Gradle

进阶

Tim Berglund 著　樊高成 译

O'REILLY®

本书深入而细腻地探讨了Gradle的各个方面，这些内容对你高效的构建至关重要。即便是Gradle老手也能从中获益良多。

Gradleware首席工程师 Luke Daley

由O'Reilly Media公司发行, 1005 Gravenstein Highway North, Sebastopol, CA 95472。

编辑: Mike Loukides & Meghan Blanchette
产品编辑: Kara Ebrahim
校对员: Kara Ebrahim
封面设计: Randy Comer
内页设计: David Futato
插画: Rebecca Demarest

首次发行版修订历史
2013-07-15： 首次发行

参见： http://oreilly.com/catalog/errata.csp?isbn=9781449304676 获取更多发行信息

ISBN: 978-1-449-30467-6 [LSI]

Gradle进阶 — 定制下一代构建系统

如果你已经通过阅读作者的上一本书《使用Gradle构建和测试》熟悉了Gradle的基础知识的话，那么这本书将为你提供一些更高级的Gradle技能以帮助你精通这款自动化编译工具。通过简单清晰的阐释和大量的‘拿来即用’的示例代码，你将逐步深入到Gradle的4个方面：文件操作，自定义Gradle插件，hook编译过程和依赖管理。

本书帮助你学习如何使用Gradle丰富的API以及如何基于Groovy的‘特定领域语言’来定制你自己的编译系统， 最终达到适应你特定的产品开发的目的。通过使用从书中学到的技能，你将有能力写出你自己特定领域的构建脚本来支持你们团队创造的代码。

本书覆盖内容

学习Gradle文件操作API， 包括文件复制，模式匹配，文件过滤和FileCollection（文件集合）接口。

掌握编译和打包一个自定义一个Gradle插件的过程

了解Gradle自身如何管理依赖和如何自定义依赖管理。

研究Gradle几款核心的插件，并学习若干关键示例（从Gradle社区收集而来）

从oreilly.com购买这本书的电子版可以享受终身免费升级。我们的电子书针对一些常见的格式做了优化，包括PDF, EPUB, Mobi和DAISY，这些都不会受到数字版权限制。

本书使用到的约定

字体约定：

斜体

指示新的术语，URL，邮件地址，文件名和文件扩展名。

等宽

用于程序清单，也用于在段落指示一些程序元素比如变量，函数名，数据库，数据类型，环境变量，代码语句和关键字。

等宽加粗

注释或者其他需要被用户手动键入的文本

等宽斜体

指示需要被用户提供的值替代或者要由上下文决定的文字。

【提示】 建议或者一般的注解。

【警告】 需要额外注意的地方。

Safari® 在线图书

Safari在线图书是一个应需而生的数字图书馆，其诣在发布技术和商业方面的专业内容，这些内容都来自于世界级的作者，包括图书和视频两种形式。

Safari在线图书被很多职业人作为首选资源用来检索、解决问题、学习和培训认证，这些人包括技术专家，软件开发者，网页设计师和商业与创新方面的专家等。

Safari在线图书为组织，政府机构和个人提供了多样的产品组合和定价策略。订阅者使用一个具备完整检索功能的数据库就可以访问成千上万的图书、培训视频和预发布手稿，提供这样数据库的公司包括O'Reilly Media，Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology等等。访问我们的官网可以获取更详细的Safari在线图书的资料。

联系我们
O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938(in the United States or Canada)
707-829-0515 (international or local)
707-829-0104(fax)

我们为本书设立了一个网页，里面包含勘误表，示例和其他额外信息。你可以通过这个地址访问：http://oreil.ly/gradle-btb。

如需询问关于本书技术方面的问题请发送邮件到：bookques tions@oreilly.com.

访问我们的网站 http://www.oreilly.com 获取更多关于书籍、课程、会议和新闻等方面的信息。

Facebook主页： http://facebook.com/oreilly
在Twitter上关注我们：http://twitter.com/oreillymedia
YouTube上我们的视频：http://www.youtube.com/oreillymedia

鸣谢

我想感谢为本书品质做了出色贡献的我的团队优秀的技术编辑们，他们包括：Jason Porter, Spencer Allain, Darin Pope和Rod Hilton. 尤其感谢Luke Daley， 他不仅参与了本书的编辑，还因为最初的版本在依赖管理一章中没有抓住主题的精髓而对其做了重要的改写。Luke同时像一个热情的助手一样，不止一次通过Skype帮我解答关于Gradle和其他技术方面的疑问。他是一个难得的朋友，我希望未来可以和他有更多的合作。

另外感谢我的朋友Mattew McCullough在早期对编译hook一章所做的贡献。Mattew在构建工具领域有很长的历史了，他对于构建元编程方面的见解对这一章的正确性提供了不小的帮助。

当然还要感谢饱受等待之苦的Hans Docktor， 为这本书他比预期多等了大约一年。和他一起工作同样令我很开心，同时他也成了我非常好的朋友。

对于我的编辑Meghan Blanchette，我有义务要感谢她，当然这个义务是我自愿接受的。如果Meghan跟我合作另一本书我想她会发明一个自动发邮件的东西经常检查我是否跟上进度。到那时我依然愿意听到她的提醒。

我倾向于很早起来写书，这样我的妻子Kari就不会看到我一直专注于这本书上。这的确很有用，但是她也体验了他的丈夫很多天晚上9点入睡以便第二天可以很早起来写书的生活。印上她的名字和对她的感谢是我所能给予的最少的东西了。

目录

---

3. hook构建过程

- Gradle生命周期回顾
- 了解构建图(Build Graph)
- 了解工程评估(Project Evaluation)
  - 全局工程加载和hook
  - 构建结束
- Rules
  - 创建Rules
  - 处理命令式Rules代码
  - 通用Rules
- 总结

4. 依赖管理

- 依赖管理是什么
- 概念
  - Configuration
  - 模块依赖
  - 动态版本
  - 文件依赖
  - 工程依赖
  - 内部依赖
- 仓库： 依赖解析
  - Maven仓库
  - 可变模块
  - Ivy
  - 仓库鉴权
  - 静态依赖
- 构建脚本依赖
- 依赖缓存
- 配置解析策略
  - 版本冲突失败处理
  - 强制使用某个版本
  - 缓存过期
- 总结

后记

前言

欢迎来到O'Reilly Gradle系列第二部 —- 《Gradle进阶》。这本书继承自前一本《使用Gradle构建和测试》，会使你更深入的掌握Gradle编程模式。我们提供了相关的技巧和语法使得Gradle不再仅仅是一个灵活的脚本， 更是一个可以使你建立稳定构建系统的工具。 Gradle和一般构建工具的区别在于：它是你自己专有的工具， 而不是一个不懂你特定需求的通用构建工具。

由于第一本书已经介绍过Gradle的基本知识，本书我们会进一步深入探究它的特性。我们会覆盖4个独立的方面：文件操作，自定义插件，hook编译过程以及依赖管理。我们假设你已经熟悉了Gradle的基本用法，基于你时间和兴趣的考虑，我们不会再讲比较简单的原理。 如果你还是一个Gradle新手，建议你先读一下《使用Gradle构建和测试》。

Gradle的API很丰富，DSL本身提供的能力在你的使用领域里已经非常充足了。想要拥有满足你特定产品需求的构建系统的路线也很清晰。那么，就让我们开始学习之旅吧。

第一章

文件操作

如果说构建过程最基本的操作是编译代码，那么列在第二位的就是文件拷贝了。真实世界的构建经常包含将文件从一个地方拷贝到另一个地方、递归遍历文件目录树、匹配文件名和对文件内容做字符串处理等操作。Gradle暴露了一些方法、任务类型和接口使得文件操作既灵活又方便。这些汇聚起来形成了Gradle的文件API。

为了探究文件API，我们先从一个使用Copy任务的实际例子出发。我们将从那里开始，然后探究到Project对象内部文件操作相关的方法，这些方法在Gradle构建脚本的任何地方都是可用的。在这个过程中我们还会学到FileCollection接口。最终我们会看到文件API是怎么被一般的Gradle插件使用的 - 这给了我们一个更广阔而深入的视角看待JAR文件和SourceSets，而不再认为它们理所当然是这样的。

Copy任务

Copy任务由Gradle内核提供。在执行阶段一个copy任务将一个或多个源的文件拷贝到目标目录中。你告诉copy任务从哪里获取文件，放在哪里，怎么通过一个配置块过滤这些文件。一个最简单的copy任务配置看起来像例1-1这样：

例1-1， 一个简单的copy任务配置

```
task copyPoems(type: Copy) {
  from 'text-files'
  into 'build/poems'
}
```

这个例子假设有一个名为text-files的目录包含了一些以诗人名字命名的文件。执行gradle copyPoems运行这个脚本就会将这些文件拷贝到build/poems目录，为后续处理做好准备。

默认情况下所有from目录下的文件都会包含在copy操作中，你可以通过模式匹配来指定包含或排除哪些文件。包含和排除规则使用的是Ant风格的写法，** 代表递归匹配所有的子目录名，*匹配文件名的任一部分。Include默认是外排的，也就是说，它会认为所有没有声明在include里的文件都会被排除掉，类似，exclude默认是内包的，它认为所有没有声明在exclude的文件都会被包含进来。

当exclude和include一起使用时，gradle采取exlude优先原则。首先收集所有声明为include的文件然后从中移除掉声明在exlude中的文件。你的include exclude逻辑叠加后呈现的结果是：一个包含范围更广的include，然后再施加一个不及原来包含性强的exlude，从而得到最终产物。

如果通过一个单独的include或者exlude不能完全表达你想要的规则，你可以在一个Copy任务配置中多次调用include或者exclude（例1-2)。对于单个方法调用你也可以传入一个逗号分割的文件列表（例1-3）。

例1-2. 拷贝除了名子包含Henley以外的全部诗人

```
task copyPoems(type: Copy) {
  from 'text-files'
  into 'build/poems'
  exclude '**/*henley*'
}
```

例1-3.只拷贝Shakespeare和Shelley

```
task copyPoems(type: Copy) {
  from 'text-files'
  into 'build/poems'
  include '**/sh*.txt'
}
```

一般的构建操作都会涉及将多个来源的文件收集拷贝到同一个地方。想要做到这一点，可以像例1-4那样多次调用from。同时每一次from调用都可以包含自己的include和exclude规则。

例1-4. 处理多个文件来源情况

```
task complexCopy(type: Copy) {
  from('src/main/templates') {
    include '**/*.gtpl'
  }
from('i18n')
from('config') {
    exclude 'Development*.groovy'
  }
  into 'build/resources'
}
```

转换目录结构

例1-4的输出是将所有来源文件都平铺式的复制到一个build/resources目录。当然你可能并不想平铺所有源目录；你可能想保留某些源目录的树结构甚至将其映射为一个新的树结构。我们可以简单的通过在from配置闭包中调用into来做到这一点。比如例1-5。

例1-5. 将源目录结构映射为新的目录结构

```
task complexCopy(type: Copy) {
  from('src/main/templates') {
    include '**/*.gtpl'
    into 'templates'
  }
  from('i18n')
  from('config') {
    exclude 'Development*.groovy'
    into 'config'
  }
  into 'build/resources'
}
```

注意外部的into调用必须有——否则构建脚本不能运行，同时注意嵌套的into调用是相对于外部配置路径的。
如果要拷贝的文件数量太多或者体积太大将会产生很大的构建时间成本。Gradle的增量构建功能有助于减轻这个问题带来的影响。Gradle会自动在第一次执行全量构建，这会很耗时，但是会在后续构建中通过减少不必要的拷贝操作来减少构建时间。

拷贝过程中重命名文件

如果你的构建涉及到文件拷贝，Gradle支持让你在拷贝过程中对文件重命名。这些文件名可能要被附上某些部署环境的标识或者要符合某种特定环境的命名标准，再或者是要以某种产品配置信息为依据。无论哪种原因，Gradle统一提供了两种灵活的方式来完成这件事情：常规表达式和Groovy闭包。

使用常规表达式重命名文件我们只需要提供一个源表达式和目标文件名就可以了。源表达式会通过文件名匹配归组这些需要被拷贝的文件，然后将其映射为以$1/$2格式表达的目标文件名。例如，拷贝一些具有某种配置标识的文件到一个暂存目录，像例1-6这样：
例1-6。 使用常规表达式重命名文件

```
task rename(type: Copy) {
  from 'source'
  into 'dest'
  rename(/file-template-(\d+)/, 'production-file-$1.txt')
}
```

如果要编程式的重命名文件，我们可以给rename方法传入一个闭包（例1-7）。闭包接收一个代表原始文件名的参数。闭包的返回值就是修改后的文件名。
例1-7。编程式地重命名文件

```
task rename(type: Copy) {
  from 'source'
  into 'dest'
  rename { fileName ->
    "production-file${(fileName - 'file-template')}"
} }
```

【提示】在Groovy中对字符串做减法操作会从第一个字符串中移除掉第二个字符串的首次匹配部分。因此“one two one four ” - “one” 会得到“two one four”。这是一个常见的快捷处理字符串的方式。

过滤和转换文件

通常在一次构建中不止包含文件的拷贝和重命名，还会包含对文件内容做转换这样的操作。Gradle提供了三个方法来处理这样的任务：*expand()*、*filter()*和*eachFile()*。我们会依次讨论它们。

关键字替换

构建过程中，一种常见的情况是将一组配置文件拷贝到暂存区并且替换掉被拷贝文件的某些内容。一个特定的配置文件可能同时包含很多不随环境变化而变化的参数和一小部分需要随之而变化的参数。如果能在这个配置文件拷贝的过程中同时替换掉那些需要变化的参数那就很方便了。Gradle提供的*expand()*方法正好能满足这个需求。

expand()方法利用了Groovy的SimpleTemplateEngine类。SimpleTemplateEngine对文本文件增加了关键字替换语法，类似于Groovy里的字符串交互语法。任何包含在花括号中并以$符开头的字符串都被当做替换对象。当在一个copy任务中声明了关键字替换时，你需要给*expand()*方法传入一个map。 这个map中的key会被用来匹配被拷贝文件中的花括号表达式， 然后将其替换为map中对应的value值。

