# 第一部分 Gradle介绍

代码下载：<https://github.com/bmuschko/gradle-in-action-source>

本书的作者Benjamin，是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。他在最近加入了Gradleware，并成为Gradle开发团队的一员。

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。

Gradle从已有构建工具中吸取经验，取其精华，并进一步提升。

传统的面向Java项目的构建工具有两个：Ant和Maven，Ant是最早的工具

Ant和Maven构建逻辑都是使用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，随着构建脚本复杂的增加，维护构建代码就成为了噩梦。

自动化构建工具有哪些：经典的make脚本语言、曾经统治了Java构建世界的Ant、颠覆了Ant统治地位的Maven、即将颠覆Maven的Gradle。如果你在使用Java语言开发软件，那么Gradle将是你工具箱中必不可少的一个得器。

传统的面向Java项目的构建工具有两个：Ant和Maven

## 1、项目自动化介绍

### 1.1 没有项目自动化的生活

### 1.2 项目自动化的好处

#### 1.2.1 防止手动介入

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。

#### 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。

#### 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，其实，IDE也许只适用于某一操作系统。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。

#### 1.3.1 按需构建

按需构建的典型用例就是在用户自己的机器上触发构建。

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

#### 1.3.3 预定构建

例如，每天凌晨1：00点或者每隔15分钟。和所有的定时作业一样，预定的自动化任何通常运行在一个专用的服务器上。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。

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）。

### 1.4 构建工具

#### 1.4.1 什么是构建工具

#### 1.4.2 构建工具剖析

通常情况下，会使用脚本语言来表达构建逻辑，这就构建文件也叫作构建脚本的原因。

### 1.5 Java 构建工具

#### 1.5.1 Apache Ant

Apache Ant（AnotherNeat 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。

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的项目Ivy很好地集成，它是一个完善且独立的依赖管理器。

一个Ant构建脚本由三个基本元素组成：一个project、多个target和可用的task。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。

target是你想要执行的task的一个集合。

project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，必须且只能定义一个project

Ant缺点：相比其他更简明的定义语言，会导致构建脚本过于臃肿和啰唆。 复杂的构建逻辑会长又难以维护的构建脚本，当尝试用xml去描述类似if-then-else的逻辑语句时，它完全就成了一种负担。在没有Ivy的情况下，使用Ant很难管理依赖。

#### 1.5.2 Apache Maven

Maven为你的项目配置和行为提供了有意义的默认值。Maven选择约定优于配置思想。常常会太过限制，也许不适合你的项目需求。

Maven缺点：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。

具有依赖管理器。Maven Central是Java社区中最流行的二进制工作仓库。

#### 1.5.3 对下一代构建工具的需求

### 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。伴随着软件交付过程中的每一步——从源码编译到打包软件，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 一个一键式和故障安全的软件发布过程。

## 2、下一代构建工具：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。

Gradle是基于JVM构建工具的新一代版本。

### 2.1 为什么要用Gradle，为什么是现在

#### 2.1.1 Java构建工具的演变

Ant的第一个正式版本是在2000年发布的。

Maven 1发布于2004年7月。

Maven 2发布于2005年10月。

Gradle结合了其它构建工具的最佳特性：

Ant：灵活性、完全控制、目标链

Ivy：依赖管理

Maven：约定优于配置、多模块项目、插件扩展

Gant：在Ant基础上提供Groovy的DSL，所以不再需要使用xml来描述构建逻辑

Ant和Maven都是用XML描述的构建逻辑的

Gradle结合了其他构建工具的最佳特性：

* Ant
  + 灵活性
  + 完全控制
  + 目标链
* Ivy：依赖管理
* Maven
  + 约定优于配置
  + 多模块项目
  + 插件扩展
* Gant：在Ant基础上提供Groovy的DSL

#### 2.1.2 为什么选择Gradle

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时间来建立项目构建代码基础设施。Gradle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。

Gradle很早就有采纳者了，甚至在1.0版本发布之前。像Groovy和Hibernate这样流利的开源项目已经完全切换到Gradle并作为它们构建的支柱。

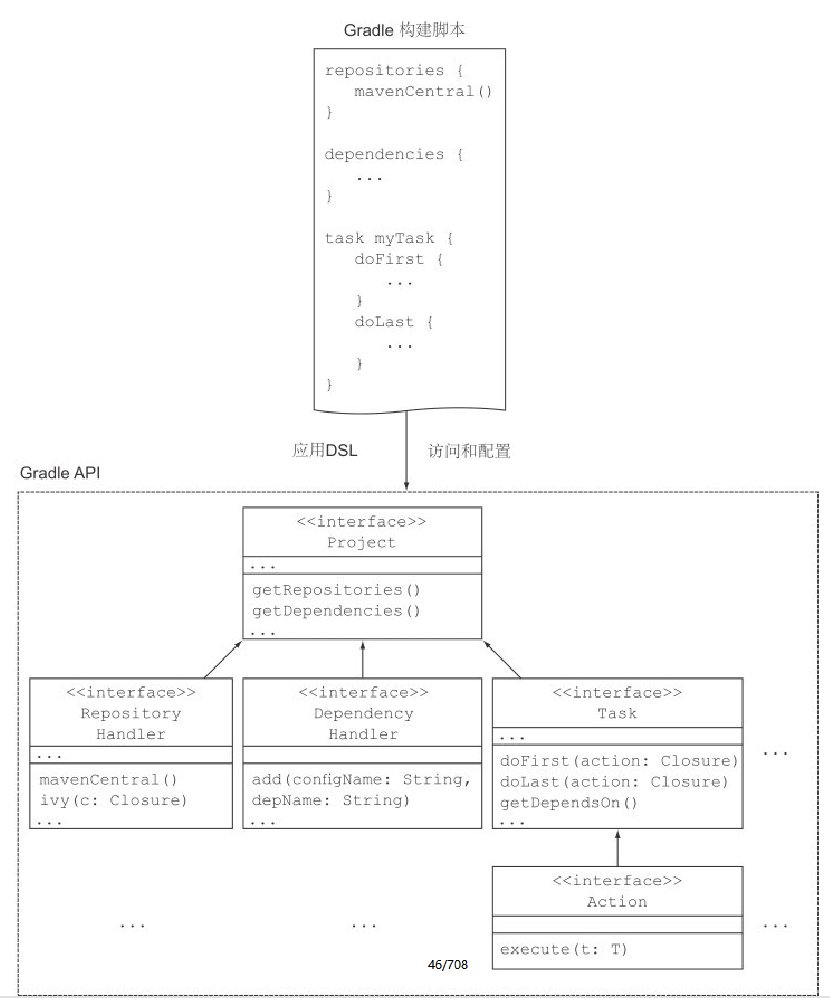
### 2.2 Gradle引人注目的特性集

Gradle的优点：

* Gradle就是Groovy
* 具有表达性的语言和强大的API
* 和其他构建工具的集成
* 社区驱动和公司支持
* 毫不费力的扩展
* 灵活的约定
* 鲁棒和强大的依赖管理
* 可扩展的构建

#### 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。如下图：**



一个构建脚本直接映射到Gradle API中Project类的一个实例。Gradle的API文档：<https://docs.gradle.org/current/javadoc/index.html> 。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Gradle脚本中每个元素都有一个与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构建语言指南**：<http://www.gradle.org/docs/current/dsl/index.html>

#### 2.2.2 Gradle就是Groovy

XML很难定义复杂的逻辑，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。

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

#### 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。

对于一个java项目，Gradle已经提供了默认的有意义的任务。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。而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的灵活性。Gradle进行了折中，Gradle即提供约定又给予你改变约定的能力

#### 2.2.4 鲁棒和强大的依赖管理

#### 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！

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。

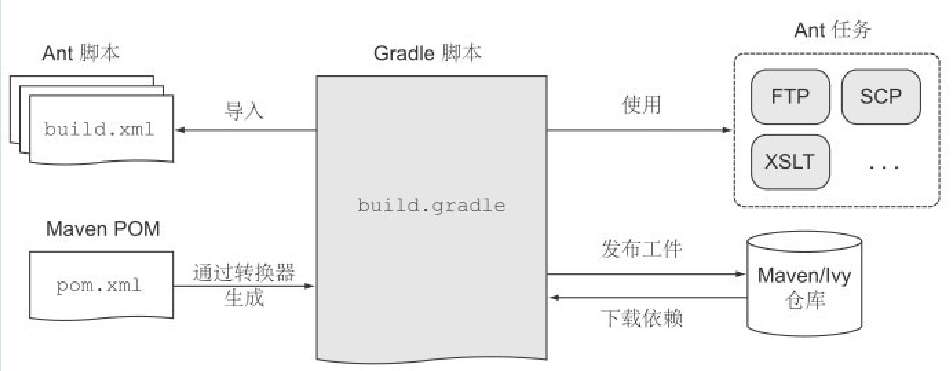
开发人员在开发过程中会多次运行构建。那意味着每次都启动一个新的Gradle进程，载入所有的内部依赖和运行构建逻辑。你会注意到，在实际执行脚本之前会有几秒钟的等待时间。要提高启动的效率，Gradle可以以守护进程模式运行。实际上，Gradle命令会fork出一个守护进程，它不仅会执行你的构建，而且会持续地在后台运行。后续的构建调用会交给这个守护进程以避免启动时的消耗。这样，你就会看到一个更快速的初始化构建执行。

