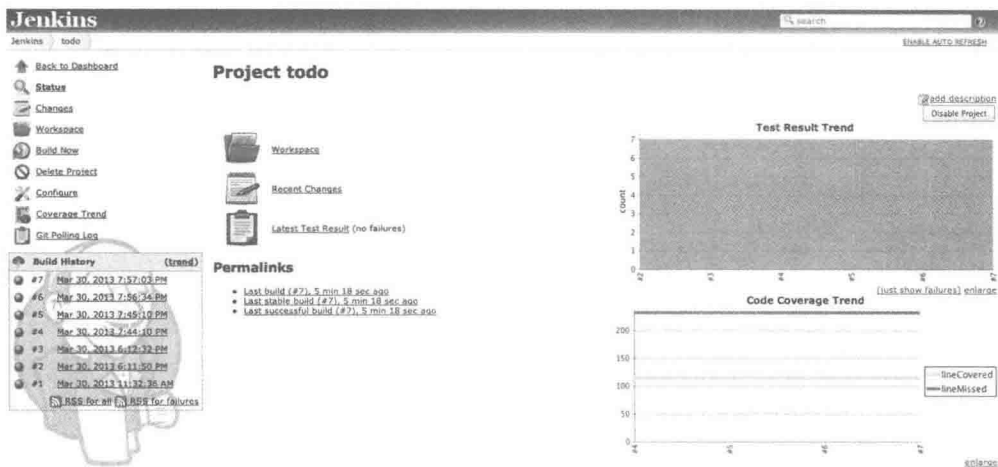


图 13.19 代码覆盖率趋势图

这就是所有关于使用 Jenkins 构建一个基本的 build job 的讨论了。你现在已经能够利用所学到的知识构建自己的项目了。在本章后面的部分中，你会学到将多个 build job 连接起来形成一个构建管道。在这之前，我们先来讨论一下基于云的 CI 解决方案。

## 13.4 探索基于云的解决方案

将 CI 服务器托管在云上面的好处非常明显。首先也是最重要的，你不需要提供基础硬件设施和维护软件。根据构建目的，对硬件资源的要求可能会很高。当你需要时，基于云的持续集成承诺提供一个可伸缩的解决方案。需要更多的 CPU 计算能力来满足多个并发的编译构建任务吗？只需要购买提供了更多硬件资源的计划来扩大规模就可以了。许多基于云的 CI 服务器能够直接与在线的代码仓库账户如 GitHub 集成。登录到你的账户中，选择一个项目，并开始构建。下面对一些流行的带 Gradle 支持的云端 CI 服务器进行概览：

- *CloubBees DEV@cloud* : Dev@cloud 服务 (<http://www.cloudbees.com/dev.cb>) 是一个标准的 Jenkins 服务器。免费版本提供有限的服务器资源和插件支持。付费版本能够让你完全访问所有标准的 Jenkins 插件。DEV@cloud 还允许你限制项目报告的可见性和配置选项的访问权限。
- *CloudBees BuildHive* : BuildHive (<https://buildhive.cloudbees.com/>) 是一个使你能够构建托管在 GitHub 上面的项目的免费服务。这个服务由拥有有限的功能集的 Jenkins 支持——比如，你不能添加更多的 Jenkins 插件，或者托管在 GitHub 之外的代码仓库。build job 的创建很容易，并且支持在合并请求之前验证拉取请求。如果你需要对开源项目的编译和测试支持，BuildHive 是一个不错的选择。
- *Travis CI* : Travis CI (<https://travis-ci.org/>) 是一个适合于开源的、小的商业化和企业级项目的 CI 服务。这个服务提供了自己生产的 CI 产品，让你能够构建托管在 GitHub 上面的项目。项目需要在源代码中提供一个配置文件，表明你想要使用的语言和执行的命令。
- *drone.io* : drone.io (<https://drone.io/>) 让你能够将 GitHub、Bitbucket 或者 Google Code 账户连接到 CI 构建项目。在免费版本中，你只能构建公共的代码仓库。付费版本也支持私有代码仓库的构建。但是报告方面的功能非常有限，drone.io 允许你自动将应用程序部署到 Heroku 或 AppEngine 这样的环境中去。

选择一个托管的 CI 服务器可能听起来没什么难度。然而，有些东西还是需要斟酌。持续集成可能会消耗大量的硬件资源，尤其是你需要构建很多应用程序而且想要获得快速反馈时。成本很有可能失去控制。如果你正在考虑使用基于云的 CI 解决方案，那么最好先试试那些免费的解决方案比较其优劣。

你已经了解了如何使用 Jenkins 来为 To Do 应用程序构建 task。如果你想要构建一个完整的管道，则需要创建多个 build job，然后将它们连接起来。接下来，我们看看如何使用 Jenkins 来实现。

## 13.5 使用Jenkins创建构建管道

虽然在一个单独的 build job 中运行 Gradle 构建的所有 task 可能很方便，但是如果构建失败了很难找到根本原因。如果将构建过程根据职责分成比较小的步骤，那么查错会变得相对容易一些。这样带来的直接结果就是清晰的关注点分离，以及更快、更具体的反馈。比如，如果你创建了一个步骤专门用来执行集成测试，然后这个步骤失败了，你就能知道两件事情。第一，你可以确定源代码可以编译通过并且所有的单元测试都能够成功运行。第二，集成测试失败的根本原因要么是一个测试的验证没有满足，要么是与其他系统组件的集成出现了问题。在本节中，你会创建构建管道的第一个步骤，如图 13.20 所示。

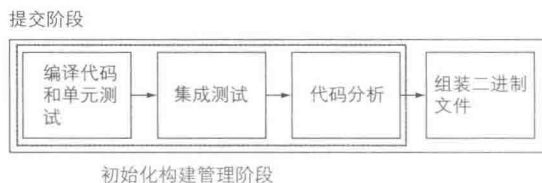


图 13.20 创建构建管道的第一个阶段

构建管道中的每个步骤之间都定义了质量关。只有当一个构建步骤的结果达到其质量关的要求时，管道才被允许继续执行下一个步骤。在你的例子中这意味着什么呢？一旦集成测试运行失败，下一步进行代码分析的步骤将不会被触发。

### 13.5.1 创建构建管道的挑战

在创建一个构建管道的时候，你面临的挑战要求你有相应的解决方案。下面列出了一些比较重要的关键点。

- 每个构建管道都是从一个单独的初始 build job 开始的。在这个 job 执行过程中项目源代码被从 VCS 仓库中检出或者更新。后续的步骤将会基于代码库的同一个修订版本进行工作，以避免引入额外的、不期望的变化。
- 我们需要用一个唯一的构建号码或标识符来清晰地识别一次构建。这个构建号码应该由构建管道的第一个 job 来分配，并传递给后续的所有步骤。所生成的工件（如 JAR 文件、报告和文档）包含了构建号码来清晰地标识它们的版本。
- 一个可交付的工件应该只被创建一次。如果后续步骤需要（如部署），则应该重用这个工件，而不是重新创建。根据构建号码从一个共享的二进制仓库中获取这个工件。
- 虽然很多构建步骤被自动触发（比如，代码提交或者前一个步骤通过质量关），但还是有些步骤需要手动启动。典型的例子就是将工件部署到一个特定的环

境中。手动触发在你想要提供一键发布功能的时候非常有用。这样，产品负责人就能决定什么时候发布功能给最终用户。

在撰写本书的时候，Jenkins 并没有提供标准的、简单易用的解决方案来实现上述需求。好消息就是你可以借助社区插件来创建成熟的构建管道。下一节中会概述这些插件的功能和用例。

## 13.5.2 探索基本的 Jenkins 插件

Jenkins 插件涉及了 600 多个功能。下面 4 个插件为创建构建管道提供了最基本的功能。在继续之前请安装它们。

### Parameterized Trigger 插件

Jenkins 为连接 build job 提供了开箱即用的功能。你需要做的是添加一个新的叫作 Build Other Projects 的 post-build 动作。这个动作允许你定义 build job 的名字，它应该在当前 build job 完成时自动被触发。问题是这种方式不能在两个 job 之间传递参数，而你需要这个功能通过初始构建的号码清晰地识别一次构建。

Parameterized Trigger 插件扩展了连接 build job 的功能，让你可以为触发的 job 定义参数。在安装这个插件后，你就能添加一个新的 post-build 动作叫作 Trigger Parameterized Build on Other Projects。在配置部分你就可以配置需要执行的 job 名字，在什么条件下被触发，以及需要传递的参数。记住，将多个 job 名字用逗号分隔会触发多个 job。

假设你想要在构建管道的第一个步骤中定义一个 SOURCE\_BUILD\_NUMER 参数来标识初始化构建号码。你可以使用 Jenkins 的内置参数 BUILD\_NUMBER 作为这个参数的值。BUILD\_NUMBER 是每个 build job 在运行时动态获得的一个唯一的值。图 13.21 演示了如何在负责编译 / 单元测试执行的 job 定义处定义触发集成测试 build job 的触发器。



图 13.21 通过传递参数来触发 build job

在触发的构建中，你现在无论在 build job 定义中还是在调用的 Gradle 构建中都

可以使用 `SOURCE_BUILD_NUMBER` 作为环境变量了。举个例子，在 Gradle 构建脚本中，你可以通过使用 `System.env.SOURCE_BUILD_NUMBER` 表达式来直接获取这个参数的值了。

如果你不能确定什么参数被传递到了当前的 build job 中，你可以安装 Jenkins 插件 `Show Build ParametersPlugin` 来查看。它通过将所有的参数及其值在构建时显示在项目页面上让你确认。

### Build Name Setter 插件

在默认情况下，每个 Jenkins 的 job 都是使用 `#{BUILD_NUMBER}` 表达式来显示构建号码的。在项目页面上，这个表达式看起来类似这样：Build #8(Apr 2, 2013 6:08:44 AM)。如果你创建了多个构建管道，你可能需要更有表达性的构建名字来清楚地标识一个构建属于哪个管道。Build Name Setter 插件让你能够修改构建名字表达式。图 13.22 显示了如何添加 `todo` 前缀到构建名字表达式中。

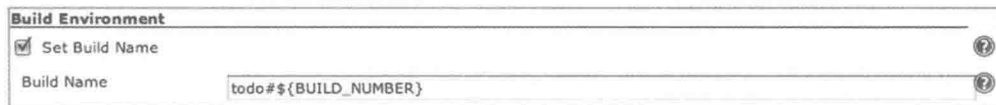


图 13.22 为第一个 job 定义构建名字表达式

构建执行完成后，名字显示成：Build todo#8(Apr 2, 2013 6:08:44 AM)。我们在后面创建完整的构建管道时还会进一步阐述这个插件的功能。

### Clone Workspace SCM 插件

正如我们前面所讨论的，你只想要在第一个 build job 执行过程中从 VCS 仓库检出源代码一次。后续的 build job 都工作在同一个修订版本上。Clone Workspace SCM 插件让你能够在其他的 job 中重用项目的工作空间。为了达到这个目的，你需要配置第一个 build job 来将检出的代码进行存档，如图 13.23 所示。



图 13.23 存档第一个 job 的工作空间

在后续的 job 中，你可以在 Source Code Management 配置部分选择 Clone Workspace 选项了。图 13.24 演示了如何在后续的 build job 中重用其父 job `todo-initial` 的工作空间。



图 13.24 在后续的 job 中克隆存档的工作空间

你不用再检出源代码了，你可以让后续的 build job 都工作在同一个工作空间了。这样你就可以访问之前创建的工件，比如编译好的类文件和项目报告。

## Build Pipeline 插件

将多个 build job 连接起来后，如果没有合适的名字，它们的执行顺序很有可能乱套。Build Pipeline 插件提供了两种功能。第一，对整个构建管道提供一个单独的视图，使其可视化。第二，让你能够配置下游的 build job 只在用户手动启动它的时候才执行。这对于一键部署非常有用。我们会在 15 章中讨论工件部署的时候使用到这个功能。

创建一个构建管道视图非常简单。安装这个插件后，点击 Dashboard 上的加号图标来添加一个新的视图。在新打开的页面中，选中 Build Pipeline View 单选钮并输入一个合适的视图名，如图 13.25 所示。

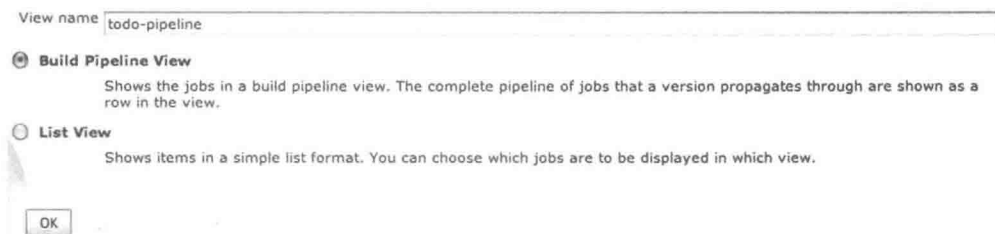


图 13.25 创建一个新的构建管道视图

单击 OK 按钮后，视图中多了一个页面。选择初始化 build job，你现在可以创建构建管道视图了。图 13.26 显示了这个插件所产生的视图示例。

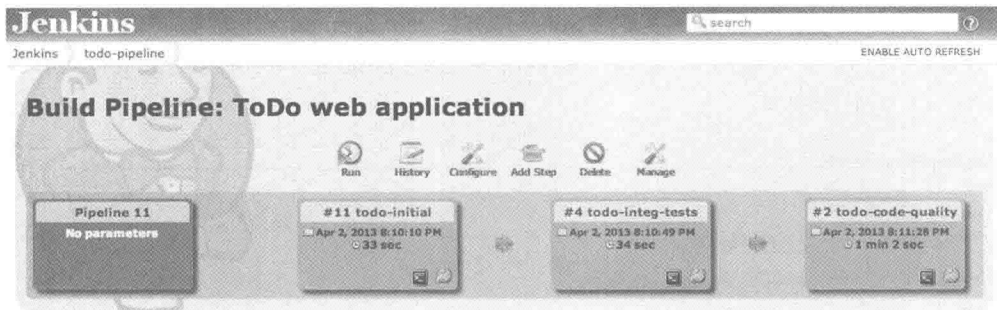


图 13.26 构建管道视图

如图所示，这个构建管道包含三个 build job。箭头表示执行的顺序。构建的状态用不同的颜色表示。

创建图形化的构建管道视图还可以选择使用 Downstream Buildview 插件。从初始化 build job 开始，将后续的 job 以分层视图显示出来。为项目选择哪个插件由你自己来定。在本书中，我们会继续使用 Build Pipeline 插件。在了解了这些插件后，你现在完全可以创建构建管道的前三个步骤了。

### 13.5.3 配置构建管道

为 To Do 应用程序创建构建管道并不需要额外的 Gradle 的 task。使用 Jenkins，你会编排一系列的 build job 来依次调用那些已经存在的 task。整个构建管道按下列顺序包含三个 build job。

- 1 todo-initial：编译源代码和运行单元测试
- 2 todo-integ-tests：运行集成测试
- 3 todo-code-quality：使用 Sonar 执行静态代码分析。

在本章前面，你设置了一个 build job 用来编译源代码和运行单元测试。只需要一点改进，这个 job 将被当作构建管道的第一个步骤。为了表明这个 job 是构建管道的入口点，将其名字改成 todo-initial。接下来用同样的命名形式为步骤 2 和步骤 3 创建新的 free-style 类型的 build job。稍后，你将赋予它们生命。

创建 Jenkins 的 build job 是一个重复的而且无聊的任务。为了简短起见，在讲解每一个构建步骤的配置时，我只将最重要的点列出来。



### 快速创建 Jenkins 的 job

在默认配置下, Jenkins 将定义好的 job 保存在 `~/.jenkins/jobs` 目录下。如果你在配置管道的时候感觉很迷茫, 也不要担心, 本书的源代码中包含了每个步骤的 job 定义。你只需要将 job 定义复制到 job 目录下, 然后重新启动服务器就好了。

#### 步骤 1 : 编译和单元测试

开始时, 需要对第一个 build job 做一些调整:

- 为了让所有后续的 job 都使用同一个工作空间, 需要添加一个 post-build 动作 Archive for Clone Workspace SCM, 并配置好表达式 `"/**/*"`。
- 使用 `todo#${BUILD_NUMBER}` 表达式定义构建名。
- 添加一个可以传递参数的 post-build 动作, 定义一个触发 `todo-integ-tests` 集成测试 job 的触发器。你也可以定义一个 `SOURCE_BUILD_NUMBER=${BUILD_NUMBER}` 参数。

#### 步骤 2 : 集成测试

只有当步骤 1 成功结束后, 集成测试步骤才会被触发。也就是在没有编译错误和所有的单元测试全部通过的时候。修改如下默认的 job 配置:

- 在 Source Code Management 配置部分, 选择 Clone Workspace 选项并指向 `todo-initial`。
- 作为构建步骤, 你想要触发执行集成测试。添加一个构建步骤来调用 Gradle 脚本中的 Gradle wrapper, 并输入运行的 task 为 `databaseIntegrationTest`。
- 将单元测试和集成测试的测试结果写到不同的目录下。为了能够发布测试报告, 使用 `"/**/build/test-results/integration/*.xml"` 表达式。你会选择 `"/**/build/jacoco/integrationTest.exec"` 文件作为代码覆盖率报告。
- 使用上游传递的构建号码参数来定义构建名: `todo#${ENV, var="SOURCE_BUILD_NUMBER"}`。
- 添加一个可以传递参数的构建动作, 定义一个构建触发器来运行静态代码分析 `todo-code-quality`。对于参数传递, 你可以选择 Current Build Parameters 选项来重用已经存在的参数。

#### 步骤 3 : 代码质量

目前代码质量构建步骤是构建管道的最后一个步骤。因此, 你不需要为其定义下游项目。在接下来的两章中, 通过添加发布 WAR 文件到仓库中和部署工件到不

同的运行时环境中两个构建步骤来进一步阐述这个构建管道。对于这个 job 的配置大多数与前面的 job 类似：

- 在 Source Code Management 配置部分，选择 Clone Workspace 选项并指向 todo-initial。
- 作为构建步骤，你想要触发执行 Sonar Runner 来产生代码质量报告。添加一个构建步骤来调用 Gradle 脚本中的 Gradle wrapper，并输入运行的 task 为 sonarRunner。
- 使用上游传递的构建号码参数来定义构建名：todo#\${ENV, var="SOURCE\_BUILD\_NUMBER"}。

很好，你使用 Jenkins 通过设置一系列 job 创建出了自己的第一个构建管道。确保你配置了至少一个可视化管道相关的插件。能够看到这些 job 按顺序执行起来真是太叫人兴奋了。

## 13.6 总结

持续集成是软件开发实践，它为你的团队和项目带来的好处非常明显。通过每天多次自动集成源代码，可以确保缺陷被引入时就被发现了。这样，交付低质量软件的风险就降低了。

在本章中，你体验到了为一个项目创建持续集成是多么容易的一件事情。你安装了开源的 CI 服务器 Jenkins，并为 To Do 应用程序创建了 build job。首先，你学到了如何从 GitHub 代码仓库定期获取源代码并触发 Gradle 构建。Jenkins 的强大功能之一就是生成报告。你配置了 build job 来显示单元测试的结果和代码覆盖率情况。在服务器上托管 Jenkins 实例需要硬件资源，也需要合适的人去维护它。我们探索了流行的基于云的 CI 解决方案，也比较了它们的优势和劣势。虽然将 CI 服务器托管到云上非常方便，但是随着 build job 和功能的增多可能会带来很多开销。

CI 服务器并不仅仅是一个用来编译和测试代码的平台。它可以被用来编排成熟的构建管道。你还学到了如何使用 Jenkins 来创建一个这样的构建管道。虽然 Jenkins 本身不提供标准化的构建管道实现，但是你可以结合一系列社区插件的功能来实现各种解决方案。我们讨论了如何为持续交付的前三个阶段创建 build job，并把它们联系起来。

在下一章中，你会学到如何为项目打包，并发布到私有的或者公共的工件仓库中去。在后面章节中，你会通过创建发布 WAR 文件和部署到特定环境中的 job 来扩展这个构建管道。

# 14

## 打包和发布

---

### 本章涵盖

- 打包和分发
- 发布到本地、远程和公共的 Maven 仓库中
- 打包和发布作为构建管道的一部分

作为开发人员，在软件开发过程中，你主要处理两种类型的工件（artifact）：源代码和构建出来的二进制包。在本书中我们已经看到过二进制包的例子了，包括 JAR、WAR 和 ZIP 文件。

版本控制系统如 Git 或者 Subversion 提供的源代码仓库被设计用来管理源代码。它们提供的功能有保存一个文件的两个版本之间的差异、分支、标签甚至更多。源代码文件通常都很小并且可以很好地被源代码仓库所管理。比较大的文件，尤其是二进制文件，有可能会降低仓库的性能，让程序员检出代码的速度变慢，并占用大量的网络带宽。

二进制仓库如 JFrog Artifactory 和 Sonatype Nexus 非常适合于存储二进制包。其中最著名的一个二进制仓库就是 Maven Central。它们被用来管理比较大的文件的二进制包，提供一种方式来组织这些文件，用基本信息描述它们，并且提供发布和

下载接口或者 API。

在本章中，我们来看看如何定义你的构建将会生成的工件，我们也会讨论如何为这些工件生成基本信息，并将其发布到本地和远程仓库中。你在第 8 章中所写的 CloudBees 插件非常适合用来演示这个功能。将插件包发布出来后，组织中的其他 Gradle 用户或者对这个插件感兴趣的任何人也能使用这个插件。

在持续交付的上下文中，打包和发布扮演了很重要的角色。一旦打包了一个特定的版本，你就可以将其部署到不同的环境中用于验收测试或者直接交给最终用户。只打包一次，然后部署到二进制仓库中，在需要的时候重用是一个非常好的实践。我们会在本章中介绍如何将这个概念运用到 To Do 应用程序中并作为上一章介绍的构建管道的一部分。

我们先回顾一下第 8 章中的插件代码，并回想它的组建过程。

## 14.1 打包和分发

在默认情况下，所有应用了 Java 插件的项目在执行 `assemble task` 的时候都会生成一个单独的 JAR 文件。在第 8 章中，你已经很好地运用了这个功能。包文件名包含一个名字（这个名字来源于基本名字，通常是项目名字）和一个版本号（如果通过 `version` 属性设置了）。如下目录树显示了为这个插件所打的包：



这个包的类型有可能根据项目的类型发生变化。例如，如果你应用了 War 插件，那么生成的包就是一个 Web 包（WAR 文件），其中包含 Web 相关的信息。

虽然创建一个单独的特定项目类型的包能够满足大多数应用的要求，但是你可能想要在组建过程中创建额外的工件。Gradle 并没有限制一个项目能够生成多少个工件。如果你熟悉 Maven，你就知道在 Maven 中为一个项目创建多个工件是多么的困难，你会发现 Gradle 的这个功能是多么的受欢迎。

那么如何为项目添加自定义的包呢？Gradle 通过 API 包 `org.gradle.api.tasks.bundling` 提供了打包的 task 如 Zip、Tar 和 Jar。第 4 章中有添加增强的 task 即打包 Zip 文件的例子。快速过一遍那个例子可能对你有帮助。

为了能够看到自定义打包 task 的输出，你需要在命令行执行，或者将其添加成另一个 task 的依赖。如果你的项目交付的非标准工件需要被其他用户或项目使用，

那么需要将它们包含在组建过程中。Gradle 提供了方便的方法来添加工件。

### 14.1.1 定义附加包

理解如何定义附加包需要一点背景知识：每个应用了 Java 插件的项目都有一个 `archives` 配置项。你可以通过调用 `dependencies` task 来检查其存在性，如下面的命令行输出所示：

```
$ gradle dependencies
:dependencies

-----
Root project
-----

archives - Configuration for archive artifacts. ← 定义输出包的标准配置
No dependencies
...

```

`archives` 配置定义了一个项目的输出包。对于 Java 项目，这个配置默认是标准的 JAR 文件。当你执行 `assemble` task 的时候，所有定义的包都被构建出来。你需要添加更多的输出包来丰富插件项目。

你会通过例子来学习如何打包。随着插件越来越流行，你会想让你的用户深入了解代码内部工作原理。插件用户特别感兴趣的是学习暴露出来的 API。如果能把源代码和相应的 Groovydocs 提供给他们岂不是更好吗？

用 JAR 文件的形式将源代码和 Groovydocs 交付给用户是一种常见的做法。对于插件项目，这意味着你需要创建两个新的 Jar 类型的 task。源代码 JAR task 需要包含所有 source set 的源文件。给这个 task 取名叫作 `sourcesJar`。groovydocJar task 创建的 JAR 文件包含了 Groovy 类的 API 文档，为了能在 JAR 文件中包含项目的 Groovydocs，你需要先生成它们。通过让 `jar` task 依赖于 `groovydocJar` task 能够很轻松地达到这个目的。图 14.1 显示了新的打包 task 作为 `assemble` task 图的一部分。

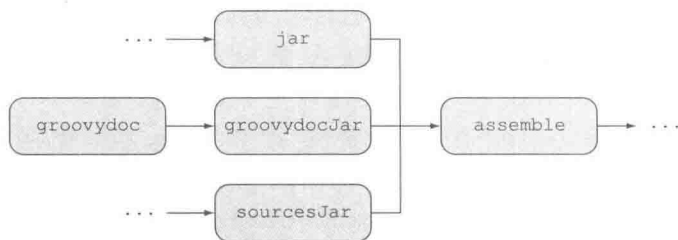


图 14.1 创建附加包的 task

现在我们来讨论一下清单 14.1 中的实现。为了能够清楚地识别所生成的 JAR 文件与插件之间的关系，最好使命名保持一致。为此，你需要为包含源代码的 JAR 文件加上 `source` 后缀（也叫作类别），为包含 Groovydocs 的 JAR 文件加上 `groovydoc` 后缀。

清单 14.1 定义附加包

```
ext.artifactBaseName = 'cloudbees-plugin'

task sourcesJar(type: Jar) {
    baseName artifactBaseName
    classifier 'sources'
    from sourceSets.main.allSource
}

task groovydocJar(type: Jar, dependsOn: groovydoc) {
    baseName artifactBaseName
    classifier 'groovydoc'
    from groovydoc.destinationDir
}

artifacts {
    archives sourcesJar
    archives groovydocJar
}
```

创建包含所有源文件的 JAR 包的 task

创建包含 Groovydoc 的 JAR 包的 task

通过指定 archives 配置来注册附加包

看看 Maven Central 中的类库，你会发现这个命名规范是一种常见的实践方式。

将新的打包 task 作为 assemble task 图的一部分有两种方式。你可以顺着必要的路线添加 `sourcesJar` 和 `groovydocJar` 作为 task 依赖。这种方式看起来最直接，也是 Ant 用户最熟悉的。

```
assemble.dependsOn sourcesJar, groovydocJar
```

这样做能够工作得很好，并且是一种合理的做法。然而，Gradle 提供了一种更具有表述性的做法。你可以表达一个项目的输出包是什么样的，而无须说明它们是如何被创建出来的。这种表述性方式有两个好处。一方面，更具有可读性和表达性；另一方面，在比较复杂的构建中，可以通过 Gradle API 直接引用其他项目的输出包。

为了能够让新的打包 task 作为 assemble task 图的一部分具有表述性，你需要使用项目的 `ArtifactHandler` 实例来注册它们。`org.gradle.api.artifacts.dsl.ArtifactHandler` 接口负责定义和发布包。使用 `Project#artifacts` 方法来注册项目生成的新包。在闭包里面，你可以将 `org.gradle.api.tasks.bundling.AbstractArchiveTask` 类型的 task（如 `Jar`、`Zip` 或 `Tar`）指定给 `archives` 配置。或者，你也可以指定 `java.io.File` 类型的实例，以防打包 task 没有生成相应的包。

archives 配置好后,执行 assemble task 就能自动创建自定义的 JAR 文件了:

```
$ gradle assemble
:compileJava UP-TO-DATE
:compileGroovy
:processResources
:classes
:jar
:groovydoc UP-TO-DATE
:groovydocJar
:sourcesJar
:assemble
```

标准的 JAR task

JAR task 打包了项目的 Groovydocs

JAR task 绑定了项目的源代码

如预期的一样,在 build/libs 目录下生成了这些标准的 JAR 文件:

```
├── build
│   ├── libs
│   │   ├── cloudbees-plugin-1.0-groovydoc.jar
│   │   ├── cloudbees-plugin-1.0-sources.jar
│   │   └── cloudbees-plugin-1.0.jar
│   └── ...
├── build.gradle
└── src
```