例1-8. 带关键字替换的文件拷贝。

```
versionId = '1.6'
task copyProductionConfig(type: Copy) {
  from 'source'
  include 'config.properties'
  into 'build/war/WEB-INF/config'
  expand([
  databaseHostname: 'db.company.com', version: versionId,
  buildNumber: (int)(Math.random() * 1000), date: new Date()
]) }
```

【提示】**SimpleTemplateEngine**还有其他的一些功能，具体可以参考: 在线文档

注意，传给expand()方法的是一个Groovy风格的map — 键和值之间用冒号分割，多个键值对之间用逗号分割，然后整体包含在方括号中。

在本例中，因为这个task专门用来为production这个产品类别准备配置文件，所以map的内容可以是写死的。真实的构建中这种方法可能就显得不够灵活了，有时可能需要从其他环境相关的配置文件中获取map的值。实际上，map的内容是可以来自于任何地方的。gradle的build文件本身就是可执行的Groovy代码，这一事实从一开始就给了你无限的灵活性。

这个时候来看一下原始的配置文件会很有帮助，我们可以清楚的看到哪里发生了替换操作。

这是“拷贝替换”操作之前的源文件

```
#
    # Application configuration file
    #
    hostname: ${databaseHostname}
    appVersion: ${version}
    locale: en_us
    initialConnections: 10
    transferThrottle: 5400
    queueTimeout: 30000
    buildNumber: ${buildNumber}
    buildDate: ${date.format("yyyyMMdd'T'HHmmssZ")}
```

这是“拷贝替换”操作之后的目标文件

```
 #
    # Application configuration file
    #
    hostname: db.company.com
    appVersion: 1.6
    locale: en_us
    initialConnections: 10
    transferThrottle: 5400
    queueTimeout: 30000
    buildNumber: 77
    buildDate 20120105T162959-0700
```

逐行过滤

expand()方法对于一般的字符串替换操作堪称完美，甚至一些轻量的内容修改也能应付。 但是有些转换操作需要在拷贝过程中单独处理文件的每一行， 这时filter()方法就派上用场了。

filter()方法有两种形式：一种是接受一个闭包，另一种是接受一个类。两种我们都会讲，先从简单的闭包形式开始吧。

当你传递一个闭包给filter()，这个闭包会被文件的每一行依次调用。闭包里可以执行任何你想执行的操作，然后返回你处理以后的新的行。例如，用MarkdownJ将Markdown文本转换为HTML文本，参考例1-9。

例1-9. 在拷贝文件的同时使用*filter()*加闭包的形式对其进行转换

```
import com.petebevin.markdown.MarkdownProcessor
buildscript {
  repositories {
    mavenRepo url: 'http://scala-tools.org/repo-releases'
  }
dependencies {
    classpath 'org.markdownj:markdownj:0.3.0-1.0.2b4'
} }
task markdown(type: Copy) {
def markdownProcessor = new MarkdownProcessor() into 'build/poems'
from 'source'
include 'todo.md'
rename { it - '.md' + '.html'}
filter { line ->
    markdownProcessor.markdown(line)
  }
}
```

被处理的源文件的内容是一首带有注释的短诗:

```
# A poem by William Carlos Williams
    so much depends
    upon
    # He wrote free verse
    a red wheel
    barrow
    # In the imageist tradition
    glazed with rain
    water
    # And liked chickens
    beside the white
    chickens
```

经过拷贝和过滤后文件注释被替换成了空行:

```
so much depends
    upon
    a red wheel
    barrow
    glazed with rain
    water
    beside the white
    chickens
```

Gradle在文件过滤上给了我们很大的灵活性，但是有一点是肯定的，它希望你始终将过滤逻辑剥离到task定义之外。 最好的办法是将过滤逻辑封装在对应的类之中，而不是将其散布在构建脚本的各个地方，这样做的好处是以后可以将这些构建代码移到独立的源文件中，同时方便了对其单独做自动化测试。让我们往这个方向迈出第一步吧。

我们可以将filter()的参数由闭包换为一个类。这个类必须是java.io.FilterReader的实例。Ant API提供了大量预写好的FilterReader实现，同时鼓励用户去复用这些类。例1-9的代码可以重写为例1-10的样子。

例1-10。使用filter()配合一个Ant Filter类对文件进行转换

```
import org.apache.tools.ant.filters.*
import com.petebevin.markdown.MarkdownProcessor

buildscript {
  repositories {
    mavenRepo url: 'http://scala-tools.org/repo-releases'
  }
  dependencies {
    classpath 'org.markdownj:markdownj:0.3.0-1.0.2b4'
} }

class MarkdownFilter extends FilterReader { MarkdownFilter(Reader input) {
    super(new StringReader(new MarkdownProcessor().markdown(input.text))) }
}

task copyPoem(type: Copy) {
  into 'build/poems'
  from 'source'
  include 'todo.md'
  rename { it - ~/\.md$/ + '.html'}
  filter MarkdownFilter
}
```

最终，MarkdownFilter类就可以从构建代码中移走放到一个自定义插件中。这是一个很重要的话题，会有专门一章来讲。

逐文件过滤

expand()和filter()方法会为所有文件应用同一套转换逻辑，但是有时我们想对每个文件做不同的处理。对于这种情况，我们需要用到eachFile()方法。

eachFile()接受一个闭包作为参数，这个闭包会在每一个文件被拷贝时执行。闭包带有一个FileCopyDetails接口实例的参数。这个参数可以让你对每一个文件单独做很多处理，而且这些处理可以在一次调用内完成。FileCopyDetails接口暴露了很多方法，包括重命名文件、改变目标路径、排除文件、创建自己的副本和以java.io.File地方式操作文件等。这些事情你都可以通过前面讲的Gradle DSL做到，但是在某些情况你可能喜欢回退到直接操作的方式。例如，你可能有一个自定义的部署程序，在上面对一个包含很多文件的目录执行拷贝操作，同时计算每一个文件内容的SHA1哈希值，最后将结果存储起来。你可以像1-11这样实现这个功能。

例1-11。使用*eachFile()*计算每一个文件的hash值

```
import java.security.MessageDigest;

task copyAndHash(type: Copy) {
  MessageDigest sha1 = MessageDigest.getInstance("SHA-1");
  into 'build/deploy'
  from 'source'
  eachFile { fileCopyDetails ->
      sha1.digest(fileCopyDetails.file.bytes)
  }

  doLast {
     Formatter hexHash = new Formatter() sha1.digest().each { b ->
        hexHash.format('%02x', b) } println hexHash
     }
}
```

文件相关方法

Gradle中有一些文件操作方法在任何地方都可以调用，这些就是Project内部的方法。这些方法为很多常见操作提供了方便，比如将一个文件路径转换为java.io.File对象、收集文件和递归遍历目录树并将其转换为文件集合等。我们下面会逐一学习这些方法。

file()

调用file()方法会生成一个java.io.File对象，并将相对路径转换为绝对路径。 file()方法接受一个参数，这个参数可以是字符串、文件、URL或者是一个闭包，如果是闭包的话返回值要是前面三种类型之一。

file()方法在task含有一个File类型的参数时会很有用。例如，Java plug-in 提供了一个叫做jar的task，它的作用是将默认sourceSet中的类和资源编译出一个JAR文件。这个task会将生成的JAR文件放到build目录下的默认位置， 但是有时我们想改变这个默认路径。Jar task有一个名为destinationDir的属性可以用来覆盖掉默认的存储路径，有人可能会像例1-12这样做：

例1-12。将一个路径赋值给destinationDir

```
jar {
  destinationDir = 'build/jar'
}
```

然而构建会失败，因为destinationDir是File类型，不是一个String。 传统的解决方案可能会像例1-13这样。

例1-13. 将一个路径赋值给destinationDir

```
jar {
destinationDir = new File('build/jar')
}
```

这样构建可以通过，但是无论如何都不能得到预期的结果。File的构造函数会使用你提供的参数产生一个绝对路径，但是这个路径是相对于JVM启动路径的。你直接调用Gradle，或者通过wrapper调用，或者通过IDE调用，亦或通过持续集成服务器调用都可能产生不同的结果。正确的方式是使用file()方法，像例1-14这样。

例1-14. 使用file()方法设置destinationDir属性

```
jar {
  destinationDir = file('build/jar')
}
```

这样做的结果是destinationDir属性值被设为了该工程根目录下的build/jar。这是预期行为，不会受到Gradle被如何调用的影响。

如果你已经有一个File对象了，file()方法会尝试用同样的方式将其转换为相对于工程的绝对路径。使用new File('build/jar')方式创建的File对象没有默认的父路径，因此file(new File('build/jar'))将会强制将其父路径设为工程的根路径。这个例子展示了一个笨拙的对象构造方式 — 正常情况下大家都想省略掉构造File对象的步骤 — 即便这样file()对于File对象的操作还是按预期正确执行了。如果你因为某种原因手头已经有一个现成的File对象时你可能就会遇到这种情况。

file()同样可以作用于protocol或者scheme是file://的java.net.URL和java.net.URI对象。文件URL平时用到很少，但经常会出现在通过ClassLoader来加载某些资源的时候。如果你在编译中恰巧遇到了文件URL，你可以轻易的使用file()方法将其转换为相对于工程的路径。

files()

files()基于传入的参数返回一个文件集合。类似于file()产生相对于工程路径的File对象，不同之处在于它是针对一批文件。它接受若干种类型的参数，见表1-1。

表1-1。files()可以接受的参数

参数类型	方法行为
String	创建一个包含单个File对象的集合，对象的生成方式同*file()*
java.io.File	创建一个包含单个File对象的集合，对象的生成方式同*file()*
java.net.URL 或 java.net.URI	根据指示的文件创建一个包含File对象的集合，只支持*file://*形式的url， 同*file()*

参数类型	方法行为

Collection, Iterable, 或 Array	根据传入参数集合创建一个对应的转换后的元素集合。集合元素通过递归遍历参数对象获得。参数集合中允许包含*files()*支持的其他任何类型。
	// Groovy风格的ArrayList
	`files(['src/main/groovy', 'src/test/groovy'])`
	// listFiles()返回一个包含File的集合对象
	`files(file('src/changelog/resources').listFiles())`

Task	根据task的输出产生一个文件集合。task是否输出文件集要根据其功能而定。Gradle核心插件提供的task一般会隐式的输出一个文件集。
	例如(使用Java plug-in) :
	// 传入Task对象
	`files(compileJava)`

Task Outputs	和直接传入task名效果相同，但是允许直接使用 TaskOutpus对象作为参数。
	例如(使用Java plugin) :
	// 传入Task Outputs
	`files(compileJava.outputs)`

正如你所看到的，files()方法简直就是多能手，可以将各种来源转换为文件集合，既能处理文件路径，又能处理文件对象，还能处理URL，甚至能处理Gradle task， 即便是以上几种组成的集合它也能处理。

初学Gradle的开发者可能期望files()会返回一个实现了List接口的集合对象。但实际上它返回了一个FileCollection对象，这个类是Gradle文件编程中的基础接口，我们会在“FileCollection接口”一章中再来认识这个类。

fileTree()

file()方法将路径转换为文件，files()方法基于file()产生一个文件集合，但如果你想收集一个目录树中的所有文件呢，fileTree()方法正是来解决这个问题的。

有三种调用fileTree()的方式，它们都很大程度的借鉴了copy task。比如它们的共同点：都必须从根目录开始，同时可以添加include和exclude规则。

最简单的用法是为fileTree()传入一个目录，然后它会自动递归遍历该目录以及所有的子目录，并将发现的文件加入到集合中。例如你想收集你Java项目工程下的所有源码文件，使用fileTree(‘src/main/java)就可以了。

有时你可能想在收集文件时显式的包含或者排除某些文件，例如你可能知道源码目录中存在一些以“~”为后缀的备份文件，然后想排除掉这些文件。亦或你知道有些本该属于resources目录的xml文件混入了源码目录中，然后你想把这些xml文件收集出来。例1-15向你展示了如何处理这两种情况。

例1-15. 使用fileTree()并配以include和exluce规则

```
def noBackups = fileTree('src/main/java') {
     exclude '**/*~'
}

def xmlFilesOnly = fileTree('src/main/java') {
     include '**/*.xml'
}
```

你还可以以map的形式添加include和exclude规则，像例1-16这样。这种写法很常见，也很重要。

```
def noBackups = fileTree(dir: 'src/main/java', excludes: ['**/*~'])

def xmlFilesOnly = fileTree(dir: 'src/main/java', includes: ['**/*.xml'])
```

FileCollection接口

如果你运行过讲files()方法那一章的例子，你会发现files()和fileTree()的返回值都没有友好的toString()实现（例1-17）。如果它们的返回值是ArrayList的话我们就可以将其内容打印出来，但实际不是。但这也隐隐预示着Gradle会提供某种比普通ArrayList更有用的功能。

例1-17。FileCollection的默认toString()实现

```
task copyPoems(type: Copy) {
  from 'text-files'
  into 'build/poems'
}
println "NOT HELPFUL:"
println files(copyPoems)
```

下面是这段代码的输出，我们看得到默认toString的行为。

```
 $ gradle -b file-collection-not-helpful.gradle
   NOT HELPFUL:
   file collection
```

例1-18。更有用的查看FileCollection的方式

```
task copyPoems(type: Copy) {
  from 'text-files'
  into 'build/poems'
}
println "HELPFUL:"
println files(copyPoems).files
```

下面是上一段代码的输出，可以看到FileCollection的内容：

```
$ gradle -b file-collection-helpful.gradle
  HELPFUL:
  [~/oreilly-gradle-book-examples/file-operations-lab/build/poems]
```

files()返回的不是一个List对象，而是一个FileCollection[注2]。FileCollection出现在Gradle的很多地方：SourceSet、task输入输出、Java classpath、dependency configurations等。了解这个类的用法将大大提升我们的文件编程的技能，无论是从Maven仓库获取的传递式的JAR依赖、Groovy工程中的源码文件还是web程序中的静态资源我们都会看到FileCollection的身影。我们接下来会研究这个接口支持的一些重要操作。