#### 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 和其它构建工具集成

Gradle与它的前辈Ant、Maven和Ivy可以做很好的集成，如下图：



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。

#### 2.2.8 社区和公司驱动

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。

Gradle社区论坛是：<http://forums.gradle.org/gradle> 或<https://discuss.gradle.org/>

Gradleware是Gradle背后的技术服务和支持公司。

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。

#### 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。

### 2.3 更大的场景：持续交付

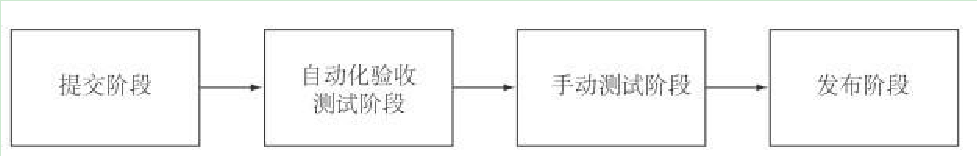
能够构建源代码仅仅是软件交付过程中的一个方面。更重要的是，你想要将产品发布到产品环境以创造业务价值。在这个过程 中，你想要运行测试，构建发布产品，分析代码以保证质量，也可能要指定一个目标环境并部署。

自动化的整个过程好处太多了，最重要的是，手动交付软件是很漫长的、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。

#### 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:



### 2.4 安装Gradle

下载地址：<https://gradle.org/install/>,包含旧版本下载：https://gradle.org/releases或者<http://services.gradle.org/distributions/>

手动安装：下载如“gradle-4.9-all.zip”，解压，然后设置GRADLE\_HOME，在Path变量中设置%GRADLE\_HOME%\bin，然后在命令行中检测安装是否成功：gradle – v

#### 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。

### 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命令中执行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！”的例子。build.gradle文件如下，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。

task helloWorld {

doLast {

println 'Hello world!'

}

}

在命令行中定位到build.gradle所在目录，然后运行：gradle -q helloWorld，效果如下：



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是这门语言的重要元素。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<<来简单的代表doLast。如下：

task helloWorld << {

println 'Hello world!'

}

下面的Demo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：

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)

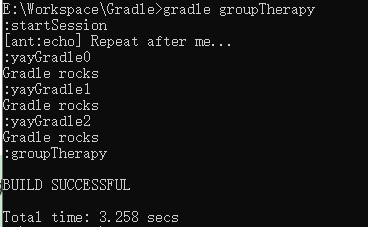


一开始你可能没有注意到，这个清单中隐藏了很多高级特性。代码中引入了关键字dependsOn来说明task之间的依赖❸。Gradle会确保被依赖的task总会在定义该依赖的task之前执行。实际上，dependsOn是task的一个方法。第4章中会涵盖一些内部task，这里不用太深入细节。

我们之前谈到的一个特性是Gradle和Ant有很好的集成❶。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字。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。

任务依赖图：



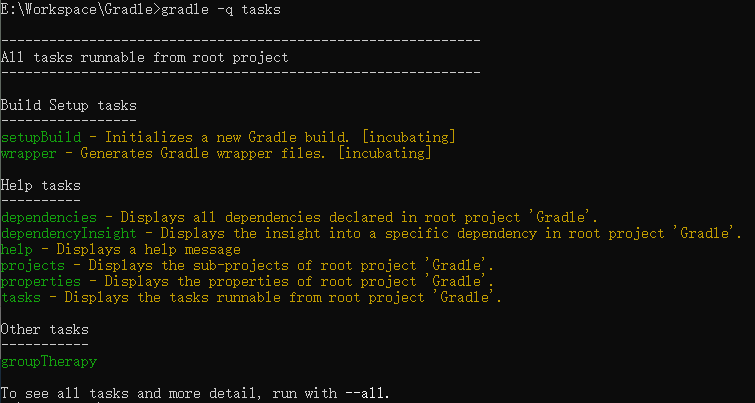
### 2.6使用Gradle的命令行

#### 2.6.1 例出项目中所有可用的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：

执行命令：gradle -q tasks

如下：

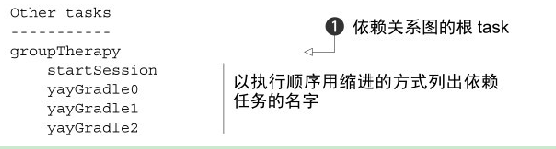




关于输出，有几点需要说明。Gradle提供了任务组的概念，你可以把它看作是多个task的集群。每个构建脚本都会默认暴露Help tasks任务组❶。如果某个task不属于一个任务组，那么它就会显示在Other tasks❷中。在这里可以找到groupTherapy❸。

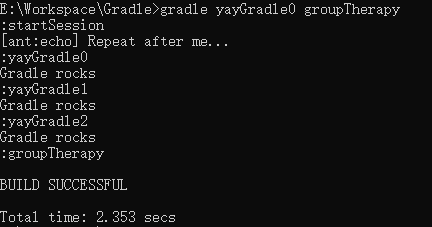
要想查看我们写的task，加上—all选项，如下：





#### 2.6.2 任务执行

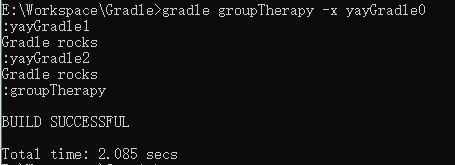
运行gradle yayGradle0 groupTherapy命令将会先执行yayGradle0任务，然后执行groupTherapy任务。执行：gradle yayGradle0 groupTherapy ，虽然这里指定了执行两个任务，但是每个任务只会输出一次，跟之前的效果一样，正确的顺序被保存并且每个任务只执行一次，效果如下：



可以以缩写的方法执行任务，但是任务的这个缩写必须是唯一的，如：gradle yG0 gT，gradle gt等同于gradle groupTherapy

**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，输出结果如下：



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。

#### 2.6.3 命令行选项

Gradle允许你同时定义一个或者多个选项。假设你想要将日记级别改变到INFO，则可以使用-i选项，或者如果想打印出在执行期间发生错误的堆栈跟踪信息，则可以使用-s选项，如：gradle groupTherapy -is或gradle groupTherapy -i -s。

-h：(help) 查看到所有可用的选项

-b：(build file) Gradle构建脚本默认命名约定为build.gradle，使用这个命令选项可以执行一个其它名字的构建脚本，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 property) 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 property) 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，如-Pmyprop=myvalue

**日志参数**

-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以获取更多信息。如果你想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

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（如：gradle groupTherapy --daemon）。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.

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-daemonvcb sk ，要手动停止守护进程，可以执行gradle –stop命令。

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。

## 3、通过范例学习构建Gradle项目

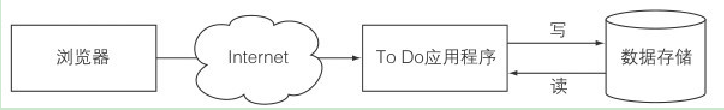
当开始开发一个全新的应用时，Java并没有指导你如何项目结构。你可能会问自己在哪里放源文件、配置文件和类库文件。如果想要将应用程序代码与测试代码分开，应该怎么办？Gradle通过引入预定义的项目布局为类似Java的项目提供了基于约定的构建方法。被那种有不同目录结构的遗留应用困住了？没问题！Gradle可以根据你的需要调整约定。