生成的 Groovydocs JAR 文件

生成的源代码 JAR 文件

包含类文件和基本数据的标准 JAR 文件

你看到了为项目生成附加包定义 task 是多么的容易,以及如何将其注册到组建生命周期过程中。在软件交付过程中,你可能想要创建一个单独的包,或者甚至多个包包含一系列独特的包文件。通常当你需要面向不同的操作系统、特定的用户群,或者多样化的软件产品风格的时候就会有这样的需求。我们来看看使用 Gradle 如何创建分发包。

### 14.1.2 创建分发包

使用 Gradle 创建自定义的分发包非常简单。对于你想创建的每一个包,你都可以添加一个增强的 Zip 或者 Tar 类型的 task。虽然这种方法对于小规模的分发包来说很实用,但是你必须有一个好的命名模式来清楚地表达它们的意图。

Gradle 的 distribution 插件提供了一种更精简和具有表述性的方法来解决这个问题。这个插件暴露出来的表达性语言能够让你描述一系列的分发包,而不需要手动声明 task。task 创建过程是由插件背后操控的。插件能够帮你生成 ZIP 或者 TAR 文件形式的分发包。

假设你想要为 CloudBees 插件项目创建一个新的捆绑了插件 JAR 文件、源代码 JAR 文件和 Groovydocs JAR 文件的分发包。下面的清单演示了如何应用 distribution 插件并指定你想要打包的目标目录。

## 清单 14.2 构建分发包

```

apply plugin: 'distribution'

distributions {
    main {
        baseName = archivesBaseName
        contents {
            from { libsDir }
        }
    }
}

```

标准分发包配置闭包默认名为 main

分发文件的基本名字

打包 build/libs 目录下的所有文件

使用这段代码，你可以决定创建 ZIP 包还是 TAR 包。distZip task 用来生成 ZIP 包，distTar 用来生成 TAR 包。通常，你只会需要一种格式。TAR 文件通常在 UNIX 操作系统上比较常见。对于跨平台的分发包，ZIP 文件通常是最优的选择。如下命令行输出显示了创建 ZIP 分发包的过程：

```

$ gradle assemble distZip
:compileJava UP-TO-DATE
:compileGroovy
:processResources
:classes
:jar
:groovydoc UP-TO-DATE
:groovydocJar
:sourcesJar
:assemble
:distZip

```

分发包被放置在 build/distributions 目录下。如下目录树显示了所生成的文件：

```

.
├── build
│   ├── distributions
│   │   ├── cloudbees-plugin-1.0.zip
│   │   └── libs
│   │       ├── cloudbees-plugin-1.0-groovydoc.jar
│   │       ├── cloudbees-plugin-1.0-sources.jar
│   │       └── cloudbees-plugin-1.0.jar
│   │       ...
│   ├── build.gradle
│   └── src

```

ZIP 分发包

源文件被包含在了分发包中

distribution 插件被设计成可以支持多个分发包。对于每个额外的分发包，你需要在 distributions 闭包中添加其他配置块。清单 14.3 演示了一个例子。第一个标准的分发包捆绑了 build/libs 目录下的所有 JAR 文件，你想要创建一个仅仅包含文档文件的分发包。对于文档文件，你分成了源文件 JAR 和 Groovydoc JAR。



清单 14.3 配置自定义的分发包

```
distributions {
    main {
        ...
    }
    docs {
        baseName = "$archivesBaseName-docs"
        contents {
            from(libsDir) {
                include sourcesJar.archiveName
                include groovydocJar.archiveName
            }
        }
    }
}
```

自定义的分发配置闭包

分发文件的基本名字

只打包 build/libs 下面的源代码和 Groovydoc JAR 文件

你可能注意到了分发包配置块的名字为 docs。插件自动为非标准的分发包创建了 task 名。除了已经存在的 task 外，你现在可以使用 docsDistZip 和 docsDistTar 来生成文档分发包了。生成分发包——这次生成一个 TAR 文件：

```
$ gradle assemble docsDistTar
:compileJava UP-TO-DATE
:compileGroovy UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:jar UP-TO-DATE
:groovydoc UP-TO-DATE
:groovydocJar UP-TO-DATE
:sourcesJar UP-TO-DATE
:assemble UP-TO-DATE
:docsDistTar
```

分发包输出目录现在还包含了 TAR 文件：

```
.
├── build
│   ├── distributions
│   │   ├── cloudbees-plugin-1.0.zip
│   │   └── cloudbees-plugin-docs-1.0.tar
│   ├── libs
│   │   ├── cloudbees-plugin-1.0-groovydoc.jar
│   │   ├── cloudbees-plugin-1.0-sources.jar
│   │   └── cloudbees-plugin-1.0.jar
│   └── ...
├── build.gradle
└── src
```

标准的 ZIP 分发包

文件打包成了 TAR 文件

你看到了如何为插件项目声明和创建分发包。虽然功能强大且具有描述性，但是你的项目可能有更复杂或者特定平台功能的需求；比如，创建一个桌面安装文件

或者生成一个 RPM 包。如果你觉得 distribution 插件不能满足这些需求，那么可以看看第三方 Gradle 插件。有些第三方插件可能提供了你想要的功能。

总结一下你目前所做的：你的项目能够生成插件 JAR 文件和两个文档压缩包，其中包含了项目的源代码和 Groovydocs。现在你可以将这个插件分享给全世界了。接下来，我会告诉你如何将这些包发布到二进制仓库中。

## 14.2 发布

在第 5 章中，我们从使用者的角度讨论了依赖管理。你学到了如何在构建脚本中定义依赖和仓库。Gradle 的依赖管理器会依次定位这些依赖，下载并存储到本地缓存中，让你能够在构建的时候使用。

在本章中，你会扮演包生产者的角色。你会学到如何发布包到本地或者远程仓库中。在发布过程中一个非常重要的步骤就是为这些包生成基本信息。基本信息通常以 XML 格式的文本文件形式存储，能够包含足够的关于对应的包的信息。下面给出了一些常用的基本信息类型：

- 包相关的通用信息，如名字、描述、开发人员以及源代码或文档的链接
- 可用的包的版本
- 包的传递性依赖
- 使用者必须拥有的许可权

Gradle 可以帮助生成这些基本信息，并上传到不同类型的仓库中。图 14.2 演示了 Gradle 构建、生成的包和一些仓库的交互关系。

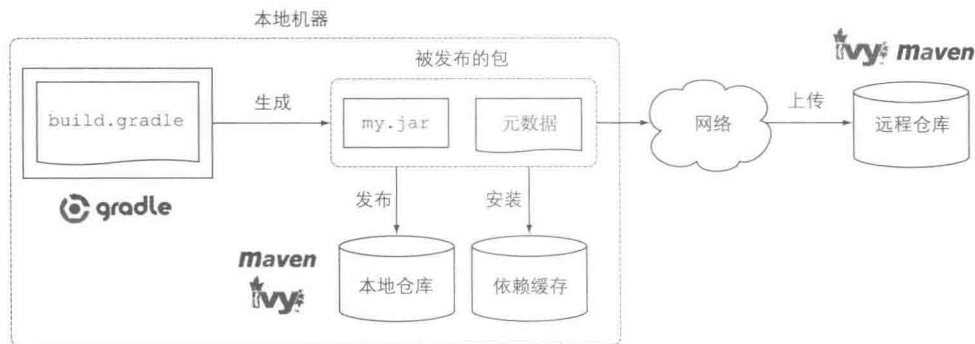


图 14.2 发布包到本地和远程仓库中

你能够找到的最常用的仓库都是基于 Maven 或者 Ivy 的。在撰写本书时，Gradle 并没有提供自己的仓库。在我们的例子中，你会看到如何发布到使用最广泛的仓库：Maven 中。

## 14.2.1 发布到 Maven 仓库中

还记得在本章前面部分，所生成的三个不同的包吗？它们被生成到了 `build/libs` 目录下：

- `cloudbees-plugin-1.0.jar`
- `cloudbees-plugin-1.0-groovydoc.jar`
- `cloudbees-plugin-1.0-sources.jar`

现在将它们发布到 Maven 仓库中。我们看看如何将它们发布到三种类型的 Maven 仓库中，如图 14.3 所示。

- 位于 `<USER_HOME>/~/.m2/repository` 目录下的本地缓存。
- 本地文件系统中任意目录下的仓库。
- 通过 HTTP(S) 可访问的远程二进制仓库。下面例子使用了流行的 JFrog Artifactory。

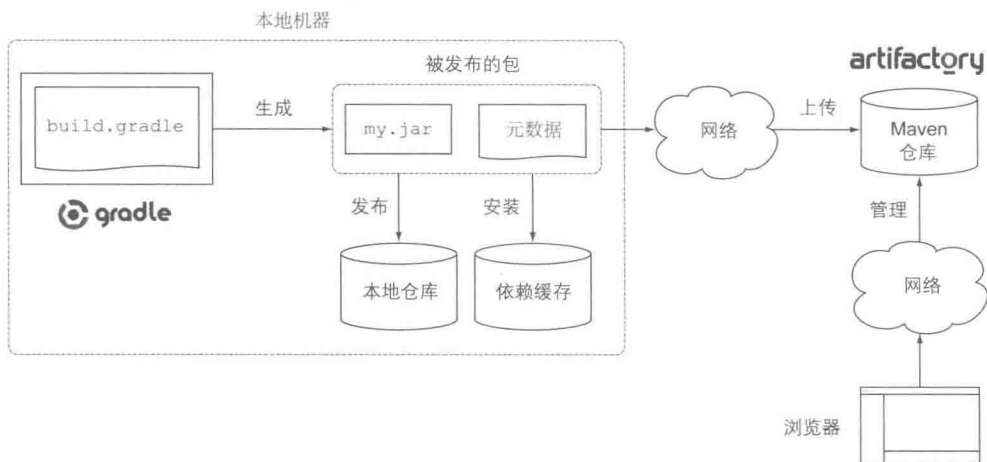


图 14.3 发布包到 Maven 仓库

企业级项目不应该依赖于 Maven Central 作为主要的依赖管理源。为了能够产生稳定的和可重复使用的构建，应该为你的项目创建一个内部仓库。

有一系列产品可以使用；比如 Sonatype Nexus (<http://www.sonatype.org/nexus/>) 和 JFrog Artifactory ([http://www.jfrog.com/home/v\\_artifactory\\_opensource\\_overview](http://www.jfrog.com/home/v_artifactory_opensource_overview))。我们不会讨论如何创建这样一个仓库。请参考相应产品的安装手册获取更多的信息。现在，我们假设你已经安装好了 Artifactory 作为依赖管理器，运行在 <http://localhost:8081/artifactory> 上。

## 14.2.2 老的和新的发布机制

发布是软件生命周期中很重要的一步。Gradle 早期支持使用 Maven 插件中的 Upload task 来发布包。虽然 Maven 插件被早期的 Gradle 用户使用并且能够很好地在产品环境中工作，但是很明显，我们需要一种更具描述性和优雅的 DSL 来描述项目的发布过程。

从 Gradle 1.3 开始，通过 maven-publish 和 ivy-publish 插件引入了新的发布机制。虽然其功能还不够成熟，但是它还是与已经存在的发布方法共存，并且会在以后的 Gradle 版本中取代它们。基于这个前景，我们的大多数讨论会基于新的发布机制，因为它使你的构建脚本不会过时。在接下来的部分中，你会使用 maven-publish 将 CloudBees 插件项目产生的包发布到 Maven 仓库中。如果你想要使用 Ivy 仓库，那么可以按照相似的方法使用 ivy-publish。两个插件暴露的 DSL 非常相似，所以只需很小的修改，你就可以使用 Ivy。我们还是来看看如何发布 JAR 包到 Maven 仓库中吧。

## 14.2.3 声明软件组件为 Maven 发布包

Gradle 项目通过应用一个特殊的插件能够在执行 assemble task 的时候预配置生成一个主要的输出包。你已经看到了这个特性的很多例子。应用了 Java 或 Groovy 插件的项目会生成一个 JAR 文件，应用了 War 插件的项目会生成一个 WAR 文件。在发布插件中，这个包被叫作软件组件。

如下清单将 Maven 发布插件应用到 CloudBees 项目中，并且使用其 DSL 描述了一个发布包：一个 Java 软件组件。当我说到 Java 软件组件的时候，我指的就是所生成的 JAR 文件。

清单 14.4 发布 JAR 组件到 Maven 仓库中

```
apply plugin: 'maven-publish'

publishing {
    publications {
        plugin(MavenPublication) {
            from components.java
            artifactId 'cloudbees-plugin'
        }
    }
}
```

定义 MavenPublication 类型的发布包名字

添加 JAR 组件到发布包列表中

声明发布包的 ID

这就是生成 JAR 文件的基本信息，并且发布到 Maven 仓库的所有配置。因为还没有定义仓库，所以你能发布到位于 `.m2/repository` 目录下的本地 Maven 缓存中。查看项目可用的发布 task 列表：

```
$ gradle tasks
:tasks
```

```
-----
All tasks runnable from root project
-----
```

```
...
```

```
Publishing tasks
-----
```

```
publish - Publishes all publications produced by this project.
```

```
publishPluginPublicationToMavenLocal - Publishes Maven publication
➡ 'plugin' to the local Maven repository.
```

```
publishToMavenLocal - Publishes all Maven publications produced by
➡ this project to the local Maven cache.
```

```
...
```

发布项目  
中所有的包

发布指定  
的包到本地  
Maven 缓存

只发布软件  
组件到本地  
Maven 缓存中

这个项目提供了三个不同的发布 task。刚开始看起来可能很迷惑，但是如果你需要对发布过程拥有更细粒度控制的话，这就清楚多了。通过合适的 task，你可以有选择性地说明想要将哪个（些）包发布到什么仓库中。你可能注意到了随意一个名字都变成了 task 名字的一部分：`plugin` 名字变成 task 名字 `publishPluginPublicationToMavenLocal` 的一部分。这使得发布的 task 名字非常具有表述性。接下来，我们来实际使用这些 task。

#### 14.2.4 发布软件组件到本地 Maven 缓存中

本地 Maven 缓存对于 Gradle 用户非常特殊，主要有两个原因。在第 5 章中，你了解到 Gradle 尝试重用本地 Maven 缓存中的包。这对于迁移的 Maven 用户来说，主要的好处就是不需要重新下载包。另一个原因是你想要发布到本地仓库时，可能在你的项目中使用了多种构建工具。一个项目可能使用 Gradle 来发布包，另一个项目可能通过 Maven 来使用。

假设你想要发布插件 JAR 文件，包招所生成的 POM 文件。如下控制台输出显示了 task 执行的结果：

```
$ gradle publishToMavenLocal
:generatePomFileForPluginPublication
:compileJava UP-TO-DATE
:compileGroovy UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
```

生成用于发布的 POM 文件

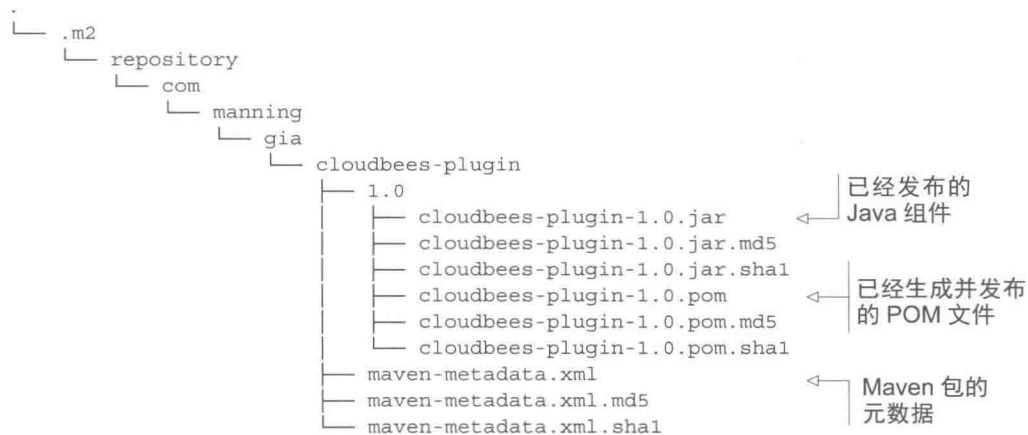
```

:jar UP-TO-DATE
:publishPluginPublicationToMavenLocal
Uploading: com/manning/gia/cloudbees-plugin/1.0/cloudbees-plugin-
1.0.jar to repository remote at file:/Users/Ben/.m2/repository/
Transferring 43K from remote
Uploaded 43K
:publishToMavenLocal

```

上传 Java 组件到本地的 Maven 仓库

在发布 JAR 文件的过程中, `generatePomFileForPluginPublication` task 生成了 POM 文件。与 `publishPluginPublicationToMavenLocal` task 类似, 它的名字来源于所定义的发布包名称。你可能想知道为什么 POM 文件在控制台输出中没有作为上传的包被列出来。别着急——它只是没有被日志记录而已。最后本地 Maven 缓存目录下的包如下:



被上传的包遵循 Maven 仓库中依赖的典型格式。目录 `com/manning/gia` 来源于项目的 `group` 属性, 包 ID (`cloudbees-plugin`) 遵循项目名字, 版本反映了项目中 `version` 属性的值。`.md5` 和 `.sha1` 文件是校验文件, 用于检查相关文件的完整性。

### 如何改变包的发布名字

在默认情况下, 发布的包名来源于项目名字。你可能记得在前面的章节中项目名与项目目录名保持一致。理解对于比如 `archiveBaseName` 属性的任何修改都不会影响发布名很重要, 即使所打出来的包是按你的要求命名的。

有时候你可能需要使用一个不同的发布名。每个 `MavenPublication` 都允许配置自定义的发布属性。其中一个属性就是包 ID (artifact ID), 如清单 14.4 所示。或者, 你也可以在 `settings.gradle` 文件中设置项目名。如下示例演示了如何将项目名修改为 `my-plugin`:

```
rootProject.name = 'my-plugin'
```

这样，发布的包名就是 my-plugin-1.0.jar，本地 Maven 缓存中的目录名为 com/manning/gia/my-plugin。

相同的命名模式也反映在了所生成的 POM 文件的依赖属性 groupId、artifactId 和 version 上。如下清单显示了 cloudbees-plugin-1.0.pom 文件的完整内容。

清单 14.5 为发布的软件组件生成的 POM 文件

```
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  ➤ http://maven.apache.org/xsd/maven-4.0.0.xsd"
  ➤ xmlns="http://maven.apache.org/POM/4.0.0"
  ➤ xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.manning.gia</groupId>
  <artifactId>cloudbees-plugin</artifactId>
  <version>1.0</version>
  <dependencies>
    <dependency>
      <groupId>com.cloudbees</groupId>
      <artifactId>cloudbees-api-client</artifactId>
      <version>1.4.0</version>
      <scope>runtime</scope>
    </dependency>
  </dependencies>
</project>
```

发布包的依赖属性

发布包能够正常工作所必需的传递性依赖

在生成 POM 文件的时候，发布插件自动确定 runtime 配置中的项目依赖并将它们定义为传递性依赖。在这里，你可以看到对于 CloudBees API 客户端类库就是这样的。记住发布 API 只会定义与 JAR 文件的使用者相关的依赖。用于构建项目的依赖（比如测试类库）不会被发布出去。尽管这与 Maven 的默认方式有一点点不同，但是这使得基本信息更加清晰简洁，并且不会有任何坏处。

非常好，你现在可以发布项目默认的包了。但是如果你想要发布额外的包该怎么做呢？你的插件项目对于提供源代码 JAR 和 Groovydocs JAR 有特殊的需求。我们来看看如何使用 DSL 来声明这些文件。

### 14.2.5 声明自定义的发布包

通过使用 API 类 MavenPublication 提供的 artifact 方法来注册额外的包。这个方法要求你提供一个参数。这个参数可以是任何 AbstractArchiveTask 类

型的 task，也就是有两个文档 JAR 文件 task 的情况，或者任何可以被转换成 `java.io.File` 的对象。发布 API 的 Javadocs 就是一个使用这两种声明方式的例子。下面的清单显示了如何定义 `sourcesJar` 和 `groovydocJar` 为 Maven 发布包。

清单 14.6 发布额外的包到 Maven 仓库

```

apply plugin: 'maven-publish'

publishing {
    publications {
        plugin(MavenPublication) {
            from components.java
            artifactId 'cloudbees-plugin'

            artifact sourcesJar
            artifact groovydocJar
        }
    }
}

```

声明 `MavenPublication` 类型的发布名称

添加项目的 JAR 组件到发布列表中

定义发布包 ID

添加自定义包到发布列表中

在定义自定义的包时有一个重要的细节需要记住。发布插件只允许发布一个没有标识符的包。通常就是软件组件，它的名字是 `cloudbees-plugin-1.0.jar`。所有其他的包都需要提供一个标识符。对于源代码 JAR 文件，标识符就是 `sources`，对于 Groovydocs JAR 文件，标识符就是 `groovydoc`。再次运行之前使用的发布 task，看看能不能发布自定义的包到 Maven 缓存中：

```

$ gradle publishToMavenLocal
:generatePomFileForPluginPublication
:compileJava UP-TO-DATE
:compileGroovy
:processResources
:classes
:groovydoc
:groovydocJar
:jar
:sourcesJar
:publishPluginPublicationToMavenLocal
Uploading: com/manning/gia/cloudbees-plugin/1.0/cloudbees-plugin-
➡ 1.0.jar to repository remote at file:/Users/Ben/.m2/repository/
Transferring 43K from remote
Uploaded 43K
Uploading: com/manning/gia/cloudbees-plugin/1.0/cloudbees-plugin-1.0-
➡ sources.jar to repository remote at file:/Users/Ben/.m2/repository/
Transferring 7K from remote
Uploaded 7K

```



```

Uploading: com/manning/gia/cloudbees-plugin/1.0/cloudbees-plugin-1.0-
groovydoc.jar to repository remote at file:/Users/Ben/.m2/repository/
Transferring 33K from remote
Uploaded 33K
:publishToMavenLocal

```

控制台输出现在显示了所有定义的发布包都被上传到了仓库中。快速地检查一下缓存目录树就能看到所期望的结果：



太好了——所有你想要发布到 Maven 缓存中的包都可以被发布了。包使用者根据默认的 POM 文件中的基本信息来识别插件 JAR 文件和其对应的传递性依赖。在 14.2 节中，我们讨论了 POM 文件中可以包含的信息。发布 API 使得这些信息可以完全被自定义。在接下来一节中，我们会讨论如何修改所生成的 POM 文件。

### 14.2.6 修改所生成的 POM 文件

仓库中的 POM 文件描述了包尽可能多的信息。至少，你应该提供包的作用、软件许可证和文档链接详细信息，以便终端用户可以知道这个包提供了什么功能，以及在自己的项目中如何使用。想要知道 POM 文件中被配置的信息，最好的方式就是查看 <http://maven.apache.org/pom.html> 上的 POM 参考指南。这个指南描述了 XML 结构和可使用的配置标签。

所生成的标准 POM 文件可以通过 `pom.withXml` 来修改。通过调用 `asNode()` 方法，你可以遍历 POM 文件的根节点。可以添加新的节点和修改已有

的节点（除了 groupId、artifactId 和 version 标识符）。如下清单显示了如何为插件的 POM 文件添加更多的信息。

**清单 14.7 发布额外的包到 Maven 仓库**

```
apply plugin: 'maven-publish'

publishing {
    publications {
        plugin(MavenPublication) {
            from components.java
            artifactId 'cloudbees-plugin'

            pom.withXml {
                def root = asNode()
                root.appendNode('name', 'Gradle CloudBees plugin')
                root.appendNode('description', 'Gradle plugin for managing
                    ➤ applications and databases on CloudBees
                    ➤ RUN@cloud.')
                root.appendNode('inceptionYear', '2013')

                def license = root.appendNode('licenses').appendNode('license')
                license.appendNode('name', 'The Apache Software License,
                    ➤ Version 2.0')
                license.appendNode('url', 'http://www.apache.org/licenses/LICENSE-
                    ➤ 2.0.txt')
                license.appendNode('distribution', 'repo')

                def developer = root.appendNode('developers')
                developer.appendNode('developer')
                developer.appendNode('id', 'bmuschko')
                developer.appendNode('name', 'Benjamin Muschko')
                developer.appendNode('email', 'benjamin.muschko@gmail.com')
            }

            artifact sourcesJar
            artifact groovydocJar
        }
    }
}
```

配置 POM XML 元素的方法

下面的清单显示了重新生成的 POM 文件，反映了基本信息的变化。

**清单 14.8 修改后插件的 POM 文件**

```
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    ➤ http://maven.apache.org/xsd/maven-4.0.0.xsd"
    ➤ xmlns="http://maven.apache.org/POM/4.0.0"
    ➤ xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.manning.gia</groupId>
    <artifactId>cloudbees-plugin</artifactId>
    <version>1.0</version>
    <dependencies>
```

```

<dependency>
  <groupId>com.cloudbees</groupId>
  <artifactId>cloudbees-api-client</artifactId>
  <version>1.4.0</version>
  <scope>runtime</scope>
</dependency>
</dependencies>
<name>Gradle CloudBees plugin</name>
<description>Gradle plugin for managing applications and databases on
    CloudBees RUN@cloud.</description>
<inceptionYear>2013</inceptionYear>
<licenses>
  <license>
    <name>The Apache Software License, Version 2.0</name>
    <url>http://www.apache.org/licenses/LICENSE-2.0.txt</url>
    <distribution>repo</distribution>
  </license>
</licenses>
<developers>
  <developer>
    <id>bmuschko</id>
    <name>Benjamin Muschko</name>
    <email>benjamin.muschko@gmail.com</email>
  </developer>
</developers>
</project>

```

至此，你已经能够发布包到本地 Maven 缓存了。如果你想要发布到本地文件系统中的 Maven 仓库或者之前设置好的 Artifactory 中该如何做呢？没有更简单的了——发布 API 也允许你定义仓库。

### 14.2.7 发布到本地 Maven 仓库中

在第 8 章中，你使用了“老”的 Maven 插件上传项目包到本地 Maven 仓库中，所以它能够被其他的 Gradle 项目所使用。你使用这种方式在本机机器上来测试插件的功能，而不需要发布到公共的仓库中。Maven 发布插件提供了同样的功能。

每个你想要发布到的目标仓库都需要在 repositories 配置块中定义并通过使用发布 DSL 暴露。如下清单定义了一个 Maven 仓库，它位于与项目目录平行的 repo 目录中。

#### 清单 14.9 发布到本地 Maven 仓库

```

apply plugin: 'maven-publish'

publishing {
  publications {
    plugin(MavenPublication) {
      ...
    }
  }
}

```

```

repositories {
    maven {
        name 'myLocal'
        url "file://$projectDir/../../repo"
    }
}

```

Maven 仓库可选名字

本地 Maven 仓库的文件 URL

仓库名字属性是可选的。如果你指定了名字，它就会成为对应的发布 task 名字的一部分。比如指定名字为 myLocal，你的 task 名字就会自动变成 publishPluginPublicationToMyLocalRepository。在同时处理多个仓库的时候这个功能是非常有用的。运行 task：

```

$ gradle publishPluginPublicationToMyLocalRepository
:generatePomFileForPluginPublication
:compileJava UP-TO-DATE
:compileGroovy UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:groovydoc UP-TO-DATE
:groovydocJar UP-TO-DATE
:jar UP-TO-DATE
:sourcesJar UP-TO-DATE
:publishPluginPublicationToMyLocalRepository
Uploading: com/manning/gia/cloudbees-plugin/1.0/cloudbees-plugin-
➤ 1.0.jar to repository remote at file:/Users/Ben/Dev/books/gradle-in-
➤ action/code/chapter14/publish-maven-local-repository/repo
Transferring 43K from remote
Uploaded 43K
Uploading: com/manning/gia/cloudbees-plugin/1.0/cloudbees-plugin-1.0-
➤ sources.jar to repository remote at file:/Users/Ben/Dev/books/gradle-
➤ in-action/code/chapter14/publish-maven-local-repository/repo
Transferring 7K from remote
Uploaded 7K
Uploading: com/manning/gia/cloudbees-plugin/1.0/cloudbees-plugin-1.0-
➤ groovydoc.jar to repository remote at
➤ file:/Users/Ben/Dev/books/gradle-in-action/code/chapter14/publish-maven-
➤ local-repository/repo
Transferring 33K from remote
Uploaded 33K

```

Gradle 已经智能到能够创建本地仓库，即使根目录还不存在。正如预期的一样，你会在正确的路径下找到上传的包：

```

.
├── cloudbees-plugin
│   └── ...
├── repo
│   ├── com
│   │   ├── manning
│   │   │   ├── gia
│   │   │   └── cloudbees-plugin