> 注2：实际上FileCollection有很多子类，这些子类在一些情况下特别有用，但是我们这里只关注最为通用的功能，也就是这个接口本身。

为了演示这些功能，我们会从一些有意思的文件集合开始，为每一个功能示例创建一个task与之对应。

例1-19。基础build文件，所有FileCollection例子都基于此。

```
apply plugin: 'java'

repositories {
  mavenCentral()
}

dependencies {
  compile 'org.springframework:spring-context:3.1.1.RELEASE'
}
```

转变为Set

我们从上面了解到FileCollection有一个files属性。它返回了个Set类型的对象，里面包含这个FileCollection所容纳的所有文件和目录。想要列出前面那个示例工程中的所有源码文件，我们可以像例1-20这样：

例1-20。一个简单的列出所有文件的方式

```
task naiveFileLister {
  doLast {
    println fileTree('src/main/java').files
  }
}
```

下面是naiveFileLister任务的输出结果：

```
$ gradle nFL
  :naiveFileLister
  [
   ~/file-collection-lab/src/main/java/org/gradle/example/ConsoleContentSink.j
ava,
   ~/file-collection-lab/src/main/java/org/gradle/example/Content.java,
   ~/file-collection-lab/src/main/java/org/gradle/example/ContentFactory.java,
   ~/file-collection-lab/src/main/java/org/gradle/example/ContentRegistry.jav
a,
   ~/file-collection-lab/src/main/java/org/gradle/example/ContentSink.java,
   ~/file-collection-lab/src/main/java/org/gradle/example/DefaultContentFactor
y.java,
   ~/file-collection-lab/src/main/java/org/gradle/example/DonneContent.java,
   ~/file-collection-lab/src/main/java/org/gradle/example/PoetryEmitter.java,
   ~/file-collection-lab/src/main/java/org/gradle/example/ShakespeareContent.j
ava
  ]
  BUILD SUCCESSFUL
```

转变为Path串

我们操作文件集合有很多目的，比如有时使用操作系统命令时需要将文件列表作为输入，在构建系统内部，核心的Java plug-in插件在执行javac编译器就有这样的需求。Java编译器有一个指定classpath的命令行参数，这个参数只接受基于操作系统的字符串格式。asPath属性的作用就是将FileCollection转换为这种基于特定操作系统的字符串。

例1-21。以path格式打印所有的编译时依赖

`println configurations.compile.asPath`

下面是输出：

```
$ gradle
  ~/.gradle/caches/artifacts-8/filestore/org.springframework/spring-context/3.
1.1.RELEASE/jar/ecb0784a0712c1bfbc1c2018eeef6776861300e4/spring-context-3.1.1.
RELEASE.jar:
  ~/.gradle/caches/artifacts-8/filestore/org.springframework/spring-asm/3.1.1.
RELEASE/jar/8717ad8947fcada5c55da89eb474bf053c30e57/spring-asm-3.1.1.RELEASE.j
ar:
  ~/.gradle/caches/artifacts-8/filestore/commons-logging/commons-
logging/1.1.1/jar/5043bfebc3db072ed80fbd362e7caf00e885d8ae/commons-logging-
1.1.1.jar:
  ~/.gradle/caches/artifacts-8/filestore/org.springframework/spring-
core/3.1.1.RELEASE/jar/419e9233c8d55f64a0c524bb94c3ba87e51e7d95/spring-core-3.
1.1.RELEASE.jar:
  ~/.gradle/caches/artifacts-8/filestore/org.springframework/spring-beans/3.1.
1.RELEASE/jar/83d0e5adc98714783f0fb7d8a5e97ef4cf08da49/spring-beans-3.1.1.RELE
ASE.jar:
  ~/.gradle/caches/artifacts-8/filestore/aopalliance/aopalliance/1.0/jar/235ba
8b489512805ac13a8f9ea77a1ca5ebe3e8/aopalliance-1.0.jar:
  ~/.gradle/caches/artifacts-8/filestore/org.springframework/spring-aop/3.1.1.
RELEASE/jar/3c86058cdaea30df35e4b951a615e09eb07da589/spring-aop-3.1.1.RELEASE.
jar:
  ~/.gradle/caches/artifacts-
8/filestore/org.springframework/springexpression/3.1.1.RELEASE/jar/1486d7787ec
4ff8da8cbf8752d30e4c808412b3f/spring-expression-3.1.1.RELEASE.jar
```

FileCollection用作模块依赖

通过打印模块依赖，我们可以了解到FileCollection是如何转path串的。值得一提的是显式的dependency configuration本身就是一个FileCollection。当依赖声明在dependencies{}中时它们会被赋给一个带有名字的configuration。而所有的configuration都是定义在configurations{}区块内的[注3]。

作为一名老师同时又是Gradle会议的常客，我有时想让学生们在没有网络的情况下也能编译Java代码。（因为与会者经常有几百人，而且他们每人都有两或三台设备需要连接无线网，但是美国的很多酒店和会议中心都没有为这样大的带宽需求做好准备。）虽然我强烈推荐依赖交由build工具负责管理，但是遇到这种情况我也只能将所有的依赖以老的Ant式[注4]的方法放在我的工程里。对于有些Java框架或者API，手动下载这些依赖JAR包将是一项很繁重的工作。通过文件集合这种依赖方式，我们可以将这项工作自动化。

如果我们将下面这段代码放在例1-19中，运行这个task后我们会发现我们需要的JAR包都被放到了lib目录里。后面增加新的依赖我们也只需要重新运行这个task，然后新的JAR包就自动出现在目录里了。注意, copy task在之前的例子里都是接收一个String, 这里我们看到它也可以接受一个FileCollection对象，见例1-22.

例1-22。使用FileCollection作为模块依赖获取JAR包

```
task copyDependencies(type: Copy) {
  from configurations.compile
  into 'lib'
}
```

> 注3：最常用到的configuration是compile和runtime， 它们都定义在Java plug-in中， 并没有显式的放在configurations块里。想要深入了解configurations和dependencies相关的知识，可以参考第4章。
>
> 注4：你可能更喜欢用Ant的方式， Gradle不会强制你使用哪种机制。

FileCollection加减法

FileCollection也可以使用+号和-号来操作。我们可以举几个例子来展示这一特性。我们的例子是一个基于Spring框架的命令行程序。Spring在精简模式下也会引入很多JAR包，想要运行程序就得通过命令行提供这些JAR，但每次都这么做会非常令人头疼。JavaExec为我们提供了一个简单的方式，只要告诉它要运行那个类，并且设置好启动JVM所需的classpath就可以了。见例1-23。

我们用到的classpath包含两种：所有编译时依赖和你的工程源码编译出的class. 前者存在于configurations.compile文件集合中，后者存在于sourceSets.main.output中。sourceSet集合我们会在下一节学习。

例1-23。使用FileCollection加法创建一个runtime classpath

```
task run(type: JavaExec) {
  main = 'org.gradle.example.PoetryEmitter'
  classpath = configurations.compile + sourceSets.main.output
}
```

从一个集合中减去另一个集合会产生一个新的相减后的集合。要创建一个除了Shelley以外的所有诗人的文件集合，我们可以像例1-24这样：

例1-24。FileCollection做减法

```
def poems = fileTree(dir: 'src/main/resources', include: '*.txt')
def romantics = fileTree(dir: 'src/main/resources', include: 'shelley*') def g
oodPoems = poems - romantics
```

将goodPoems的file属性打印出来（或者通过其他方式查看）可以发现它包含了除以shelley开头外的所有src/main/resources下的文件。在实际的构建中，这种情况可能会采用exclude的方式，但是使用相减的方式有时可以提供更精确的操控，比如在编译一个Java web程序时从一个由WAR task打包的依赖集中减去容器已经提供的JAR包，这个时候减法操作就会显得很有用了。

FileCollections用作SourceSets

在之前的例子中我们使用fileTree()方法创建所有工程源码的文件集，实际上Gradle为我们提供了更简单的方式，那就是**SourceSets**。SourceSets属于Gradle的范畴内的对象，用来代表源码文件集。它以FileCollection的形式暴露出了源码输入文件集和编译输出文件集。

SourceSet对象的allSource属性包含了所有的输入文件集：待编译的源码文件和所有的资源文件。例1-25打印出了这个属性。

例1-25。打印main Java SourceSet的源文件集

`println sourceSets.main.allSource.files`

```
[~/file-collection-lab/src/main/resources/application-context.xml,
    ~/file-collection-lab/src/main/resources/chesterton.txt,
    ~/file-collection-lab/src/main/resources/henley.txt,
    ~/file-collection-lab/src/main/resources/shakespeare.txt,
    ~/file-collection-lab/src/main/resources/shelley.txt,
    ~/file-collection-lab/src/main/java/org/gradle/example/ConsoleContentSink.
java,
    ~/file-collection-lab/src/main/java/org/gradle/example/Content.java,
    ~/file-collection-lab/src/main/java/org/gradle/example/ContentFactory.jav
a,
    ~/file-collection-lab/src/main/java/org/gradle/example/ContentRegistry.jav
a,
    ~/file-collection-lab/src/main/java/org/gradle/example/ContentSink.java,
    ~/file-collection-lab/src/main/java/org/gradle/example/
      DefaultContentFactory.java,
    ~/file-collection-lab/src/main/java/org/gradle/example/DonneContent.java,
    ~/file-collection-lab/src/main/java/org/gradle/example/PoetryEmitter.java,
    ~/file-collection-lab/src/main/java/org/gradle/example/ShakespeareContent.
java]
```

同样，编译输出由outputs属性提供。outputs不会冗杂的给出所有编译生成的文件，而是给出存放这些编译文件的目录 — 要拿到这些目录首先得运行编译器编译这些源码。

例1-26。 打印main source set的所有编译输出目录

`println sourceSets.main.output.files`

```
[~/file-collection-lab/build/classes/main,
~/file-collection-lab/build/resources/main]
```

文件懒加载

当使用文件集时我们可能倾向于认为它就是一个静态的列表，例如在调用fileTree()时会立即扫描文件系统，然后生成一个只读的文件列表。只读数据结构有它的好处， 但是在Gradle里并没有多大意义。所以结果是，Gradle会在你真正想用的时候才去实例化这个FileCollection接口，并去加载相关的文件。例如，有一个task调用fileTree()去收集build/main/classes下所有符合**/*Test.class的文件。如果这个集合在configuration阶段创建（这很有可能）[注6]，你想找的文件可能这时还不存在，因为这些文件要在execution阶段才会生成。实际上，文件集是为静态地描述集合这一语义而设计的，真正的内容要在执行阶段某个需要用到它的时候才会被填充。

总结

这一章里我们看到了Gradle对文件操作的充分支持。我们研究了copy任务，通过不同难度的例子了解了她的各项功能：移动文件，匹配文件，甚至过滤文件内容。我们学习了关键字替换和文件重命名，这些都会成为我们信手拈来的技术。我们回顾了三种重要的文件处理方式，最后还学习了最最重要的FileCollection接口 — 它在Gradle领域内扮演着非常重要的角色。Gradle给了你丰富的文件API，可以让你做任何你想做的事情，而不是简单的给你一个File对象。这些技术可以帮助你创建自己的下一代自动化构建系统。

> 注6：Gradle构建分为三个阶段：初始化阶段(initialization)，配置阶段(configuration)和执行阶段(execution)。配置阶段为Gradle准备好任务图(Task Graph)。真正的操作像拷贝、编译和打包等都是发生在执行阶段。

第二章

自定义插件

插件思想

Gradle希望借助其标准的领域语言(DSL)和插件，使自己成为一款即使没有任何额外扩展也能独立运行的强大构建工具。通过Gradle，大多数常见的构建任务只需要通过简单的配置就可以运行。我们很容易想到将这些常见任务写成通用的形式，但是要写的“足够”通用，还是要花点功夫的。

一些比较大的项目（大量代码和资源）很容易演化成复杂的多工程形式， 它们产出标准的归档格式需要涉及很多的操作：数据库迁移，静态资源转换，自动化部署校验等，除此之外还有很多标准配置无法满足的自动化过程。

创造这样的构建系统是一种特殊形式的软件开发。这里涉及的自动化不是项目本身业务逻辑的自动化，而是项目源码构建逻辑的自动化。这种软件的开发正是Gradle大力推进的。

经验较少的Gradle开发者在实现这样的构建软件时往往会在doLast里写出大量的命令式的代码。然而，这种代码最终会变得无法测试，也无法读懂，这种代码也正是其他一些构建工具被诟病所在。我们强烈推荐你不要这样做，因为我们为你提供了插件API。

插件API

Gradle插件是一种可以用来扩展其核心功能并且可以被发布的软件。插件从三个方面扩展了Gradle的功能：
首先，它可以像你真正的build文件一样操作你的Project对象。通过应用插件你可以增加或修改Task、SourceSets、dependencies和repositories等Gradle元素。

其次，它可以为你引入其他模块帮助你完成特定的工作。例如，一个为Java web service工程生成WSDL文件的插件不应该以代码的形式包含在项目里，而应该以library依赖的方式使用，同时要提供一种机制可以让构建系统从在线仓库中获取到这个插件。

最后，插件可以为你的构建脚本引入新的关键字和领域对象。Gradle DSL中不存在和你业务紧密相连的一些东西，比如，你的部署服务器地址、你的应用所关联的数据库shema、你的源码控制工具所暴露出的操作等。的确，标准DSL不可能预见开发者碰到的所有场景，Gradle只能提供一些有着良好文档的API，让开发者根据自己的场景去定制扩展Gradle的功能，以满足自身的需求。这也是Gradle作为一款构建工具所具备的核心强项。它可以让你以自己习惯的风格写出精简的、声明式的构建代码，并具备丰富、专用的功能。

插件示例

这一章里我们将实现一个Gradle插件，用来自动使用开源数据库重构工具Liquibase。Liquibase是一个用Java写的命令行工具，用来管理关系型数据库shema的变更。它通过对数据库shema做逆向工程来生成XML格式的变更日志，然后用变更日志和正在运行的数据库实例作对比从而得出数据库是否需要被重构。对于那些不喜欢XML而喜欢Groovy的开发者可以使用开源的Groovy Liquibase DSL作为替代。