### 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

#### 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：



#### 3.1.2 任务管理用例

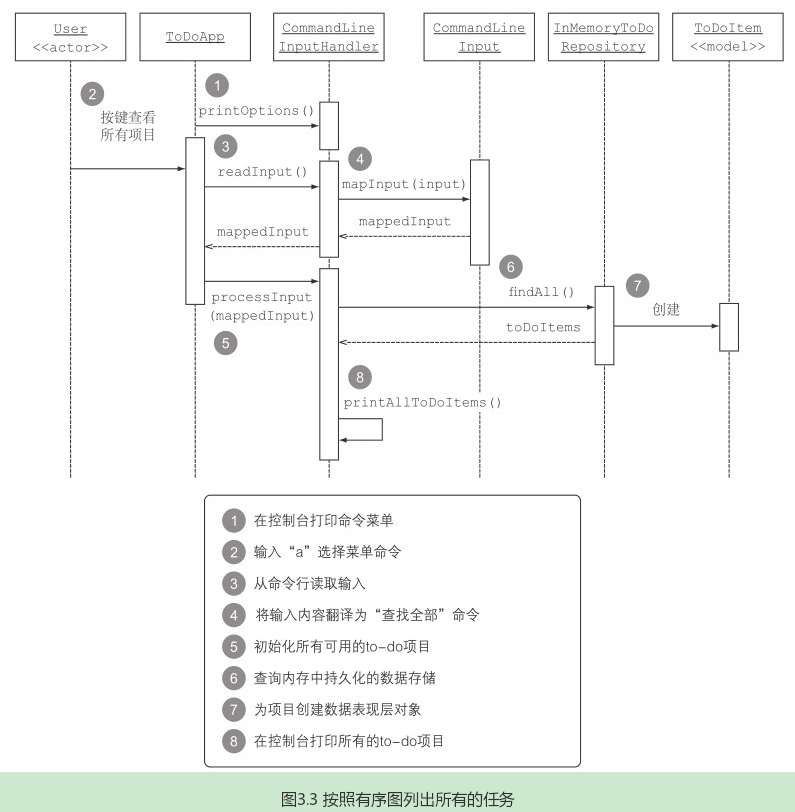


#### 3.1.3 检查组件交互

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。



#### 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。

具体代码在笔记目录下的todo-app.zip中都写了，不需要去写这里的代码，我们的目的是实际自动化构建，而不是代码功能。

**To Do模型类**

每一个ToDoItem类的实例都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读取和存储它。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;

public int compareTo(ToDoItem other) {

return id.compareTo(other.id)

}

}

**领域模型在内存中的持久化**

仓库类接口，为了能够方便地切换实现，你需要创建一个接：ToDoRepository

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

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.Collections;

import java.util.ArrayList;

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<>()

@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.id, toDoItem);

}

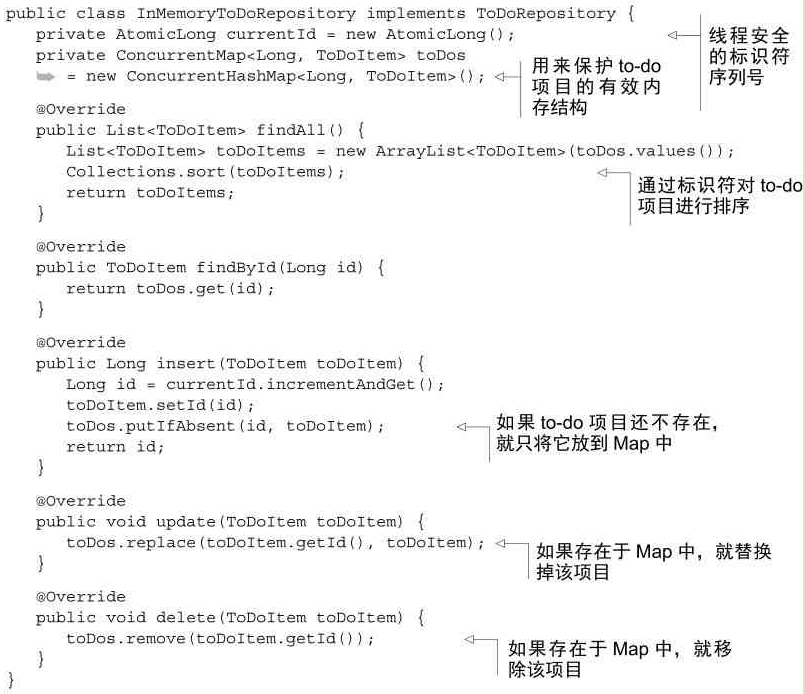
@Override

public void delete(ToDoItem toDoItem) {

toDos.remove(toDoItem.id);

}

}



**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

实现主体类：

package com.manning.gia.todo;

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

import com.manning.gia.todo.utils.CommandInputHandler;

public class ToDoApp {

public static final char DEFAULT\_INOUT = '\u0000';

public static void main(String[] args) {

CommandLineInputHandler commandLineInputHandler = new CommandLineInputHandler();

char command = DEFAULT\_INOUT;

while (CommandLineInput.EXIT.getShortCmd() != command) {

commandLineInputHandler.printOptions();

String input = commandLineInputHandler.readInput();

char[] inputChars = input.length() == 1 ? input.toCharArray() : new char[] { DEFAULT\_INOUT };

command = inputChars[0];

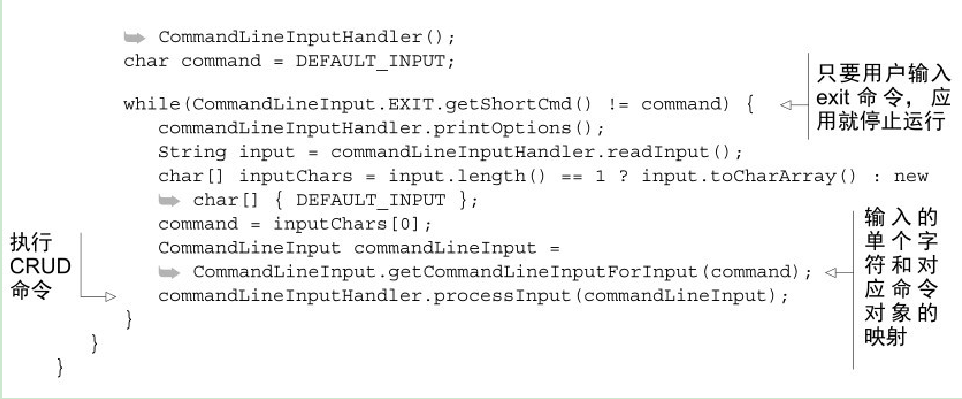
CommandLineInput commandLineInput = CommandLineInput.getCommandLineInputForInput(command);

commandLineInputHandler.processInput(commandLineInput);

}

}

}



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。

### 3.2 构建Java项目

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。

#### 3.2.1 使用Java插件

创建build.gradle文件，并像下面这样告诉它使用Java插件：

apply plugin: 'java'

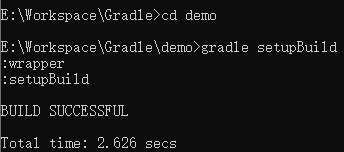
一行代码足够构建你的Java代码，但是Gradle怎么知道去哪里找源文件呢？Java插件引入的约定之一就是源代码的位置。在默认情况下，插件会到src/main/java目录下查找，你需要将所有关于To Do应用程序的类文件放在正确的目录下。

**自动化项目生成**

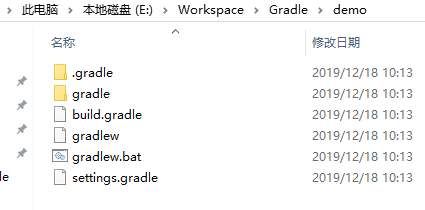
如果不需要你手动创建源文件目录，是不是很棒？Maven有一个概念叫作project archetypes，它是一个插件，用来从现有的模板生成项目结构。遗憾的是，在写这本书的时候，这个功能还没有成为Gradle的核心特性。Gradle模板插件是由Gradle社区提出的，为的就是解决这个问题。该插件可以在https://github.com/townsfolk/gradle-templates上找到。build setup插件实现了初始化Gradle项目的功能，你甚至不需要构建脚本就可以使用。这个插件允许你生成项目文件（和之后你会了解的其他相关文件）。要生成Gradle构建脚本，只需要在命令行执行gradle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

在一个空目录下执行：

gradle setupBuild



执行之后会生成如下内容：



这里并没有生成java项目的结构，所以如果想创建Java的项目结构可以使用gradle-templates：<https://github.com/townsfolk/gradle-templates>，在该网站上有说如何使用。

随便创建一个目录，如：demo3，然后创建build.gradle，内容如下：

buildscript {

repositories {

maven {

url 'http://dl.bintray.com/cjstehno/public'

}

}

dependencies {

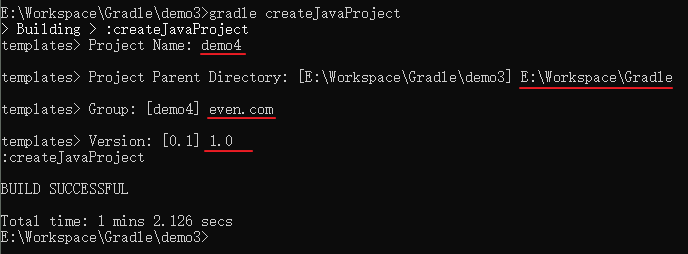
classpath 'gradle-templates:gradle-templates:1.5'