```

插件项目发布包

与插件项目并行的本地 Maven 仓库

```
├── 1.0
│   ├── cloudbees-plugin-1.0-groovydoc.jar
│   ├── cloudbees-plugin-1.0-groovydoc.jar.md5
│   ├── cloudbees-plugin-1.0-groovydoc.jar.sha1
│   ├── cloudbees-plugin-1.0-sources.jar
│   ├── cloudbees-plugin-1.0-sources.jar.md5
│   ├── cloudbees-plugin-1.0-sources.jar.sha1
│   ├── cloudbees-plugin-1.0.jar
│   ├── cloudbees-plugin-1.0.jar.md5
│   ├── cloudbees-plugin-1.0.jar.sha1
│   ├── cloudbees-plugin-1.0.pom
│   ├── cloudbees-plugin-1.0.pom.md5
│   └── cloudbees-plugin-1.0.pom.sha1
├── maven-metadata.xml
├── maven-metadata.xml.md5
└── maven-metadata.xml.sha1
```

在接下来的一节中，我们会看看如何通过 HTTP 发布同样的包到远程仓库中。

## 14.2.8 发布到远程的 Maven 仓库中

如果想要让你的包对其他团队或者组织中的相关人员可用，远程仓库能够提供帮助。出于测试目的，我们在本地机器上创建了 Artifactory 实例。记住这个仓库可以被创建在任何通过 HTTP(S) 可访问的企业的网络范围内。Artifactory 默认预配置了两个 Maven 仓库：

- `libs-snapshot-local`：用于发布还处于开发阶段的包，会加上 `-SNAPSHOT` 后缀
- `libs-release-local`：用于发布已经成型的包，将会去掉 `-SNAPSHOT` 后缀

这两个仓库都暴露了一个专用的 HTTP URL，并且能够控制谁可以上传包，怎样上传。一个安全的仓库需要你提供相应的认证凭证。如果你不提供这些属性，构建将会失败，因为其无法认证你的上传。为了防止失败，需要将这些敏感数据放到版本控制系统中，你可以通过 `gradle.properties` 文件来指定相应的属性。Gradle 会在运行时自动读取这个文件的内容，并且使这些属性对构建脚本可用。目前，你将使用默认的 Artifactory 管理员认证信息：

```
artifactoryUsername = admin
artifactoryPassword = password
```

接下来，你会编写一些代码根据 `version` 属性将包发布到其中一个仓库中。下面的清单跟之前的例子看起来有些不同。最大的不同就是你提供了凭证来认证上传动作。

## 清单 14.10 发布到远程 Maven 仓库

```

apply plugin: 'maven-publish'

ext {
    artifactoryBaseUrl = 'http://localhost:8081/artifactory'
    artifactorySnapshotRepoUrl = "$artifactoryBaseUrl/libs-snapshot-local"
    artifactoryReleaseRepoUrl = "$artifactoryBaseUrl/libs-release-local"
}

publishing {
    publications {
        plugin(MavenPublication) {
            ...
        }
    }
}

repositories {
    maven {
        name 'remoteArtifactory'
        url project.version.endsWith('-SNAPSHOT') ?
            ➡ artifactorySnapshotRepoUrl : artifactoryReleaseRepoUrl ➡

        credentials {
            username = artifactoryUsername
            password = artifactoryPassword
        }
    }
}

```

定义  
Artifactory  
仓库 URL  
的属性

Maven 仓库的  
可选名字

根据项目版本  
选择快照或者  
发布仓库

设置上传凭证

你命名远程仓库为 `remoteArtifactory`。发布插件将仓库名字整合到了上传项目包到 Artifactory 使用的 task 名字中：

```

$ gradle publishPluginPublicationToRemoteArtifactoryRepository
:generatePomFileForPluginPublication
:compileJava UP-TO-DATE
:compileGroovy
:processResources
:classes
:groovydoc
:groovydocJar
:jar
:sourcesJar
:publishPluginPublicationToRemoteArtifactoryRepository
Uploading: com/manning/gia/cloudbees-plugin/1.0/cloudbees-plugin-
➡ 1.0.jar to repository remote at
➡ http://localhost:8081/artifactory/libs-release-local
Transferring 43K from remote
Uploaded 43K
Uploading: com/manning/gia/cloudbees-plugin/1.0/cloudbees-plugin-1.0-
➡ sources.jar to repository remote at
➡ http://localhost:8081/artifactory/libs-release-local
Transferring 7K from remote
Uploaded 7K

```

```
Uploading: com/manning/gia/cloudbees-plugin/1.0/cloudbees-plugin-1.0-  
groovydoc.jar to repository remote at  
http://localhost:8081/artifactory/libs-release-local  
Transferring 33K from remote  
Uploaded 33K
```

上传成功后，你应该能够通过所提供的 group ID 来浏览仓库。你会看到一个用特定版本号命名的目录，其包含 CloudBees 插件对应的 POM 文件和 JAR 文件（图 14.4）。

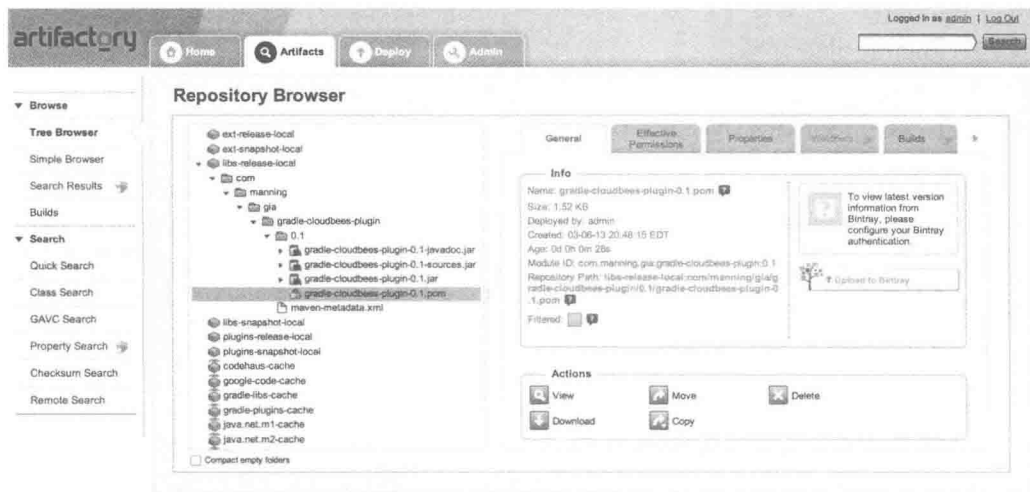


图 14.4 Artifactory 的 Repository Browser 中显示的上传包

### 使用针对特定产品的插件简化发布代码

一些二进制仓库产品提供了 Gradle 插件来简化发布过程。它们的插件提供了标准的、针对特定产品的 DSL，并且不能够被其他仓库使用。可以将其看作 Gradle 发布 API 中更高级别的 API，同时添加了更多的特性（如构建信息）。使用这些插件能够减少你为发布包编写的代码。只有当你知道项目只会发布到某种类型的仓库中时，你才可以使用这些插件。

JFrog 提供了这样的插件。在此我们不会讨论其用法，但是你可以到其 Wiki 页面 <http://wiki.jfrog.org/confluence/display/RTF/Gradle+Artifactory+Plugin> 查看相关信息。

## 14.3 发布到公共的二进制仓库

发布到企业网络范围内的仓库中对于在团队中共享是一个可行的解决方案。同时也为企业的知识产权提供了保护。

与开源软件利益相关的组织喜欢为 Gradle 社区或者对使用他们的代码感兴趣的用戶做贡献。这可以通过让它们的二进制仓库能够通过互联网访问，或者将二进制包上传到公共仓库中来达到。最流行的公共仓库就是 JFrog Bintray 的 JCenter 和 Maven Central。在本章中，你会学到如何将 CloudBees 插件发布到这两个仓库中。我们从 Bintray 开始介绍。

### 14.3.1 发布到 JFrog Bintray 中

JFrog Bintray (<https://bintray.com/>) 在公共仓库中算是一个新秀。Bintray 不只是一个托管二进制仓库的提供商。它提供了一个 Web Dashboard，集成了搜索功能、社交功能和下载统计，以及用来管理仓库和二进制包的 RESTful API。Bintray 于 2013 年年初发布，在其发布后前几个月就获得了巨大的用户群增长。现在，你可以看到很多开源项目都已经迁移到 Bintray 上。在将包发布到 Bintray 之前，你需要先设置一个账户。

#### 设置 Bintray

在开始之前，Bintray 要求你创建一个账户、一个仓库和一个包。简要步骤如下：

- 1 打开浏览器，访问 <https://bintray.com/>，如果你还没有账户，先注册一个。
- 2 注册完成后，登录。单击 New Repository 按钮来配置一个仓库。给仓库取名为 gradle-plugins 并选择 Maven 仓库格式。
- 3 点击仓库并新建一个包叫作 gradle-cloudbees-plugin。包作为容器来盛装项目所生成的二进制包。在包里面可以放置任意多个二进制包。
- 4 发布到包里的所有二进制包都只能被当作 Gradle 项目依赖被下载，而且需要在仓库配置中提供相应的认证信息。为了让所有的 Gradle 用户都能够访问这个包，你可以将其与免费的、公共的 Maven 仓库 Bintray 的 JCenter 进行同步。

如果你在配置过程中遇到新的问题也不要担心。Bintray 用户指南 (<https://bintray.com/docs/help/bintrayuserguide.html>) 可以给你提供更多的信息，让你了解 Bintray 术语，并快速入门。下面介绍将插件二进制包发布到 Bintray 中。

#### 将二进制包上传到 Bintray 包中

在架构上，上传包到 Bintray 与上传包到远程 Maven 仓库看起来没有什么不同(图 14.5)。主要不同的地方就是可以将私有仓库中的包与公共仓库 JCenter 共享。



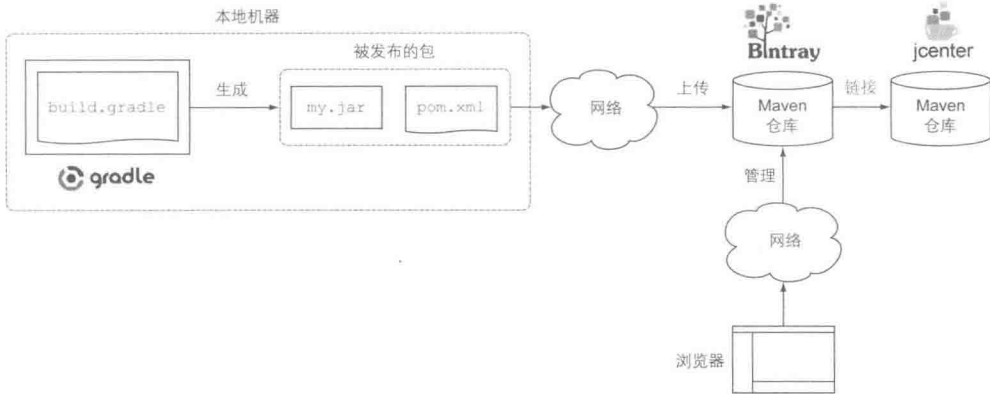


图 14.5 发布包到 Bintray

在上传包之前 Bintray 并不要求你为其签名，但是你可以上传之后再签名。因此你只需要对原始的构建脚本做很小的修改，如下面的清单所示。你需要定义一个新的 Maven 仓库，提供仓库的 URL，并给出 Bintray 账户的凭证信息。

#### 清单 14.11 发布到 Bintray 仓库

```

ext {
    bintrayBaseUrl = 'https://api.bintray.com/maven'
    bintrayUsername = 'bmuschko'
    bintrayRepository = 'gradle-plugins'
    bintrayPackage = 'gradle-cloudbees-plugin'
}

apply plugin: 'maven-publish'

publishing {
    publications {
        ...
    }

    repositories {
        maven {
            name 'Bintray'
            url "$bintrayBaseUrl/$bintrayUsername/$bintrayRepository/"
            ➡ "$bintrayPackage"

            credentials {
                username = bintrayUsername
                password = bintrayApiKey
            }
        }
    }
}

```

构建 Bintray API URL 的属性

上传用的凭证

组装 Bintray API URL 并赋值给 Maven URL 属性

出于安全考虑，将凭证信息放到主目录下的 `gradle.properties` 文件中。这个文件的内容与如下属性类似：

```
bintrayUsername = bmuschko
bintrayApiKey = 14a5g63385ad861d4c8210da795
```

在 `credentials` 配置块中，使用 API key，而不是密码。你可以在 Bintray 的 Dashboard 中的你的账户 >Edit>API Key 链接下找到 API key。所有设置都完成了，是时候发布插件包让世界变得更美好了：

```
$ gradle publishPluginPublicationToBintrayRepository
:generatePomFileForPluginPublication
:compileJava UP-TO-DATE
:compileGroovy UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:groovydoc UP-TO-DATE
:groovydocJar UP-TO-DATE
:jar
:sourcesJar UP-TO-DATE
:publishPluginPublicationToBintrayRepository
Uploading: org/gradle/api/plugins/gradle-cloudbees-plugin/0.1/gradle-
➤ cloudbees-plugin-0.1.jar to repository remote at
➤ https://api.bintray.com/maven/bmuschko/gradle-plugins/gradle-
➤ cloudbees-plugin
Transferring 179K from remote
Uploaded 179K
Uploading: org/gradle/api/plugins/gradle-cloudbees-plugin/0.1/gradle-
➤ cloudbees-plugin-0.1-sources.jar to repository remote at
➤ https://api.bintray.com/maven/bmuschko/gradle-plugins/gradle-
➤ cloudbees-plugin
Transferring 23K from remote
Uploaded 23K
Uploading: org/gradle/api/plugins/gradle-cloudbees-plugin/0.1/gradle-
➤ cloudbees-plugin-0.1-groovydoc.jar to repository remote at
➤ https://api.bintray.com/maven/bmuschko/gradle-plugins/gradle-
➤ cloudbees-plugin
Transferring 83K from remote
Uploaded 83K
```

看起来上传成功了！你也可以在 Bintray 的 Dashboard 中查看已上传的包。包 `gradle-cloudbees-plugin>Versions>0.1>Files` 下的文件浏览器类似于图 14.6。

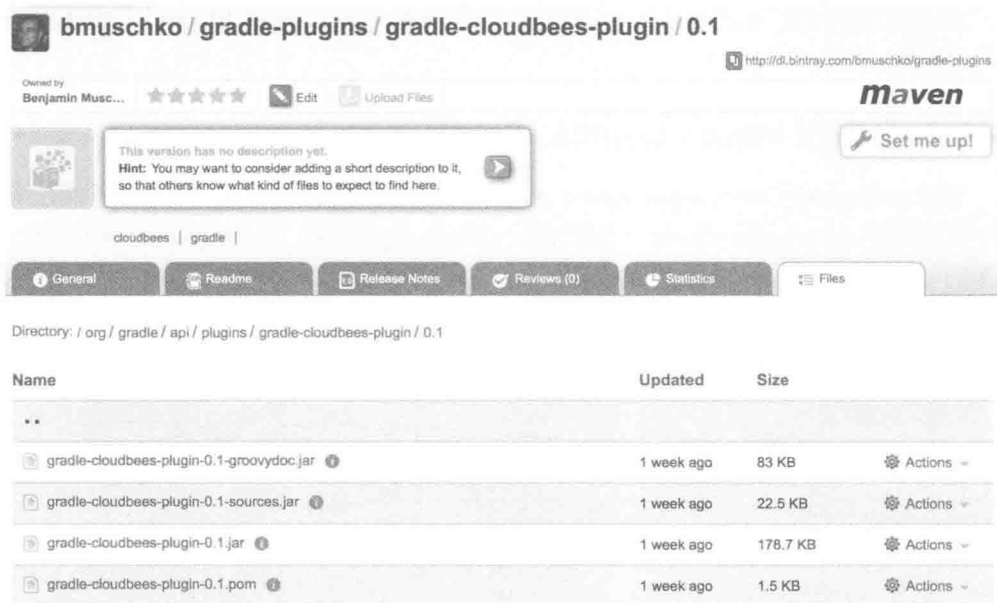


图 14.6 Bintray 的 Dashboard 中显示的已上传的包

### 管理公共二进制包

永远不要小看开源的力量。一旦你将包发布到公共仓库中，其他的程序员就可以开始使用了。Bintray 并不会阻止你删除已经发布的二进制包。尽量不要删除，因为删除可能会破坏其他程序员的构建。如果你需要修复代码中的漏洞，则可以发布一个新的版本并给出相应的漏洞修复说明。

### 使用 Bintray 的 JCenter 中的依赖

你发布到 JCenter 中的包可用了。为了配置 Gradle 项目使用 Bintray 中的插件，声明一个新的依赖引用地址 URL 为 `http://jcenter.bintray.com`，如下面的清单所示。

#### 清单 14.12 使用发布到 Bintray 中的插件

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.manning.gia:gradle-cloudbees-plugin:0.1'
    }
}

apply plugin: 'cloudbees'
```

← 定义 Bintray 的 JCenter 仓库

定义之前上传的插件 JAR 文件的依赖

看起来很简单，是吧？我们也来看看将同一个插件发布到 Maven Central 中需要哪些步骤。

### 14.3.2 发布到 Maven Central

Maven Central (<http://repol.maven.org/maven2/>) 可能是最流行的公共仓库，尤其是开源项目。Sonatype Nexus 负责管理二进制包，这些构成了一个仓库的骨干功能。图 14.7 显示了与 Sonatype OSS 的交互。

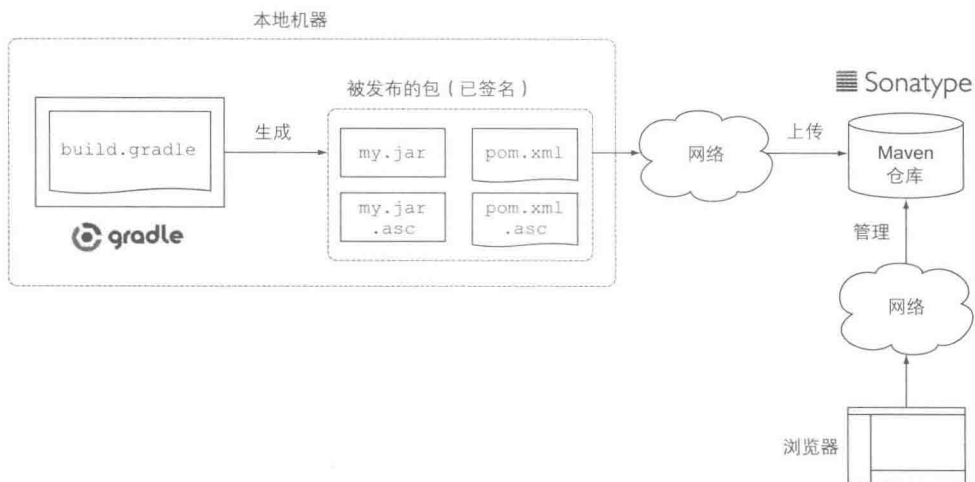


图 14.7 发布包到 Sonatype OSS 中

对于发布包 Sonatype OSS 比 Bintray 有更多的要求。如下清单（使用指南 <https://docs.sonatype.org/display/Repository/Sonatype+OSS+Maven+Repository+Usage+Guide> 中描述了更详细的步骤）解释了如何发布包并使其可用。

- 1 在 <http://issues.sonatype.org/> 上注册一个新的 Sonatype JIRA 账户。你需要等待账户通过审核。
- 2 在 <https://issues.sonatype.org/browse/OSSRH/> 上新建一个 JIRA ticket 用来描述包的基本信息（如 groupId、项目 URL 等）。一旦请求通过审核，你就可以使用请求的 groupId 进行发布了。如果要使用不同的 groupId 进行发布，你需要创建另一个 JIRA ticket。
- 3 为想要发布的包生成 GNU Privacy Guard (GPG) 签名（图 14.7 中的 .asc 文件）。为了生成签名，你可以使用 Gradle 的 signing 插件。你可以到插件文档页面获取更多的信息。在撰写本书时，新发布的 API 并没有提供开箱即用的签名支持。你可能要降级到老的 API 来达到这个目的。

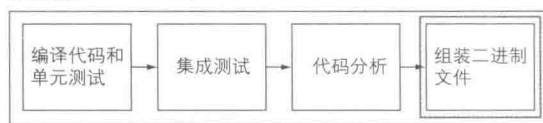
- 4 使用 Gradle 进行发布。
- 5 从 <https://oss.sonatype.org/> 登录 Sonatype OSS 用户界面，跳转到 Staging Repositories 页面，选择包，然后单击 Close 按钮。选择完成后，就可以发布了。单击 Release 按钮开始最后的发布。
- 6 在已发布的包能够通过 Maven Central 被访问之前，Sonatype OSS 需要同步到 Maven Central 仓库。这通常需要花费数小时的时间。
- 7 一旦包发布后，你就不可以对它进行删除或修改。

发布到 Sonatype OSS 的过程需要更多的手动步骤和更多的耐心来等待审核。如果你不能确定为项目选择哪个公共仓库，我推荐使用 Bintray。接下来，你会学习如何将所学知识应用到 To Do 应用程序中。

## 14.4 打包和发布作为构建管道的一部分

在上一章中，你学到了如何安装和配置 Jenkins 的 job 来为 To Do 应用程序创建构建管道。在了解了打包和发布的核心概念之后，你现在可以将所学知识应用到 Web 应用程序中了。首先，你需要对构建脚本做一些扩展，然后通过配置 Jenkins 的 job 扩展构建管道。图 14.8 显示了这个过程。

提交阶段



当前阶段

图 14.8 在构建管道的上下文中创建分发包

在持续交付的过程中，有一些很重要的实践需要讨论。它们最终决定你会如何实现打包和发布的过程。

### 14.4.1 构建一次

在将应用程序部署到目标环境中之前——比如 UAT（user acceptance test）环境——QA 团队用来做手动测试，或者产品环境中，将产品交付给最终用户，可交付的包，也叫作可部署的包，必须要构建出来。团队为每个环境都重新构建这个可交付的包的情况并不少见。这种情况通常基于它们需要包括与环境相关的配置；比如，指向一个为特定环境建立的专用数据库。尽管这种方式能奏效，但是其带来了不必要的风险，可能为每个环境交付了不同的产品。比如，依赖管理器可能引入了新版本的第三方库并且可用，但这并不是你的本意。为了避免任何副作用，你应该只构建一次，然后将可交付的包存储在一个中央的位置。正如你前面所学到的，一个二进制仓库就是一个绝佳的位置。

### 14.4.2 发布一次并重用

二进制仓库要求你发布一个带有唯一属性的包。在第 5 章中，你学到了 Maven 仓库将这些属性描述为 `groupId`、`artifactId` 和 `version` 来做唯一性标识，通常也被叫作坐标。区别构建出来的包的主要属性是版本。对于其他两个属性 `groupId` 和 `artifactId`，在你选定了一个有意义的值后就很少会改变了。随着时间的推移，你会发现越来越多版本的包被存储进来。所上传的包会一直存储在仓库中，直到你删除它们。图 14.9 演示了发布递增版本的流程。

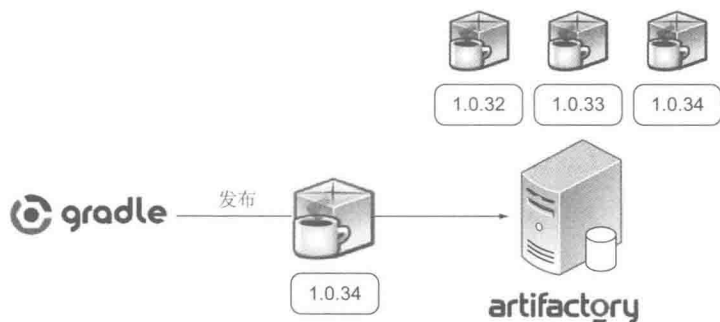


图 14.9 将不同版本的 WAR 包发布到 Artifactory 中

一旦带有具体版本的二进制包被上传到了二进制仓库后，你就可以通过这些属性来获取它并且在构建管道中重用了。一个典型的用例就是部署到不同的环境中。为了通过 HTTP(S) 下载二进制包，许多仓库产品都暴露了 RESTful API。URL 中包括了用来唯一标识二进制包的依赖属性。图 14.10 显示了从 Artifactory 中下载特定版本的包进行部署。

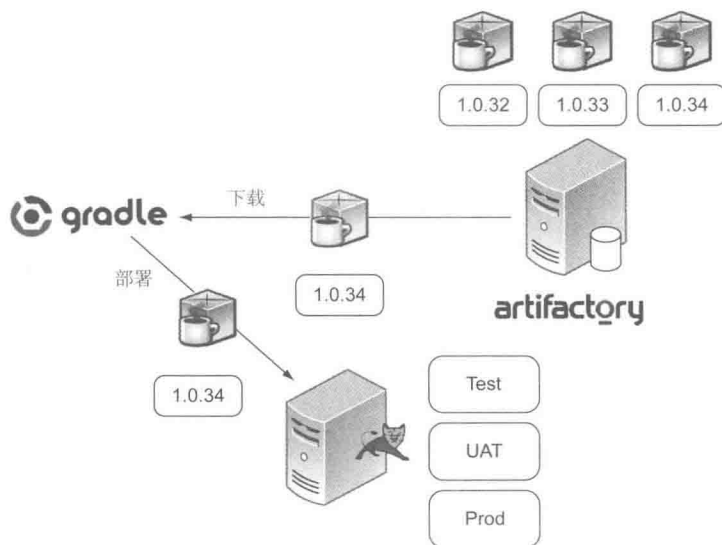


图 14.10 从 Artifactory 中获取 WAR 包进行部署

有了这些背景知识，你可以开始为项目制定版本方案了。

### 14.4.3 选择一个合适的版本管理方案

在组建的过程中包的版本管理变得非常重要。在本节中，我们会讨论在持续交付的核心原则下哪种版本管理方案工作得好，以及如何使用 Gradle 实现一个合适的版本管理策略。一些构建工具为版本管理方案提供了标准格式。我们来看看 Maven 是怎么做的。

#### Maven 的版本管理方案

从概念上讲，Maven 用 snapshot 和 release 来区分版本。一个包的 snapshot 版本意味着其还处于开发阶段，不适用于产品环境。这种状态通过文件名后缀 -SNAPSHOT（比如 todo-webapp-1.0-SNAPSHOT.war）来标识。无论什么时候将包发布到二进制仓库中，它都带有相同的版本。这个包的任何消费者都只能获得最新的 snapshot 版本。因为没有具体的固定版本信息，你不能把其与 VCS 中一个具体的修订版本联系起来。这在调试一个问题的时候成为了主要的瓶颈。因为这个原因，可交付包的 snapshot 版本一定不能部署到产品环境中。

在开发的某个时间点，确定软件的所有功能都完成了。一旦通过了 QA 阶段，它就可以发布到产品环境中了。这时 -SNAPSHOT 后缀就从版本信息中去掉并发布到产品环境中。现在你面对的就是包的 release 版本了。最终，这意味着必须修改 POM 文件中的版本属性并检入到 VCS 中。图 14.11 演示了这种版本控制方案。

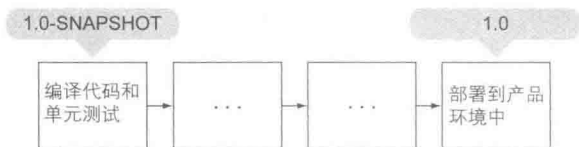


图 14.11 在构建管道中应用 Maven 的标准版本管理方案

一个新的开发周期通过增加项目的主版本号和 / 或次版本号开始了；后缀 -SNAPSHOT 又被加上了。这种指定项目版本的方式有什么不好的地方呢？这是持续交付的核心原则给出的最好的解释。

#### 每次提交都变成发布

持续交付的一个重要原则就是在版本控制下每次提交到代码库都可能变成一次发布被部署到产品环境中。当然，这只能发生在通过了团队定义的质量标准之后。

Maven 的版本管理思想与这种发布策略截然相反。其假设你在一定时间段内工作在一个功能上面，直到其发布上线。为了能够在开发过程中以及真实环境中唯一标识一个版本，你需要在构建管道最开始阶段设置一个合适的版本。这个版本将一直使用到部署到产品环境这个构建管道阶段（图 14.12）。

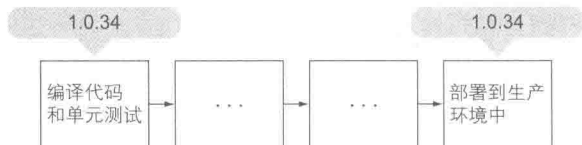


图 14.12 在构建管道的最开始阶段设置一个动态的、递增的版本号

接下来，我们来看看如何使用 Gradle 来实现这样的版本管理方案。

### 持续交付中的版本管理方案

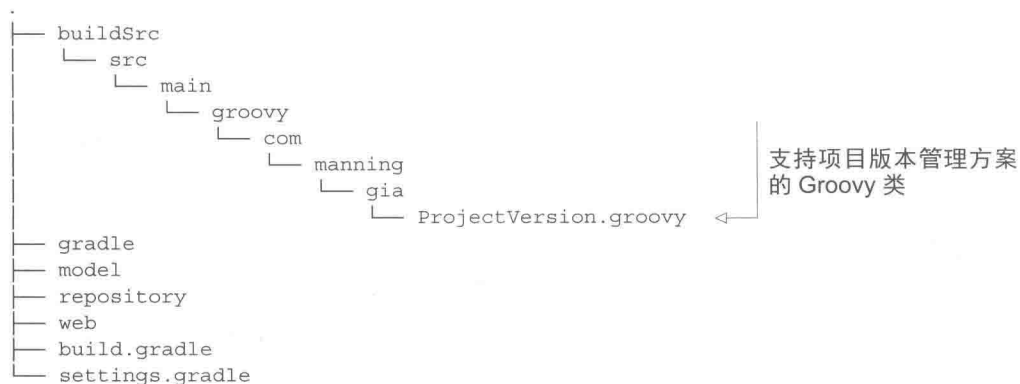
在第 4 章中，你学到了如何指定一个自定义类的实例来确定项目版本。你可以直接将这个知识应用到为 To Do 应用程序构建项目版本中。版本格式示例如图 14.13 所示。它包含三个属性：主版本(major version)、次版本(minor version)和构建号(build number)。



图 14.13 To Do 应用程序的版本管理方案

主版本和次版本属性可以通过 Gradle 来配置。它们会很少改变（比如标识一个新功能）而且必须手动地增加。版本管理方案中动态变化的部分是构建号。每次构建开始的时候自动增加——即，在构建管道的第一个阶段。

我们来看看 Gradle 中版本管理方案的真实实现。Groovy 类 `ProjectVersion` 位于 `com.manning.gia` 包中，用于支持版本表示因为你想要在构建的项目中共享这个类，所以你将这个类创建在 `buildSrc` 目录下，如下面的目录树所示：





我们来看看这个类，如下清单显示了 `ProjectVersion.groovy` 文件的内容。

清单 14.13 表示项目版本的 Groovy 类