你可以在这里了解到更多的Liquibase相关的知识：Liquibase快速开始

Liquibase功能很强大，但是直接通过命令行使用却显得很笨拙，如果能有一层包装来简化使用过程就好了。因为自动化构建和部署一直是我们的目标，所以我们自然而然倾向于将Liquibase的操作嵌入到构建过程中。

本章的目标包含以下几点：

- 创建generateChangeLog、changeLogSync和update命令的相应的Gradle task。
- 使用Groovy DSL替代XML格式的Changelog。
- 将Gradle task重构为自定义task类型
- 引入extensions概念用来描述Changelog和数据库配置。
- 将插件打包为可发布的JAR包

> Liquibase插件刚开始会以gradle build文件的形式呈现。在你还不知道你的代码最终是什么样子的时候采用这种方式会比较容易上手。这也是开发一个新的自动化构建软件比较典型的工作流。随着插件成型，我们会慢慢对它重构，直到变为一个包含自己生命周期的可发布的插件工程。这样演进式的开发是一种比较完美的学习Gradle API同时发现自身需求的方式。

设置

要运行这一章的例子你得先有一个可以供Liquibase连接的数据库。本书的示例程序通过build文件配置了H2数据库。使用Git克隆http://github.com/gradleware/oreilly-gradle-book-examples 这个仓库， 然后切换到plugins/database-setup目录下。运行以下两个task：

```
$ gradle -b database.gradle createDatabaseScript
$ gradle -b database.gradle buildSchema
```

第一个任务会提供一个叫做starth2的平台相关的脚本，在你开发过程中可以随时通过运行这个脚本进入到内嵌的管理员控制台，进而来检测数据库的schema。第二个任务会创建一个迫切需要重构的示例数据库schema — 仅仅是一个测试环境，供我们学习插件开发之用。

【提示】你需要将database.gradle文件移到你开发插件的工作目录中，然后在该目录下执行buildSchema以确保H2数据库放在正确的位置并且插件找得到它。或者，你也可以把数据库放在工作目录以外的其他地方，然后编辑JDBC URL让它指向正确的路径，但是这种方式留给读者做练习。

从头开始写一个你自己的插件

我们的Liquibase插件会从执行Changelog逆向、Changelog同步和Changelog更新三个任务开始。研究Liquibase API后我们发现最好的执行这三个命令的方式是调用liquibase.integration.commandline.Main.main()方法。这个方法接收一个参数数组，用来指示连接哪个数据库和执行哪个子命令。对于每一个Liquibase操作任务，我们都会构造好参数数组然后调用前面的main()方法。

考虑一下最终的task会是什么样子？由于我们要在gradle里支持Liquibase的三个命令 — generateChangeLog、chageLogSynche和update — 所以我们打算为每一个命令创建一个同名的task。在某些场景下你可能要考虑避免和其他插件产生命名冲突，采用命名空间的方式给每一个task名加一个lb或者liquibase的前缀，但是在这里我们为求简洁就先不考虑这么多了 — 我们的关注点在插件开发上。

自定义Liquibase task

我们的插件最终会加入一些功能完备的task用来调用Liquibase，这些task很少或者完全不用声明式的配置。在做到这些之前我们得先自定义一个task类型。这个task的目的是将task参数转换为Liquibase命令行所需要的参数，然后调用main()方法。实现如下（例2-1）

例2-1。自定义Liquibase task类

```
import org.gradle.api.DefaultTask
import org.gradle.api.tasks.TaskAction
import liquibase.integration.commandline.Main

class LiquibaseTask extends DefaultTask {
    String command
    String url, password, username
    File changeLog

    @TaskAction
    def liquibaseAction() {
          def args = [
              "--url=${url}",
              "--password=${password}",
              "--username=${username}",
              "--changeLogFile=${changeLog.absolutePath}",
              command
          ]
    Main.main(args as String[]) }
}
```

记住，自定义task类型需要实现org.gradle.api.Task接口或者更常见的是继承自org.gradle.api.DefaultTask类。LiquibaseTask为后续命令执行提供了基础接口。LiquibaseTask的属性在之后会被具体的task用来配置需要执行的动作。

写好自定义类型后，我们就可以创建自己的task了，只需要将类型声明为这个类并做好相应的配置即可。注意，例2-2中我们可以使用Gradle的配置语法来设置task的成员变量，同时我们是通过非常标准的赋值语法为url、username、password、changelog和command属性赋值的。

例2-2. 实例化自定义Liquibase task

```
task generateChangeLog(type: LiquibaseTask) {
  url = 'jdbc:h2:db/gradle_plugins'
  username = 'secret'
  password = 'sa'
  changeLog = file('changelog.xml')
  command = 'generateChangeLog'
}
```

应用插件

我们实现了自定义task类型，同时也拥有了一个可以运行的Liquibase task。那现在就让我们稍稍往后退一步来看看怎么样创建一个Gradle插件吧。最简单的Gradle插件是实现org.gradle.api.Plugin接口。这个接口只定义了一个方法：apply(Project project)。我们可以通过例2-3看到这个接口的用法。

【提示】Plugin接口是泛型的，因为它理论上可以被应用于任何Gradle对象。将其应用于Project是目前最常见的用法，也是我们在本书看到的唯一用法。

例2-3。第一版Liquibase插件apply()的用法

```
class LiquibasePlugin implements Plugin<Project> {

    void apply(Project project) {
        project.task('generateChangeLog', type: LiquibaseTask) {
        group = 'Liquibase'
        command = 'generateChangeLog'
    }

    project.task('changeLogSync', type: LiquibaseTask) {
        group = 'Liquibase'
        command = 'changeLogSync'
    }

    project.task('update', type: LiquibaseTask) {
        group = 'Liquibase'
        command = 'update'
} }
}
```

这里提醒一下，如果你是第一次像我们这样写插件，你可以将代码直接写在build.gradle文件中。在这个时间点，你的主要精力是用来学习插件API、插件作用域和如何设计插件这些插件本身相关的事情的。部署和打包插件是非常重要的内容，但是我们现在还不用考虑这个。

这个例子创建了3个新的task：generateChangeLog、changeLogSync和update[注2]。因为Liquibae插件是使用Groovy实现的，所以我们在声明新的task时可以采用Gradle风格的语法。实际上，即使脱离插件，直接将这些代码定义在build文件中，也会产生一样的运行效果。Gradle老手可以不写Groovy插件，但这是得益于直接书写build文件和和写插件的相似性，并且也是得益于Groovy是一个Java语言的效率增强版这一事实。

Extensions

此时我们的插件已经可以拿来做一些事情了。但是它的配置方式看起来太过于学究（例2-4） — 我们必须要每一个task配置数据库username、password、URL和changelog文件。

例2-4。勉强可用的Liquibase task

```
generateChangeLog {
  url = 'jdbc:h2:db/gradle_plugin'
  password = 'secret'
  username = 'sa'
  changeLog = file('changelog.xml')
}

changeLogSync {
  url = 'jdbc:h2:db/gradle_plugin'
  password = 'secret'
  username = 'sa'
  changeLog = file('changelog.xml')
}

update {
  url = 'jdbc:h2:db/gradle_plugin'
  password = 'secret'
  username = 'sa'
  changeLog = file('changelog.xml')
}
```

> 注2：generateChangeLog用来逆向工程一个数据库schema，changeLogSync用来将一个新的数据库置于Liquibase管理之下，update用来将新的变更同步到数据库。

很明显，我们不应满足于此。Gradle插件的真正强大之处不仅在于其将命令式的代码隐藏于插件背后 — 这个特性Ant和Maven早在十多年前就有了，那时Gradle1.0版本都还没发布 — 而是在于其扩展构建语言的能力。Extension API就是满足这种扩展能力的主要方法之一。

设计extension之前要想明白你期望中的build文件是什么样的。我们要想象哪些事情会以扩展语言的方式和build脚本交互。在这个例子中，答案很简单：build脚本需要知道database和changelog。

database是一个特殊的与JDBC建立连接的数据库实例。一个自动化的Liquibase数据库迁移脚本会涉及几个独立的对象分别用来代表不同的用途，这其中包括：本地数据库沙盒、测试专用的暂存服务器数据库实例和产品数据库实例等。

changelog是一个XML格式的文件，里面包含一个有序的数据库重构记录表。你可以到Liquibase网站上查看更多关于Liquibase changelog 的信息。例2-5只有一个changelog文件，但是真正使用Liquibase的构建系统可能会将其分为好几个独立的文件。我们的扩展语言必须能支持任意数量的changelog文件。

例2-5。我们自己的DSL

```
liquibase {
  changelogs {
    main {
      file = file('changelog.groovy')
    }
  }

  databases {
      sandbox {
        url = 'jdbc:h2:db/liquibase_workshop;FILE_LOCK=NO'
        username = 'sa'
        password = ''
      }
      staging {
        url = 'jdbc:mysql://staging.server/app_db'
        username = 'dev_account'
        password = 'ab87d24bdc7452e557'
      }
  }

  defaultDatabase = databases.sandbox
}
```

注意到没有，这里我们在使用自己创造的Gradle语法，用来描述我们特定领域的数据对象，而这些是Gradle设计者肯定考虑不到的。所有的语法都圈定在一个固定的命名空间里(liquibase), 但是里面内容我们是可以完全控制的，在这里我们将其设计成可以满足我们数据描述需求的样子。通过这样一个简单的例子我们可以窥探到Gradle核心价值 — 自定义构建语言。Gradle不仅允许我们定制自己的构建脚本，而且允许我们定制自己的构建语言。这正是Gradle可以用来管理构建复杂性的关键。

例2-6。 Liquibase插件的extension类

```
import org.gradle.api.NamedDomainObjectContainer

class LiquibaseExtension {
    final NamedDomainObjectContainer<Database> databases
    final NamedDomainObjectContainer<ChangeLog> changelogs
    Database defaultDatabase
    String context
    LiquibaseExtension(databases, changelogs) {
        this.databases = databases this.changelogs = changelogs
    }

    def databases(Closure closure) {
        databases.configure(closure)
    }

    def changelogs(Closure closure) {
        changelogs.configure(closure)
     }
 }
```

extention类用来保存例2.5看到的数据集合，它定义了两个方法、两个非final属性[注3]和两个NamedDomainObjectContainer类型的final属性。NamedDomainObjectContainer是一个泛型集合，它的两个对象使用了不同的参数类型 — Database和ChangeLog， 这两个参数类型必须要像例2-7那样定义。Database和ChangeLog类和普通Java对象（POJOs或者叫POGOs）的不同之处是它们必须有一个name属性，并且构造函数要接收一个String类型的参数，同时要在构造函数中用这个参数初始化name，否则它们不会继承或实现任何Gradle的基类和接口。在前面的例子中，Gradle之所以能保存database和changelog数据就是因为这两个类和NamedDomainObjectContainer配合使用的结果。

```
class ChangeLog {
    def name
    def file
    def description
    ChangeLog(String name) {
        this.name = name
    }
}

class Database {
    def name
    def url
    def username def password
    Database(String name) {
       this.name = name
    }
}
```

> 注3：在Groovy和Java中，final域可以在构造函数中初始化，但是之后就不能再改变。这两个final域是对象集合，集合中的对象是可以在运行时改变的，但是对于集合实例本身，一但所在的对象构造完成后就不能再改变了。
>
> 注4：POJO代表“普通的Java对象”。它是指那些只包含属性、方法和一个普通构造函数的对象类型，它不会参与处理任何额外的业务需求。

对LiquibasePlugin.apply()稍作修改，加上创建extension的相关代码（见例2-8最后的extensions.create调用），我们就可以得到一个增强版的apply方法。create方法的第一个参数代表这个extention的名字，这里是liquibase，最后两个参数是NamedDomainObjectContainers的实例，代表需要包含在extenion内的扩展对象。

例2-8。 带创建extension的apply方法

```
class LiquibasePlugin implements Plugin<Project> {

    void apply(Project project) {
    // Create and install custom tasks
    project.task('generateChangeLog', type: LiquibaseTask) {
        group = 'Liquibase'
        command = 'generateChangeLog'
    }
    project.task('changeLogSync', type: LiquibaseTask) {
        group = 'Liquibase'
        command = 'changeLogSync'
    }
    project.task('update', type: LiquibaseTask) {
        group = 'Liquibase'
        command = 'update'
    }
    // Create the NamedDomainObjectContainers
    def databases = project.container(Database)
    def changelogs = project.container(ChangeLog)
    // Create and install the extension object
    project.configure(project) {
         extensions.create("liquibase",
     }
  }

}
```

有了整体了解后，我们来仔细分析一下extension类本身。它有一个defaultDatabase属性，是Database类的实例，在liquibase块中可以被正常赋值。 在例2-5中可以你看到defaultDatabase = databases.sandbox这一行。这句代码告诉LiquibaseTask如果没有明确指定其他的database，就默认使用这个叫sandbox的database。extention类有两个方法：databases和changelogs， 它们都只接收一个Closure参数。这两个方法都会调用集合对象的configure()方法并传入Closure对象， 这么做有两个作用：首先它会创建一个扩展对象的实例(Database或者ChangeLog)， 然后用build脚本里声明的属性值去初始化这个扩展对象的对应字段。我们通过例2-9再来看一下这段代码：

例2-9。扩展对象创建和配置

```
liquibase {
    databases {
      sandbox {
        url = 'jdbc:h2:db/liquibase_workshop;FILE_LOCK=NO'
        username = 'sa'
        password = 'secret'
    }
  }
}
```

sandbox代码块创建了一个名为“sandbox”的Database类型的扩展对象实例。（请记住，因为我们使用的是命名的对象容器 — NamedDomainObjectContainer，因此所有的内部对象都必须有名字）。sandbox花括号内的赋值语句最终都会映射到真正的Database实例相应的字段中去。

sandbox块的外层闭包作为参数传给了databases()方法。如果你想定义更多的database对象 — 比如一个暂存服务器的对象或者产品对象 — 你可以将它们定义在与sandbox平级的地方。

以上例子的完整代码可以在GitHub Gist 中找到。注意，我们所有的代码都是直接写在build文件中的，不需要编译就能运行。这是一种高效创建原型的方法， 当然了，一个正规插件是要经过编译的，而且要经过测试、定版本号和发布部署等步骤，这样的过程才趋向于一个成熟的软件开发。这些我们会在下一节讲到。