}

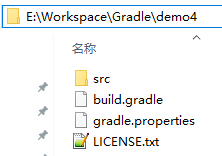
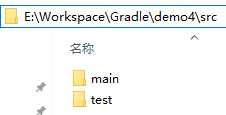
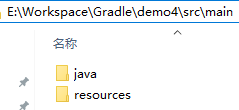
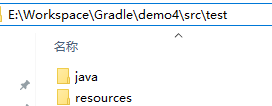
}

apply plugin:'templates'

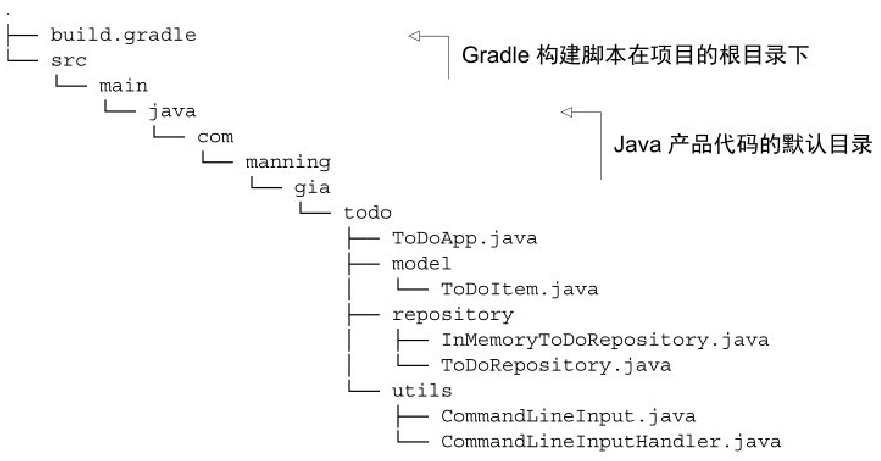
然后在cmd中导航到demo3目录，并执行：gradle createJavaProject 命令，它会提示你输入项目名称、项目位置、组织、版本，如下：

**\**

如上图，中括号中的应该是默认值，所以可以不输入直接回车使用默认值。命令执行后生成的内容如下：

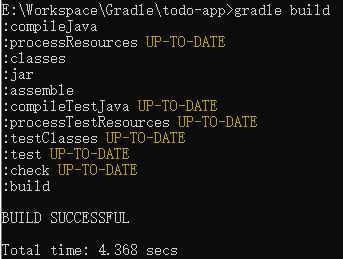
在创建完构建脚本和将源代码移动到正确的位置之后，你的项目结构应该是这样的：

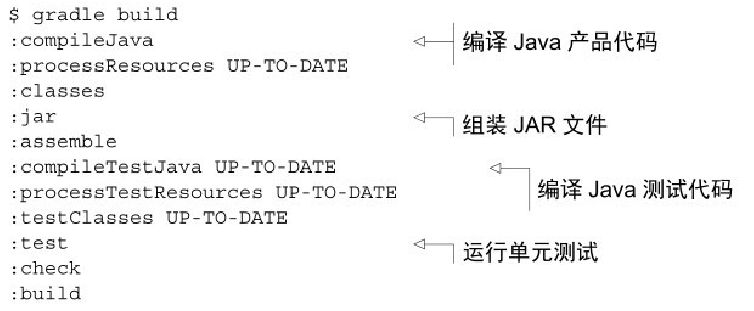


这里build.gradle文件中只有一行代码：apply plugin: 'java'，todo所有的文件就是上图列出来的这些

**构建项目**

Java插件有一个任务叫作build，这个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：





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。

执行gradle build命令后生成的内容如下：

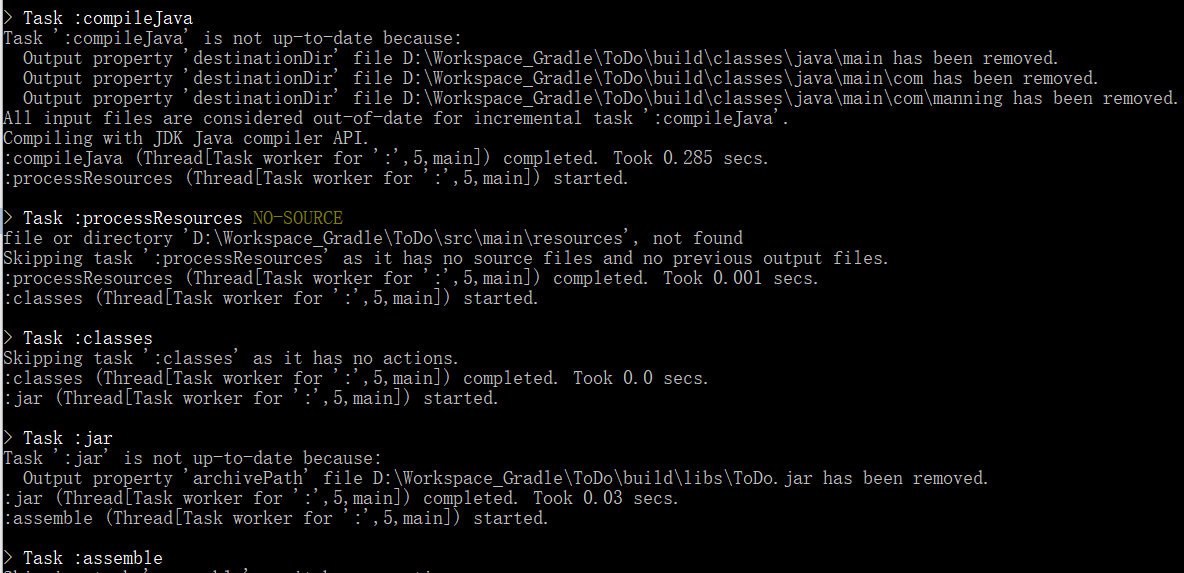


build目录保存了生成的class文件和jar文件，详细内容如下：



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。

要想知道执行这个任务时都执行了哪些依赖任务，可执行：gradle build -i



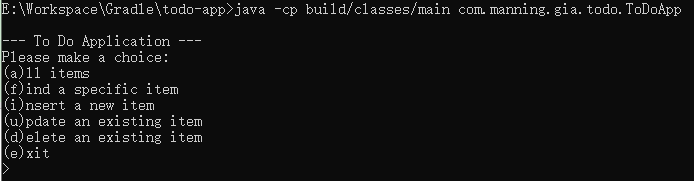
某些任务标记为UP-TO-DATE消息，这意味着这个任务被跳过了，Gradle的增量式构建支持自动鉴别不需要被运行的任务。特别是在大型企业级项目中，这个特性是节省时间的好帮手。在输出中，你可以看到被跳过的任务是：comipleTestJava和testClasses。因为我们没有在src/test/java下提供单元测试。

在项目根目录下会看到生成了一个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。进入到\build\classes\java\main执行：java com.manning.gia.todo.ToDoApp即可运行程序。

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。

**运行项目**

在项目的根目录下执行如下命令：

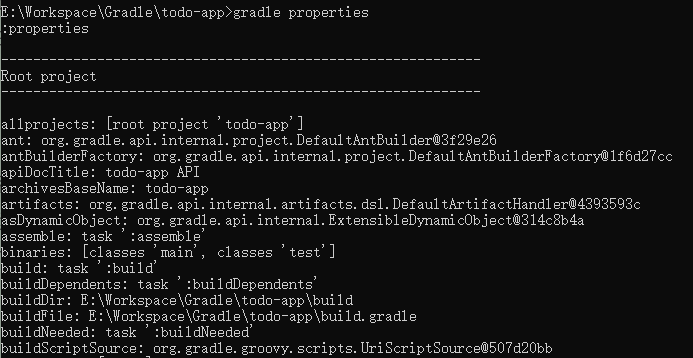


**对独立运行Java程序的支持**

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

#### 3.2.2 定制你的项目

Java插件是一个灵活的框架。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，同时Gradle也给予你定制这些约定的能力，如何知道什么是可配置的？可查看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。效果如下：



**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。

修改属性变量并添加JAR文件头：

apply plugin: 'java'

version = 0.1

sourceCompatibility = 1.6

jar {

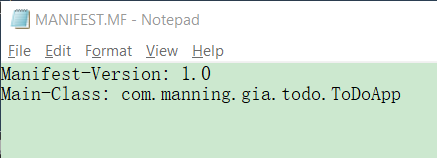
manifest {

attributes 'Main-Class': 'com.manning.gia.todo.ToDoApp'