```
package com.manning.gia

class ProjectVersion {
    final Integer major
    final Integer minor
    final String build

    ProjectVersion(Integer major, Integer minor, String build) {
        this.major = major
        this.minor = minor
        this.build = build
    }

    @Override
    String toString() {
        String fullVersion = "$major.$minor"

        if (build) {
            fullVersion += ".$build"
        }

        fullVersion
    }
}
```

代表主次版本、构建号的属性

构建项目版本的字符串表现形式

你知道 Gradle 在执行构建的时候会自动编译 `buildSrc` 下的每个类。这个编译类现在可以被用在任何构建脚本中来实现版本管理方案。为了更好地分离关注点，你需要在 `gradle` 目录下创建一个新的脚本插件 `versioning.gradle`。如下面的清单所示，你引入了 `ProjectVersion` 类，并实例化，然后赋值给 `org.gradle.api.Project` 的 `version` 属性。

清单 14.14 设置构建信息作为脚本插件

```
import com.manning.gia.ProjectVersion

ext.buildTimestamp = new Date().format('yyyy-MM-dd HH:mm:ss')
project.version = new ProjectVersion(1, 0, System.env.SOURCE_BUILD_NUMBER)
```

设置一个构建时间戳属性来确定构建的初始化日期

将一个新的 `ProjectVersion` 的实例赋值给项目的版本属性

这里设置的构建号是一个名为 `SOURCE_BUILD_NUMBER` 的环境变量。这个变量在你将构建作为 Jenkins 构建管道的一部分执行的时候自动有效。在上一章中，你使用 `Parameterized Trigger` 插件在配置初始化 `job` 的时候设置了这个值。你可能想要跳回到 13.5.2 节快速复习一下。有了这个版本管理脚本插件后，你现在可以通过 `allprojects` 配置块将它应用到所有项目中了，如下面的清单所示。

清单 14.15 为所有项目提供版本

```
allprojects {
    ...
    apply from: "$rootDir/gradle/versioning.gradle"
}
```

在无须执行 Jenkins 构建管道的情况下你可以模拟 Jenkins 构建 WAR 包了。你只需要在命令行设置 SOURCE\_BUILD\_NUMBER 环境变量即可。下面的例子演示了如何为构建号赋值 42：

- *\*nix*: `export SOURCE_BUILD_NUMBER=42`
- *Windows*: `SET SOURCE_BUILD_NUMBER=42`

如果你现在组建 WAR 文件，你会看到 WAR 包有了正确的版本号。你可以在 `web/build/libs` 目录下找到正确的 `web-1.0.42.war` 文件。

为包标识一个唯一的版本有两个原因。它暗示了这里一个发布版本，并且让你能够通过 VCS 中打上标签把二进制包和源代码对应起来。同样的版本管理信息应该包含在二进制包中用于自描述。

#### 14.4.4 在可部署包中加入构建信息

版本管理信息不仅仅在构建的时候有用。在某些场合下，你可能想知道在某个环境中部署的是应用程序的哪个版本。这可以通过将构建信息包含在由一个新的 `task createBuildInfoFile` 生成的属性文件中来达到。在运行时，这个属性文件可以被读取并显示在 Web 应用程序中。如下清单显示了 web 子项目的构建文件中所需要的 task 配置。

清单 14.16 在 WAR 文件中加入构建信息

```
task createBuildInfoFile << {
    → def buildInfoFile = new File("$buildDir/build-info.properties")
    Properties props = new Properties()
    props.setProperty('version', project.version.toString())
    props.setProperty('timestamp', project.buildTimestamp)
    props.store(buildInfoFile.newWriter(), null)
}

war {
    dependsOn createBuildInfoFile
    baseName = 'todo'

    from(buildDir) {
        include 'build-info.properties'
        into('WEB-INF/classes')
    }
}
```

用于存储构建信息的属性文件

添加属性来表示项目版本和构建时间

将属性写到文件中

添加对写构建信息文件 task 的依赖

将构建信息文件绑定到 WAR 文件的 WEB-INF/classes 目录

看看下面的例子，你就知道这个文件的大概样子了：

```
#Tue Apr 23 06:44:34 EDT 2013
version=1.0.134
timestamp=2013-04-23 06:44:11
```

我们不会讨论怎样写自定义的代码在 To Do 应用程序中读取并显示这些属性。请参考本书的代码示例来看看如何通过 Java 实现这一功能。准备好二进制包后，你现在可以将其发布到内部的二进制仓库中用于后续的部署了。接下来，你会采用在本章开始的时候编写的发布代码来上传 WAR 文件。

### 14.4.5 发布 To Do 应用程序 WAR 文件

将 WAR 文件发布到内部的 Artifactory 仓库没有什么神奇的地方。你已经知道如何做了。主要区别是这里你要发布的软件组件是 WAR 文件而不是 JAR 文件，如下面的清单所示。

清单 14.17 发布到远程的 Maven 仓库

```
apply plugin: 'maven-publish'

ext {
    artifactoryBaseUrl = 'http://localhost:8081/artifactory'
    artifactoryReleaseRepoUrl = "$artifactoryBaseUrl/libs-release-local"
}

publishing {
    publications {
        toDoWebApp(MavenPublication) {
            from components.web
            artifactId 'todo-web'

            pom.withXml {
                def root = asNode()
                root.appendNode('name', 'To Do application')
                root.appendNode('description', 'A simple task management
                    application.')
            }
        }
    }
}

repositories {
    maven {
        name 'remoteArtifactory'
        url artifactoryReleaseRepoUrl

        credentials {
            username = artifactoryUsername
            password = artifactoryPassword
        }
    }
}
```

Artifactory 仓库 URL

定义 web 软件组件为发布包

定义发布包 ID

上传 WAR 文件只需运行 `task publishToDoWebAppPublicationToRemoteArtifactoryRepository`。记住这个 `task` 名字是发布 API 自动生成的。如果检查 POM 文件，你会发现 WAR 包没有定义任何外部依赖。这主要是因为：依赖已经被绑定在 `WEB-INF/lib` 目录下的 WAR 文件中了。在下一节中，你会将构建管道的这个阶段配置到 Jenkins 的已有 job 上。

### 14.4.6 扩展构建管道

在交付 To Do 应用程序的过程中很重要的一个步骤就是创建分发并发布到二进制仓库中。你已经通过 Gradle 实现了这个功能。是时候通过在 Jenkins 上面配置相应的 job 来扩展构建管道了。在上一章中，你可能还记得你配置了三个 Jenkins job 以下面的顺序执行：

- 1 `todo-initial`：编译源代码并运行单元测试。
- 2 `todo-integ-tests`：运行集成测试。
- 3 `todo-code-quality`：使用 Sonar 执行静态代码分析

打开 Jenkins 的 Dashboard 并添加一个名为 `todo-distribution` 的新 job。为了让创建过程变得简单，你可以克隆 `todo-code-quality`。创建完成后，你需要修改构建管道第 3 步的配置。

#### 步骤 3：代码质量

添加一个参数化的构建动作用来定义一个构建触发器来触发 `todo-distribution`。至于参数传递，你可以通过选择 **Current Build Parameters** 选项来重用已有的参数。

太好了，你将第 3 步和作为下游项目的第 4 步连接起来了。接下来，你会配置刚刚创建的 job。

#### 步骤 4：打包和发布

这个 job 的配置跟前面的类似。看如下清单来确认你的配置正确：

- 在 **Source Code Management** 配置部分，选择 **Clone Workspace** 选项并选择父项目 `todo-initial`。
- 在构建步骤中，你需要组建 WAR 文件并发布到 Artifactory 仓库中。使用 **Wrapper** 添加一个调用 Gradle 脚本的构建步骤，并执行 `assemble` `publish` 两个 task。
- 通过使用上游项目的构建号定义构建名：`todo#${ENV, var="SOURCE_BUILD_NUMBER"}`。

有了这个配置后，Jenkins 中的构建管道视图应该看起来类似于图 14.14。为了

保证其正确行为，请确保通过执行构建管道来测试。

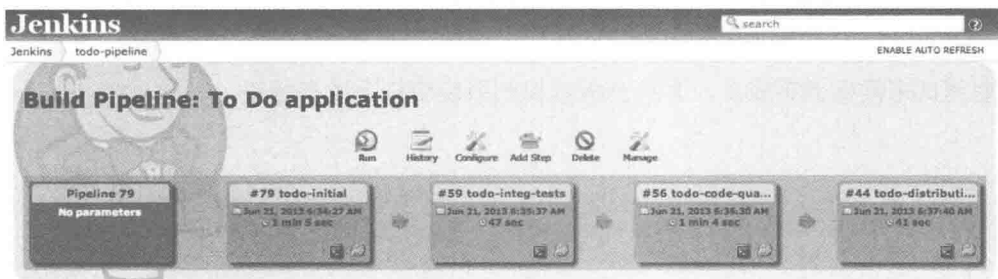


图 14.14 构建管道视图

基于构建管道目前的状态，你已经非常接近部署 To Do 应用程序到不同的目标环境中了。请继续阅读来了解如何完成这个任务。

## 14.5 总结

大多数软件在部署到目标环境中或者安装到指定位置之前都需要经历打包的过程。在打包的过程中，独立的软件组件应以特定的形式组合在一起。软件交付并不局限于交付一个单独的二进制包。通常其包含多个二进制包或者分发包，它们针对不同的组织或运行时环境。

Gradle 通过核心的自定义 task 如 Jar 或者 Zip 支持创建多种格式的压缩包，如果你应用了合适的插件，它们中的一些可以自动预配置。除了这个功能，Gradle 打包插件支持通过具有表述性的 DSL 来描述自定义分发包，不需要在构建脚本中定义 task。有了这些工具，打包的过程变得很简单，也能很灵活地响应交付过程中的需求变化。

一旦可交付的产品构建出来后，它就可以被其他项目、团队或者理论上每个程序员所共享。二进制仓库提供了基础设施用于上传、管理、浏览和使用任意数量或者类型的包。在本章中，你学到了如何使用 Gradle 提供的发布插件与本地或者远程的 Maven 仓库交互。你使用了第 8 章中的插件项目，为其组建 JAR 文件，生成个性化的基本信息，并上传到 Artifactory 中，Artifactory 是一个流行的二进制仓库。分享开源项目最方便的做法就是将其发布到公共的、通过互联网可访问的仓库中。我们讨论了如何设置 Sonatype OSS（即 Maven Central）和 JFrog Bintray 账户以及各自的发布过程。

在持续交付上下文中，打包和发布扮演了很重要的角色。如果可能，你打包一次，从而避免潜在的副作用。包被上传到仓库中后，它们就可以在后续的部署阶段被重

用。你学到了如何实现灵活的版本管理策略来清楚地识别一系列的二进制包。之后，你通过新建打包和发布 To Do 应用程序所产生的 WAR 文件的 `jop` 扩展了构建管道。

在本书最后一章中，你会将 To Do 应用程序部署到不同的目标环境中，并编写冒烟测试来确保部署成功，并且在版本控制系统中打上发布标签。

# 15

## 基础环境准备和部署

---

### 本章涵盖

- 通过 Gradle 准备基础环境
- 自动部署到不同的目标环境中
- 使用冒烟测试和验收测试验证部署结果
- 集成部署到构建管道中

软件部署需要重复并且稳定地执行。错误部署导致的任何服务器中止——对生产系统的影响最大——都会导致组织的经济损失。自动化是规划和精简部署流程的下一个逻辑和必需的步骤。在本章中，我们会以 To Do 应用程序为例讨论如何使用 Gradle 自动化部署流程。

在部署之前，目标环境必须要预配置好必需的软件基础环境。以前，这项工作由系统管理员完成，系统管理员会手动准备好物理机器并安装所需要的软件组件。这种安装可以通过 Puppet 或者 Chef 之类的工具定义成代码，用版本控制系统来管理，并像常规软件一样进行测试。使用这种基础设施即代码的方式，可以避免人为的错误，并且能减少基于同一个软件栈的新环境的运转周期。但是 Gradle 并不提供这样的原生工具，你可以通过使用其他工具来完成。

作为构建大师，部署软件不只是复制文件到服务器上。在构建中，你需要配置和指向不同的环境。自动化整个部署周期通常需要清除之前的部署文件，以及重启远程的运行时环境，如 Web 容器。本章介绍了一种可行的方式来达到这个目的。

一旦部署了一个新版本的软件，你需要验证部署结果。自动化的冒烟测试和验收测试能够帮助检测软件的功能是否正常。你会设置一系列的测试并使用 Gradle 执行它们。

所有的这些流程——部署到不同的环境中并验证部署是否成功——都需要作为构建管道的一部分。在建立好支持的 Gradle task 后，你可以在 Jenkins 中通过相应的 job 调用它们。部署软件应该简单到只需要按下一个按钮即可。你会通过部署到不同环境中的 job 来扩展构建管道。在深入部署软件的细节之前，我们来看看帮助准备基础环境的一些工具。

## 15.1 准备基础环境

在应用程序能够被部署之前，托管的基础环境需要准备好。当我说到准备基础环境的时候，在传统意义上是指设置好硬件，以及安装和配置好所需要的操作系统和软件组件。

如今，我们看到了在云端准备基础环境的趋势。与传统方式不同，云提供商通常都会以虚拟服务器的形式来分配预配置好的硬件资源。服务器虚拟化就是把物理服务器分成很多较小的虚拟服务器，从而帮助最大化服务器资源的利用。根据所提供的服务，操作系统和软件组件由云提供商管理。

在本节中，我们会讨论如何利用第三方的开源工具自动创建虚拟服务器和软件基础环境。这些工具会帮助你安装和配置好 To Do 应用程序所需要的目标环境。之后，你会学到 Gradle 如何与这些工具集成。

### 15.1.1 基础设施即代码

开发人员通常工作在一个自包含的环境中——他们的开发机器。软件基础环境需要手动设置。请回想一下 To Do 应用程序，这包括 Java 运行时环境、Web 容器和数据库。这对于单个开发人员来说不会有问题，但是随着团队规模的增长这可能演变成一个巨大的问题。现在，每个开发人员都需要确保他们安装了相同软件包的相同版本，并使用了相同的配置（最好是在相同的操作系统上）。

在其他环境（如 UAT 和产品环境）中安装配置硬件和软件基础环境遵循类似的过程。在大企业中，这个任务传统上通常由运营团队负责。没有有效的沟通和充足的文档，准备环境有可能变成一个冗长且费力的过程。甚至更糟糕的是，如果需要



更改任何配置，都必须手动传播到所有的环境。

虽然 shell 脚本是减轻这个痛苦的第一种手段，但它并不能达到完全的自动化。基础环境即代码的模式目标是要在软件开发和系统管理之间建立起桥梁。通过使用一些工具，可以将机器配置描述成可执行的代码，然后将其放入版本控制系统中进行管理并且在不同的人员之间共享。任何时候你想要创建一个新的环境，只需要执行基于基础环境代码的相应指令即可。最后，允许你像其他软件开发项目一样来处理基础环境代码，可以进行版本控制、测试，以及检查潜在的语法问题。

过去的几年，许多商业的和开源的工具参与到了自动化基础环境准备中来。我们将关注最流行的两种开源的自动化工具：Vagrant 和 Puppet。接下来会从架构方面看看如何利用这两种工具从零开始创建一个虚拟机。最终结果就是一个可以运行 To Do 应用程序的运行环境。

### 15.1.2 使用 Vagrant 和 Puppet 创建虚拟机

Vagrant (<http://www.vagrantup.com>) 是一个用来配置和创建虚拟环境的工具。可以通过执行 Vagrant 命令来管理机器。比如，你可以通过简单的一行 shell 脚本来启动和停止一台机器。你甚至可以直接 SSH 到这台机器并且控制机器，就像想任何其他远程 \*nix 服务器一样。

通过 shell 脚本或者如 Chef 和 Puppet 这样的工具来描述机器的软件配置。你所使用的工具通常取决于个人偏好和所掌握的工具知识。我们会把注意力集中在 Puppet 上。

Puppet (<https://puppetlabs.com/puppet/>) 提供了基于 Ruby 的 DSL 来描述软件组件以及它们在目标机器上所需的状态。如果你回想一下 To Do 应用程序所需的运行时环境，就会确定需要下面的软件包及其配置：

- Java 运行时环境 (JRE) 安装程序。使用 JRE 6。
- 一个 Servlet 容器用来部署 Web 应用程序。使用 Apache Tomcat 7。
- 一个 H2 数据库用来管理程序数据。为了能够正常工作，需要先设置好数据库基本结构。

完整地描述建立这样场景的配置不在本书的讨论范围内。然而，你可以在本书源代码中找到一个实际的例子。我们来看看 Vagrant 项目的基本结构，从而提高认识。



图 15.1 演示了一个 Vagrant 项目的每个独立部分是如何一起工作的。至少，每个 Vagrant 项目都需要包含一个 Vagrantfile。基于这个文件来创建和配置虚拟机。你将采用 Puppet 作为配置工具。虚拟机需要的配置被设置在 Puppet 清单文件中，在 Vagrantfile 中引用了这个清单文件。在这个例子中，这个清单文件叫作 tomcat.pp。为了能够管理版本以及与其他开发人员共享 Vagrant 项目，你需要将其检入到版本控制系统中。

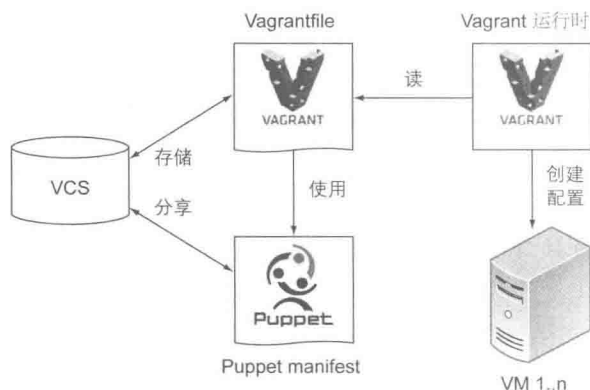


图 15.1 使用 Vagrant 和 Puppet 创建测试环境

在你的基础设施代码能够生成虚拟机之前，你需要安装 Vagrant 执行环境和 Virtual Box (<https://www.virtualbox.org>)。请参考 Vagrant 文档获得更多信息。在成功安装后，你可以通过命令行运行 Vagrant。

我们来看看一些常用的命令。跳转到 Vagrant 项目并执行 `vagrant up` 命令来启动 Vagrant 机器。Vagrant 的输出很冗长，所以在这里就不给出其命令行输出了。一段时间后，你就应该能看到虚拟机启动成功的提示。

在 Vagrantfile 中，你配置了虚拟机可以通过 IP 地址 192.168.1.33 访问。作为配置代码的一部分，你定义了 Tomcat 运行在 8080 端口。为了验证 Web 容器安装成功，请打开浏览器输入 `http://192.168.1.33:8080/`，你应该能看到 Tomcat 7 的 Dashboard。使用 `vagrant destroy` 命令关闭虚拟机。在接下来的一节中你会学

到如何从 Gradle 启动 Vagrant 命令。

### 15.1.3 从 Gradle 执行 Vagrant 命令

这时，你可能会想，“为什么我需要在 Gradle 中执行 Vagrant 命令呢？”答案是自动化。与 Vagrant 提供的虚拟机交互的都需要能够从 Gradle 调用相应的命令。为了演示一个简单的工作流，我们假设你想要在一个虚拟机上执行功能测试，其模仿了一台生产服务器的基础环境设置。需要涉及如下步骤：

- 1 使用 `vagrant up` 命令启动虚拟机。
- 2 部署 Web 应用程序到 Tomcat 服务器。
- 3 执行一系列的功能测试。
- 4 使用 `vagrant destroy` 命令关闭虚拟机。

这个用例相当高级并需要复杂的设置。现在，从一些简单的东西开始，你会封装一些 Vagrant 命令使构建能够用 Gradle task 来调用。你需要自定义一个 task。这个 task 将你想要执行的 Vagrant 命令定义成输入参数。另外，你需要告诉这个 task 去哪找 Vagrant 项目。如下清单演示了一个可重用的 task 用来执行 Vagrant 命令。

清单 15.1 为执行 Vagrant 命令自定义 task

```
package com.manning.gia.vm

import org.gradle.api.DefaultTask
import org.gradle.api.GradleException
import org.gradle.api.tasks.Input
import org.gradle.api.tasks.TaskAction

class Vagrant extends DefaultTask {
    static final String VAGRANT_EXECUTABLE = 'vagrant'

    @Input
    List<String> commands

    @Input
    File dir

    @TaskAction
    void runCommand() {
        commands.add(0, VAGRANT_EXECUTABLE)
        logger.info "Executing Vagrant command: '${commands.join(' ')}'"
    }
}
```

Vagrant 提供的命令(不包括 Vagrant 本身)

Vagrant box 所在的目录

```

def process = commands.execute(null, dir)
process.consumeProcessOutput(System.out, System.err)
process.waitFor()

if(process.exitValue() != 0) {
    throw new GradleException()
}
}
}

```

以外部进程的方式执行 Vagrant 命令（等待其结束）

如果进程没有成功结束，抛出异常

根据配置的复杂性，一些 Vagrant 命令（尤其是 `vagrant up`）可能需要几分钟的时间来完成。如果你有一个相互依赖的 task 链，则需要确保 task 在 Vagrant 完成其工作后再执行。Task 实现通过让当前线程等待直到 Vagrant 命令响应一个退出值来满足这个需求。接下来，测试 Vagrant task。如下清单演示了使用自定义 task 在 Gradle 构建中执行重要的 Vagrant 命令。

### 清单 15.2 执行重要的 Vagrant 命令的增强 task

```

import com.manning.gia.vm.Vagrant

ext.targetedVagrantProjectDir = file('../vagrant-tomcat-box')

task vagrantUp(type: Vagrant) {
    commands = ['up']
    dir = targetedVagrantProjectDir
}

task vagrantDestroy(type: Vagrant) {
    commands = ['destroy', '--force']
    dir = targetedVagrantProjectDir
}

task vagrantSshConfig(type: Vagrant) {
    commands = ['ssh-config']
    dir = targetedVagrantProjectDir
}

task vagrantStatus(type: Vagrant) {
    commands = ['status']
    dir = targetedVagrantProjectDir
}

task vagrantSuspend(type: Vagrant) {
    commands = ['suspend']
    dir = targetedVagrantProjectDir
}

task vagrantResume(type: Vagrant) {
    commands = ['resume']
    dir = targetedVagrantProjectDir
}

```

指向目标 Vagrant box 的目录

根据 Vagrantfile 创建和配置机器

停止正在运行的 Vagrant 机器并销毁其所有资源

SSH 配置文件的输出配置（需要 SSH 到正在运行的机器上）

关于 Vagrant 机器状态的报告（如正在运行、挂起）

通过创建当前状态的镜像将一台正在运行的 Vagrant 机器挂起

恢复一台之前挂起的 Vagrant 机器

祝贺你，你刚刚完成了将 Vagrant 集成到构建中！在本地机器上运行 Vagrant 可以很好地模拟类产品环境。当需要与除本地机器以外的其他已经存在的环境打交道时，你的构建需要能够配置连接信息。在接下来的一节中，我们会探讨一种灵活的存储和读取环境相关配置的方法。

## 15.2 针对部署环境

持续交付的主要准则就是将软件从程序员的机器上尽可能快且频繁地交付到终端用户手中。但这并不是说你需要马上组建可交付产品并把它部署到产品环境中去。在这些步骤之间，构建管道通常会在不同的环境中验证功能性和非功能性需求，如图 15.2 所示。

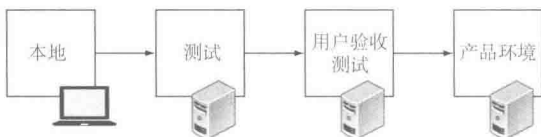


图 15.2 将软件部署到不同的环境中

在本章开头，你在开发机器上创建了虚拟机。虽然这个虚拟机有类产品环境的配置，但是你使用这个环境只能出于测试目的。测试环境从团队中多个开发人员处获取代码放到一起。因此，其可以看作是运行代码的第一个集成点。在测试环境中部署应用程序，你可以运行自动化的验收测试来验证功能性和非功能性需求。用户验收测试（UAT）环境通常存在的目的就是用于探索和手工测试。一旦 QA 团队认为软件代码的当前状态满足要求，就可以被部署到产品环境中了。产品环境直接面向最终用户并使新特性可用。

如果你想要使用同样的代码来部署到所有的环境，则需要能够在构建时动态地指定环境。自然地，测试、UAT 和产品环境运行在不同的服务器上，有不同的端口号和认证信息。在构建脚本中你可以将这些配置保存成扩展属性，但是很快就会将文件的逻辑弄乱。或者，你可以将这些信息保存在 `gradle.properties` 文件中。不管哪种方式，最后都会得到一堆结构混乱的属性列表。在构建时，你不得不根据命名规范读取一堆属性。听起来不是很灵活，是吧？利用标准的 Groovy 特性，我们可以有更好的方式来存储和读取配置。

### 15.2.1 在 Groovy 脚本中定义配置

配置，尤其是有很多配置，应该尽可能地具有可读性和结构化。Groovy 语言特性之一是允许在 Groovy 脚本中使用闭包来定义属性。如下清单显示了以迷你 DSL

的形式包含环境相关配置的一个例子。

### 清单 15.3 基于 Groovy 的环境相关配置

```
environments {  
    local {  
        server {  
            hostname = 'localhost'  
            sshPort = 2222  
            username = 'vagrant'  
        }  
  
        tomcat {  
            hostname = '193.168.1.33'  
            port = 8080  
            context = 'todo'  
        }  
    }  
  
    test {  
        ...  
    }  
  
    uat {  
        ...  
    }  
  
    prod {  
        ...  
    }  
}
```

The diagram consists of four arrows pointing from text labels to specific parts of the Groovy code. The first arrow, labeled '按照环境分类配置的元素' (Elements configured by environment), points to the 'environments' block. The second arrow, labeled '开发环境的配置' (Development environment configuration), points to the 'local' block. The third arrow, labeled '测试环境的配置' (Test environment configuration), points to the 'test' block. The fourth arrow, labeled 'UAT 环境的配置' (UAT environment configuration), points to the 'uat' block. Additionally, there is a label '产品环境的配置' (Production environment configuration) with an arrow pointing to the 'prod' block.

在 `environments` 配置块中为每个环境定义了属性。对于每个环境，你配置了一个具有描述性名字的闭包。比如，你可以定义服务器主机名、SSH 端口号和用户名登录到服务器。使用这个配置数据，并将其保存到 `gradle/config/buildConfig.groovy` 脚本文件中，如下面的项目目录树所示：