打包插件

将我们这些不成体的代码整合到一个独立的插件工程中并不难。过程主要涉及将我们之前写的那些代码片段归类、增加一些meadata文件并设置好Gradle API的classpath等，下面我们会挨个来讲。

我们将所有代码整理到了7个类文件中，并放到了com.augusttechgroup.gradle.liquibase和com.augusttechgroup.gradle.liquibase.tasks两个包下面。由于这些是Groovy类，我们需要将它们放在src/main/groovy目录里。整理后的源码可以在GitHub上看到。

放好代码后我们需要再添加一个文件用来指派插件ID，这个ID在应用插件时会用到。在前面例子中我们都是用”apply plugin: LiquibasePlugin”这种方式直接通过类名应用插件的。一个打包好的插件应该像Gradle核心插件那样使用，用“apply plugin: ‘liquibase’”的方式。插件ID是通过一个metadata文件提供的。

我们要在工程下的src/resources/META-INF/gradle-plugins目录里新增一个名为liquibase.properties的文件。这个目录会被自动打包进jar包的META-INF文件夹。这个文件的目的是将插件ID转换为实现了这个插件的那个类的全路径。文件内容如例2-10所示。

例2-10。liquibase.properties文件的内容

```
implementation-class=com.augusttechgroup.gradle.liquibase.LiquibasePlugin
```

最后，插件代码需要构建一下。很自然会想到要用Gradle来构建插件， 完整示例中的build文件包含了很多额外的功能，像打包和部署到Central仓库，但是这些都很简单。主要问题是如何添加Gradle API所在的各种依赖。（这个插件同时依赖了Liquibase模块，但是我们通过阅读《使用Gradle进行构建和测试》一书已经掌握了如何引入普通Maven风格的依赖）

“如何添加Gradle API的依赖”这个问题的答案是使用gradleApi()方法。只需将这个函数的返回值作为运行时依赖（compile）来添加，它就会为你引入编译一个插件用到所有Gradle内部类。 例2-11展示了这个build文件的一个简化版。

例2-11。 一个简化后的插件build文件

```
apply plugin: 'groovy'

repositories {
  mavenCentral()
}

dependencies {
  groovy 'org.codehaus.groovy:groovy:1.8.6'
  compile gradleApi()
  compile 'org.liquibase:liquibase-core:2.0.3'
}
```

总结

Liquibase插件是一个很好的例子。它向我们展示了Gradle在增强功能和自定义扩展语法方面的能力。插件封装了构建代码，同时为我们引入了很多高级的功能，这些功能可以有自己的语法，甚至可以通过扩展自己来容纳一个新的不存在于原生构建中的数据类型。

当插件独立构建和打包时，它就以一个软件工程的形式存在了。我们可以为它引入自动化测试，甚至更理想一点，我们可以把它当做项目的一部分来进行开发。我们可以为它部署持续集成环境，我们也可以为它应用一些在常规软件开发上才使用的管理流程，比如制定版本号、制定仓库标签等。简而言之，插件为我们的系统带来了新的语言，帮助我们完成很多功能，它完全可以成为我们软件开发实践中很重要的一环。至此，我们已经掌握了Gradle插件最基本的内容，相信它会在你的项目释放出强大的能量。

第三章

hook构建过程

Gradle是一个可深度定制的构建工具。它有着出色的语法和约定，可以满足你高标准的构建需求，同时也有着非常多的新奇的特性，可以让你实现很多意想不到的功能。在定制构建系统的这条路上，Gradle还有很多方面有待挖掘。

Gradle为你提供了hook构建过程的能力，无论是配置阶段还是执行阶段的事件它都可以hook。hook本质上是一段可以插在构建过程中的代码，当某种事件发生时它就会被调用执行，比如task被添加、工程被创建等时机。Gradle还提供了元编程能力，比如后面要讲到的rules, 你只要遵循相关的范式就可以通过元编程的方式动态创建task。这一章里我们将学习hook和rules这两个特性。它们都是管理构建复杂性的有力工具。

Gradle生命周期回顾

很多读者可能已经熟悉了Gradle生命周期，不过为了能更深入的理解这些生命周期，我们会花一些篇幅来回顾一下它们。任何一个Gradle构建过程都具有三个固定的阶段：初始化阶段、配置阶段和执行阶段。

在初始化阶段Gradle会找到所有需要处理的build文件，这么做有一个重要的目的就是区分这个工程是单工程还是多工程。如果是单工程，Gradle会找出build文件然后将之传递到下一个阶段。如果是多工程，Gradle会找出所有的build文件然后交由下一阶段处理。

下一个是配置阶段，在这个阶段里，Gradle会将上一个阶段传入的那些build文件当做Groovy脚本来执行。配置阶段执行这些build脚本其实不会产生真正的动作，它只会根据build文件描述的对象生成一个有向无环图(DAG)[注1]， 真正的动作是在下一个阶段做的。很多hook方法在生成这张图的时候可以被触发执行。

最后一个是执行阶段。在执行阶段Gradle找出所有DAG图里要被执行的task，然后按照它们声明的依赖顺序依次执行[注2]。所有的构建动作（比如编译源码、拷贝文件和上传最终产物等）都是发生在执行阶段。有一些hook方法是在执行阶段触发执行的。

了解构建图(Build Graph)

在2000年初的时候，AOP编程范式变成了管理复杂的企业级Java软件的一种比较流行的但又有点剑走偏锋的方式。AOP给了大家一种在不影响原有业务逻辑的同时增强相关逻辑单元的能力 — 如果是Java语言，逻辑单元就是它的方法。例如一个处理web请求的方法需要在每次执行前执行一段安全性检查，或者在它执行完后打印一些日志，再或者写数据库之前需要对它先做一些配置，写完后还要执行一些事务提交逻辑。以上这些都可以使用AOP来解决。

AOP形成了一套自己的术语，并且出现了一些实现了它的思想的框架。由于Gradle hook的思想和AOP非常像，所以本书中保留了AOP的一些术语。某一段在原始方法之前或者之后执行的代码被称作“建议”(advice)。“建议”的代码经常被称作和原始代码“正切”，意思是说这段代码和原始代码之间没有任何同向关系 — 即便它们是顺序执行的。有人也将这种实现思路称作“面向切面”。使用这个思想最常见的例子就是日志输出和数据库操作管理。

在我们关于Gradle task hook的讨论中，我们尽量维持”建议”这一术语，我们的例子也会将关注点放在“正切”或者说“面向切面”这个伴随AOP发展了10多年思想上。

> 注1：如果是多工程，还会有一个工程对象DAG。
>
> 注2：依赖顺序的意思是：如果task A依赖于task B，则B先执行。

了解工程评估(Project Evaluation)

build文件的作用是为每一个工程定义相关的配置和task。这个定义的过程发生在配置阶段，而task的执行发生在执行阶段。project.beforeEvaluate()和project.afterEvaluate()方法分别是对build文件评估前和评估后的hook。

例3-1是一个hook build文件评估完成的例子。build评估完后，hook会去检查grammars目录是否存在。如果存在，则创建一个名为testGrammars的task，在里面模拟对目录内容做语法检查操作。尝试在有grammars和没有grammars目录的情况下运行build，观察两种情况的输出。

例3-1. hook build评估完成

```
afterEvaluate {
  if (file('grammars').isDirectory()) {
    println "'grammars' directory found"
    task testGrammars << {
      println "Running grammar tests"
    }
  } else {
    println "'grammars' directory not found"
  }
}

task helloWorld {
  doLast {
    println "hello, world"
  }
}
```

beforeEvaluate()在这个例子中完全不会起作用，因为任何build文件在评估前是不可能知道自己要hook“评估前”这个事件的[注3]。beforeEvaluate()只对多工程有用。例3-2有三个工程：一个父工程和两个子工程。语法检查的task放在beforeEvaluate()中时只会作用于子工程，通过allprojects这个方法可以为每一个工程添加评估前hook。allprojects的闭包会应用于所有工程（包括子工程），由于评估父工程的时候子工程还尚未评估，因此hook子工程评估前会起作用。如果你分别在有grammar目录和没有grammar目录的情况下运行这个示例，你会发现两种情况下父目录都无法创建testGrammars task，因为beforeEvaluate hook不能运行在它自己所在的工程中。

例3-2。hook工程评估前

```
allprojects {
    beforeEvaluate {
        if(project.file('grammars').isDirectory()) {
            println "'grammars' found in ${project.name}"
            task testGrammars << {
                println "Running grammar tests in ${project.name}"
            }
        } else {
            println "'grammars' not found in ${project.name}"
        }
    }
}

task helloWorld {
    doLast {
        println "the parent says hello"
    }
}
```

注意，这里我们用的是project.file()检测文件是否存在，替代之前的file()，因为gradle会去所有工程目录里寻找文件，而不是只在父工下寻找。

> 注3：当hook代码被评估的时候就已经晚了，因为其所在build文件的评估已经开始了。

全局工程加载和hook

前面的例子展示了beforeEvaluate()和afterEvaluate()通过allprojects闭包一次性应用给所有的工程情况。如果你需要针对性的hook某个单独的子工程，做法也同样简单。此外，如果你需要hook所有的工程评估前和评估后两个时机，那就要用到下面要讲的gradle.projectsLoaded()和gradle.projectsEvaluated()了。

gradle.projectsLoaded()发生在所有工程加载完成时，也就是初始化阶段执行完成时，这个时候任何工程的评估还没有开始。当然，这个时候能用的Gradle对象不太多 — 因为它们大多还没开始创建，但是还是可以做一些有用的事情的。闭包接收了一个gradle对象作为参数（例3-3中命名为g），你可以正常对它进行操作。例3-3展示了通过配置buildscript为工程添加一些自定义插件依赖的情况。在一些比较简单的工程中我们有时觉得buildscript的配置代码有点烦人，这种方法正好可以将这些代码移到单独的地方。

例3-3。hook初始化阶段结束事件

```
gradle.projectsLoaded { g ->
  g.rootProject.buildscript {
    repositories {
      mavenCentral()
    }
    dependencies {
      classpath 'com.augusttechgroup:gradle-liquibase-plugin:0.7'
      classpath 'com.h2database:h2:1.3.160'
    }
  }
}
```

projectEvaluate()同样接收一个闭包，但它是运行在所有工程评估完成之后[注4] — 也就是配置阶段完成后执行阶段开始前的这个时间点。这个时候工程图已经生成并且可以确保任何task都尚未执行。

> 注4：回想下之前对Gradle生命周期的讨论

构建结束

你可能想知道构建在什么时候结束，成功了还是失败了。当然了，Gradle命令行或者持续集成框架给你一个反馈，但是想象下是否可以通过其他方式获取到这个状态，并且在此基础上做一些事情?

如果构建失败，肯定是因为哪儿的构建逻辑抛了异常。顺利执行完并且没有抛异常就说明构建成功了。这个状态可以通过buildFinished()的闭包参数BuildResult获取到。见例3-4。

例3-4。hook构建结束

```
gradle.buildFinished { buildResult ->
  println "BUILD FINISHED"
  println "build failure - " + buildResult.failure
}
task succeed {
  doLast {
    println "hello, world"
  }
}
task fail {
  doLast {
  throw new Exception("Build failed") }
}
```

运行gradle succeed输出如例3-5所示。

例3-5。succeed task输出

```
$ gradle succeed
:succeed
hello, world
BUILD SUCCESSFUL
Total time: 1.374 secs
BUILD FINISHED
build failure - null
```

注意最后两行的输出。buildFinished()会在gradle输出Total time: 1.374 secs后执行，报告构建结束，同时打印出buildResult.failure的值，这里是null。

运行gradle fail会抛出一个异常（例3-6）。注意，即使编译失败也会输出BUILD FINISHED。无论构建成功与失败我们的hook都会执行，只是有错误需要报告时gradle.failure属性才有值。gradle.failure是Gradle的一个内部类型，封装了异常的源头。我们可以将它显示到构建控制台或者直接log输出，通过对其分析我们可以找到失败的根源。

例3-6。fail task输出

```
$ gradle fail
:fail FAILED
FAILURE: Build failed with an exception.
* Where:
Build file '/Users/tlberglund/Documents/Writing/Gradle/oreilly-gradle-book-exa
mples/ hooks-lab/build-finished/build.gradle' line: 14
* What went wrong:
Execution failed for task ':fail'. > java.lang.Exception: Build failed
* Try:
Run with --stacktrace option to get the stack trace. Run with --info or
--debug option to get more log output.
BUILD FAILED
Total time: 1.453 secs
BUILD FINISHED
build failure - org.gradle.api.internal.LocationAwareException: Build file '/U
sers/ tlberglund/Documents/Writing/Gradle/oreilly-gradle-book-examples/hooks-l
ab/build- finished/build.gradle' line: 14
Execution failed for task ':fail'.
```

Rules

通常情况下，一个task就是一个有着特定名字的特定操作。例如Java工程中的build task用来编译和测试代码。一个特定的task始终做一件事情，并通过固定的名字调用，就像Java类方法一样，专用并且可识别。

但是如果你的task行为是不可预测的呢？这种情况一般出现在你想执行一些行为逻辑需要根据环境变化的任务，而且这些任务具有一个通用的模板的时候。例如你可能想要一批task来执行部署操作，这些操作的逻辑相同，但是有各自的host，或者他们有相同的二进制仓库，但是有着不同的产出文件。

瞬间想到的是为task传入参数，但是这是行不通的。简单的方式是通过System属性或者环境变量或者build脚本中定义的临时变量来进行，但这样会导致task代码变成命令式的风格，增加了逻辑复杂度，相当于是将task逻辑和它之外的build脚本耦合了起来。在Gradle中，只要有一点可能性，我们都希望能将独立且具名的task暴露给使用者。解决这个问题的办法是使用Rules。

任何时候你调用一个不存在task时，Gradle都会去咨询task rules。task rules可以以任意方式回应“task不存在”的请求，但是常见的情况是根据你调用的名字动态创建一个task。动态创建task这样的特性非常有用。你如果熟悉Groovy的methodMissing和propertyMissing或者Ruby的method_missing的话，你可能已经想到我要说的了。