}

}

执行gradle build命令，然后进入到\build\libs目录会看到生成了带版本号的jar，执行java -jar ToDo-0.1.jar即可启动应用。可利用winrar软件打开ToDo-01.jar，查看META-INF中的MANIFEST.MF文件，如下：



**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。假设你需要把源代码放在src目录下，而不是src/main/java，另外，你需要把结果放在out目录下，而不是build目录下，请看下面的例子：

apply plugin: 'java'

version = 0.1

sourceCompatibility = 1.6

jar {

manifest {

attributes 'Main-Class': 'com.manning.gia.todo.ToDoApp'

}

}

sourceSets {

main {

java {

srcDirs = ['src']

}

}

test {

java {

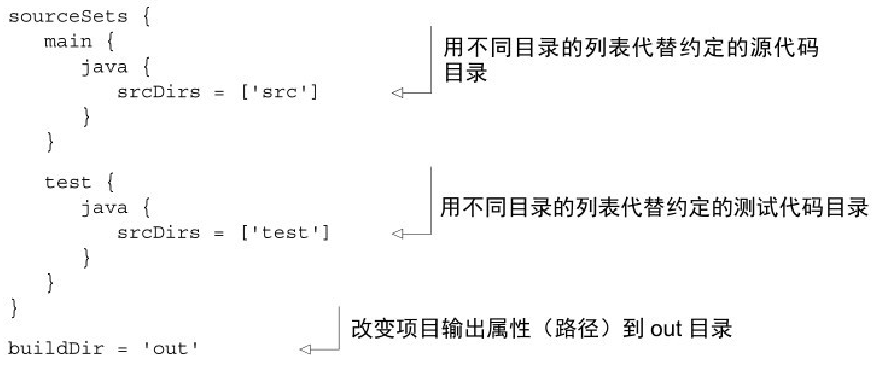
srcDirs = ['test']

}

}

}

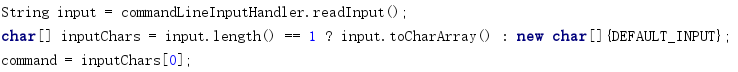
buildDir = 'out'



执行gradle build查看结果。**定制一个构建的关键就是了解潜在的属性和DSL元素。**

#### 3.2.3 配置和使用外部依赖

ToDoApp类中的main方法，通过读取用户在控制台的输入，然后将输入的第一个字符做为命令，如下：



获取字符串的第1个字符的功能可以通过使用外部库来实现。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则返回一个默认字符，下面的代码显示了使用外部库来获取输入的第一个字符：



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

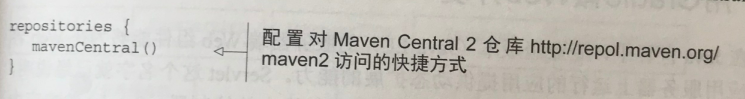
**定义仓库**

在Java世界，类库都是以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的类库。为此，你需要使用公共的可访问的仓库Maven Central：

repositories {

mavenCentral()

}



定义好仓库之后，你就可以声明类库了。

**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：

dependencies {

compile group: 'org.apache.commons', name: 'commons-lang3', version: '3.1'

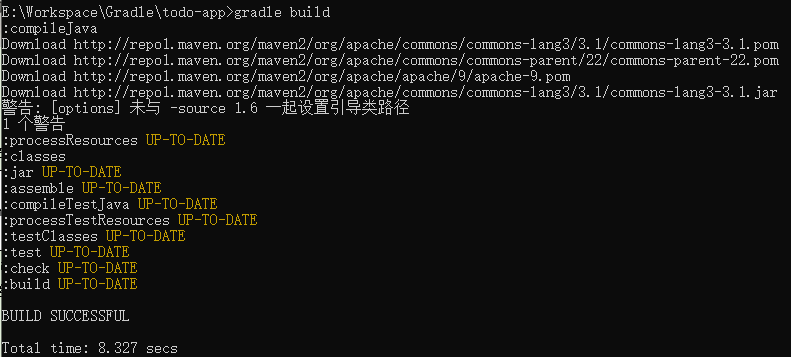
}

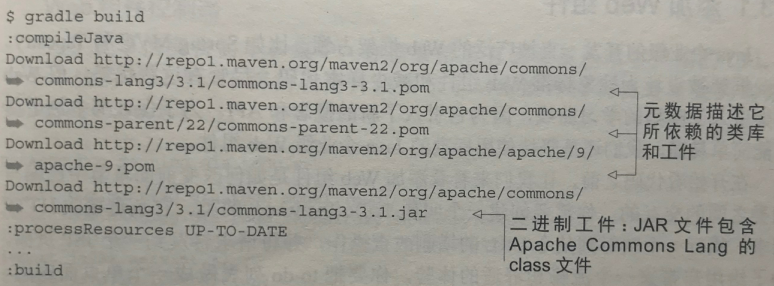
在Gradle中，依赖是由configuration分组的。Java插件引入的一种configuration是compile。你可以通过configuration的名字看出它是给编译源代码使用的。

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：





### 3.3 用Gradle做Web开发

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。

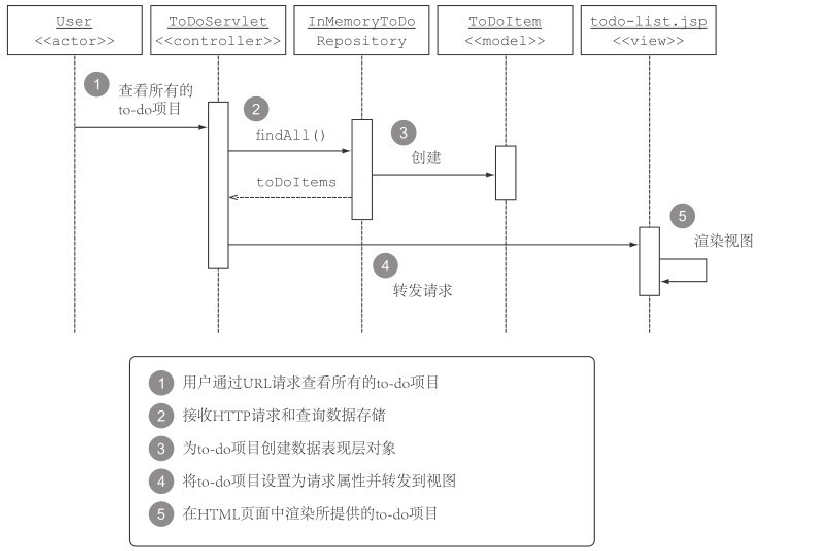
Gradle提供了开箱即用的插件，用来组装WAR文件和将Web应用部署到本地Servlet容器中。之前，我们看到了如何应用和配置这些插件，你所需要的就是将一个独立运行的Java应用变成一个Web应用。

不需要写任何代码，直接使用笔记目录下的todo-webapp-jetty.zip即可。

#### 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。在todo包下创建一个web包，从源代码中复制ToDoServlet.java过来。

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。下图显示了在新系统中获取和渲染所有to-do项目的流程。





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。

什么是Jetty?

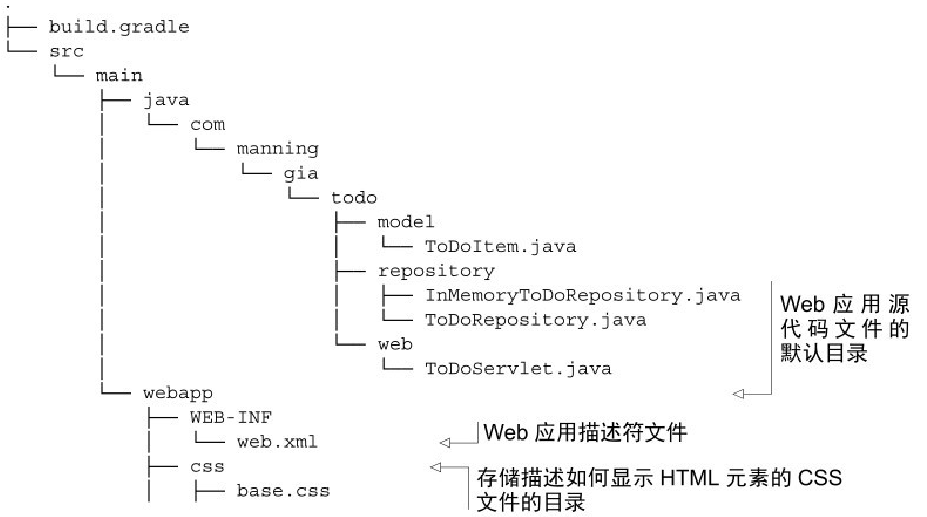
        简单来讲Jetty就是一个开源的HTTP服务器和Servlet引擎，它可以为JSP和Servlet提供运行时环境，由于其轻量、灵活的特性，Jetty也被应用于一些知名产品中，例如ActiveMQ、Maven、Spark、GoogleAppEngine、Eclipse、Hadoop等。