```
.  
├── buildSrc  
├── gradle  
│   └── config  
│       └── buildConfig.groovy  
├── model  
├── repository  
├── web  
├── build.gradle  
└── settings.gradle
```

The diagram shows a project directory tree. A bracket on the right side of the tree, labeled '基于 Groovy 的环境相关配置文件' (Environment-related configuration files based on Groovy), groups the `buildSrc`, `gradle`, and `model` directories. An arrow points from this bracket to the `buildConfig.groovy` file within the `gradle/config` directory.

你现在拥有了一个基于 Groovy 的配置文件了，但是如何从 Gradle 构建中读取其内容呢？Groovy 提供了一个方便的 API 类叫作 `groovy.util.ConfigSlurper`，用于解析树形的数据结构。我们来仔细看看它的功能。

## 15.2.2 使用 Groovy 的 ConfigSlurper 读取配置

ConfigSlurper 是一个工具类，用来读取 Groovy 形式的脚本配置数据。配置可以在脚本根级别被定义成属性，或者在 environments 闭包中被包装成环境相关属性。一旦配置被解析，就可以通过点号来遍历属性图了。我们来实际看看。

从 Gradle 中读取配置文件需要一些思考。你需要确保在所有 task 动作被执行之前读取配置信息。还记得我们在第 4 章中讨论的 Gradle 的构建生命周期阶段吗？最好在配置阶段做这件事情，如图 15.3 所示。

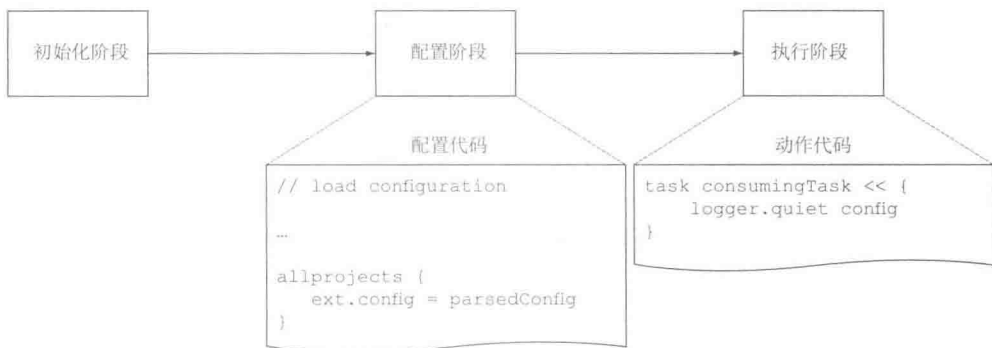


图 15.3 在 Gradle 的配置阶段读取 Groovy 脚本

读取 Groovy 脚本的 task 不需要有可执行的动作。你会在这个 task 的配置块中创建一个 ConfigSlurper 类的实例，并在它的构造器中提供了你想要读取的特定环境信息。parse 方法指向了配置文件的位置。如下清单演示了如何根据所提供的属性 env 解析 Groovy 脚本。

### 清单 15.4 在配置阶段读取配置信息

```

def env = project.hasProperty('env') ? project.getProperty('env') : 'local'
logger.quiet "Loading configuration for environment '$env'."

def configFile = file("$rootDir/gradle/config/buildConfig.groovy")
def parsedConfig = new ConfigSlurper(env).parse(configFile.toURL())

allprojects {
    ext.config = parsedConfig
}
  
```

包含配置的 Groovy 文件

将读取到的配置赋值给扩展属性

读取关于特定环境的配置文件

解析好的配置通过扩展属性 config 对构建中的所有项目可用。接下来，你会在另一个 task 中通过这个属性使用解析好的配置。

### 15.2.3 在构建中使用配置

清单 15.3 中显示的这些配置的典型使用模式是部署 To Do 应用程序到特定环境中。指向特定环境的关键是在命令行属性 `env` 指定值。比如,你想要使用 UAT 环境,你需要指定项目属性 `-Penv=uat`。这个项目属性的值直接对应于 Groovy 配置脚本中的闭包 `uat`。运用这种简单的机制,将使构建运行相同的 `task` 逻辑——比如,通过环境相关配置进行代码部署,如图 15.4 所示。

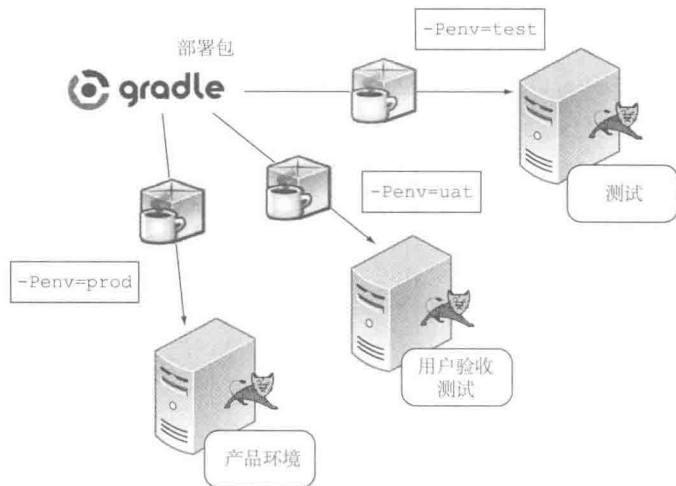


图 15.4 通过提供一个项目属性指向特定环境

你需要模拟一个部署任务来看看这种机制是否能正常工作。在 web 项目中,你会创建一个新的 `task` 叫作 `deployWar`, 如清单 15.5 所示。现在你无须考虑具体的部署逻辑。为了确保各个环境的配置都被正确地解析了, 这个 `task` 的 `doLast` 动作使用 Gradle 的日志功能将读取到的主机名和端口号打印出来。

#### 清单 15.5 使用扩展属性的配置

日志输出  
配置的  
服务器名和  
端口

```
task deployWar << {
    logger.quiet "Deploying WAR file to $config.server.hostname via SSH
                 on port $config.server.sshPort."
}
```

当你在 `local` 环境运行 `deployWar` `task` 的时候, 你可以看到相应的配置被解析并打印到了终端:

```
$ gradle deployWar -Penv=local
Loading configuration for environment 'local'.
:todo-web:deployWar
Deploying WAR file to 'localhost' via SSH on port 2222.
```



仅仅打印出这些配置信息太无趣了。在接下来的一节中，你会真正地使用这些配置将 To Do 应用程序的 WAR 包部署到服务器上。

## 15.3 自动部署

每个构建管道的最终目的都是在软件经过了所有的自动化或者手动测试阶段过后，将其部署到产品环境中。部署过程应该是可以重复的，并且是可靠的。无论怎样，在产品环境中安装新版本软件的时候，你都想要尽可能地避免人为的错误。软件部署失败可能会带来不可预期的副作用，或者是直接的经济损失。

我认为没有人不觉得软件的部署任务是一件非常重要的事。自动化部署是实现这个目标重要的和必需的步骤。用于自动化部署过程的代码不应该直接在产品环境下开发和测试，以减少破坏的风险。相反，应该是在本地机器上的类产品环境中或测试环境中进行测试。你已经用 Vagrant 设置好了这样的环境。它使用的基础环境定义几乎跟产品环境一样。使用虚拟机模拟一个类产品环境来开发部署代码变得简单、容易管理，并且不会干扰构建管道中的其他环境。一旦你觉得这个解决方案没有问题了，你就可以执行那些代码将软件部署到构建管道中最关键的环境中了。随着代码按照预期运行，你就拥有了更多的自信，接下来你就可以将软件部署到更多的不同环境如 UAT 和产品环境中了。

编写部署代码并不是一成不变的。根据软件类型和部署的环境不同而不同。比如，需要部署到 Linux 机器上的 Web 应用程序和在 Windows 机器上运行的客户端安装软件的需求是不同的。在撰写本书时，Gradle 并没有提供一个统一的部署软件方案。稍后我们将讨论将 Web 应用程序部署到 Tomcat 容器中的方法。

### 15.3.1 从二进制仓库中获取包

在上一章中，你学到了如何将包上传到二进制仓库中。为了部署，你现在需要获取到包。现在你已经有了 Groovy 配置文件，你也可以在里面添加 Artifactory 仓库的 URL。在这个例子中，你只使用一个不针对于环境的仓库。ConfigSlurper 还会读取不依赖于 env 属性在 environments 闭包外定义的所有属性值。如下清单演示了如何定义通用的配置——也就是二进制仓库地址。

清单 15.6 在 buildConfig.groovy 中添加二进制仓库设置信息

```
binaryRepository {  
    baseUrl = 'http://localhost:8081/artifactory' ← Artifactory 基础 URL
```

```

    releaseUrl = "$baseUrl/libs-release-local"
}
environments {
    ...
}

```

← 用于发布的 Maven 仓库的 URL

在清单 15.7 中, 你使用 `buildConfig.groovy` 文件中的设置信息从 Artifactory 仓库中下载项目当前版本的 WAR 文件。在这个例子中, Gradle 的依赖管理机制做了下载的繁重工作。执行 `fetchToDoWar` task 会将 WAR 包下载到 `build/download/artifacts` 目录下。请注意并不一定要使用 Gradle 的依赖管理机制来获取文件。你也可以使用 Ant 的 task `Get` 和在 Groovy 中写一个更底层的实现。

### 清单 15.7 从远程仓库中下载 WAR 文件

```

repositories {
    maven {
        url config.binaryRepository.releaseUrl
    }
}

configurations {
    todo
}

dependencies {
    todo group: project.group, name: project.name,
        version: project.version.toString(), ext: 'war'
}

ext.downloadDir = file("$buildDir/download/artifacts")

task fetchToDoWar(type: Copy) {
    from configurations.todo
    into downloadDir
}

```

← 定义 Artifactory 为仓库

← 为应用程序相关依赖引入新配置

← 定义 To Do 应用程序 WAR 文件为依赖

← 扩展属性用来定义目标下载目录

← 用于从 Artifactory 下载 WAR 文件的 task

试着运行这个 task。假设你的项目版本是 1.0.42。执行完这个 task 后, 你会在下载目录下找到所下载的文件:

```

├─ build
│   └─ download
│       └─ artifacts
│           └─ todo-web-1.0.42.war

```

当然, 只下载一次是非常有意义的, 即使你要部署到不同的环境中。`fetchToDoWar` task 自动实现了增量构建功能。第二次执行这个 task 会将其标志成 UP-TO-DATE, 如下面的命令行输出所示:

```

$ gradle fetchToDoWar
:fetchToDoWar UP-TO-DATE

```

现在事情开始变得有趣了。你已经下载了正确版本的包了。在部署到 Tomcat 之前，你应该计划一下所需要的步骤。

### 15.3.2 确定必需的部署步骤

将 Web 应用程序部署到过程服务器的流程需要遵循一定的工作流，以保证在版本更新的时候能够平滑地过渡。你需要考虑哪些方面的问题呢？

首先，你需要确保所有老的版本，比如 WAR 文件都被正确地删除了。任何时候，你都不应该将新版本和老版本混合在一起。

有些部署解决方案如 Cargo (<http://cargo.codehaus.org/>) 允许在容器运行的时候进行部署，这被称为热部署技术。虽然这听起来很有吸引力，因为你不需要重启服务器，但是热部署对生产系统来说不是一个可行的方案。随着时间的推移，长时间运行的 JVM 进程会出现 `OutOfMemoryError` 错误，导致进程冻结。原因是 JVM 不会回收前一次部署的类的实例，即使它们不会被使用到了。因此，强烈推荐在新版本部署之前完全停止 Web 容器的 JVM 进程。

一个高效的部署过程可以包含如下步骤：

- 1 将新的二进制包推送到服务器。
- 2 停止 Web 容器进程。
- 3 删除老的包和相应的解压后文件。
- 4 部署新的二进制包
- 5 启动 Web 容器进程

在下一节中，你会使用 Gradle 实现这个过程。之前创建的 Vagrant 实例会作为部署脚本的实验台。

### 15.3.3 通过 SSH 命令部署

我们前面并没有对所建立的虚拟机的操作系统有过多的说明。假设其是基于 Linux Ubuntu 的。你可能知道运行 SSH 守护进程将文件传输到远程服务器可以通过安全拷贝 (Secure Copy, SCP) 来达到。SCP 与 SSH 使用相同的安全机制。出于认证目的，SCP 会询问密码或通行短语。或者，可以通过提供私钥文件来认证用户。Vagrant 自动将认证文件放到了 `<USER_HOME>/ .vagrant.d` 目录下。

你可以将整个部署过程建模到一个 shell 脚本中，然后在 Gradle 中通过创建增强的 Exec 类型的 task 来调用。这肯定是一种实现必需步骤的合理方式。然而，在本节中我们会讨论如何使用相应的 Gradle task 来建模每一个步骤。

## 使用 SCP 传输文件

你会从使用 SCP 传输文件开始。如果你熟悉 Ant 的话，你可能之前用过 SCP task。Ant task 在 SSH 的一个纯 Java 实现 JSch (<http://www.jcraft.com/jsch/>) 上做了很好的抽象。下面的清单显示了在 buildSrc 项目中如何使用自定义的 Gradle task 封装 Ant 的 SCP task。

清单 15.8 自定义 task 封装可选的 Ant 的 SCP task

```
package com.manning.gia.ssh

import org.gradle.api.DefaultTask
import org.gradle.api.GradleException
import org.gradle.api.file.FileCollection
import org.gradle.api.tasks.*

class Scp extends DefaultTask {
    @InputFiles
    FileCollection classpath

    @InputFile
    File sourceFile

    @Input
    String destination

    @Input
    File keyFile

    @Input
    Integer port

    @TaskAction
    void transferFile() {
        logger.quiet "Copying file '$sourceFile' to server."
        ant.taskdef(name: 'jschScp', classname:
            ➤ 'org.apache.tools.ant.taskdefs.optional.ssh.Scp',
            ➤ classpath: classpath.asPath)
        ant.jschScp(file: sourceFile, todir: destination, keyfile:
            ➤ keyFile.canonicalPath, port: port, trust: 'true')
    }
}
```

使用 Ant  
的 SCP  
task 将  
文件传  
输到服  
务器

声明 Ant 的  
SCP task

你会使用 web 项目构建脚本中的这个 SCP 抽象将 WAR 文件传输到远程位置。在清单 15.9 中，你使用了自定义配置 jsch 对 JSchAnt task 的依赖进行了声明。这个依赖被传递给了负责将 WAR 文件传输到远程服务器的增强 task 的 classpath 属性。在构建的配置阶段，也会读取服务器的设置信息。

清单 15.9 通过 SCP 将 WAR 文件传输到服务器

```
configurations {
    jsch
}
```

```
dependencies {
    jsch 'org.apache.ant:ant-jsch:1.9.1'
}

ext {
    warFile = configurations.todo.singleFile
    tomcatRemoteDir = '/opt/apache-tomcat-7.0.42'

    userHome = System.properties['user.home']
    vagrantKeyFile = file("$userHome/.vagrant.d/insecure_private_key")
    remoteTmpDir = "$config.server.username@$config.server.hostname:/tmp"
}

import com.manning.gia.ssh.Scp

task copyWarToServer(type: Scp, dependsOn: fetchToDoWar) {
    classpath = configurations.jsch
    sourceFile = warFile
    destination = remoteTmpDir
    keyFile = vagrantKeyFile
    port = config.server.sshPort
}
```

在通过 SCP 将 WAR 文件传输到服务器之前，确保其已经从 Artifactory 下载

将可选的 Ant task 赋值给 task 的 classpath

定义下载的 WAR 包为传输的源文件

定义 WAR 文件被传输到的远程目标目录

这个清单只实现了部署流程中的第一个步骤。还有 4 个步骤有待实现。所有其他的步骤都需要在远程服务器上面执行 shell 命令。

### 使用 SSH 执行远程命令

SSH 命令使得操作变得很容易。SSH 可以在远程机器上运行命令并打印输出。以下清单显示了自定义 SshExec task 内部封装 Ant 的 SSH task 的过程。

**清单 15.10 自定义 task 封装可选的 Ant 的 SSH task**

```
package com.manning.gia.ssh

import org.gradle.api.DefaultTask
import org.gradle.api.GradleException
import org.gradle.api.file.FileCollection
import org.gradle.api.tasks.*

class SshExec extends DefaultTask {
    @InputFiles
    FileCollection classpath

    @Input
    String host

    @Input
    String username

    @Input
    String command

    @InputFile
    File keyFile
}
```

```

@Input
Integer port

@TaskAction
void runSshCommand() {
    logger.quiet "Executing SSH command '$command'."
    ant.taskdef(name: 'jschSshExec', classname:
        ➡ 'org.apache.tools.ant.taskdefs.optional.ssh.SSHExec',
        ➡ classpath: classpath.asPath)
    ant.jschSshExec(host: host, username: username, command: command,
        ➡ port: port, keyfile: keyFile.canonicalPath,
        ➡ trust: 'true')
}
}

```

使用 Ant 的 SSH task 将文件传输到服务器

声明 Ant 的 SSH task

在接下来的清单中，将使用这个自定义的 SShTask 在 Vagrant 虚拟机上面运行一系列的 shell 命令。从总体上看，这个脚本实现了我们前面讨论的整个部署 workflow。

#### 清单 15.11 用于管理 Tomcat 和部署 WAR 文件的 SSH 命令

```

import com.manning.gia.ssh.SshExec

tasks.withType(SshExec) {
    classpath = configurations.jsch
    host = config.server.hostname
    username = config.server.username
    keyFile = vagrantKeyFile
    port = config.server.sshPort
}

task shutdownTomcat(type: SshExec, dependsOn: copyWarToServer) {
    command = "sudo -u tomcat $tomcatRemoteDir/bin/shutdown.sh"

    doFirst {
        logger.quiet "Shutting down remote Tomcat."
    }
}

task deleteTomcatWebappsDir(type: SshExec, dependsOn: shutdownTomcat) {
    command = "sudo -u tomcat rm -rf $tomcatRemoteDir/webapps/todo"
}

task deleteTomcatWorkDir(type: SshExec, dependsOn: shutdownTomcat) {
    command = "sudo -u tomcat rm -rf $tomcatRemoteDir/work"
}

task deleteOldArtifacts(dependsOn: [deleteTomcatWebappsDir,
    deleteTomcatWorkDir]) {
    doFirst {
        logger.quiet "Deleting old WAR artifacts."
    }
}

task copyWarToWebappsDir(type: SshExec, dependsOn: deleteOldArtifacts) {
    command = "sudo -u tomcat cp /tmp/$warFile.name
        ➡ $tomcatRemoteDir/webapps/todo.war"
}

```

配置所有 SshExe 类型的 task

关闭 Tomcat JVM 进程

删除已经存在的 To Do 应用程序目录

删除 Tomcat 临时工作目录

复制 WAR 包到 Tomcat 的 webapps 目录

```

doFirst {
    logger.quiet "Deploying WAR file to Tomcat."
}
}

task startupTomcat(type: SshExec, dependsOn: copyWarToWebappsDir) {
    command = "sudo -u tomcat $tomcatRemoteDir/bin/startup.sh"

    doFirst {
        logger.quiet "Starting up remote Tomcat."
    }
}

task deployWar(dependsOn: startupTomcat)

```

启动 Tomcat JVM 进程

部署生命周期 task

一个完整的部署过程就是这样的。将 Vagrant 启动起来，试试这些命令。下面的命令行输出显示了正在执行这些步骤：

```

$ gradle deployWar -Penv=local
...
Loading configuration for environment 'local'.
:todo-web:fetchToDoWar
:todo-web:copyWarToServer
Copying file 'todo-web-1.0.42.war' to server.
:todo-web:shutdownTomcat
Shutting down remote Tomcat.
Executing SSH command 'sudo -u tomcat /opt/apache-tomcat-7.0.42/
bin/shutdown.sh'.
:todo-web:deleteTomcatWebappsDir
Executing SSH command 'sudo -u tomcat rm -rf /opt/apache-tomcat-
7.0.42/webapps/todo'.
:todo-web:deleteTomcatWorkDir
Executing SSH command 'sudo -u tomcat rm -rf /opt/apache-tomcat-
7.0.42/work'.
:todo-web:deleteOldArtifacts
Deleting old WAR artifacts.
:todo-web:copyWarToWebappsDir
Deploying WAR file to Tomcat.
Executing SSH command 'sudo -u tomcat cp /tmp/todo-web-1.0.42.war
/opt/apache-tomcat-7.0.42/webapps/todo.war'.
:todo-web:startupTomcat
Starting up remote Tomcat.
Executing SSH command 'sudo -u tomcat /opt/apache-tomcat-7.0.42/
bin/startup.sh'.
:todo-web:deployWar

```

在重启 Tomcat 后，Web 应用程序运行起来可能需要一定的时间。在给 Tomcat 足够的时间来解压 WAR 文件并启动服务后，在浏览器中打开 <http://192.168.1.33:8080/todo>，你就能看到一个新建 To Do 任务的清单了。

运行 SSH 命令并不是实现自动化部署的唯一方式。还有很多其他方式可以达

到相同的效果。在这里我们就不一一展示了。你可以自己尝试。只要你选择的过程是可重复的、可靠的，并且能满足公司需求，那么你做的事情就是正确的。

## 15.4 部署测试

应用程序的每一次部署都应该做一个基本的测试，用来验证操作是成功的，系统工作状态在预期中。这种测试被称为部署测试。

如果部署不论任何原因失败了，你都想要知道——很快地知道。在最坏的情况下，如果部署到产品环境失败了，最终用户不应该是第一个告诉你应用程序 down 掉的人。很明显，你应该彻底避免发生这种情况，因为这会破坏信誉和造成经济损失。

在一些特殊情况下，即使做了精心的准备，部署也可能会失败。尽可能地了解失败也是有价值的。这样，你就能够采取措施将系统恢复到可操作状态；如回滚到应用程序的上一次可用的版本。

除了这些 fail-fast 测试之外，自动化验收测试用来验证部署的应用程序的重要特性和用例能够正确实现。为此，你可以使用在第 7 章中所写的功能测试。你会配置它们指向其他环境。我们首先来看看如何实现最基本的部署测试：冒烟测试。

### 15.4.1 使用冒烟测试验证部署成功

部署完成后，自动化部署代码应该对系统做一个基本功能状态的验证。这种测试就被称为冒烟测试。为什么叫这个名字？冒烟测试类比了硬件如集成电路的安装。如果你打开电源，发现电子器件中有烟雾冒出，那就说明安装出问题了。同样可以用来类比软件。

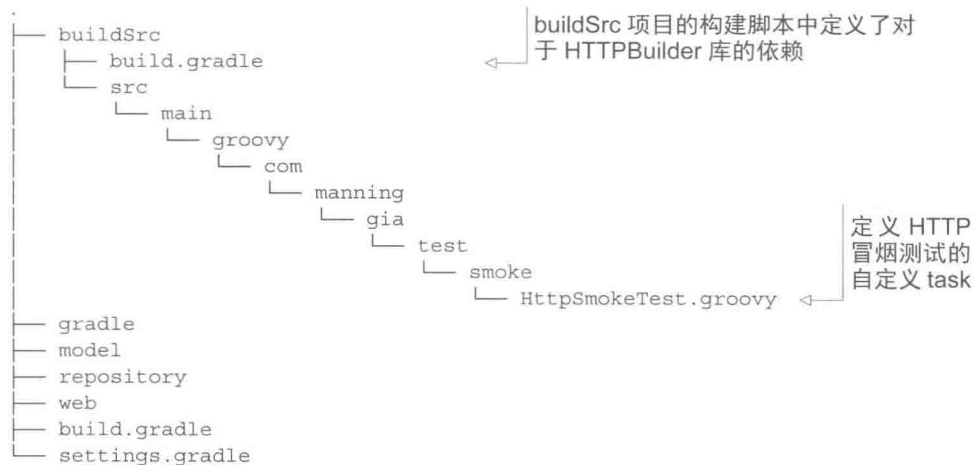
部署后，目标环境可能需要一定的反应时间来达到完全工作的状态。比如，如果部署过程重启了 Web 容器，很明显它不可能立即就接收服务请求。如果你的情况也是这样的，请确保在执行冒烟测试之前留一些回旋余地。

对于 Web 应用程序比如 To Do 应用程序的冒烟测试是什么样的？很简单——你可以发送最基本的 HTTP 请求来看看 Tomcat 服务器是否在运行。你还需要验证应用程序主页 URL 响应 HTTP 请求时返回 HTTP 状态码 200，这代表是正确的。

在 Gradle 的 task 中发送 HTTP 请求可以简单地通过使用 Java 的标准 API 类 (`java.net.HttpURLConnection`) 或者第三方库如 `ApacheHttpComponents` (`http://hc.apache.org/`) 来实现。使用 Groovy 库 `HTTPBuilder` (`http://groovy.codehaus.org/modules/http-builder/`) 会更加简单。`HTTPBuilder` 采用 DSL 风格的配置机制封装了 `HttpComponents` 所提供的功能，使你需要编写的代码大大地减少了。你会在自定义的 `HttpSmokeTest` task 中使用 `HTTPBuilder` 作为抽象层来进行 HTTP 调用。



为了让所有项目都能使用这个 task，实现类变成了 buildSrc 项目的一部分，如下面的目录树所示：



在 buildSrc 下的任何类都能使用 HTTPBuilder 库之前，你需要在构建脚本中定义它。如下清单采用的是 0.5.2 版本。

清单 15.12 buildSrc 项目的构建脚本

```

repositories {
    mavenCentral()
}

dependencies {
    compile 'org.codehaus.groovy.modules.http-builder:http-builder:0.5.2'
}
  
```

声明对 HTTPBuilder 库的依赖

正如你所想象的，以后你可以实现其他类型的冒烟测试；比如，测试数据库的功能。为此，你需要将冒烟测试分组并放在 com.manning.gia.test.smoke 包中。我们来仔细看看下面清单中对发送 HTTP 请求进行冒烟测试的实现。

清单 15.13 执行 HTTP 冒烟测试的自定义 task

```

package com.manning.gia.test.smoke

import org.gradle.api.DefaultTask
import org.gradle.api.GradleException
import org.gradle.api.tasks.Input
import org.gradle.api.tasks.TaskAction
  
```

```
import groovyx.net.http.HTTPBuilder
import static groovyx.net.http.ContentType.TEXT
```

```
class HttpSmokeTest extends DefaultTask {
    @Input
    String url

    @Input
    String errorMessage

    @TaskAction
    void letThereBeSmoke() {
        boolean success = isUp(url)

        if(!success) {
            throw new GradleException(errorMessage)
        }
    }
}
```

验证 HTTP GET 请求并检  
查响应状态码是否是 OK

如果响应状态码不是 OK，则  
使冒烟测试的 task 失效

向提供的  
URL 类发送  
HTTP GET  
请求并解析  
返回状态码

```
private boolean isUp(String url) {
    def http = new HTTPBuilder(url)
    def responseStatus = http.get(ContentType: TEXT) { resp, reader ->
        resp.status
    }

    responseStatus != HttpURLConnection.HTTP_OK
}
```

在构建脚本中，你可以设置任意多的冒烟测试。然而，这些提供了 Web 应用程序的访问点的 URL 是从之前配置的 Groovy 配置文件中读取的。图 15.5 显示了如何使用 env 项目属性来指向特定的环境。

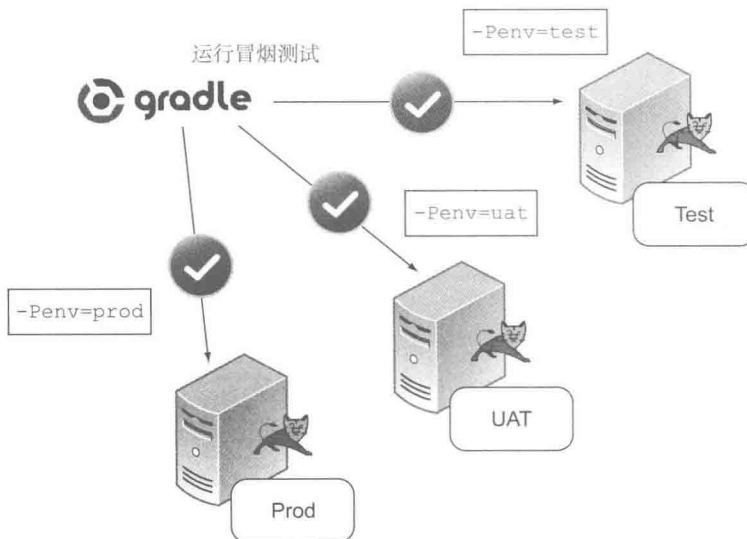


图 15.5 针对不同的环境运行冒烟测试

我们来看一些冒烟测试的典型例子。如下清单显示了两个不同的冒烟测试：一个用于验证 Tomcat 是否在运行；另一个用于检查 Web 应用程序是否部署成功。

**清单 15.14 为需要验证的 URL 执行冒烟测试**