Java插件通过rules方式提供了build、上传二进制仓库和clean三个task。我们从这三个中最简单的clean rule来看看rules是怎么工作的。

每一个Java build文件都有一个通用格式的clean task，其作用是删除build目录。然而你可能只想删除某个task产生的特定目录。在Java工程里，compileJava和processResources这两个task都会在build目录里生成文件：前者默认将文件放在build/classes/main下，后者默认放在build/resources/main下。我们可以通过编译这个示例工程 查看输出。下面是编译完成后的build目录：

```
build
|___classes
|      |___main
|              |___org
|                     |__gradle
|                            |___poetry
|                                   |___PoetryEmitter.class
|
|—— libs
|      |___ java-build-with-resources.jar
|
|__ resources
      |__ main
            |___ chesterton.txt
            |___ henley.txt
            |___ shakespeare.txt
            |___ shelley.txt
```

可以看到，Java类被编译了，静态资源被拷贝了，JAR文件也生成了。此时直接运行clean会删掉整个build目录，但是运行cleanJar只会删掉JAR文件，运行cleanResources只会针对性的删掉build/resources目录。

下面是运行cleanResources后的目录结构：

```
build
 | ——— classes
 |      |___main
 |              |___org
 |                     |__gradle
 |                           |___poetry
 |                                  |___ PoetryEmitter.class
 |__ libs
       |__ java-build-with-resources.jar
```

这个简单的例子可能没展出什么高级的行为 — 小心的使用rm命令也能做到这些 — 但是它演示了rules的强大，很多令人激动的可能性已经呼之欲出了。

创建Rules

让我们来创建一个ping任意服务器的rule，然后将结果存起来。我们将结果通过log输出，同时将其存在task内的一个扩展变量里，以便其他task可以根据ping的结果改变自己的逻辑。

ping在Java里只是一行操作（往广了说，在Groovy里也一样），但是我们不想为每一个要ping的服务器都写一个专门的task。再者，我们也不想写太长的代码来处理不同情况的错误。这是那种常见的刚开始很简单，随着时间慢慢变复杂的例子。为了让复杂度可控，我们把代码都放在了同一个地方，没有让其扩散到很多个task中。见例3-7。

例3-7。一个检测任意服务器http到达性的rule

```
ext {
  pingLogDir = "${buildDir}/reachable"
}

tasks.addRule('Rule Usage: ping<Hostname>') { String taskName ->
   if(taskName.startsWith('ping')) {
   task(taskName) {
      ext.hostname = taskName - 'ping'
      doLast {
          def url = new URL("http://${ext.hostname}")
          def logString
          try {
              def pageContent = url.text
              // 使用正则表达式对pageContent进行过滤
              logString = "${new Date()}\t${ext.hostname}\tUP\n"
              ext.up = true
          } catch(UnknownHostException e) {
              logString = "${new Date()}\t${ext.hostname}\tUNKNOWN HOST\n"
              ext.up = false
          } catch(ConnectException e) {
              logString = "${new Date()}\t${ext.hostname}\tDOWN\n"
              ext.up = false
          }
          file(pingLogDir).mkdirs()
          file("${pingLogDir}/ping.log") << logString
      }
    }
  }
}
```

使用HTTP检测可达性要伴随一些对返回消息体的解析操作，这个使用Groovy的正则辅助很容易做到。现在这个类已经有点长了，我们得找一个更好的方式来管理这些命令式的代码。

处理命令式的Rules代码

我们的rule定义已经有18行代码了，从代码的可重构和可维护方面来说这已经太长了。这段代码还不能测试，截止写这本书的时候，大多数构建工具还不支持在Gradle build文件中直接开发Groovy脚本，也不支持使用常见的Groovy类。解决方法是使用Rule接口。

你可能还记得我们的例子工程里有时会有一个buildSrc目录，里面放着一个Groovy工程。如果这个Groovy工程存在的话，它会先于主工程构建，构建生成的类会加入到主工程build script的classpath中。这是我们可以创建和测试Rule类的基础。

例3-8是一个基础的Rule的骨架。Rule接口包含两个方法: 一个getter方法，用来获取这个rule的描述，会被tasks任务用来记录当前构建中都有哪些可用的rule， 和另一个叫做apply()的方法，是用来创建task的地方。这个类写在我们示例工程的buildSrc/src/main/groovy/org/gradle/examples/rules目录下。

例3-8。一个最简单Rule

```
import org.gradle.api.Rule

class HttpPingRule implements Rule {
  String getDescription() {
    'Rule Usage: ...'
  }
  void apply(String taskName) {
  }
}
```

现在我们的build文件一下变得简单了不少。所有命令式的代码都被移走放到了一个源文件中，于是我们的build文件就变成了例3-9这样。

例3-9。Rule定义在类中后build文件小了很多

```
import org.gradle.examples.rules.HttpPingRule
tasks.addRule(new HttpPingRule(project))
```

运行gradle tasks就能看到我们的task rule的定义（列在静态task之后）。

我们现在来填充一下apply()方法以便测试。（记住，这个例子task rule作用是捕获不存在的task，然后根据匹配规则动态创建相应的task）。为apply()添加功能后如例3-10所示：

例3-10。 在apply()方法中创建task的Rule类

```
import org.gradle.api.Rule

class HttpPingRule implements Rule {
    def project
    HttpPingRule(project) {
        this.project = project
    }
    String getDescription() {
        'Rule Usage: ping<Hostname>'
    }
    void apply(String taskName) {
        if (taskName.startsWith('ping')) {
            project.task(taskName) {
            ext.hostname = taskName - 'ping'
            doLast {
              println "PING ${ext.hostname}"
            }
        }
    }
}
```

可以再进一步完善这个类，将println()调用换成网络检测代码，以使它具备比较完整的功能。此时，因为它是用类实现的，所以我们可以对它做针对性的测试了。

通用Rules

到目前为止我们写的rule都是针对task的。但实际上，rule API并不只是针对task级别，而是针对所有有名称的可扩展对象级别的。也就是说，Gradle中任何有名称的元素集合 — SourceSets、configurations和自定义扩展对象等 —— 都可以通过rule创建。

总结

当你对自己的构建代码有完全的控制权，或者你在写自己的插件时，通常最简单的增加新功能的方式是直接将代码写在自定义插件里。但是有时当你看不到build源码，或者你不方便修改这些源码时，Gradle通过引入面向切面和元编程的思想，为你提供了hook和rule这两种强大的机制协助你实现自己的想法。

第四章

依赖管理

无论是对于Java、Ruby还是其他工程，依赖都是一个很大的挑战。涉及到一个东西依赖另一个东西，就意味着你得接受这个东西不可用，或者很难获取，或者不能按照预期工作这些风险。程序员们对这些风险都有着深刻的感受。

依赖看起来很令人头疼，实际上，Maven受到的大部分不应得的批评都是和依赖管理有关，但是对于一个复杂的生态系统这是不可避免的，但同时这也反映了这个系统正在尝试以合理的方式分割他们的工作。因此，Gradle拥抱依赖管理。

我们将主要从Java的角度研究依赖管理，因为Java开发者曾提出了大量的依赖管理问题，同时也有效的解决了很多。早年由于Java的开源衍生出了成千上万的独立模块，一个模块很快会依赖很多其他的模块，而同时这些被依赖的模块又会依赖到很多别的模块。

当然，Java依赖管理的解决方案也可以应用于其他基于JVM的语言，比如Groovy和Scala就可以原封不动的使用本书所讲的技术。

依赖管理是什么

你可能曾经不假思索的将某个目录(一般是lib)的所有jar包加到工程的编译路径中。这样做随着时间流逝和项目发展不可避免的会出问题，而且随着项目越大，删除或者修改某一个依赖的成本会越大，你越来越难搞清楚变更依赖后会产生什么影响。通过将依赖定义成你软件构建的一部分，这些依赖才会变得容易理解和分析。直接添加lib目录里的jar包本身没有问题，问题是出在盲目的使用这些jar包和这些jar包之间有着不明不白的关系上。依赖管理解决了这个问题。

通常我们会把“依赖管理”和自动下载直接依赖或者间接依赖库(依赖的依赖)划等号。这是依赖管理的好处之一，但不是全部。通过声明和组织依赖，Gradle可以利用这些信息自动处理依赖问题，比如下载依赖库和检测间接依赖冲突等。

Gradle拥抱依赖管理，并对依赖自动化提供了出色的支持。

概念

在Gradle中，你可以通过DSL声明依赖。（例4-1）。

例4-1。声明依赖

```
repositories {
    mavenCentral()
}
configurations {
    compile
}
dependencies {
    compile 'org.springframework:spring-core:3.0.5'
}
```

上面的例子出现了Gradle依赖管理中的三个概念：configurations(compile)、dependencies(org.springframework:spring-core:3.0.5)和repositories(mavenCental())。一个“configuration”代表一组依赖，Gradle构建没有它也可以。一个“repository”是一个依赖源（即获取依赖的仓库）。依赖通常通过指定属性的方式声明，通过这些属性Gradle知道怎么从仓库中找到这些依赖。

在一个典型的Gradle构建中，configuration声明通常是隐式的，插件将configuration隐藏起来并打理好了一切。然而如果没有透彻的理解configuration的话，你就不可能完全理解Gradle的依赖管理。因此我们从configuration的概念开始讲起。

Configuration

Configuration是一个包含依赖并且有名称的容器，它是FileCollection的一个具体类型。回想下之前讲的，FileCollection只是一个文件集合的描述，当需要用到这些文件时（例如获取文件操作）这些文件才会被加载到一个具体的列表中。Configurations也是一样的原理，除了它不是一个普通文件集合的描述，而是“依赖”的描述，这些依赖可能存在于本地，也可能存在于网络上或者其他抽象的地方。不同的依赖类型会以不同的方式解析，这是configuration的基本作用，同时它也提供查询声明的依赖和自定义解析过程等功能。

Java插件会引入以下6种configuration

- archives
- default
- compile
- runtime
- testCompile
- testRuntime

【提示】archives和default这两个configuration实际上是由base插件创建的，base插件会被Java插件以隐式的方式应用到构建中。后面的讨论中我们会刻意忽略掉这个细节。

compile configuration包含了编译代码所需要的所有依赖。当Java插件的compileJava task调用javac编译器时，它必须为之提供一个classpath，而在一个拼凑命令行参数的构建中（典型的可能像Ant构建那样），你必须得手动收集编译器用来解析代码import语句的所有JAR文件，现在可喜的是通过compile configuration指定编译时依赖也能完成这样的工作了，因为compileJava task所需的classpath早早就已经在compile configuration中组织好了(例4-2)。

例4-2。声明default configuration

```
configurations {
  compile
  runtime
  testCompile.extendsFrom('compile')
  testRuntime.extendsFrom('runtime', 'testCompile')
  default.extendsFrom('runtime')
}
```

这里注意到，自动化测试的依赖和主代码稍有不同，它除了需要主代码的所有依赖外，可能还需要其他一些东西：一个单元测试框架、一个模拟框架、一个测试友好的数据库驱动或者其他同类的东西。不止这些，自动化测试还要依赖于主代码的编译输出。Java插件通过testCompile来管理这些依赖。当compileTestJava task运行javac编译器时，它会通过这个configuration获取到所需的classpath。

通常，工程可能会依赖一些在编译时不需要而在运行时才会用到的模块。常见的例子是JDBC驱动，代码在编译时依赖的是标准库提供的接口，在运行时才会使用第三方提供的具体实现。Java插件提供了runtime和testRuntime可以用来满足这个需求。这两个configuration用于最终打包和执行测试，不会加到编译期classpath中。另外，runtime configuration要包含compileJava task的输出，因为运行时会用到工程中的所有类。

default configuration很少用到，但是会参与工程间依赖中。当一个工程依赖于另一个工程，Gradle的默认行为是包含所有被依赖工程的configuration文件到依赖工程中。default默认继承自runtime。

【提示】Gradle的war插件提供了providedCompile和providedRuntime configuration，可以达到和Apache Maven的provided同样的效果。具体见Gradle用户指南。

configuration继承

当我们看到例4-2中default继承自runtime, 我们可能一下就明白了这个机制，但是这里还是要解释一下。要弄清楚这里面的细节，让我们来分析一下testCompile是怎么为compileTestJava task提供依赖的。Gradle在编译单元测试代码的时候，除了需要用到自身的一些依赖（例如，产品模拟框架和测试框架等），还要用到主代码的所有编译时依赖，准确的讲，testCompile是compile的超集，因此testCompile可以为compileTestJava提供所需的所有依赖。Gradle支持一个configuration继承自另一个configuration。

你可以在例4-2中看到继承语法。调用configuration的extendsFrom方法意味着它会包含被继承configuration的所有文件，外加自己显式声明的文件。

configuration之间的”extends”关系是可以改的。如果因为某些原因你可能不想runtime继承自compile了，你可以简单的将它从extendsFrom集合中移除掉（例4-3）。

例4-3。修改继承关系

```
configurations {
  runtime.extendsFrom.remove(compile)
}
```

模块依赖

现在我们对依赖管理已经有个大致的了解了，让我们把注意力集中到一种最常见的依赖上：模块依赖。模块就是其他工程编译出的JAR包。各种Java开源工程就是很好的例子，像Commons Logging，Hibernate或者Spring Framework等[注1]。

模块一般由三个字段来标识：group、name和version。group是指负责这个模块的组织，一般(并非总是)用反过来的域名标识（像org.apache.solr和org.springframework，或者特殊的像junit)。name是指这个模块的唯一名称，通常和工程名一致(像solr-core、spring-web或者junit)。release是指这个模块发布时工程的版本号(像1.4.1、3.0.2或4.8)。这三个元素可以以冒号分隔组成一个可以标识唯一模块的字符串（见例4-4）。

例4-4。使用group:name:version的形式声明依赖

```
dependencies {
  compile 'org.apache.solr:solr-core:1.4.1'
  compile 'org.springframework:spring-core:3.0.5'
  testCompile 'junit:junit:4.8'
}
```

或者还可以用Groovy map的形式表达一个依赖(map的key要显式的写出来)，参考例4-5. 注意只有name字段是必须要有的，group和version可有可无。