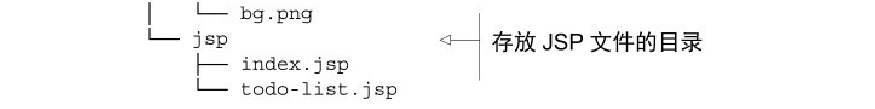
**War插件**

War插件扩展自Java插件，因此，只要使用了War插件则不需要使用Java插件（不过即使你也使用了Java插件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。在build.gradle文件中使用War插件，如下：

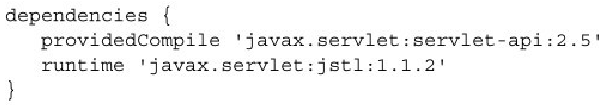
apply plugin:’war’

使用了这个插件后，你的项目除了Java插件提供的约定外，也会有Web应用的约定，它会知道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，所有的Web资源文件都放在这个目录下，结构如下：





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：



**构建项目**

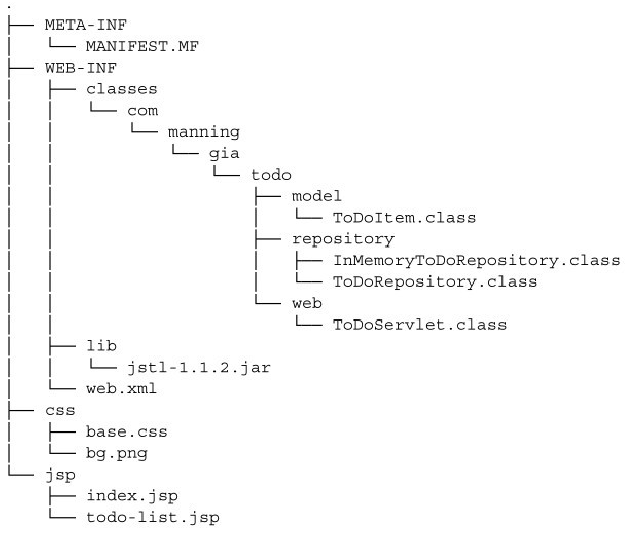
在Gradle中构建一个Web应用就和构建一个独立运行的Java应用程序一样。运行gradle build命令后，组装的WAR文件可以在build/libs目录下找到。将一个独立运行的Java应用改变成一个Web应用，jar任务被war任务代替，输出结果如下：





生成的war文件在 build\libs目录下：todo-webapp.war

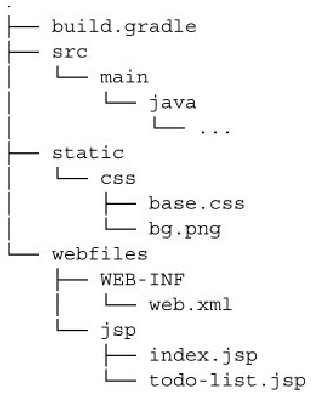
War插件确保组装的WAR文件遵循由Java EE规范定义的标准结构。war任务将Web应用源代码目录src /main/webapp的内容原封不动地拷贝到WAR文件的根目录。编译的class文件最终会放置在WEB-INF/classes下，通过依赖闭包定义的运行时类库会放置在WEB-INF/lib下。下面显示了在运行命令jar tf todo-webapp-0.1.war之后WAR文件的目录结构：



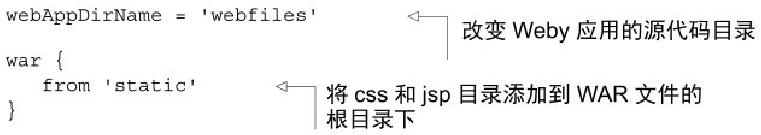
在默认情况下，WAR文件的名字会继承自项目的目录名字。即便你的项目没有遵循Gradle的标准约定，该插件也依然可以构建WAR文件。让我们看一些定制化选项吧。

**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将需要的目录添加到WAR文件中，如下：



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**

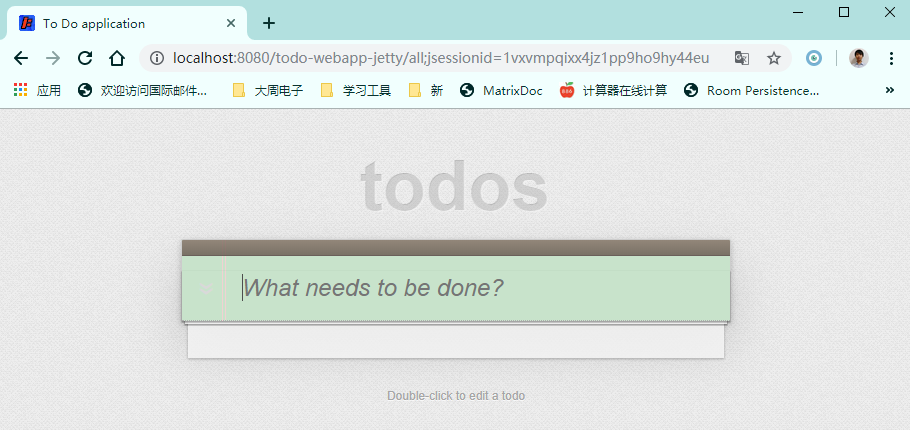
一个嵌入式Servlet容器并不了解你的应用，直到你提供了Web应用程序确切的classpath和相关的源代码目录。通常，你需要用编程的方式做这件事。在这里，Jetty插件帮你做了这件事。因为War插件暴露了所有的信息，Jetty插件可以在运行时访问它们。这是Gradle中一个典型的例子，通过Gradle的API，一个插件可以访问另一个插件的配置。在你的构建脚本中，像这样使用插件：



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：



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，并输入这个url。终于，你可以看到To DoWeb应用程序，如下：



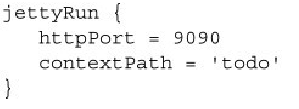
Gradle会一直让应用程序运行，直到你按“Ctrl+C”键。Jetty是如何知道在什么端口和上下文中运行应用的呢？还是约定。Jetty插件运行一个Web应用的默认端口是8080，上下文路径 todo-webapp-jetty是从项目名字继承的。当然，这些都是可配置的。

**快速应用开发**

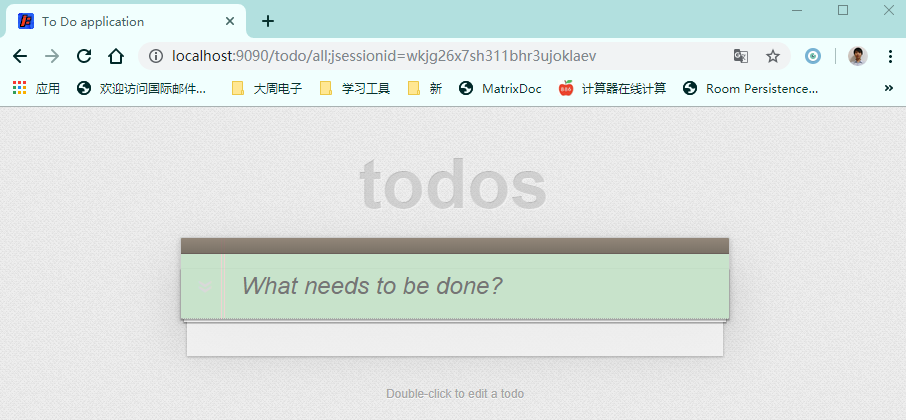
每次改变项目代码都需要重新启动容器是非常麻烦和耗时的。Jetty插件允许你在不启动容器的情况下，改变静态资源和JSP文件。另外，还针对class文件的改变配置像Jrebel这样的字节码交换技术来执行热部署。

**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，而且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：



运行效果如下：



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。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。