```
import com.manning.gia.smoke.HttpSmokeTest

ext {
    tomcatUrl = "http://$config.tomcat.hostname:$config.tomcat.port"
    toDoAppUrl = "$tomcatUrl/$config.tomcat.context"
}

task checkTomcatUrl(type: HttpSmokeTest) {
    url = tomcatUrl
    errorMessage = "Tomcat doesn't seem to be up."
}

task checkApplicationUrl(type: HttpSmokeTest,
    dependsOn: checkTomcatUrl) {
    url = toDoAppUrl
    errorMessage = "Application doesn't seem to be up."
}

task smokeTests(dependsOn: checkApplicationUrl)
```

从配置文件创建环境相关的 Tomcat URL

验证 Tomcat 容器主页可以访问

验证部署的 To Do 应用程序 URL 可访问

我确信你可以想象到在现实世界的应用中冒烟测试得到了广泛应用。只要写起来简单并且能够快速执行，我们就应该实践。

如果所有的冒烟测试都通过了，你就可以认为应用程序部署成功了。但是其功能能正常工作吗？下一步，你应该验证应用程序所提供的功能是否能够如预期般正常工作。

## 15.4.2 使用验收测试验证应用程序功能

功能测试，也叫作验收测试，专门用来验证最终用户的需求是否得到了满足。在第 7 章中，你学到了在浏览器自动化工具 Geb 的帮助下如何为 Web 应用程序实现功能测试。

当然，你需要针对在其他环境中部署的应用程序运行这些测试。验收测试通常在持续交付构建管道的自动化验收测试阶段运行。在构建管道中这是第一次将开发团队的工作集成到一起，部署到测试服务器上，并以自动化形式验证功能是否满足最终用户的需求。在构建管道的后续阶段中，运行验收测试可以得到在功能层面部署是否成功的快速反馈。测试的质量越好，对于验证的结果就越有信心。

在清单 15.13 中，你添加了一个新的 Test 类型的 task 针对远程服务器来运行功能测试。Geb 允许通过设置系统属性 geb.build.baseUrl 来指向 HTTP 的端点。这个系统属性值是从环境配置中读取出来的，如下面的清单所示。

清单 15.15 添加一个 Test 类型的 task 针对远程服务器来执行功能测试

```

ext {
    functionalTestReportDir = file("${test.reports.html.destination/functional}")
    functionalTestResultsDir = file("${test.reports.junitXml.
        ↳ destination/functional}")
    functionalCommonSystemProperties = ['geb.env': 'firefox',
        ↳ 'geb.build.reportsDir':
        ↳ reporting.file("${name/geb}")]
}

task remoteFunctionalTest(type: Test) {
    testClassesDir = sourceSets.functionalTest.output.classesDir
    classpath = sourceSets.functionalTest.runtimeClasspath
    reports.html.destination = functionalTestReportDir
    reports.junitXml.destination = functionalTestResultsDir
    systemProperties functionalCommonSystemProperties
    systemProperty 'geb.build.baseUrl', toDoAppUrl
}

```

用于在不同环境运行功能测试的 Test 类型的 task

定义部署的 Web 应用程序的 HTTP 访问点

## 15.5 将部署集成到构建管道中

在前面的章节中，我们讨论过构建管道提交阶段的目的和解决方案。我们编译代码，运行自动化单元测试和集成测试，生成代码质量报告，组装二进制包，发布到仓库中以备后续使用。请参考之前的章节快速回忆一下前面配置的 Jenkins 的 job。

提交阶段检验的软件在技术层面工作的标准，验收阶段检验的是软件能否满足功能性和非功能性需求。为了达到这个目的，你需要从二进制仓库中获取包，并部署到类产品环境的测试环境中。在运行自动化验收测试来检验应用程序功能对最终用户可用之前，你运行了冒烟测试来确保部署成功。在后续的阶段中，你重用了已下载的二进制包，并部署到其他环境中：UAT，用来做手动测试；产品环境，将软件交到最终用户手中。

我们来看看 Jenkins 中的这些阶段。复制一个已有的 Jenkins 的 job 作为模板。现在不要关心它们的配置。你会修改它们的设置的。最终结果是下面的 Jenkins job：

- todo-acceptance-deploy
- todo-acceptance-test
- todo-uat-deploy
- todo-uat-smoke-test
- todo-production-deploy
- todo-production-smoke-test

这些 job 组成了构建管道的阶段，如图 15.6 所示。

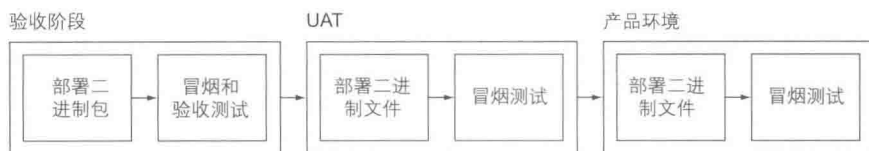


图 15.6 构建管道的验收、UAT 和产品环境阶段

在接下来一节中，我们会对每个阶段进行逐一介绍。让我们从验收阶段开始。

### 15.5.1 自动部署到测试环境

首先，通过配置 `todo-acceptance-deploy` 这个 Jenkins job 来自动部署 WAR 文件到测试服务器。图 15.7 展示了构建管道中的后续部署阶段。

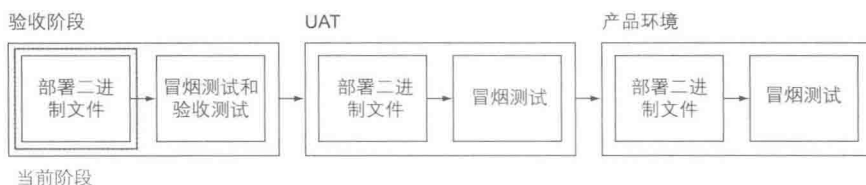


图 15.7 将 WAR 文件部署到测试服务器用于进行验收测试

快速地看看下列清单，检查一下是否都设置正确了：

- 在 Source Code Management 配置部分，选择 Clone Workspace 选项和父项目 `todo-distribution`。
- 作为构建步骤，你需要从 Artifactory 仓库下载 WAR 文件并部署到测试环境中。添加一个构建步骤来使用 Gradle wrapper 调用构建脚本并输入 `deployWar` task。在 Switches 框中，提供了相应的环境属性：`-Penv=test`。
- 根据上游项目的构建号参数定义构建名称：`todo-${ENV,var="SOURCE_BUILD_NUMBER"}`。
- 添加一个参数化的构建动作来定义一个构建触发器，来触发运行部署测试这个 `todo-acceptance-test` job。至于参数传递，可以通过选择 Current Build Parameters 选项来重用已经存在的参数。

### 15.5.2 部署测试

部署测试应该紧随部署应用程序到测试环境之后（图 15.8）。

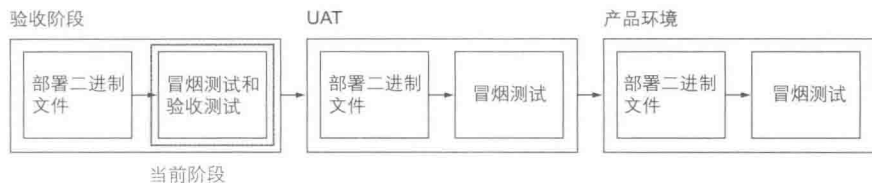


图 15.8 针对已部署的 WAR 文件进行部署测试

在配置 Jenkins 的 job 时有两个重要的点需要考虑。job 执行需要有一定的延迟来等待测试环境启动。还有就是下游项目（部署到 UAT）不应该自动执行。下列清单解释了必需的配置步骤：

- 在 Source Code Management 配置部分，选择 Clone Workspace 选项和父项目 `todo-acceptance-test`。
- 在 Advanced Project Options 配置部分，勾选 Quiet Period 复选框并在输入框中输入 60。这个选项会使 job 执行延迟一分钟，以确保 Tomcat 服务器正确启动。因为这种方式比较脆弱，你可能需要更好的机制来验证服务器是否启动成功。
- 作为构建步骤，你想要在测试环境中运行冒烟测试和验收测试。添加构建步骤来使用 Gradle wrapper 调用构建脚本并输入 `smokeTests` 和 `remoteFunctionalTest` 两个 task。在 Switches 框中，提供了相应的环境属性：`-Penv=test`。
- 根据上游项目的构建号定义构建名称：`todo#${ENV,var="SOURCE_BUILD_NUMBER"}`。
- 添加一个参数化的构建动作来定义一个手动的构建触发器，来触发部署 WAR 文件到 UAT 环境这个 `todo-uat-deploy` job。为了定义这个手动触发器，在 Add Post-build Action 下拉菜单中选择 Build Pipeline Trigger → Manually Execute Downstream Project 选项。构建管道视图会为此 job 显示一个 Play 按钮来表示这需要手动触发。

当在 Jenkins 中运行整个构建管道的时候，你会发现部署到 UAT 这个 job 需要手动干预。只有当你触发这个部署时，管道才会继续执行——也就是，直到点击另一个按钮，或者下游 job。

### 15.5.3 按需部署到 UAT 和产品环境

在第 6 步中，你已经配置了一个手动按钮。在部署二进制包到产品环境中的时候，同样需要这样的配置，如图 15.9 所示。

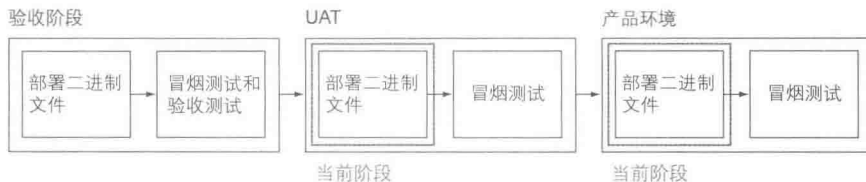


图 15.9 通过执行按钮发布到 UAT 和产品环境

我们不会过多讨论这些 job 的详细配置。实际上，它们的配置跟在验收阶段建立的 job 的配置非常相似。最大的区别就是目标环境不一样。在 Gradle 构建步骤中，部署到 UAT 的 job 需要设置 `-Penv=uat`。部署到产品环境的 job 设置是 `-Penv=prod`。

Jenkins 中的构建管道视图可以被配置成保留构建执行历史。如果你想要快速地浏览失败的和成功的构建，这个功能非常方便。这个视图也能使构建相关人员选择部署的版本。这个用例典型的场景如下：

- 生产团队决定在一个特定的版本中开启一个新的功能。
- 由于部署失败或者功能不可用，需要在产品环境中将应用程序版本回滚到上一次可工作的状态。
- QA 团队需要部署特定功能对应的版本到 UAT 中进行手动测试。

当你点击发布按钮时，Jenkins 需要知道要部署的是哪个版本。幸运的是，parameterized build 插件让你能够在 job 中指定版本信息。对于所有部署的 job，做如下配置。选择 This Build Is Parameterized 复选框。在 Add Parameter 下拉菜单中选择 String Parameter。在 Name 输入框中，输入 `SOURCE_BUILD_NUMBER`。

## 15.6 总结

软件部署需要可重复且稳定可靠。任何形式的部署失败——尤其是在生产系统中——都会导致直接的经济损失。自动化是部署过程中必需的步骤。

可部署的二进制包通常看起来有些不同的特征，它们要遵循不同的项目规定，针对不同的运行时环境。尽管部署软件没有万能解药，但是 Gradle 被证明是一个实现部署策略的灵活的工具。

一个配置好的环境是软件部署的前提。本章开始时，我们讨论过了基础环境即代码的重要性，用来自动创建和配置所需的环境和提供服务。Vagrant 在创建和测试基础环境的时候扮演了重要的角色。你学到了如何通过使用 Gradle task 封装 Vagrant 管理命令来启动虚拟机。然后，你使用 SSH 命令在 Vagrant 的虚拟机上模拟了部署和执行基本功能的过程。

为了能够重复部署，在不同的环境中，你可以使用同样的部署代码。这意味着自动化部署的逻辑需要使用动态的属性值来指向特定的环境。当使用 Groovy 的闭包机制时，环境相关的配置变得结构清晰且可读。Groovy 的 API 类 `ConfigSlurper` 提供了方便使用的机制来解析这些配置。为了能够在构建的所有项目中使用这些配置，你编写了 `task` 用来在 Gradle 的配置生命周期阶段读取 Groovy 脚本。

每一次部署的结果都需要被检验。自动化部署测试，在部署完成后执行，可以提供快速反馈。冒烟测试实现起来容易，且能快速地发现不足。功能测试，也叫验收测试，是冒烟测试的良好扩展，其验证软件是否满足了功能性和非功能性需求。

本章最后，通过配置手动部署功能扩展了构建管道。在 Jenkins 中，你配置了三个部署 `job` 来完成测试环境、UAT 环境和产品环境的部署，包括它们相对应的部署测试。最后一步完成后，你就构建了一个功能完整、端到端的构建管道。我们一起探索了使用 Gradle 和 Jenkins 实现自己的构建管道的一些必不可少的工具和方法论。



# A 驾驭命令行

Gradle 的命令行接口（CLI）是 Gradle 用户了解可用选项、检查项目和控制执行行为的首选工具。它包含三部分，探索或帮助相关的 task、构建设置 task 和配置输入。gradle 命令的用法如下：

```
gradle [option...] [tasks...]
```

## A.1 探索类task

很多探索类 task 都提供了构建信息。如果你刚上手一个项目，它们就是你开始了解项目配置最好的地方。它们以 Gradle task 的形式实现。每个 Gradle 构建最开始都提供探索类 task，如表 A.1 所示。

表 A.1 对所有 Gradle 项目可用的探索类 task

名字	描述	详细
dependencies	列出你的项目依赖，包括传递性依赖。声明和自定义依赖相关的讨论请跳到第 5 章。	5.4.2 节

名字	描述	详细
dependencyInsight	解释在依赖图中一个依赖如何被选择，为什么会被选择。检查一个特定的依赖，需要提供 <code>--dependency</code> 参数。检查 compile 依赖以外的依赖，使用 <code>--configuration</code> 参数。如： <code>gradle dependencyInsight--dependency apache-commons</code> 。	5.7.2 节
help	显示 Gradle 命令行的基本用法；比如，列出所有 task 和运行一个特定的 task。如果你运行 <code>gradle</code> 命令而未指定任何 task， <code>help task</code> 就会被自动执行。	Gradle 在线手册
projects	显示在多项目构建中的所有子项目。单项目构建没有子项目。	6.2 节
properties	列出项目中所有可用的属性。有些属性是由 Gradle 的 Project 对象提供的。其他属性是用户自定义的属性，可能来自于属性文件、属性命令行选项，或者直接在构建脚本中定义的。	4.1.3 节
tasks	显示项目中所有可运行的 task，包括它们的描述信息。应用于项目的插件提供了额外的 task。显示可用 task 的附加信息，可以使用 <code>--all</code> 选项。	2.6.1 节

## A.2 构建设置task

每个 Gradle 项目至少都需要一个 `build.gradle` 文件来定义构建逻辑。这个文件可以被手动创建或者通过构建设置插件的 task 来方便地生成。表 A.2 中显示了构建设置 task 如何初始化一个新的 Gradle 构建。

表 A.2 用来初始化 Gradle 构建的构建设置 task

名字	描述	详细
setupBuild	通过创建 <code>build.gradle</code> 和 <code>settings.gradle</code> 来初始化一个 Gradle 项目，设置 wrapper 文件。如果 Gradle 找到了一个 <code>pom.xml</code> 文件，Gradle 会尝试从这个 Maven 基本数据中继承项目信息（参见 <code>maven2Gradle</code> ）。	Gradle 在线手册
generateBuildFile	为构建 Java 项目创建一个带有标准设置的 <code>build.gradle</code> 文件，只有在项目目录下没有 <code>pom.xml</code> 文件时这个 task 才可用。	3.2.1 节
generateSettingsFile	创建一个 <code>settings.gradle</code> 文件，通常用来配置多项目构建。只有在项目目录下没有 <code>pom.xml</code> 文件时这个 task 才可用。	Gradle 在线手册
maven2Gradle	通过分析项目目录下的 POM 文件将一个 Maven 项目转换成 Gradle 项目（通常作为构建迁移的一部分）。这个 task 运行完成后， <code>build.gradle</code> 和 <code>settings.gradle</code> 文件就被自动创建了。只有在没有 <code>pom.xml</code> 文件时这个 task 才可用。	9.2.2 节
wrapper	在 Gradle 项目目录下生成 Gradle Wrapper 文件，使用与 Gradle 运行时相同的版本。	3.4.1

## A.3 配置输入

构建配置信息可以通过 CLI 提供。不需要提供值的选项可以被组合起来使用；比如，`-is` 用来运行构建并且配置日志级别为 INFO，如果有错误发生则将堆栈跟踪信息打印出来。

### A.3.1 常用选项

表 A.3 中显示了一些常用的命令行选项，它们不属于某个功能分组。这些选项在你的 Gradle 日常使用当中可能非常有用，所以去探索一下它们吧。

表 A.3 常用的命令行选项

名字	描述	详细
<code>-?, -h, --help</code>	打印出所有可用的命令行选项，包含描述信息。	Gradle 在线手册
<code>-a, --no-rebuild</code>	避免重新构建多项目构建中的所有子项目（也叫作部分构建）。通过使用部分构建，可以节约检查子项目模型的开销，降低构建执行时间。	6.3.3 节
<code>-b, --build-file</code>	Gradle 构建脚本默认的命名约定是 <code>build.gradle</code> 。使用这个选项执行其他名字的构建脚本（比如， <code>gradle -b test.gradle build</code> ）。	8.3 节
<code>-c, --settings.file</code>	Gradle 设置文件默认的命名约定是 <code>settings.gradle</code> 。使用这个选项执行非标准设置文件名的构建（比如， <code>gradle -c mySettings.gradle build</code> ）。	Gradle 在线手册
<code>--continue</code>	在一个 task 执行失败后 Gradle 会继续执行。在多项目构建中这个选项极其有用。它让你在构建时发现所有可能的问题，并一起修复它们，而不必一一修复。	Gradle 在线手册
<code>--configure-on-demand</code>	这个选项的目的是优化初始化多项目构建的配置时间。这种模式尝试只配置跟正在请求的 task 相关的项目。这个选项可以通过在 <code>gradle.properties</code> 文件中设置 <code>org.gradle.configure-ondemand</code> 属性来激活。	Gradle 在线手册
<code>-g, --gradle-user-home</code>	Gradle 的默认 home 目录位于用户 home 目录下的 <code>.gradle</code> 目录中。如果你想要指向不同的目录，则使用这个选项。	Gradle 在线手册
<code>--gui</code>	运行一个基于 Swing 的图形化用户界面。	Gradle 在线手册
<code>-I, --init-script</code>	设置一个初始化脚本用于构建。这个脚本会在所有构建 task 执行前被执行。	Gradle 在线手册
<code>-p, --project-dir</code>	默认的，Gradle 会在当前目录下执行构建。通过这个选项，你可以指定不同的目录来执行构建。	Gradle 在线手册

名字	描述	详细
--parallel	并行构建参与多项目构建的子项目。Gradle 会自动确定最优的线程数。这个选项可以通过在 gradle.properties 文件中设置 org.gradle.parallel 属性来激活。	Gradle 在线手册
--parallel-threads	当并行构建多项目构建的时候，这个选项可以被用来重写线程数（比如，--parallel-threads=5）。	Gradle 在线手册
-m, --dry-run	打印 task 的执行顺序，而不必真的执行它们。如果你想要快速地确定 task 的执行顺序，这个选项会很方便。	Gradle 在线手册
--profile	除了每次构建时输出总的构建时间，你可以将构建时间拆分得更小。profile 选项在 build/reports/profile/ 目录下生成了详细的 HTML 报告，其中列出了所有 task 的执行时间和在配置阶段所用的时间。	Gradle 在线手册
--rerun-tasks	重新运行 task 执行图中所有确定的 task。这个选项会忽略前面 task 执行的任何 UP-TO-DATE 状态。	Gradle 在线手册
-u, --no-search-upwards	告诉 Gradle 不要在父目录中寻找设置文件。在有深层次嵌套的项目结构中，这个选项被用来避免在父目录中搜索，从而节约时间。	6.2.4 节
-v, --version	打印出 Gradle 运行时的版本信息	2.4 节
-x, --exclude-task	指定某个 task 在构建的时候不执行。一个比较典型的例子就是如果你想执行一个完整的构建，但是不执行所有的单元测试（比如，gradle -x test build）。	2.6.2 节

A.3.2 属性选项

属性提供了一种在命令行配置构建的方式。除了标准的 Java 系统属性以外，Gradle 定义了项目属性。表 A.4 中描述了它们的特定用例。

表 A.4 为 Gradle JVM 进程或 Gradle 项目提供的属性

名字	描述	详细
-D, --system-prop	Gradle 以 JVM 进程的形式运行。与所有的 Java 进程一样，你可以指定系统属性如：-Dmyprop=myvalue。	4.1.3 节
-P, --project-prop	项目属性是在构建脚本中能够被使用的变量。你可以使用这个选项从命令行直接将一个参数传递到构建脚本中（比如，-Pmyprop=myvalue）。	4.1.3 节

### A.3.3 日志选项

Gradle 允许访问构建产生的所有日志信息。根据情况，你可以提供日志选项来过滤相关的重要消息，如表 A.5 所示。

表 A.5 控制运行时的日志级别

名字	描述	详细
<code>-i, --info</code>	在默认设置下 Gradle 构建并不会输出大量的日志信息。通过这个选项将日志级别设置成 INFO 来获得更多的日志信息。	7.3.1 节
<code>-d, --debug</code>	以 DEBUG 日志级别运行 Gradle 构建，将会产生大量的日志信息，包括堆栈跟踪信息，这在查错的时候特别有用。	Gradle 在线手册
<code>-q, --quiet</code>	减少构建输出只剩下错误信息。	2.6.1 节
<code>-s, --stacktrace</code>	如果构建出错了，你会希望知道哪里出错了。如果有异常抛出，这个 <code>-s</code> 选项会打印出一个简单的堆栈跟踪信息，这在调试失败构建的时候非常有用。	Gradle 在线手册
<code>-S, --full-stacktrace</code>	打印出完整的异常堆栈跟踪信息	Gradle 在线手册

### A.3.4 缓存选项

Gradle 利用多级别的缓存机制来提高构建性能。使用表 A.6 中提供的缓存选项，你可以改变默认的缓存行为。

表 A.6 管理 Gradle 的缓存功能

名字	描述	详细
<code>--offline</code>	通常你的构建定义了一些依赖，如果这些依赖在本地没有存储，并且运行构建的时候没有网络连接就会引起构建失败。使用这个选项，可以让构建采用离线模式运行，并且只检查本地存储的依赖。	5.6.2 节
<code>--project-cache-dir</code>	默认的依赖缓存目录位于用户目录中的 <code>.gradle</code> 目录下。这个选项可以被用来指定一个不同的目录。	Gradle 在线手册
<code>--recompile-scripts</code>	Gradle 默认编译所有的脚本并存储在本地缓存中以提高构建性能。使用这个选项来清空这些缓存。	Gradle 在线手册
<code>--refresh-dependencies</code>	手动刷新缓存中的依赖。这个标志强制检查依赖的版本。	5.7.4 节

A.3.5 后台守护进程选项

守护进程以后台进程的形式运行 Gradle。一旦开始，gradle 命令会重用已经获得的后台进程执行后续构建，从而避免每次启动时的开销。表 A.7 中给出了控制后台进程的选项概览。

表 A.7 管理后台守护进程

名字	描述	详细
--daemon	使用后台守护进程模式执行构建可以提高构建性能。如果后台进程已经存在，则会重用它，如果不存在，则会启动一个新的后台进程。后台守护进程可以通过在 gradle.properties 文件中设置 org.gradle.daemon 属性来激活。	2.6.4 节
--foreground	在终端中运行 Gradle 后台守护进程，用于调试和监控目的。	2.6.4 节
--no-daemon	不使用已有的 Gradle 后台守护进程执行构建。	2.6.4 节
--stop	终止一个已经存在的 Gradle 后台守护进程。	2.6.4 节

# Gradle用户所需要了解的 Groovy

---

Gradle 的核心功能是使用 Java 实现的。在这些功能之上，有一个使用动态编程语言 Groovy 编写的领域特定语言（DSL）。当编写 Gradle 构建脚本时，你可以自动使用 DSL 所暴露出来的语言特性来表达所需的构建指令。Gradle 构建脚本就是可执行的 Groovy 脚本，但是它们却不能通过 Groovy 运行时环境运行。当需要实现自定义的逻辑时，可以使用 Groovy 的语言特性在 Gradle 构建脚本中直接构建所需的功能。这个附录是了解 Groovy 语言的入门，并解释了 Gradle 用户学习 Groovy 语言的重要性。稍后，我还会演示 Gradle 的一些配置元素是怎样通过 Groovy 实现的。

## B.1 什么是Groovy

Groovy 是针对 Java 虚拟机（JVM）的一种动态编程语言。其语法与 Java 类似。这种语言与已经存在的 Java 类或者类库集成，这对于 Java 程序员来说，学习起来非常容易。Groovy 不仅仅基于 Java 的优势，它还提供了很多强大的编程特性，这些特性是从 Ruby、Python 等其他语言中借鉴而来的。Groovy 是一种脚本语言，不用编译其代码。当然，Groovy 代码也可以被编译成 Java 二进制代码。在本书中，你使用了两种方式。

虽然这个附录会告诉你一些 Groovy 最重要的语言特性，但还是强烈推荐你自己去进一步探索。有很多书或者在线资源可供使用。Dierk Konig 等人所著的 *Groovy*

*in Action, Second Edition* (Manning, 2009) 就是 Groovy 权威指南。它详细地解释了所有的语言特性。DZone 针对 Groovy 的参考手册提供了一个对每个 Groovy 初学者都非常方便的清单。你可以从 <http://refcardz.dzone.com/refcardz/groovy> 免费下载。探索 Groovy 另一个很好的地方是由 Hubert A. Klein Ikkink 维护的 *Groovy Goodness* 博客, 地址是 <http://mrhaki.blogspot.com/search/label/Groovy%3AGoodness>, 每篇博客都通过例子解释了一个独特的 Groovy 语言特性。作者将其博客汇集到了一本叫作 *Groovy Goodness Notebook* (Lenanpub, 2013) 的书中。

## B.2 我需要知道多少Groovy知识

如果你是 Gradle 新手, 可能想知道需要学习多少 Groovy 知识才能编写出第一个构建脚本。答案很简单, 就是很少。然而, 强烈推荐你学习一些 Java 知识, Groovy 几乎 100% 兼容 Java。在构建脚本中实现 task 动作的时候, 你可以选择使用纯 Java 代码或者使用 Groovy 的具有表述性的语言构造。

我们来看一个例子。假设你想要知道一个目录下的所有文件并将它们的名字写到一个新的 `allfiles.txt` 文件中。听起来很简单, 对吧? 如下清单演示了 Java 版本的 `doLast` task 动作。

清单 B.1 使用 Java 语法编写的 Gradle task

```
task appendFileNames << {
    File inputDirectory = new File("src");
    File outputFile = new File(getBuildDir(), "allfiles.txt");
    File outputDirectory = outputFile.getParentFile();

    if(!outputDirectory.exists()) {
        outputDirectory.mkdirs();
    }

    outputFile.createNewFile();
    FileWriter fileWriter = new FileWriter(outputFile, true);

    try {
        for(File file : inputDirectory.listFiles()) {
            fileWriter.write(file.getName() + "\n");
        }
    } finally {
        fileWriter.close();
    }
}
```

← Gradle task 定义

Java 实现的  
task 动作

除了 task 本身的定义使用 Groovy 的 DSL 定义外, 实现这个场景不需要使用 Groovy 代码。接下来, 我们会将这个 task 的实现与 Groovy 版本的进行比较。在下



面的清单中，task 动作的代码比 Java 版本的更短、更简洁。

#### 清单 B.2 使用 Groovy 语法编写的 Gradle task

```
task appendFileNames << {  
    def inputDirectory = new File('src')  
    def outputFile = new File(buildDir, 'allfiles.txt')  
    def outputDirectory = outputFile.parentFile  
  
    if(!outputDirectory.exists()) {  
        outputDirectory.mkdirs()  
    }  
  
    outputFile.createNewFile()  
    inputDirectory.eachFile { outputFile << "$it.name\n" }  
}
```

← Gradle task 定义

Groovy 实现的  
task 动作

在构建中混合使用 Java 和 Groovy 可以工作得很好，尤其是在学习 Groovy 的过程中。更复杂的构建包括自定义 task 和插件，你需要知道更多的关于这种语言的知识。并且如果想要深入理解 Gradle 的工作原理，那么知道 Groovy 的高级特性是关键。

## B.3 比较Java和Groovy语法

在上一节中，我们比较了一个用 Java 和 Groovy 实现的 task。在 Groovy 中你可以高效地工作，同时编写更少的代码。为了更好地理解，我们来讨论一下这两种语言的主要区别。我们会以本书第 4 章中的 ProjectVersion.java 类为例。这个类描述了 Gradle 项目的主版本和次版本。下面的清单显示了这个类的简化版本。

#### 清单 B.3 一个典型的 Java POJO 类