> 注1：Hibernate和Spring实际上都是由若干个单独模块组成的。这些模块之间可能有依赖，也有可能是完全独立的（只依赖了框架以外的模块）。

例4-5。使用Groovy map语法声明依赖

```
dependencies {
  compile group: 'org.apache.solr', name: 'solr-core' version: '1.4.1'
  compile group: 'org.springframework', name: 'spring-core', version: '3.0.5'
  testCompile group: 'junit', name: 'junit', version: '4.8'
}
```

默认情况下，Gradle的依赖是传递性的。依赖传递在Maven仓库一节56页有详细说明，但是现在你只需要知道当一个模块依赖了另一个模块，Gradle会发现这些“依赖的依赖”，并下载好它们。这为我们节省可观的时间，但是有时也会引发问题。如果你依赖了一个版本为1的模块A和一个版本为2的模块B，模块A同时又依赖了版本为3的模块B，你可能就不想Gradle帮你决定使用哪个版本的依赖。如果处理不好，错误版本的JAR可能会加进你compile或者runtime的classpath， 大多数Java开发者知道这将多么令人崩溃。

幸运的是，你可以通过三种方式来干预传递依赖的解析：1，你可以禁掉所有的传递依赖(例4-6)，2，你可以禁掉某一个传递依赖(例4-7)，3，你可以强制让两个有版本冲突的模块中的一方胜出(例4-8)。让我们看一下它们各自的语法。

例4-6。下载Spring Core 3.0.6, 并禁止所有的传递依赖

```
dependencies {
  compile('org.springframework:spring-core:3.0.6') {
    transitive = false }
}
```

例4-7。从Sring Core的传递依赖中排除掉commons-logging

```
dependencies {
  compile('org.springframework:spring-core:3.0.6.RELEASE') {
      exclude name: 'commons-logging'
  }
}
```

例4-8。强制所有Spring Core使用3.0.6版本

```
dependencies {
  compile('org.springframework:spring-core:3.0.6.RELEASE') {
    force = true }
}
```

动态版本

只要仓库不出问题，声明一个特定的版本号，比如3.0.6，Gradle可以确保每次构建时下载的都是相同的模块，这一点是依赖管理的基石。但是有时我们想做一些微调，比如我们想让Gradle每次都下载某一个模块最新的版本。

使用动态版本就可以做到，但是使用动态版本有一定风险，因为每次模块发布新版都会导致生成不一样的产品 — 即使你的源码一点没动。构建高手们可能还是会用，它们有把握，当然，Gradle对此完全支持（参考例4-9动态版本的使用）。这个功能和依赖缓存有很重的交互，在使用动态版本前你应该先理解好这个缓存机制。

例4-9。声明动态依赖的两种语法

```
dependencies {
  compile 'org.springframework:spring-core:3.0.+'
  testCompile 'junit:junit:4.8+'
}
```

文件依赖

采用Maven风格声明依赖的方法很实用(group:name:version)，Gradle会从对应仓库中(像Maven Central)解析到正确的模块。然而，即使在一个大多数依赖都用这种方式管理的常规工程里，你还是能找出一些你想用但是仓库中又没有的模块。这些可能是一些第三方的JAR包，只能通过直接下载获取，或者是一些你不方便部署到内部仓库的二进制模块。无论什么原因，这些模块都可以通过Gradle提供的文件依赖来管理。

由于管理依赖的configuration实际上是一个文件集合，所以很容易通过暴力手段直接添加文件进去。用这种方式，你可能会在工程根目录下创建一个lib目录，把所有JAR包放到里面，然后像例4-10这样直接添加文件到依赖中。

例4-10。显式添加一个本地文件依赖

```
dependencies {
  compile files('lib/hacked-vendor-module.jar')
}
```

当然，这种方式会一发不可收拾，你很快会发现你的lib目录里出现越来越多的模块，好像又回到了Ant构建中遇到的情况。所以并不推荐这种方式，实际上你可以通过一句声明包含所有的文件，即使这些文件是在lib目录的子目录里（例4-11）。

例4-11。递归依赖lib下的所有文件

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

工程依赖

多工程构建中build文件之间可能有依赖关系，所有build文件都包含了这个工程的某些configuration，它们无一例外都会在构建期间被评估。工程图中的Project对象之间也有相互关系。Gradle没有对这些关系做任何规定，但是提供了一种语法可以用来表达这种关系。当你要描述两个工程之间的关系时，你可以说一个工程“依赖”于另一个工程。

查看这个 多工程示例，会发现他是由一个顶层的命令行驱动工具工程，加上一个提供文本内容的和另一个提供编码服务的子工程组成。在这个多工程构建中，顶层工程依赖于那两个子工程，子工程之间没有依赖。摘取顶层工程build文件的一分部分如例4-12所示。

例4-12。声明对子工程的依赖

```
dependencies {
  compile project(':codec')
  compile project(':content')
}
```

从例4-12看到，顶层工程有两个编译时依赖：':codec'和':content'。当然，一个工程不是一个文件集，但是一个依赖最终要转换为一个文件集以便被加入到某一个具体的configuration中，比如compile或者runtime。当你像示例程序那样依赖于一个工程，Gradle会默认为你使用default configuration，它会包含被依赖工程所有编译输出文件和所有编译时与运行时模块依赖。default configuration在49页Configurations一节有细讲。

当然，你可能不想包含default的所有文件。你可能为子工程专门设计了一个依赖configuration用来发布特定的文件到依赖它的工程中。例4-13中，project()方法接收了一个map — 包含一个指定工程路径的键值对和一个指定依赖configuration的键值对。

例4-13。显式声明依赖子工程的哪个configuration

```
dependencies {
  compile project(path: ':codec', configuration: 'published')
  compile project(':content')
}
```

内部依赖

Gradle依赖DSL提供了一些很方便的方法帮助你扩展Gradle自身。它们是gradleApi()和localGroovy();

如果你正在写自定义task或者自定义插件，你将会直接使用到Gradle 的API，同时会声明一个Gradle的编译时依赖。这种依赖和你平时调用其他API使用到的编译时依赖是相似的。

但是由于你导入的类正好和解释运行build脚本用到的类在同一个包下，所以你可以直接访问这些类，不用额外声明外部依赖（例4-14）

例4-14。依赖Gradle API

```
dependencies {
compile gradleApi()
// other dependencies...
}
```

同样，你可以使用任何一种JVM语言扩展Gradle脚本，很多Gradle开发者偏爱使用Groovy。用Groovy写的插件和自定义任务和Gradle build文件有着一样的语法，作为一款高效的动态语言，Groovy将开发者服务的非常周到。

Gradle的Groovy插件使得写构建脚本变得简单了一些，但是这也需要一项额外的工作：配置对Groovy某一个版本的依赖（例4-15）。通常通过调用groovy方法并传入某一个Groovy版本的依赖标识就可以完成。

例4-15。典型的Groovy构建配置

```
apply plugin: 'groovy'

// Repository declaration
dependencies {
    groovy 'org.codehaus.groovy:groovy:2.0.5'
   // other dependencies...
}
```

如果你写的代码是在扩展Gradle本身，就不需要特别声明Groovy的版本。因为Gradle build文件本身就是Groovy脚本，只要集成了Gradle环境都已经有一个相同版本的Groovy了。你的自定义task、插件或者其他Gradle扩展都可以在这个版本的Groovy上运行。Gradle提供了一个localGroovy()方法让你可以使用到这个默认版本的Groovy，这样你就不用显式的声明Groovy外部依赖了（见例4-16）。

例4-16。依赖Gradle内置的Groovy版本

```
apply plugin: 'groovy'

dependencies {
    groovy localGroovy()
    // other dependencies...
}
```

仓库：依赖解析

按照定义，依赖，是一种构建之外的东西。一旦声明，Gradle就会从某些地方获取它们。在Gradle中，存放依赖的地方称作“仓库”。

每一个Gradle工程内部都有一个仓库列表。这个列表默认是空的，我们可以通过*repositories*代码块为它添加内容。Gradle在执行阶段使用这些仓库获取依赖并将其存到缓存里。Gradle目前支持三种仓库类型：Maven仓库、Ivy仓库和静态目录。我们会依次说明这些仓库，同时将注意力放在它们的配置DSL上。

Maven仓库

Maven最有价值的创新之一就是Maven Central仓库。它不仅作为一个公共的仓库容纳了成千上万Java模块供开发者自由访问，而且开放了它的组织和访问协议。同样，世界上还有很多其他的Maven风格的仓库和很多公司内部的仓库，Gradle都将它们视作一等公民。

像模块依赖一章所讲的那样，Maven仓库中的模块通常用三个字段标识：group、name和version。这种自描述的格式起初是按照Maven格式引入到Java生态系统的。除此之外，模块也可以用*classifier*和*type*标识。classifier通常用来区分模块是依赖库、JavaDoc还是源码归档。type用来指示这个模块是Jar还是其他格式。classifier和type经常被省略掉，但它们也是Maven格式所支持的一部分。

Maven仓库的主要价值不是它提供了标准的从网络存取这些模块(库、源码和文档)的机制，而是它很好地描述了这些模块之间的依赖关系。每一个Maven标识都对应一个叫做POM(Project Object Model —工程对象模型)的XML描述文件。POM文件包含了关于这个模块的各种元数据，尤其是它依赖的其他模块列表。默认情况下，Gradle会递归式的从仓库和本地缓存中获取这些依赖。

由于Maven仓库无非是一个提供POM文件和模块并具有固定目录结构的网站，因此声明一个Maven仓库只需要告诉Gradle这个仓库的URL就行了。最常见的Maven仓库是Central仓库，地址是*http://repo1.maven.org/maven2/*，为避免你直接写死这个地址，Gradle提供了一个更简单灵活的方法声明仓库（见例4-17）。

例4-17。声明Maven Central仓库

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

Central可能是最常用的Maven仓库，但绝不是唯一选择。网络上存在很多开源的Maven仓库，它们更多是用在企业的私有网路里（自有构建系统），Sonatype的Nexus和JFrog的Artifactory就是这样的两个例子，它们可以用来部署私有的Maven仓库。声明这样的仓库只需要简单地提供一个URL即可（例4-18）。

例4-18。声明内部Maven仓库

```
repositories {
  maven {
    url = 'http://download.java.net/maven/2'
  }
  maven {
    name = 'JBoss Repo' //optional name
    url = 'https://repository.jboss.org/nexus/content/repositories/releases'
  }
}
```

Maven仓库具有良好的URL命名规范，这点让它很容易使用。实质上，它还是一个提供POM文件的网站。在某些情况下我们可能想从一个网站上获取POM（比如一个集中式的企业仓库），但是从另一个仓库镜像里下载依赖（见例4-19）。我们这么做的动机可能是想减轻带宽压力，或者加快下载速度。无论基于何种原因，Gradle都可以满足你的需求，同时让这件事变得很简单。

例4-19。从不同的URL获取模块和POM文件

```
repositories {
    // Overriding artifacts for an internal repo maven {
    url = 'http://central.megacorp.com/main/repo'
    artifactUrls = [ 'http://dept.megacorp.com/local/repo' ]
  }
  // Obtain Maven Central artifacts locally
  mavenCentral artifactUrls: ['http://dept.megacorp.com/maven/central']
}
```

Maven会将所有依赖缓存到著名的~/.m2目录，这个目录也被称作本地Maven仓库。Gradle会维护好缓存机制，但是有时你想自己使用本地Maven缓存来解析依赖。最常见的案例是你想以Maven的方式依赖一个自己还在开发中的模块 — 这个模块尚未成熟发布。让Gradle基于本地Maven缓存解析依赖并不难。见例4-20。

例4-20。从本地Maven缓存(~/.m2目录)获取依赖

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

可变模块

Maven仓库同样可以存放快照，也就是Gradle术语里的可变模块。可变模块的版本一直处于变更状态，因此不能固定于某一个版本号。当你的模块正在开发中，你可能时不时会发布一个新版，这时就可以使用这种方式。可变模块的管理方式和正常模块基本一样，但是因为它们很容易变，因此在缓存机制方面会有所不同。

正常情况下，Gradle一但缓存了某个模块就不会再去下载它，因为固定版本号的模块是不可变的。然而可变模块是会变化的，默认情况下，可变模块会每隔24个小时重新下载一次。你可以通过配置全局的”依赖解析策略”来改变这一值。

Ivy

Apache的Ivy可用于扩展Ant构建，它提供了Maven风格的模块管理机制。它没有Maven仓库应用广泛，但在一些适配企业级Ant构建方面还有在用。

大体功能上，Ivy和Maven非常相似。Ivy的模块也是通过group、name和version拼接串来标识的。仓库地址可以是HTTP URL或者文件系统。模块的元数据使用ivy.xml文件表达，可以包含传递依赖的信息。同样的，声明一个Ivy仓库也很简单(例4-21)。

例4-21。使用默认风格声明一个Ivy仓库

```
repositories {
  ivy {
    url 'http://build.megacorp.com/ivy/repo'
  }
}
```

Ivy和Maven的不同之处是它不是所有仓库都遵循同一种目录结构。Maven仓库固定为group、name和version三级结构，但是Ivy仓库的结构是多种多样的。如果我们没有提供模块定义或者Ivy映射(见例4-21），Gradle会假设其使用Maven结构。当然我们也很容易自定义仓库结构。

例4-22展示了模块和ivy.xml基于同一个URL，但是存放在不同的路径上的情况。用方括号括起来的内容表示会替换为对应的依赖描述字段，具体映射见表4-1。

表4-1。Ivy字段和默认依赖描述字段之间的映射

Ivy字段	依赖字段
[organisation]	group
[module]	name
[revision]	version
[artifact]	name
[ext]	type
[classifier]	classifier

例4-22。声明特殊目录结构的Ivy仓库

```
repositories {
  ivy {
    url 'http://build.megacorp.com/ivy/repo'
    layout 'custom', {
        artifact "artifacts/[module]/[revision]/[artifact].[ext]"
        ivy "ivy/[module]/[revision]/ivy.xml"
    }
} }
```