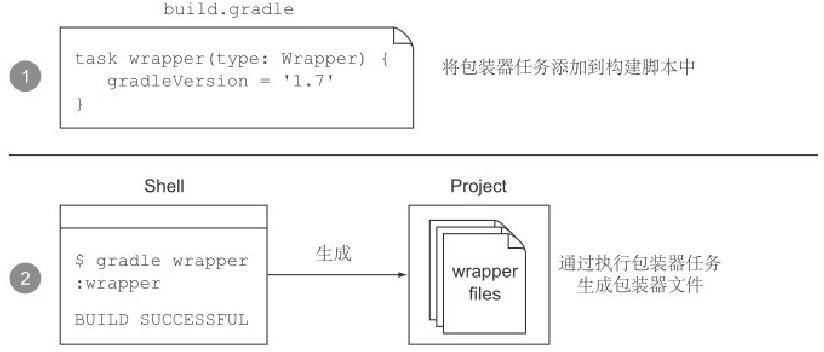
**什么时候使用包装器**

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）。



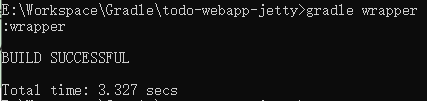
为了能够让项目可以下载压缩过的Gradle运行时文件，定义一个类型为Wrapper的任务，通过gradleVersion属性指定你想要使用的Gradle版本：

task wrapper(type: Wrapper) {

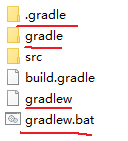
gradleVersion = '1.2'

}

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：



执行了这个命令后会生产这些内容：





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：

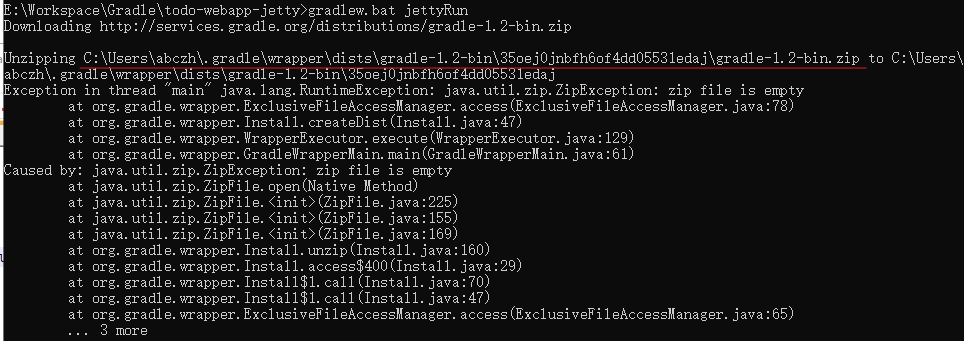


#### 3.4.2 使用包装器

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：

真实运行这个任务时是会报错的，如下：



如上红线位置的文件是一个空文件，文件大小为0，这是因为[Gradle停用了HTTP服务](https://blog.gradle.org/decommissioning-http) (具体可查看这个连接)。这意味着，如果您的[PROJECT\_PATH]/gradle/wrapper/gradle-wrapper.properties文件具有以下条目：

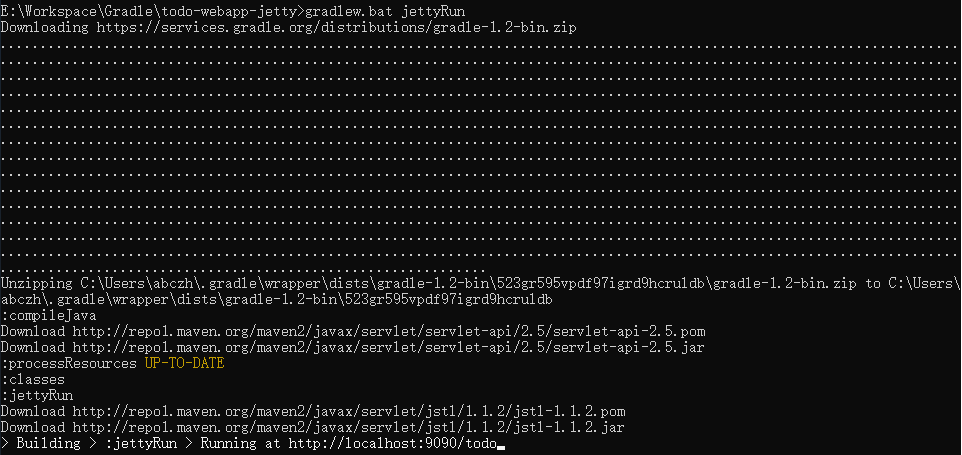
distributionUrl=http\://services.gradle.org/distributions/gradle-{version}.zip

...然后，该条目将尝试ZIP从不再为该文件提供服务的服务中检索文件（http基于ZIP文件），并且它将静默返回一个空文件；因此，这将是尝试解压缩该空ZIP文件的任何过程的错误源。

要解决此问题，只需更改您的[PROJECT\_PATH]/gradle/wrapper/gradle-wrapper.properties文件以使其distributionUrl使用即可https：

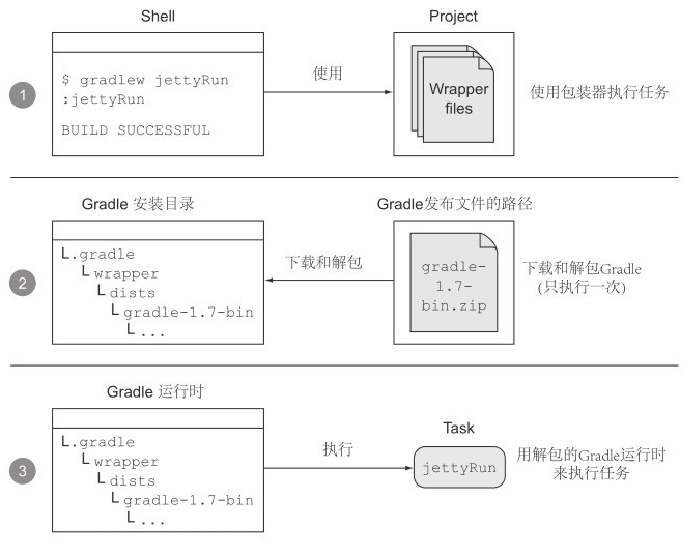
distributionUrl=https\://services.gradle.org/distributions/gradle-{version}.zip

修成https后再执行gradlew.bat jetttyRun，效果如下：

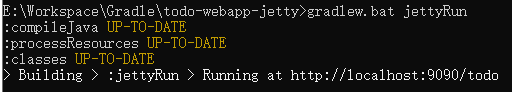


上面那些点代表正在下载gradle-1.2-bin.zip，有时候下载很慢，解决办法是可以手工下载，然后放到对应的位置，如：C:\Users\abczh\.gradle\wrapper\dists\gradle-1.2-bin\523gr595vpdf97igrd9hcruldb\gradle-1.2-bin.zip，然后再执行之前的命令即可，这样它就不会再下载了。