```
package com.manning.gia;  
  
public class ProjectVersion {  
    private Integer major;  
    private Integer minor;  
  
    public ProjectVersion(Integer major, Integer minor) {  
        this.major = major;  
        this.minor = minor;  
    }  
  
    public Integer getMajor() {  
        return major;  
    }  
  
    public void setMajor(Integer major) {  
        this.major = major;  
    }  
  
    public Integer getMinor() {  
        return minor;  
    }  
}
```

```
public void setMinor(Integer minor) {  
    this.minor = minor;  
}
```

作为 Java 程序员，你可能每天都会看到这样的代码。Plain Old Java Object (POJO)，没有实现或扩展第三方类库的类，通常被用来表示数据。这个类通过 getter 和 setter 方法暴露了两个私有属性。如下清单演示了在 Groovy 中如何表达同样的逻辑。

#### 清单 B.4 使用 Groovy 编写的 ProjectVersion 类

```
package com.manning.gia  
  
class ProjectVersion {  
    Integer major  
    Integer minor  
  
    ProjectVersion(Integer major, Integer minor) {  
        this.major = major  
        this.minor = minor  
    }  
}
```

我想我们同意 Groovy 版本的类更加简洁一些。Groovy 为你所编写的类假定了合理的默认值，尤其是以下优化：

- 表达式后面的分号是可选的。
- 每个类、构造器和方法默认是 public 的。
- 在 Groovy 中，方法体中的最后一个表达式的值会被作为返回值。这意味着 return 语句是可选的。
- Groovy 编译器自动加上 getter/setting 方法，所以你不需要自己写。
- 类的属性可以通过点号来获取，看起来好象它们在 Java 中是共有的一样。然而，在底层 Groovy 调用的是自动生成的 getter/setting 方法。
- 如果你用 == 比较两个类的实例，在底层 Groovy 会自动调用 equals() 方法。这个操作也避免了可能的 NullPointerExceptions，即空指针异常。

## B.4 高效的Groovy特性

至此，我们只是看到了 Groovy 特性的皮毛。在本节中，我们会探索在使用 Groovy 编程时常用的语言方面的基本功能。我们接下来讨论的特性并没有特定的顺序。你可以在自己的计算机上面试一下每个例子。有两个特别好用的工具用来执行代码片段。

在安装好 Groovy 运行时环境后，Groovy 终端就自动可以使用了。它提供了交互式的用户界面用于输入和执行 Groovy 脚本。可以通过运行 `groovyConsole` 命令或者位于 `<GROOVY_HOME>/bin` 目录下的 `groovyConsole.bat` 来打开 Groovy 终端。

Groovy Web 终端 (<http://groovyconsole.appspot.com/>) 为执行 Groovy 代码片段提供了更加方便的方式。你可以在 Web 终端运行 Groovy 脚本，而不需要安装 Groovy 运行时环境。记住，你在脚本中所使用的语言特性跟 Groovy 版本的 Web 终端是绑定在一起的。

### B.4.1 assert 语句

如果你熟悉 Java，可能会知道 `assert` 关键字。它用来验证代码中的前置或者后置条件。在 Java 中，`assert` 只有在设置了运行时标志 (`-ea` 或者 `-enableassertion`) 来进行断言检查时才有用，而 Groovy 的 `assert` 语句一直有用。如下清单显示了一个例子。

#### 清单 B.5 Groovy 的 power assert

```
def version = 12
assert version == 12
version++
assert version == 12
```

制造一次失败的断言

细心的读者会发现第二个 `assert` 语句会失败，因为 `version` 变量已经加 1 了。Groovy 的 `assert` 语句，也被叫作 *power assert*，提供了有用的输出，用于查找问题的根源。如下显示了一个 `assert` 的输出

```
Exception thrown
Jul 29, 2013 8:06:04 AM org.codehaus.groovy.runtime.StackTraceUtils
sanitize
WARNING: Sanitizing stacktrace:
Assertion failed:
assert version == 12
    |             |
    13          false
```

在接下来的代码例子中会使用 `assert` 语句作为工具来验证所期望的行为。

### B.4.2 可选类型定义

Groovy 并不强制你显示声明变量类型、方法参数或者返回类型。你可以使用 `def` 关键字进行简单的标识，它作为 `java.lang.Object` 的一个占位符。在运行时，Groovy 通过所赋的值分析出其类型。下面的清单演示了一些可选类型的例子。

### 清单 B.6 变量和方法的可选类型

```
def buildTool = 'Gradle'
assert buildTool.class == java.lang.String

def initProjectVersion(major, minor) {
    new ProjectVersion(major, minor)
}

assert initProjectVersion(1, 2).class == com.manning.gia.ProjectVersion
```

这个特性在某种情况下非常有用，但是也许你还是喜欢显式的强类型。特别是在暴露公共 API 的项目中，强类型自动提高了文档的完整性，使所提供的参数类型更加明显，也使 IDE 能够提供方便的代码补全功能。出于同样的原因，如果一个方法没有返回值，则应该声明 `void` 而不是 `def` 作为返回类型。

## B.4.3 可选的括号

在 Groovy 中如果方法签名需要至少一个参数的话，则方法调用可以省略括号。在不涉及太多细节的情况下这个特性通常被用来创建更自然的 DSL，一种可以被领域专家理解可读的语言。如下清单比较了有括号和没有括号的两种方法调用。

### 清单 B.7 对于顶层的表达式省略括号

```
initProjectVersion(1, 2)
initProjectVersion 1, 2

println('Groovy is awesome!')
println 'Groovy is awesome!'
```

## B.4.4 字符串

在 Groovy 中有三种不同的方式可以定义字符串。带单引号的字符串通常创建出等效于 Java 的 `String` 类型。第二种方式与在 Java 中创建字符串的方式相同，即使用双引号包起来。多个字符串，使用三个双引号包起来，这在赋值长文本或者格式化（比如，多行 SQL 语句）时非常有用。下面的清单显示了在 Groovy 中创建字符串的所有方法。

### 清单 B.8 在 Groovy 中字符串表示方法

```
def myString1 = 'This is a single-quoted String'
def myString2 = "This is a double-quoted String"
def myString3 = """
    This
    is a
    multiline
    String
    """
```

## B.4.5 Groovy 字符串（GString）

在 Groovy 中带双引号的字符串比传统的 Java 字符串更加强大。它们可以插值到变量或表达式中，通过美元符号（\$）和花括号来表示。在运行时，Groovy 会计算其中的表达式并组成一个字符串。在 Groovy 中这种字符串通常被叫作 GString。如下清单给出了用例。

### 清单 B.9 使用 GString 进行字符串插值

```
def language = 'groovy'
def sentence = "$language is awesome!"
assert sentence == 'groovy is awesome!'

def improvedSentence = "${language.capitalize()} is awesome!"
assert improvedSentence == 'Groovy is awesome!'
```

## B.4.6 集合 API

Groovy 为集合 API 的实现提供了简洁的语法，使得其比 Java 中类似的使用更加方便。接下来，我们会讨论 List 和 Map。

### List


在方括号中放入一串以逗号分隔的值，就可以初始化新的 List。在底层，Groovy 创建了一个 `java.util.ArrayList` 实例。Groovy 也添加了一些语法糖来简化 List 的使用。一个完美的例子就是可以通过左移操作符（<<）向 List 中添加一个新的元素。在底层，Groovy 调用了 `add` 方法。如下清单演示了这个功能。

### 清单 B.10 在 Groovy 中管理 List

```
def buildTools = ['Ant', 'Maven']
assert buildTools.getClass() == java.util.ArrayList
assert buildTools.size() == 2
assert buildTools[1] == 'Maven'

buildTools << 'Gradle'
assert buildTools.size() == 3
assert buildTools[2] == 'Gradle'

buildTools.each { buildTool ->
    println buildTool
}
```



### Map

处理 Map 比处理 List 更加容易。在方括号中放入一串用逗号分隔的键值对就可以创建一个新的 Map。Map 的默认实现是 `java.lang.LinkedHashMap`。如下清单演示了 Map 的使用。

清单 B.11 在 Groovy 中管理 Map

```
def inceptionYears = ['Ant': 2000, 'Maven': 2004]
assert inceptionYears.getClass() == java.lang.LinkedHashMap
assert inceptionYears.size() == 2
assert inceptionYears.Ant == 2000
assert inceptionYears['Ant'] == 2000

inceptionYears['Gradle'] = 2009
assert inceptionYears.size() == 3
assert inceptionYears['Gradle'] == 2009

inceptionYears.each { buildTool, year ->
    println "$buildTool was first released in $year"
}
```

遍历 Map 中的值

## B.4.7 命名参数

之前我们讨论过简单的 `ProjectVersion` 类。这个类暴露了一个构造器来初始化其中的属性。假设你没有定义构造器。Groovy 提供了一样更加方便的方式来设置属性值，叫作命名参数。这种机制首先调用类的默认构造器，然后为每个参数调用对应的 `setter` 方法。如下清单演示了如何通过命名参数来设置 `major` 和 `minor` 属性的值。

清单 B.12 使用命名参数设置属性值

```
class ProjectVersion {
    Integer major
    Integer minor
}

ProjectVersion projectVersion = new ProjectVersion(major: 1, minor: 10)
assert projectVersion.minor == 10
projectVersion.minor = 30
assert projectVersion.minor == 30
```

## B.4.8 闭包

闭包是一个类型为 `groovy.lang.Closure` 的代码块，与其他编程语言的 `lambdas` 特性类似。闭包可以被赋值给变量，作为参数传递给方法，并且像普通方法一样来调用。

### 隐式的闭包参数

每个没有显式定义任何参数的闭包都可以访问一个隐式的参数 `it`。`it` 代表调用这个闭包的时候第一个传递进来的参数。如果没有提供参数，那么参数的值就是 `null`。我们来看一个例子让这个概念不那么抽象。如下清单显示了闭包的定义和调用，并且包含隐式的参数 `it`。

**清单 B.13 定义带有单一的、隐式参数的闭包**

```
def incrementMajorProjectVersion = {  
    it.major++  
}  
  
ProjectVersion projectVersion = new ProjectVersion(major: 1, minor: 10)  
incrementMajorProjectVersion(projectVersion)  
assert projectVersion.major == 2
```

← 显式的闭包参数 it

← 调用闭包并提供所需的参数

**显式的闭包参数**

除了使用隐式的闭包参数，你可以自定义更具描述性的参数名。如下清单定义了一个名为 version 的 ProjectVersion 类型的参数。

**清单 B.14 定义带有单一的、显式参数的闭包**

```
def incrementMajorProjectVersion = { ProjectVersion version ->  
    version.major++  
}  
  
ProjectVersion projectVersion = new ProjectVersion(major: 1, minor: 10)  
incrementMajorProjectVersion(projectVersion)  
assert projectVersion.major == 2
```

← 隐式命名闭包参数

← 调用闭包并提供所需参数

记住，在 Groovy 中类型是可选的。你可以使用标识符 version 而不指定类型。Groovy 没有限制闭包可以指定的参数数量。如下清单显示了如何定义带有多个无类型参数的闭包。

**清单 B.15 定义带有多个无类型参数的闭包**

```
def setFullProjectVersion = { projectVersion, major, minor ->  
    projectVersion.major = major  
    projectVersion.minor = minor  
}  
  
ProjectVersion projectVersion = new ProjectVersion(major: 1, minor: 10)  
setFullProjectVersion(projectVersion, 2, 1)  
assert projectVersion.major == 2  
assert projectVersion.minor == 1
```

← 声明三个无类型的闭包参数

← 调用闭包并提供所需参数

**闭包返回值**

闭包总是会返回一个值。返回值是闭包的最后一条语句的值（如果没有显式的 return 语句），或者是可执行的 return 语句的值。如果闭包的最后一条语句没有值，就返回 null。如下清单中显示的闭包返回了最后一条语句的值，即项目的次版本号。

## 清单 B.16 闭包返回值

```
ProjectVersion projectVersion = new ProjectVersion(major: 1, minor: 10)
def minorVersion = { projectVersion.minor }
assert minorVersion() == 10
```

闭包返回最后一条语句的值

## 闭包作为方法参数

之前提到过，你也可以将闭包作为方法参数。如下清单显示了一个例子。

## 清单 B.17 闭包作为方法参数

```
Integer incrementVersion(Closure closure, Integer count) {
    closure() + count
}

ProjectVersion projectVersion = new ProjectVersion(major: 1, minor: 10)
assert incrementVersion({ projectVersion.minor }, 2) == 12
```

调用闭包并用其返回值计数

定义闭包为第一个参数的方法

调用方法并提供闭包作为第一个参数

## 闭包委托

闭包代码在委托的闭包上执行。默认的，这个委托就是闭包的所有者。比如，如果你在 Groovy 脚本中定义了一个闭包，那么所有者就是一个 `groovy.lang.Script` 实例。闭包的隐式变量 `delegate` 允许你重新定义默认的所有者。考虑下面清单中的场景。你设置了委托给 `ProjectVersion` 实例，这意味着所有闭包都在这个实例上执行。

## 清单 B.18 设置闭包委托

```
class ProjectVersion {
    Integer major
    Integer minor

    void increment(Closure closure) {
        closure.resolveStrategy = Closure.DELEGATE_ONLY
        closure.delegate = this
        closure()
    }
}

ProjectVersion projectVersion = new ProjectVersion(major: 1, minor: 10)
projectVersion.increment { major += 1 }
assert projectVersion.major == 2
projectVersion.increment { minor += 5 }
assert projectVersion.minor == 15
```

设置闭包的委托

ProjectVersion 的内部方法用闭包作为参数

只处理委托引用

调用闭包

调用 ProjectVersion 实例的方法，其内部执行了一个闭包

## B.4.9 Groovy 开发工具库

Groovy Development Kit (GDK) 扩展了 Java Development Kit (JDK)，其为标



准的 JDK 类提供了很多方便的方法。你可以在 <http://groovy.codehaus.org/groovy-jdk/> 上找到一个 Javadoc 风格的 HTML 文档。你会在如 String、Collection、File 和 Stream 这些类里面找到很多有用的方法。在这些方法中很多都使用闭包作为参数，为这门语言添加了很多函数式语言的味道。你已经在清单 B.10 和 B.11 中看到了这样的方法：each。你使用这个方法遍历了 Collection 中的所有元素。在下面的清单中，我们来看看其他的例子。

### 清单 B.19 通过 GDK 添加的方法示例

```
def buildTools = ['Ant', 'Maven', 'Gradle']
assert buildTools.find { it == 'Gradle' } == 'Gradle'
assert buildTools.every { it.size() >= 4 } == false
assert 'gradle'.capitalize() == 'Gradle'
new File('build.gradle').eachLine { line ->
    println line
}
```

添加到 Collections 的方法

方便的字符串处理方法

简化遍历文件的每一行

## B.5 在Gradle构建脚本中使用Groovy

Gradle 构建脚本就是合法的 Groovy 脚本。在构建脚本中，你可以使用 Groovy 语言的所有特性。这意味着代码与 Groovy 语法严格吻合。不合法的代码会在执行构建时导致运行时错误。

Gradle 使用 Groovy 编写的 DSL 来建模典型的构建关系。在第 4 章中你学到了每个构建脚本都至少有一个对应的 `org.gradle.api.Project` 实例。在大多数情况下，在构建脚本中调用的属性和方法都自动委托给了这个 Project 实例。

我们来看看清单 B.20 中显示的构建脚本示例。在通过例子了解了 Groovy 的一些语言特性后，你可能已经知道了其内部是如何工作的。我希望这个例子可以为 Gradle 和 Groovy 的初学者揭秘其中的“魔法”。

### 清单 B.20 在 Gradle 构建脚本中应用 Groovy 语法