此外，Ivy仓库还可以将ivy.xml和模块本身存储在不同的URL里。如果没有一个共用的基础URL就可以使用例4-23所展示的语法。

例4-23。使用不用的基础URL声明Ivy仓库

```
repositories {
  ivy {
    artifactPattern "http://build.megacorp.com/ivy/repo/artifacts/[organisatio
n]/ [module]/[revision]/[artifact]-[revision].[ext]"
    ivyPattern "http://build.megacorp.com/ivy/repo/ivy-xml/[organisation]/[mod
ule]/ [revision]/ivy.xml"
  }
}
```

仓库鉴权

大多数在线仓库一开始就是为开源模块服务的，因此任何人都可以自由访问无需任何认证。同样的，大多企业仓库都是基于可信赖的内部网络建成的，公司员工可以自由访问。但是有时基于安全考虑我们希望仓库具有基本的用户名和密码验证功能。Maven和Ivy都支持这个功能，同时Gradle有一个通用的配置DSL用来提供仓库鉴权（见4-24）。

例4-24。增加仓库鉴权声明（不太安全的方式）

```
repositories {
  maven {
    url = 'http://central.megacorp.com/main/repo'
    credentials {
      username 'repouser'
      password 'badly-protected-secret'
    }
  }
}
```

这样简单声明的问题是它把密码以明文的方式暴露在了build脚本里，同时加到了代码版本控制。这显然不是一个谨慎的密码管理方式。更好的办法是将密码存在环境变量或者外部属性文件中。Gradle在启动时会自动读取gradle.properties文件，并将里面的属性值应用到build脚本里。这是一个方便的存储密码的方式，当然，前提是这个文件不需要加入到代码版本控制。参见例4-26。

例4-25。在gradle.properties文件里存放密码

```
megacorpRepoPassword=well-protected-secret
```

例4-26。增加仓库鉴权声明，并从属性文件中读取密码

```
repositories {
  maven {
    url = 'http://central.megacorp.com/main/repo'
    credentials {
      username 'repouser'
      password megacorpRepoPassword
    }
  }
}
```

gradle.properties可靠的前提是它必须不在版本控制中，比如将文件加入到.gitignore文件或svn:ignore属性，或者使用其他版本控制工具类似的机制。如果这个文件必须包含一些其他内容导致你无法忽略这个文件，你可以创建一个自定义属性文件，然后使用**java.util.Properties** API和Groovy相关语法读取。

静态依赖

大多数当代的开源项目或者企业级构建都使用Maven风格的仓库来管理依赖，但是也不是所有项目都这么做。一个老的基于Ant的构建可能使用一个包含了所有JAR包的lib目录来静态管理这些依赖。新的设计当然不推荐这样的做法，但是对于老的设计也不忙对其依赖管理做重构，应该优先采用平滑迁移的方式，谨慎地一个一个处理遇到的问题。

我们一般会尽可能采用新的依赖管理方式，但是有时也因为一些编程上的原因不得不会退到老的方式。例如，你需要在字节码级别hack一个已经不被支持的三方JAR包，或者你对一些外部模块的构建有着高度的定制，因此你更倾向以源码的方式控制这些JAR包 — 这种场景最有可能是因为你的内部仓库或者发布机制还没有建好。

无论你是因为老的技术原因还是编程方面的考虑，Gradle都已经有所准备。Gradle支持从一个本地目录中解析依赖，就像这个目录是一个Maven或者Ivy风格的模块一样。例4-27展示了如何声明这种目录式仓库。

例4-27。以目录方式声明一个仓库

```
repositories {
  flatDir dirs: "${projectDir}/lib"
}
```

这句声明会告诉Gradle去工程目录下的lib目录里寻找JAR，然后用10多年前Ant构建的套路去解析这些文件。但是这里也提出了一个问题，这些文件没有group、name和version，Gradle是怎么确定依赖标识的呢？答案是Gradle使用[group]-[name]-[version]-[classifier]. [type]公式。从表4-2的一些例子可以窥探一二。

表4-2。JAR包文件名和依赖标识之间的映射规则

依赖标识	文件名

‘org.apache.solr:solr-core:1.4.1’	org.apache.solr-solr-core-1.4.1.jar
name: ‘commons-codec’, version: ‘1.6’	commons-codec-1.6.jar
name: ‘ratpack-core’, version: ‘0.8’, classifier: ‘source’, type: ‘zip’	ratpack-core-0.8-source.zip
name: ‘commons-logging’	commons-logging.jar

如你所见，目录中的文件并不能很好的匹配常规Maven仓库的的“group:name:verion”格式。如果你的文件名有版本号，你一般可以使用group和name组合的方式声明依赖，如果没有版本号，就只能使用name了，Gradle会去仓库目录中尝试解析这个文件。

偶尔，你可能有多个静态目录供Gradle读取。声明多个目录的语法可以参见例4-28。这个例子还展示了如何为flatDir仓库指定一个名字，方法同Maven和Ivy相应的声明。

例4-28。声明多目录仓库

```
repositories {
  flatDir name: 'staticFileRepo'
          dirs: [ "${projectDir}/api-libs",
                  "${projectDir}/framework-libs" ]
}
```

构建脚本依赖

大多数依赖和仓库声明都考虑的是怎么样才能构建代码 — 编译代码需要什么模块，执行测试需要哪些模块，运行代码又需要哪些模块，等等之类的。实际上，由于Gradle构建属于Groovy编程范畴，我们也可以为构建脚本本身引入外部的依赖库。

我们来考虑一种场景，假如我们的工程中有一组Mardown文件，我们写了一个task将它们转换为HTML文件。这是Copy task的完美应用场景，只需要添加一个filter即可，但是这个filter会有大量的任务要做，它基本要变成一个功能完备的Markdown解析器，同时还得是一个HTML渲染器。显然，我们不想自己来做这件事。

幸运的是，MarkdownJ工程可以用来完成这项工作，但是我们得让MarkdownJ的相关类在构建脚本中可见，以便构建脚本可以正常解释和执行（注意这次不是为了编译或者执行工程代码），我们可以像例4-29这样做。

例4-29。为构建脚本本身添加依赖模块并使用

```
buildscript {
  repositories {
    mavenCentral()
  }
  dependencies {
    classpath 'org.markdownj:markdownj:0.3.0-1.0.2b4'
  }
}

import com.petebevin.markdown.MarkdownProcessor

class MarkdownFilter extends FilterReader {
  MarkdownFilter(Reader input) {
  super(new StringReader(new MarkdownProcessor().markdown(input.text))) }
}

task markdown(type: Copy) {
  from 'src/markdown'
  include '*.md'
  into 'build/labs'
  rename { it - '.md' + '.html' }
  filter MarkdownFilter
}
```

注意，buildscript内的repositories和dependencies声明和之前章节里讲的方式是一样的。唯一不同的是它使用了一个新的依赖configuration：classpath。这个configuration只会在 buildscript中用到。它表示在执行脚本时依赖模块中的类可以被classloader加载，这正是我们所需要的。buildscript代码块之后导入了MarkdownProcessor类，这个类会在配置task filter时用到。我们可以采用同样地方式导入任意我们想要的模块。

我们还可以用这个方法引入外部插件。像在第二章里所讲的那样，插件只是一个JAR包，包含了形成插件API的相关代码。想要应用一个插件，这个插件必须要被加入到构建脚本自身的classpath中。如果一个插件被正常打包并部署在像Central这样的仓库里，我们就可以依照例4-30所示的方法引入这个插件。

例4-30。从仓库中引入并应用一个外部插件

```
buildscript {
  repositories {
    mavenCentral()
  }
  dependencies {
    classpath 'com.augusttechgroup:gradle-liquibase-plugin:0.7'
  }
}
apply plugin: 'liquibase'
```

依赖缓存

可以运用简明的语法声明并解析依赖是一回事，在保持这一点的基础上又能具备很快的执行速度是就是另一回事了。为了达到这一目标，Gradle提供了复杂的依赖缓存机制。

像Ivy和Maven这样的仓库都为我们暴露出了元数据(例如，pom.xml和ivy.xml文件)和模块本身(例如，JAR文件、源代码和Java文档)。作为一个重要的创新，Gradle将元数据和模块分开进行存放，这么做避免了很多非重复性构建的问题。

依赖缓存的具体实现是将下载的文件按照标识符分组(以group:name:version的形式)。但是，这个简单的方法会漏掉不同版本号的同一模块存放在不同仓库的情况。显然，这么做是存在风险的，Gradle构建过程中可能会提高其中一个模块的版本号，导致意料之外的结果，但是这也是一种现实存在的情况。

Gradle通过group、name和version来缓存依赖元数据(pom.xml和ivy.xml)和仓库信息。说一个依赖被解析意思就是说这个依赖会被从仓库中下载并缓存。从内部仓库和从Central解析commons-codec:commons-codec:1.6依赖会涉及不同的操作。这会增加构建中的网络请求，但是只是针对元数据文件，只有在这个依赖模块需要被下载的时候它才会去下载。

Gradle为每个仓库对应一个缓存元数据文件，但是所有相同的依赖只会维护一份模块文件。模块以SHA-1哈希值的方式存储，所有元数据缓存在共享的本地存储中。在之前的例子中，内部仓库可能有一份comms-codec:comms-codec:1.6，Maven Central可能也有一份。如果两个都提供同一个JAR，而且具有相同的SHA-1哈希码，Gradle就只会下载一此，并且存放在磁盘上某一个地方。Gradle深度依赖哈希来优化下载性能，它会从仓库中下载数据量非常小的哈希值，如果它对应的模块已经存在于缓存中就会跳过下载。

有几个可以影响依赖缓存行为的命令行参数。–offline告诉Gradle禁止从网络上解析依赖。这个在你网络连接断开的时候会非常有用。–refresh-dependencies与之相反，强制Gradle从网络上重新解析所有依赖。但如果相同哈希的模块已经存在就不会去重复下载。你如果想让让可变依赖或者动态版本依赖模块更新到最新，这个就能派上用场。

配置解析策略

每一个依赖配置都有一个关联的解析策略。在大多数构建中，解析策略都是运行在幕后的，对开发者不可见，但是当出现依赖问题时就需要对其做一些配置了。目前为止有三种情况可以通过配置解析策略来满足：1，当出现依赖版本冲突时，2，强制使用某一个依赖版本，3，改变可变依赖和动态版本依赖的缓存过期策略。这三种情况我们会依次介绍。

版本冲突失败处理

由于Gradle会解析传递依赖，两个独立的模块依赖于同一个模块时就有可能发生冲突。例如同一工程下有一个web表单处理模块依赖于commons-collections:commons-collections:3.2.1， 同时有一个老的三方企业集成模块依赖于commons-collections:commons-collections:2.1。 Gradle会检测依赖图中有着不同版本号的相同模块，然后默认选择使用最新的版本。但是企业版集成模块可能在Commons Collections 3.2.1下不能正确执行。于是你可能想自己来处理版本冲突问题。当然，这是可行的 — 你既可以通过全局配置，也可以针对单configuration配置。见例4-31。

例4-31。当检测到版本冲突时抛出错误并停止编译

```
configurations.all {
  resolutionStrategy {
    failOnVersionConflict()
  }
}
```

强制使用某个版本

在前例的基础上，如果你明确知道声明新版本依赖的目的是为了使用最最新的依赖元数据，代码本身都是兼容的，那么当版本冲突时你可以选择不报错，可以强制Gradle使用2.1版本的Commons Collections (例4-32)。

例4-32。强制使用某个版本的依赖

```
configurations.all {
  resolutionStrategy {
    force 'commons-collections:commons-collections:2.1'
  }
}
```

缓存过期

像commons-collections:commons-collections:3.2.1和commons-collections:commons-collections:2.1这样的依赖都是不可变的模块。这些模块可以发布到仓库，并且可以通过我们之前讲到的机制解析下载等。一旦下载，就立即被存入缓存，同时标识为唯一模块并无法再改变。然而，动态版本依赖和可变依赖是可以改变的，因此它们的缓存策略也比较容易定制。

动态和可变依赖模块同样会被缓存，但由于这种模块会过期，Gradle在某个时间周期后会就刷新缓存。默认两种依赖都是每隔24小时刷新一次，但是它们都可以通过配置设置任意时间（见例4-33）。

例4-33。设置依赖缓存过期策略

```
configurations.all {
  resolutionStrategy {
    cacheDynamicVersionsFor 1, 'hour'
    cacheChangingModulesFor 0, 'seconds'
  }
}
```

总结

由于在JVM的世界里的充分演进，Gradle具备了丰富并可高度定制的依赖管理支持。它充分支持Maven和Ivy仓库，同时支持老的静态目录依赖管理。此外，依赖缓存功能还解决了企业级构建的重复性问题。当然，更深入的定制光依靠依赖管理是不够的，可能还需要其他技术的配合。

后记

如果你通读了本书并尝试了书中的所有示例，你应该已经为Gradle打下了坚实的基础。书中覆盖到的文件操作、插件、hook和依赖管理等这些技术需要充分的理解和消化，只有这样你才能运用这些技术完成你的定制需求。

开发构建系统也是开发软件，如果你认为它只是把一堆脚本搓在一起，不算“真正”的软件开发，那你就大错特错了。如果你学会了本书中的技术，你就能完成针对你特定领域的高度定制的构建系统，这个系统可以支持你们团队创建的所有代码，并给它们一个高效合理的构建流程。

我希望本书能让你在Gradle之路上更进一步，让你可以为你的团队创建出优秀的软件构建和发布流程。这样的软件真的非常重要，因此不断深入地学习Gradle同样非常重要。我很高兴可以为你的终极目标贡献出一份绵薄之力。

关于作者

Tim Berglund是一个全栈多能并且富有激情的老师，它热爱编程，热爱分享，并且热衷于和大家一起工作。他是August Technology Group（一家专注于JVM相关技术的咨询公司）的创始人兼首席工程师。他是美国No Fluff Just Stuff巡回大会的国际发言人，是畅销书《O’Reilly Git Master Class》的共同出品人，同时也是Denver Open Souce User Group的联席总裁。他最近在研究自动化构建、非关系型数据存储和抽象思维(像如何创建蚁群式的软件架构这样的课题)。他和他的妻子还有他的三个孩子住在Littleton, Colorado。