```
apply plugin: 'java'
version = '0.1'
repositories {
    mavenCentral()
}
dependencies {
    compile 'commons-codec:commons-codec:1.6'
}
```

调用闭包委托的 Maven Central() 方法 (见 B.4.8 节)

调用 Project 的 apply 方法，关于参数，提供一个只有一个键值对的 Map (见 B.4.6 节)。方法调用省略括号 (见 B.4.3 节)

通过调用 Project 的 setter 方法为项目设置版本属性 (见 B.3 节)

调用 Project 的 repositories 方法，并传递一个闭包参数 (见 B.4.8 节)

调用 Project 的 dependencies 方法，并传递一个闭包参数 (见 B.4.8 节)

调用闭包委托的对象的 compile 方法，用一个 String 作为参数 (见 B.4.8 节)，方法调用省略括号 (见 B.4.3 节)



# 索引

## Symbols

---

: (colon) 138  
-? option 45

## A

---

-a option 147  
acceptance stage  
    automatic deployment to test  
        environment 417  
    deployment tests 417  
    description 416  
    on-demand deployment 418  
acceptance tests 415  
AJAX (asynchronous JavaScript  
    and XML) 284  
--all option 43  
allprojects method  
    application of 151  
    configuration 389  
    example 152  
    purpose 143  
analysis  
    external Java library 290–291  
    JavaScript 284  
annotations 91  
Ant  
    build script  
        components 12, 14  
        example 14–15, 226  
    characteristics 12, 15  
    comparison to Maven 21  
    Gradle integration 224, 236  
    history of 24  
    migration 23, 34, 233, 236  
        purpose 12  
        SCP task 408  
        task 113  
Ant Get task 232  
ant property  
    definition 41  
    Gradle build script 226  
AntBuilder class 224  
Apache Commons Lang library  
    Ant build script 226  
    CharUtils class 59  
    configuration 59  
Apache license 35  
API (application programming  
    interface)  
    dependency 114, 129  
    Maven repository, custom 124  
    Project API 77, 143–144  
    RepositoryHandler interface 121  
    Settings API 139, 155  
    task 78  
    test class 171  
API (Cloudbees)  
    information retrieval 199  
    properties 198  
API key 195  
application plugin 57  
application programming interface.  
    *See* API  
apply method 210  
archive  
    configuration 361  
    file type 360  
arrow  
    definition 9  
    dependsOn 100

- artifact
  - change detection 127
  - declaration 361, 363, 372–373
  - deployment 238
  - download, reduction of 128
  - filename creation 360
  - outcome 243
  - public artifact 382
  - publication name 370
  - publication process 366, 369, 371
  - quantity of 360
  - registration
    - artifact method 372
    - process of 362
  - retrieval 18, 405–406
  - signature 381, 384
  - See also* deployable artifact
- artifact method 372
- artifactId attribute 385
- Artifactory
  - installation 367
  - libs-release-local repository 377
  - libs-snapshot-local repository 377
  - preconfigured repository 377
  - publication process 385
  - retrieval 405–406
- Asgard 38
- assemble task
  - artifact 361
  - JAR file 360
- assertion
  - code coverage 312
  - creation of 163
  - example 433
  - failure 164–165
- asynchronous JavaScript and XML.
  - See* AJAX
- automated acceptance test stage
  - definition 37
  - example 415
- automated testing. *See* test automation
- automation. *See* project automation

## B

- BDD (behavior-driven development) 167
- beforeMerged hook 256
- bidirectional dependencies 302
- Bintray
  - artifact consumption 383
  - artifact uploading 382
  - description 380
  - setup of 380
- branch coverage 313
- build by convention
  - adaptation 48
  - definition 31
- build comparison plugin
  - purpose 243
  - report 245
  - use of 245
- build engine 11
- build execution 340
- build file
  - content of 10
  - root project 153
  - subproject 154
  - See also* build script
- build job (Jenkins)
  - creation of 344
  - definition 344
  - execution 347–348
  - pipeline jobs configuration 356, 358
- build lifecycle
  - configuration phase 87
  - functional testing 189
  - Gradle 87
  - initialization phase 87
  - Maven 16
  - test report 170
- Build Name Setter plugin (Jenkins) 354
- build number 388
- build pipeline
  - challenges 352
  - definition 338
  - deployment 416, 419
  - environment 401
  - Maven versioning scheme 387
  - model 352
  - view 355–356
  - See also* deployment pipeline
- Build Pipeline plugin (Jenkins) 355–356
- build script
  - build.gradle 40
  - buildSrc directory 98
  - class source files 98
  - classpath 197
  - command-line command 55
  - continuous integration environment 339
  - extendibility 34
  - external library 197
  - Gradle wrapper 70–71
  - Groovy 439
  - initialization script 103
  - plugin 210
  - POGO, addition of 85
  - properties file 85
  - script plugin 196
  - smoke tests 414

build script (*continued*)  
 standalone object plugin 219  
 test listener implementation 177  
 TestNG 167

*See also* build file

build setup plugin 240

build step 346

build task

purpose 55  
 root project 146–147

build tools

Ant 12, 16  
 components of 10, 12  
 conversion 224  
 definition 8–9  
 evolution of 24, 26  
 Gradle features 20, 23  
 legacy project 58  
 Maven 16, 19

build trigger

code integration process 339  
 configuration (Jenkins) 345  
 Parameterized Trigger plugin 353

build-announcements plugin 103

buildDependents task 149

--build-file option 45

build.gradle file

build script 40  
 dependency definition 110  
 joint compilation 303  
 name of 154  
 project 76  
 project-specific build 144  
 root project 153  
 Settings instance 139  
 settings.gradle file 138  
 subproject 153

buildNeeded task 148

buildSrc directory

custom task class 202, 204  
 incremental build 202  
 use of 98–99  
 versioning scheme 389

## C

-c option 141

C++ plugin 307

cache

dependency 32, 127, 198  
 features 127–128  
 manual refresh 131  
 offline mode 128  
 snapshot 131  
 structure 126–127

Cargo

cache structure 126  
 dependency 110, 116  
 file dependency 120  
 flat directory repository 125  
 hot deployment 407  
 Ivy repository 124  
 Maven repository 121  
 purpose 110  
 repository 117  
 version resolution 119

Cascading Style Sheets. *See* CSS

Checkstyle plugin

Sonar report 330  
 tool comparison 320  
 use of 322, 324

CI (continuous integration)

benefits 338, 340  
 cloud-based CI servers 351–352  
 definition 7, 337

CI server

cloud-based 351–352  
 continuous integration environment 339  
 definition 338

class

build script 98  
 class source files 98  
 compiled class 99

classifier attribute

definition 116  
 publication 372

classpath

build script 197  
 custom task class 202

.classpath file

definition 251  
 generation of 253

cleanEclipse task 253

cleanIdea command 261

CLI (command-line interface)

definition 421  
 description 36  
 use of 42, 45

Clojure plugin 307

Clone Workspace SCM plugin  
 (Jenkins) 354

closure

definition 436  
 delegation 438  
 explicit closure parameter 437  
 method parameter 438  
 return value 437

cloud provisioning 396

cloud-based CI servers 351–352

- CloudBees
  - account creation 193, 195
  - API key 195
  - application provisioning 195
  - BuildHive 351
  - DEV@cloud 351
  - hibernate status 200
  - integration testing 277, 280
  - platform as a service 192
  - plugin creation 196, 201
- Clover
  - Sonar report 335
  - tool comparison 313
- Cobertura plugin
  - report reusability 335
  - tool comparison 313
  - use of 317, 319
- code analysis
  - quality metrics 311
  - report reusability 330
- code coverage analysis
  - definition 311
  - example 314–315
  - graph (Jenkins) 350
  - metrics 312
  - percentage 312–313
  - report (Jenkins) 349
  - tool comparison 313
- code integration process 339
- CodeNarc 320
- cohesion 134
- Cohn, Mike 159
- Collections API 435
- colon 138
- command-line interface. *See* CLI
- command-line options
  - caching options 426
  - common options 45–46, 423
  - daemon options 427
  - logging options 426
  - property options 425
- commit stage
  - build pipeline 385
  - code analysis phase 311
  - definition 37
- compilation
  - compiler daemon 297
  - dependency management 234
  - enhanced task 205, 308
  - manual process 4
  - project automation 5
  - See also* joint compilation
- complexity metrics 313
- conditional logic 15
- ConfigSlurper class 403
- configuration
  - access to 114
  - customization 112
  - dependency grouping method 60
  - Gradle wrapper 71
  - group 112
  - standard configurations, disadvantages of 112
  - subproject 142, 152
  - visibility 113
  - War plugin 65
- configuration block
  - addition of 86
  - creation of 144
  - dependencies 110, 120, 145
  - execution sequence 86
  - repositories 110, 121
- configuration phase
  - build lifecycle 87
  - ConfigSlurper class 403
  - custom task properties 215
  - input and output 89
- ConfigurationContainer class 111
- continuous delivery
  - Maven versioning scheme 387
  - on-the-fly instrumentation 315
  - stages 36
  - versioning scheme 388, 390
- continuous integration. *See* CI
- controller web component 62
- conventions
  - configuration 67
  - Java project 31, 296
  - plugin creation 214
- coordinates 385
- coupling 134
- cron job 7
- cross-project task dependency 150
- CRUD (create, read, update, destroy)
  - definition 50
  - execution 54
- CSS (Cascading Style Sheets) 293
- custom task
  - advantages 90
  - characteristics 209
  - creation of 90–91
  - parent class 206
  - testing 207–208
  - use of 91–92
  - See also* task type
- customization
  - custom task 90, 92, 201
  - dependency 262
  - distribution creation 363, 366
  - Eclipse project files 254, 258
  - functional testing configuration 186

- customization (*continued*)
  - idea plugin 260, 262
  - project 58, 154–155
  - property 58
  - Sonar configuration 332
  - Sublime Text plugin 264, 267
- cyclomatic complexity
  - Cobertura plugin report 317
  - definition 313

## D

- D option 45
- daemon mode
  - background process 46
  - expiration 46
  - startup performance 33
- daemon option 46
- DAG (directed acyclic graph)
  - definition 9
  - task dependency 100
- data persistence 52, 179
- database
  - control of 183
  - instance 180
- DDD (domain-driven design) 76
- def keyword 433
- DefaultTask class
  - built-in task types 93
  - custom task class 90, 202
  - extension 224
  - fields 88
- deliverable 385
- <dependencies> tag 17
- dependency 421
  - API 114
  - application of 83
  - configuration block 30, 110, 120, 145
  - configuration, grouped by 60
  - declaration 114
  - definition 11
  - dependency insight report 130
  - dependency report 130
  - dependency tree 117
  - dependsOn keyword 41
  - detection 60
  - functional testing 186
  - Ivy configuration 108
  - JavaScript library 286
  - list of 114
  - local cache 126
  - Maven configuration 108
  - POM generation 371
  - version 107–108
- dependency attribute 116

- dependency graph 328
- dependency insight report 130
- dependency management
  - Ant to Gradle migration 234
  - artifact retrieval 406
  - automation 107–110
  - definition 11
  - Gradle 23, 32
  - Ivy 12, 25
  - Java 106
  - JavaScript library 285, 287
  - Maven 17–18, 25
  - project 33
  - role in build tool 12
  - subproject 33
- dependency notation 116
- dependency tree 117
- <dependency> tag 17
- dependencyInsight task 422
- dependent task
  - build engine 11
  - build file 10
  - definition 9, 78
  - inferred dependency 94
- dependsOn
  - application of 83
  - directed acyclic graph 100
  - example 41
  - execution sequence 84
- deployable artifact
  - definition 385
  - version information 390
- deployment
  - manual process 5
  - scenario 89
  - SSH commands 407, 412
  - WAR file 200
- deployment pipeline
  - acceptance stage 416, 419
  - code analysis phase 311
  - project automation 37
  - stages 36
  - steps of 407
  - tasks within stages 37
- deployment tests 412
- description property 82
- directed acyclic graph. *See* DAG
- directed edge
  - definition 9
  - dependsOn 100
- directory tree
  - example 136
  - functional testing 187
  - modularization 135
- discovery tasks 421

- distribution plugin
  - creation of 363, 366
  - customized distribution 365
  - definition 93
  - description 363
  - use of 364
- documentation
  - annotations 91
  - configuration options 46
  - plugin functionality 72
- doFirst action 229
- doLast action 40, 229
- DOM (domain object model) 256, 258
- domain-driven design. *See* DDD
- domain-specific language. *See* DSL
- double-quoted String 434
- Downstream Buildview plugin (Jenkins) 356
- drone.io 351
- DSL (domain-specific language)
  - IDE features 268
  - project settings 250
  - SpringSource STS 270
  - use in Gradle 23
  - vocabulary in Gradle 30
- dynamic task definition
  - definition 41
  - example 40

## E

- Eclipse
  - definition 251
  - imported projects 258
  - plugins 250–251, 258
  - project files 251
- eclipse plugin
  - application of 252
  - configuration properties 254
  - purpose 252
- eclipse task 253
- Eclipse web tools platform (WTP)
  - purpose 252
  - subproject configuration 256
- eclipse-wtp plugin 252
- Emma
  - Sonar report 335
  - tool comparison 313
- enhanced task
  - advantages 90, 92
  - Ant to Gradle migration 235
  - custom task class 204–205
  - definition 90
  - reusability 92

- env property
  - purpose 404
  - smoke tests 414
- environment
  - build pipeline 401
  - configuration of 401–402
  - deliverable 385
  - deployment 386
  - env property 404
- event
  - listener 103
  - test event logging 175
- exclusion
  - attribute 119
  - transitive dependency 118
- execution
  - build engine 11
  - configuration phase 87
  - finalizer task 84
  - sequence 86
  - task dependency 84
  - up-to-date task 88
- execution phase 87
- execution state 9
- explicit closure parameter 437
- Extensible Markup Language. *See* XML
- extension model 214
- extension objects
  - customization 212
  - registration 215
- extensions() method 215
- external dependency
  - Ant to Gradle migration 234
  - configuration 59–60
- external library 197
- external module dependency 115
- external script reusability 199

## F

- facet
  - configuration 256
  - definition 256
  - merge hooks 257
- failure
  - assertion 164–165
  - exception stack trace 174
  - test report 166
- file dependency configuration 120
- finalizer task 84
- FindBugs plugin
  - Sonar report 330
  - tool comparison 320
  - use of 326–327



- flat directory repository
  - description 121
  - use of 125
- fork 341
- forked test process 175
- functional testing 184, 189
  - definition 158
  - example 415
- functTestCompile configuration 186
- functTestRuntime configuration 186

## G

- Gant 25
- GDK (Groovy Development Kit) 438
- Geb
  - dependency 186
  - description 185
  - system properties 188
- generateBuildFile task 423
- generateSettingsFile task 423
- Git plugin (Jenkins) 342
- Git setup 340, 342
- GitHub
  - account creation 340
  - cloud-based CI servers 351
  - host for Gradle 35
  - plugin 35
  - remote repository 341
  - repository fork 341
- GNU Privacy Guard signature 384
- Golo plugin 307
- Google Closure Compiler 287
- Google Hosted Libraries 285
- GPG signature 384
- Gradle
  - advantages 26, 28
  - Ant integration 224, 236
  - CSS plugin 293
  - development roadmap dashboard 210
  - domain-specific language 439
  - features 28, 36
  - forum 35
  - JavaScript plugin 292
  - profile selection 239
  - Sonar interaction 329
  - test results 160
- GRADLE\_OPTS environment variable 39
- Gradle plugin 268, 343
- Gradle Templates plugin 55
- Gradle wrapper
  - best practice 69
  - customization 71
  - Jenkins 346

- purpose 35
- set up 69–70
- use of 70–71
- GradleConnector class 280
- gradle.properties file
  - API key 195–196
  - security 377, 382
- Gradleware
  - consultation 35
  - professional services company 28
- graphical user interface 36
- Groovy
  - buildSrc directory 98
  - comparison to Java 430, 432
  - definition 429
  - distribution 300
  - Eclipse 251
  - features 432, 438
  - Gradle DSL 30
  - Groovy projects 298, 303
- Groovy Development Kit. *See* GDK
- Groovy plugin
  - application of 299
  - functional testing 187
  - Groovy base plugin 298
  - SpringSource STS 268
- Groovy project 301
- groovy.util.AntBuilder class 224
- group
  - configuration 112
  - dependency 60
- group attribute
  - definition 115
  - exclusion 119
- group property 82
- groupId attribute 385
- Growl 103
- Grunt 293–294, 296
- Gruntfile.js file 293

## H

- h option 45
- H2 (open source database engine) 329
- Hello world! 40
- help option 45
- help task 422
- hibernate status 200
- hook
  - domain object model 256
  - selection of 257
- hot deployment 407
- HTML (Hypertext Markup Language)
  - conversion from XML 321, 323
  - sample report 316

HTTPBuilder library 412  
Hudson 342

## I

- i option 45
- IDE (integrated development environment)
  - development within 4
  - features 268
  - project management 267, 277
  - remote debugging 171
- .idea directory 273
- idea plugin
  - customization 260, 262
  - definition 250
  - merge hooks 262
  - use of 259
- .iml file 259
- implementation class 212
- importation
  - Ant script 225, 228
  - Ant target migration 233
  - IntelliJ IDEA 263, 271, 273
  - NetBeans IDE 275
  - project 258
  - SpringSource STS 269
  - Sublime Text 266
- importBuild method
  - Ant script 225
  - Gradle build script 226
- incremental build
  - advantages 56
  - Ant target 230–231, 233
  - buildSrc directory 202
  - configuration phase 87
  - continuous integration 346
  - definition 33
  - input and output 89
  - lifecycle 87
  - task type 95
- INFO logging level
  - Ant task output 228
  - example 165
  - exception stack trace 174
- info option 45
- infrastructure
  - description 396
  - infrastructure as code 397
  - provisioning 396
  - Vagrant 397, 401
- initialization phase
  - build lifecycle 87
  - execution sequence 146
  - initialization script 103
  - project name 243
  - settings execution 140
  - task execution 146–147
- initialization script
  - build script 103
  - purpose 104
- in-memory persistence 52
- input
  - annotations 91
  - configuration phase 89
  - definition 88
  - incremental build 89
  - validation 91
- installation 38
- instrumentation
  - continuous delivery 315
  - definition 313
  - instrumentation methods 314–315
- integrated development environment. *See* IDE
- integration
  - Ant and Gradle 34, 41
  - manual process 4
  - Maven and Gradle 35
- integration testing 178, 184
  - definition 158
  - JaCoCo plugin report 317
  - pipeline jobs configuration (Jenkins) 357
  - report (Jenkins) 349
  - tooling API 277, 280
- IntelliJ IDEA
  - definition 258
  - Gradle 271
  - importation 263, 271, 273
  - JetGradle plugin 274
  - project files 259
- .ipr file 259
- Ivy 124
  - Ant 16
  - definition 12
  - dependency configuration 108
  - dependency management 32
  - repository types 121
  - transitive dependency 25
- ivy-publish plugin 368
- .iws file 259

## J

- JaCoCo plugin
  - report (Jenkins) 349
  - report reusability 334
  - tool comparison 313
  - use of 315, 317
- JAR file
  - Ant integration 230
  - creation 57

- JAR file (*continued*)
    - custom task class 202
    - dependency 59
    - Groovydocs 361
    - Maven repository 121
    - object plugin 221
    - program execution 58
    - version 106
  - Java
    - buildSrc directory 98
    - compilation with Groovy 301, 303
    - conventions 296
    - conversion to Scala 304, 306
    - dependency management techniques 106
    - program execution 57
  - Java Code Coverage plugin. *See* JaCoCo plugin
  - Java Development Kit. *See* JDK
  - Java development tools. *See* JDT
  - Java EE (enterprise edition) 61
  - JAVA\_OPTS environment variable 39
  - Java plugin
    - capabilities 296, 298
    - comparison to Maven scope 237
    - configuration 112
    - features 209
    - purpose 54
    - test configuration 161
    - test tasks 161
    - testing with 160
    - use of 54, 57, 261
  - Java project
    - convention properties 31
    - conversion to Groovy 299, 303
  - Java Secure Channel. *See* JSch
  - Java Server Page. *See* JSP
  - Java Virtual Machine. *See* JVM
  - JavaScript
    - dependency management 285, 287
    - library retrieval 285–287
    - optimization 289
    - popularity 283
    - purpose 283
  - JDepend plugin
    - overview 327–328
    - tool comparison 320
  - JDK (Java Development Kit)
    - development tools in 54
    - Groovy Development Kit 438
  - JDT (Java development tools) 254
  - Jenkins
    - build job definition 344, 346
    - build job execution 347–348
    - description 342
    - distribution and publication 392
    - history of previous builds 418
    - pipeline jobs configuration 356, 358
    - plugin installation 342, 344
    - plugins 353, 356
    - report 348, 350
    - startup 342
  - JetGradle plugin
    - features 274
    - use of 262
  - Jetty plugin
    - customization 68
    - definition 63
    - file location 287
    - functional testing 188
    - use of 67
  - joint compilation
    - bidirectional dependencies 302
    - definition 302
    - Java and Groovy 301, 303
    - Java and Scala 307
  - jQuery 284
  - JSch (Java Secure Channel) 408–409
  - JSHint
    - definition 290
    - example 290–291
    - Grunt JSHint plugin 295
  - jsOptimized property 289
  - JSP (Java Server Page) 61
  - JUnit
    - dependency 163
    - functional testing 185
    - Spock 167, 169
    - test results 160
    - TestNG integration 169
    - use of 162, 166
  - JVM (Java Virtual Machine)
    - installation 38
    - options 39
- 
- ## K
- 
- Kotlin plugin 307
- 
- ## L
- 
- layout (project)
    - customization of 58
    - flat layout 141
    - Groovy 301
    - hierarchical layout 141
    - Scala 306
    - test source files 160
  - legacy project 58

- library
  - dependency management 107–108
  - project dependency 146
  - transitive dependency 12
- libs-release-local repository 377
- libs-snapshot-local repository 377
- lifecycle
  - assemble task 360
  - build lifecycle 16, 87
  - build script 30–31
  - callback method 100
  - incremental build 87
- lifecycle event
  - build phase 99
  - listener 176
  - location 103
- lifecycle hook
  - example 101, 176
  - purpose 100
  - task 101
- listener
  - feedback 99
  - interface 101
  - lifecycle event 99, 176
  - lifecycle hook 101
  - registration 101
  - test listener 177
- Lists (Collections API) 435

## M

---

- main class
  - definition 50
  - implementation 53
- maintainability 90
- major version 388
- manifest file 398
- manual test stage 37
- Maps (Collections API) 435
- Maven
  - build lifecycle 16
  - cache, publishing to 369, 371
  - Central repository
    - addition of 122
    - availability 109
    - definition 237
    - dependency search 60
    - example 59
    - publication to 383
    - purpose 18, 359
  - comparison to Ant 21
  - comparison to Gradle 26, 237, 240
  - dependency configuration 108
  - dependency manager 17–18
  - directory layout 16
  - history of 25
  - integration with Gradle 35, 236
  - migration to Gradle 240, 243
  - plugin 368
  - profile 238
  - purpose 16
  - repositories 124
    - definition 237
    - object plugin consumption 221
    - publication to 366
    - publication to local 375, 377
    - publication to remote 377, 379
    - repository type 121
  - scope 237
  - shortcomings 19, 24
  - site generation 240
- Maven Polyglot 25
- Maven2Gradle task
  - description 423
  - use of 241–242
- MavenPublication class 372
- maven-publish plugin
  - description 368
  - example 375
- memory
  - data storage 52
  - settings 173
- merge hook
  - idea plugin 263
  - selection of 258
  - types of 256
- merging
  - Google Closure Compiler 287
  - Grunt 293
  - purpose 283
- metadata
  - cache features 127
  - content for artifacts 366
  - dependency 108, 134
  - disadvantages 109
  - .idea directory 273
  - Ivy repository 124
  - Maven repository 121
  - transitive dependency 119
- method coverage 313
- method parameter 438
- metrics
  - methods 314–315
  - publication to Sonar 333, 335
  - static code analysis 319
- migration
  - Ant 34, 233, 236
  - Gradle 23

- minification
  - example 288
  - Grunt 293
  - library retrieval 286
  - minifyJS task 293
  - purpose 283
- minor version 388
- Model-View-Controller. *See* MVC architecture
- modularization
  - cohesion 134
  - coupling 134
  - directory tree 136
  - functionality, based on 136
  - Spring 134
- module
  - definition 18
  - identification 135–136
  - organization 133
  - See also* project
- module attribute 119
- Mojo 25
- Mozilla Firefox 185
- multiline String 434
- multiplatform build
  - assembly of 137, 142
  - common requirements 142
  - compiler daemon 297
  - definition 133
  - initialization phase 87
  - script plugin 315
  - support for 78
- MVC (Model-View-Controller)
  - architecture 61

## N

---

- name
  - initialization phase 243
  - migration 234, 236
  - plugin 211, 217
- name attribute
  - definition 115
  - repository 376
- named parameter 436
- NetBeans IDE
  - Gradle 274
  - importation 275
  - installation 275
  - use of 276
- no-daemon option 46
- node
  - definition 9
  - dependsOn 100
- node package manager. *See* NPM
- no-rebuild option 147–148

- no-search-upward option 141
- notification
  - code integration process 340
  - configuration 346
  - continuous integration 339
  - push mode 345
- NPM (node package manager) 285, 293

## O

---

- object plugin
  - application of 210
  - characteristics 209
  - creation 213–214
  - definition 196
  - extension objects 212
  - implementation class 212
  - location 212
  - options 212
  - plugin descriptor 212
  - testing 217
  - web project 221
  - See also* standalone object plugin
- offline bytecode instrumentation 314
- offline option 45
- on-demand build 6
- on-the-fly instrumentation
  - continuous delivery 315
  - definition 314
  - JaCoCo plugin 315, 317
- optimization
  - CSS 293
  - JavaScript 289
- org.gradle.api.AntBuilder class 224
- outcome 243
- output
  - annotations 91
  - configuration phase 89
  - definition 88
  - directory 95
  - incremental build 89
- output directory 57

## P

---

- P option 45
- PaaS (platform as a service) 192
- Package Explorer 270
- package.json file 293
- parallel test execution
  - forked test process 175
  - support for 33
- Parameterized Trigger plugin (Jenkins) 353

- parent class
  - creation of 205
  - custom task class 206
- parentheses omission 434
- partial build
  - advantages 147
  - definition 33
- phases
  - definition 16
  - lifecycle event 99
  - See also* build lifecycle
- plain old Groovy object. *See* POGO
- plain old Java object. *See* POJO
- platform as a service. *See* PaaS
- plugin
  - application methods 210
  - build-announcements plugin 103
  - conventions 214
  - customization 192
  - descriptor 212
  - extension 34, 214, 217
  - GitHub 35
  - Gradle architecture 209
  - Gradle wiki page 210
  - migration 234, 236
  - name 217
  - product-specific plugin 379
  - project archetypes plugin 55
  - short identifier 217
  - source files location 55
  - standard plugins 209
  - task rule 98
  - third-party plugins 210
  - types of 196
- PMD plugin
  - Sonar report 330
  - tool comparison 320
  - use of 324, 326
- POGO (plain old Groovy object)
  - addition of 85
  - extension model 214
  - reusability 92
- POJO (plain old Java object)
  - conversion to Groovy 300
  - definition 50
  - use of 85
- polyglot programming 25
- POM (project object model)
  - creation of 219
  - definition 25
  - Maven profile 238
  - modification of 373, 375
- pom.withXml hook 373
- portability 6
- println command 40
- processes, list of running 46
- project
  - Ant 13
  - creation of 219
  - customization 154–155
  - definition 33, 76
  - repository 121
  - See also* module
- project archetypes plugin 55
- project automation
  - benefits 5–6, 36
  - continuous delivery 36, 38
  - deployment 405, 412
  - reasons for automation 4–5
  - types of 6, 8
  - Vagrant 399, 401
- project dependency 145
- project files
  - definition 249, 251
  - generation 267
  - generation of 253
- Project instance
  - configuration 111
  - creation of 87
  - multiproject build 139
  - ProjectBuilder class 208
  - testing 207
  - use of 77
- project method
  - example 144
  - project-specific build 143
- project object model. *See* POM
- project structure
  - example 56
  - flat layout 141
  - hierarchical layout 141
  - integration testing 181
  - multiproject build 138
- project variable 77
- ProjectBuilder class 208
- project-prop option 45
- projects task
  - description 422
  - example 138
- ProjectVersion class
  - Groovy syntax 432
  - Java syntax 431
  - versioning scheme 388
- properties file
  - build script, use with 85
  - version property 89
- properties task
  - description 422
  - project customization 58
  - purpose 46

- property
  - access to 82
  - addition of 79
  - configuration 217
  - definition 79
  - Eclipse project files 255
  - example 85
  - extra properties 79, 215
  - idea plugin 260
  - importation 229
  - inheritance 145
  - injection of 79
  - modification 58, 229
  - options 425
- providedCompile configuration 65
- public artifact management 382
- public binary repository 380
- publication
  - local Maven repository 375, 377
  - Maven Central repository 383
  - product-specific plugin 379
  - remote Maven repository 377, 379
- pull mode 339
- Puppet
  - project setup 398
  - purpose 397
- push mode
  - code integration process 339
  - notification 345

---

**Q**

---

- q option 46
- QA (quality assurance) 4
- quality metrics
  - build pipeline 352
  - code analysis 311
  - instrumentation methods 314–315
  - pipeline jobs configuration (Jenkins) 357
  - report (Jenkins) 350
- quality profile 330
- quiet option 40, 46

---

**R**

- R plugin 307
- RAD (rapid application development)
  - Jetty plugin 68
  - web application 63
- refactoring 205, 207
- release stage
  - definition 37
  - version 88
- release version (Maven) 387
- remote debugging 171

- report
  - Checkstyle plugin 323
  - clickable URL 166
  - Cobertura plugin report 317, 319
  - FindBugs plugin 326
  - JaCoCo plugin report 316
  - JDepend plugin 327
  - Jenkins 348, 350
  - PMD plugin 325
  - report configuration 326
  - reusability 333, 335
  - static code analysis 321
- repositories configuration block 375
- repository
  - availability 109
  - cache 130
  - configuration (Jenkins) 345
  - configuration block 30, 117
  - coordinates 385
  - definition 11
  - dependency 59
  - example 52
  - external dependency 32
  - fork 341
  - Gradle requirement 59
  - Maven repository, custom 124
  - Maven repository, local 123
  - Maven repository, remote 18, 377
  - name attribute 376
  - standalone object plugin 219
  - types of 121
- repository interface (Groovy) 302
- RepositoryHandler interface
  - API 121
  - Maven repository 122
- require.js 285
- return on investment. *See* ROI
- reusability
  - custom task 92
  - enhanced task 90
  - refactoring 205, 207
- Rhino
  - definition 290
  - example 290–291, 296
- ROI (return on investment)
  - factors 310
  - test automation pyramid 159
  - testing 190
- root project
  - build file 153
  - definition 138
  - Eclipse project files 255
  - IntelliJ IDEA 271, 276
  - project property configuration 261
  - SpringSource STS 270

- root project (*continued*)
  - Sublime Text plugin 265
  - task execution 146–147
- RUN@cloud. *See* CloudBees
- runtime behavior control 172, 174
- runtime dependency 65
- runtime environment
  - functional testing 185
  - web application 192

## S

---

- s option 46
- .scala file 305
- Scala plugin 303
- Scala projects 303, 307
- scheduled build 7
- SCP (Secure Copy)
  - description 407
  - file transfer 408–409
- script plugin
  - Checkstyle plugin 322
  - Cobertura plugin 318
  - creation of 197, 201
  - definition 196
  - FindBugs plugin 326
  - JaCoCo plugin 315
  - PMD plugin 325
- Secure Copy. *See* SCP
- security
  - gradle.properties file 382
  - repository 377
- Selenium 185
- sequence diagram 50
- server virtualization 396
- Servlet 61
- settings file
  - build file, name of 155
  - content of 254
  - definition 251
  - generation of 253
  - purpose 138
  - search for 140
  - Settings instance 139
  - subproject, addition of 138
- settings-file option 141
- settings.gradle file
  - example 138
  - initialization phase 140
  - purpose 139
- setupBuild task 422
- setupWrapper task 423
- short identifier 217
- signature (artifact) 381, 384
- single-page application 61
- single-quoted String 434
- smart exclusion 45
- smoke tests
  - definition 412
  - example 415
- snapshot
  - cache 131
  - up-to-date task 88
  - version 88, 387
- Snarl 103
- software component
  - definition 368
  - POM 371
  - publication 369, 371
- software functionality separation 315
- Sonar
  - installation 330
  - purpose 328
- Sonar Runner plugin
  - configuration 331
  - functionality 330
  - use of 332, 335
- Sonatype OSS 384
- SOURCE BUILD NUMBER environment
  - variable 389
- source code instrumentation 314
- source set integration testing 182
- Spock
  - functional testing 185
  - JUnit integration 169
  - TestNG integration 169
  - use of 167–168
- Spring 134
- Spring Tool Suite. *See* STS
- SSH deployment 407, 412
- SSH remote commands 409, 411
- stacktrace option 46
- standalone object plugin 218, 221
- statement coverage 313
- static code analysis
  - definition 319
  - tool comparison 320
- Strings
  - double-quoted String 434
  - GStrings 435
  - multiline String 434
  - single-quoted String 434
- structure. *See* project structure
- STS (Spring Tool Suite) 268–269
- Sublime Text plugin
  - project files 265
  - use of 264, 267
- .sublime-project files 264
- .sublime-workspace files 264



- subproject
  - build file 154
  - configuration 142, 152
  - dependency management 33
  - hierarchy 138
  - property 256
  - property inheritance 145
  - settings file 138
- subprojects configuration block
  - Checkstyle plugin 322
  - Cobertura script plugin 318
  - customization 261–262
  - Eclipse project files 255–256
  - JaCoCo script plugin 316
  - JDepend plugin 328
  - static code analysis 321
- subprojects method
  - application of 151
  - example 152
  - purpose 143
- system-prop option 45

## T

- TAR file creation 364, 366
- target (Ant)
  - definition 13
  - dependency 229
  - Gradle build script 226
  - Gradle task 227–228
  - incremental build 230
  - input and output 230
  - list of 228
  - migration to Gradle 233, 235–236
  - modification 229–230
- target directory 57
- task
  - Ant 13
  - Ant and Gradle 232
  - build script 34
  - build tools 9
  - characteristics 201, 208
  - definition 78
  - deployment pipeline 37
  - input and output 10, 88, 90
  - list of available tasks 42–43
  - migration 234, 236
  - target (Ant) 227
  - type, default 80
  - vocabulary in Gradle 30
- task (Ant)
  - Google Closure Compiler 287
  - Gradle integration 231
  - use of 232
- task action
  - addition of 82
  - declaration 81
  - definition 81
  - task configuration 104, 204
- task chaining
  - example 40
  - execution 84
  - functional testing 189
- task class creation 201, 208
- task configuration
  - definition 86
  - example 86
  - execution sequence 86
  - task action 104
- task dependency
  - all option 43
  - execution phase 87
  - graph example 93
  - inferred dependency 94
- task exclusion
  - command-line interface 44
  - smart exclusion 45
- task execution
  - command-line interface 43
  - cross-project task dependency 150
  - default sequence 149
  - root project 146–147
- task group
  - definition 43
  - task rule 97
  - values 83
- task name
  - abbreviation example 44
  - command-line interface 44
  - distribution task names 365
  - example 96
  - pattern 96
- task rule 95–96
- task type
  - built-in task types 93
  - definition 90
  - example 91
  - incremental build 95
  - use of 93
- tasks task
  - definition 42, 46
  - description 422
  - task group 43
  - task rule 97
- test automation
  - detection 162
  - functional testing 189
  - types of 158
- test automation pyramid 159

- test class
  - creation of 163, 179
  - integration testing 182
  - naming 168, 171, 179, 181
- test configurations 161
- test coverage analysis. *See* code coverage analysis
- test environment 401
- test listener implementation 177
- test logging control 174–175
- test report
  - build lifecycle 170
  - clickable URL 166
  - example 165
  - testing tools aggregation 169
- test results
  - Gradle 165
  - XML 160
- test source directory
  - creation of 162
  - functional testing 187
  - integration testing 181
- testCompile configuration
  - dependency, addition of 163
  - purpose 161
  - Spock 168
- testing
  - build process 33
  - custom task 207–208
  - execution 164–165
  - execution configuration 170, 178
  - Grunt 293
  - INFO logging level 165
  - JavaScript 284
  - manual process 4
  - object plugin 217
- testing tools
  - JUnit 162, 166
  - Spock 167–168
  - TestNG 167
- TestNG
  - functional testing 185
  - JUnit integration 169
  - Spock integration 169
  - use of 167
- testRuntime configuration 161
- ThoughtWorks 27–28
- toChar method 59
- Tomcat plugin 64
- tooling API
  - definition 250
  - integration testing 277, 280
  - purpose 277
- transitive attribute 119

- transitive dependency
  - definition 32
  - dependency management 12, 107
  - dependency tree 117
  - example 25
  - exclusion of 118
  - metadata 109, 119
  - POM generation 371
  - project dependency 146
  - transitive dependency graph 109
- transpiling 284
- Travis CI 351
- triggered build 7

## U

---

- u option 141
- UAT (user acceptance test) 401
- unit of work. *See* task
- unit testing 162, 170
  - definition 158
  - integration testing 180
  - JaCoCo plugin report 317
  - pipeline jobs configuration (Jenkins) 357
  - report (Jenkins) 349
  - Sonar dashboard 333
- untyped parameter 437
- up-to-date task
  - definition 88
  - example 231
  - UP-TO-DATE message 56
- user acceptance test. *See* UAT

## V

---

- Vagrant
  - execution 399, 401
  - installation 398
  - project setup 398
  - purpose 397, 401
- VCS (Version Control System)
  - code integration process 339
  - continuous integration 338–340
  - on-demand build 6
  - project files 250
  - snapshot version 387
  - triggered build 7
- version
  - build comparison plugin 245
  - class 80
  - conflict resolution 118, 128
  - dependency 107
  - dynamic version 119–120
  - enforcement 129
  - exclusion of 118

- version (*continued*)
  - JAR file 106
  - release 80, 88
  - snapshot 88
  - specification 58
  - troubleshooting 129
- version attribute
  - coordinates 385
  - definition 115
  - dynamic version 119
- version control
  - Ant 24
  - automation 81
  - Gradle wrapper 70
  - manual techniques 106
- Version Control System. *See* VCS
- version property 360
- versioning scheme
  - continuous delivery 388, 390
  - Maven 387
- virtual server
  - cloud provisioning 396
  - creation of 397, 401
- VirtualBox installation 398
- web application
  - build 65
  - conversion to 61, 63
  - definition 61
  - directory, default 64
  - functional testing 185
  - Gradle support for 63
  - port, default 67
  - runtime environment 192
- web application archive file. *See* WAR file
- web container
  - definition 61
  - Jetty plugin 67
- web framework 61
- whenMerged hook
  - domain object model 256
  - example 257
- withXml hook
  - domain object model 256
  - facet 256
  - idea plugin 262
  - POM modification 373
  - use of 263
- workspace reusability (Jenkins) 354
- wrapper. *See* Gradle wrapper
- WTP. *See* Eclipse web tools platform

---

## W

- WAR (web application archive) file
  - application provisioning 195
  - deployment of 200
  - example 65
  - name 66
  - publication of 391
  - purpose 61
  - retrieval 406
  - War plugin 65
- War plugin
  - comparison to Maven scope 237
  - customization 66
  - definition 63
  - use of 64, 66

---

## X

- XML (Extensible Markup Language)
  - characteristics 15
  - comparison to Gradle 24
  - conversion to HTML 321
  - dependency configuration 108
  - project files 249
  - test results 160

---

## Z

- ZIP file creation 364–365

# 实战Gradle

Gradle是一个通用的自动化构建工具。其继承了先驱者Ant和Maven所建立的使用模式，并且允许构建具有表达性、可维护性和容易理解。使用灵活的基于Groovy的DSL，Gradle提供了声明式的可扩展语言元素，让你能够随心所欲地建模项目需求。

《实战Gradle》是关于使用Gradle实现端到端的项目自动化的综合指南。从基础知识开始介绍，具有实践性，易于阅读，讨论了在实际项目中如何建立高效的完整的构建过程。在这个过程中，涵盖了一些高级话题，如测试、持续集成和代码质量监测。你还会体验到像设置目标环境和部署软件这样的具体任务。

## 本书内容包括

- 关于Gradle的综合指南
- 真实的实践案例
- 从Ant和Maven过渡
- 深入讲解插件开发
- 使用Gradle实现持续交付

本书假设读者具有Java基础背景，但是不要求具备Groovy知识。

Benjamin Muschko是Gradleware工程师团队的成员，同时也是很多流行的Gradle插件的作者。

如需下载免费的PDF、ePub和Kindle形式的电子书，本书的拥有者可以访问manning.com/GradleInAction。



策划编辑：张春雨  
责任编辑：葛娜  
封面设计：李玲

“一本权威指南。”

— 摘自Hans Dockter之序，Gradle和Gradleware的创建者

“自动化构建的新方式，你永远不会怀念旧的。”

— Nacho Ormeno, startupXplore

“多语言程序员必读！”

— Rob Bugh, ReachForce

“最好的Gradle参考！充满了真实案例。”

— Wellington R. Pinheiro 巴西沃尔玛电商

“帮助所有开发者使用Gradle所缺少的一本书。”

— Samuel Brown, Blackboard, Inc.

上架建议：软件工程/程序设计

ISBN 978-7-121-26925-7



定价：89.00元

[General Information]

书名=实战Gradle

作者=(美) 马斯可著

页数=463

SS号=13843986

DX号=

出版日期=2015.09

出版社=北京电子工业出版社