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：



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：



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。

**总结：**

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）。

# 第2部分 掌握基本原理

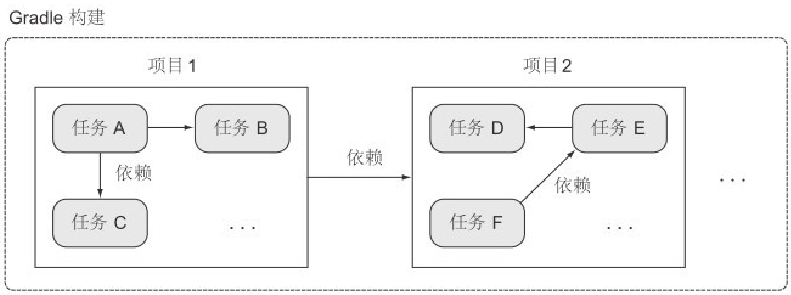
## 4、构建脚本概要

**了解构建生命周期挂接**

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。

### 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。

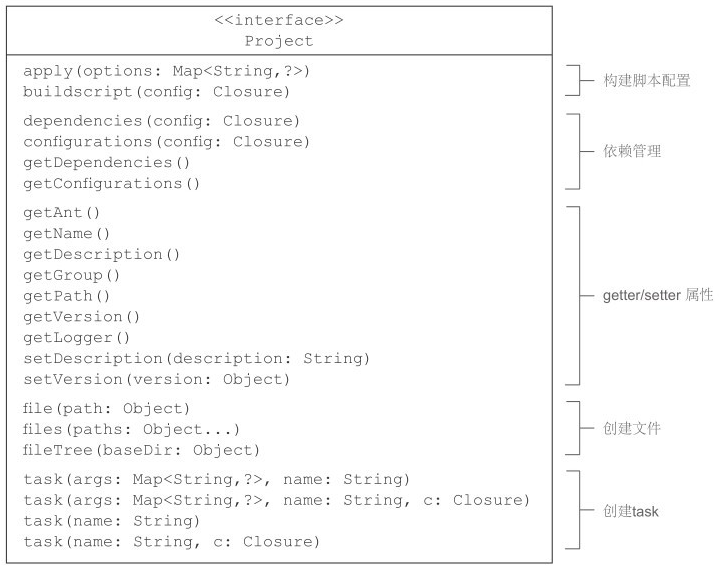


在多项目构建中一个project可以依赖于其他的project。相似的，task可以形成一个依赖关系图来确保它们的执行顺序。

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

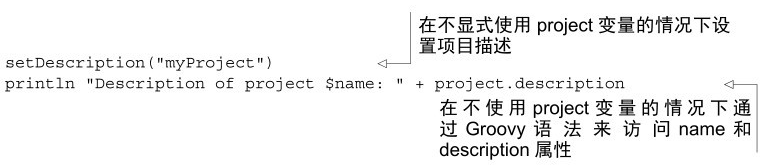
在Gradle术语中，一个项目（project）代表一个正在构建的组件（比如，一个JAR文件），或一个想要完成的目标，如部署应用程序。如果你使用过Maven，那么这个概念应该听起来很熟悉。Gradle的build.gradle文件相当于Maven的pom.xml。每个Gradle构建脚本至少定义一个项目。当构建进程启动后，Gradle基于build.gradle中的配置实例化org.gradle.api.Project类，并且能够通过project变量使其隐式可用。图4.2显示了API接口及其最重要的方法。



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：



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

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。

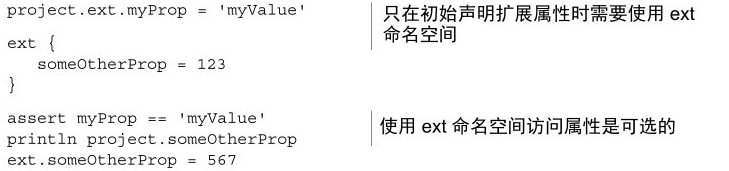


#### 4.1.3 属性

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。



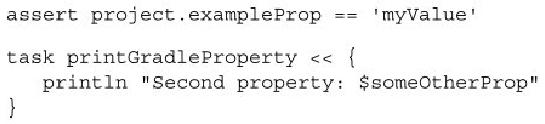
类似地，额外的属性也可以通过属性文件来提供。

**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：



你可以按照如下方式访问项目中的这两个变量：



**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

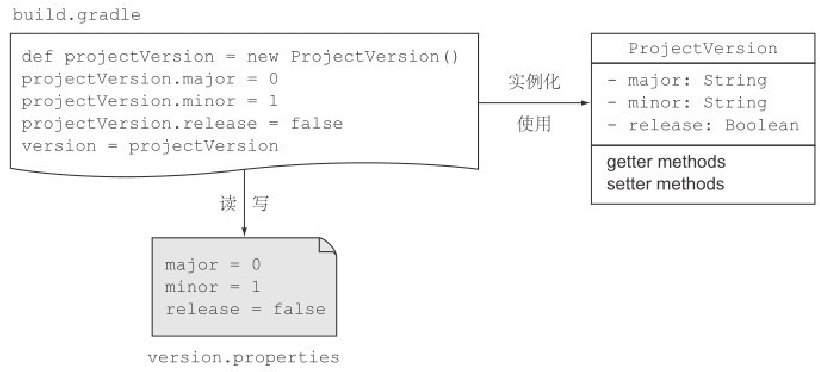


在这里对于这些声明属性的可选方式不再给出具体的例子，但是在需要时你可以使用它们。如果你想进一步了解，在线的Gradle用户指南上有非常详细的例子可供参考。在本章的后续部分，你将充分使用task和Gradle的构建生命周期。

### 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



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"

}

}

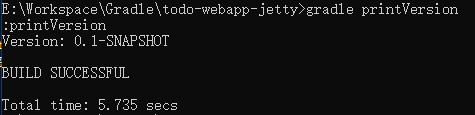
version = '0.1-SNAPSHOT'

task printVersion << {

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：



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 {

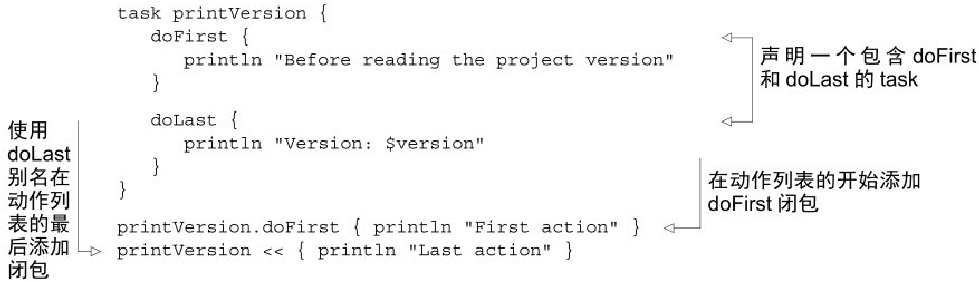
println "Version: $version"

}

}

**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 "doFirst..."

}

doLast {

println "doLast..."

}

}

printVersion.doFirst { println "First action" }

printVersion << { println "Last action" }

执行：gradle printVersion，效果如下：



从结果看，扩展的doFirst和doLast并没有覆盖原有的doFirst和doLast，而是都执行。扩展的doFirst比原来的doFirst先执行，扩展的doLast比原有的doLast后执行。

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DefauItTask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：

version = '0.1-SNAPSHOT'

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 "group = $version, description = $description"

}

结果如下：



从结果看，goup的赋值并未生效。

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 "group = $version, description = $description"

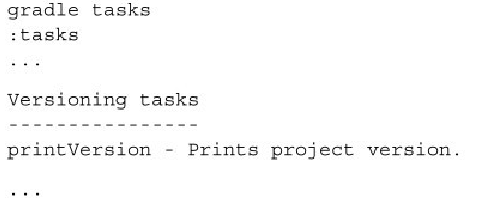
}

}

运行结果如下：



当运行gradle tasks时，我们可以看到task正确的分组和描述：



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。

task first << { println "first" }

task second << { println "second" }

version = '0.1-SNAPSHOT'

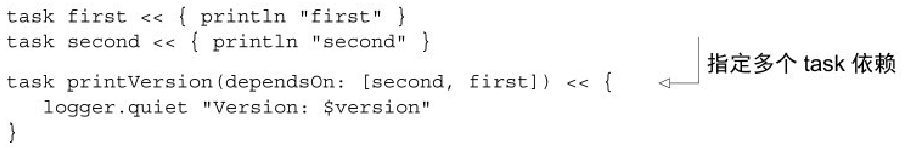
task printVersion(dependsOn: [second, first]) {

doLast { println "Version: $version" }

}

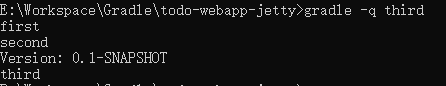
task third << { println "third" }

third.dependsOn('printVersion')





在命令行通过调用task third来执行依赖链上的其他task：



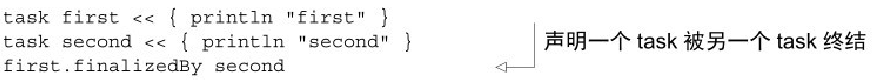
如果仔细看看task的执行顺序，你可能会对结果感到惊讶。task printVersion声明了依赖first和second task。难道你不期望second在first执行之前得到执行？在Gradle中，task执行顺序是不确定的。

**task依赖的执行顺序**

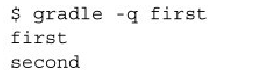
理解Gradle 并不能保证task依赖的执行顺序是很重要的。dependsOn方法只是定义了所依赖的task需要先执行。Gradle的思想是声明在一个给定的task执行之前什么该被执行，而没有定义它该如何执行。如果你之前使用过像Ant这种命令式地定义依赖的构建工具的话，那么Gradle的这个概念就有点难以理解了。在本章的后续部分你将看到，在Gradle中，执行顺序是由task的输入/输出规范自动确定的。这种构建设计有很多好处。一方面，你不需要知道整个task依赖链上的关系是否发生改变，这样可以提高代码的可维护性和避免潜在的破坏。另一方面，因为构建没有严格的执行顺序，也就是支持task的并行执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



你会看到执行first会自动触发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。



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'}"

}

}

task printVersion << {

logger.quiet "Version: $version"

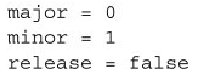
}

上面这些代码可直接写在build.gradle文件中

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：

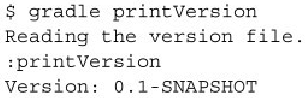


**添加task配置块**

下面的代码声明了一个名为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。



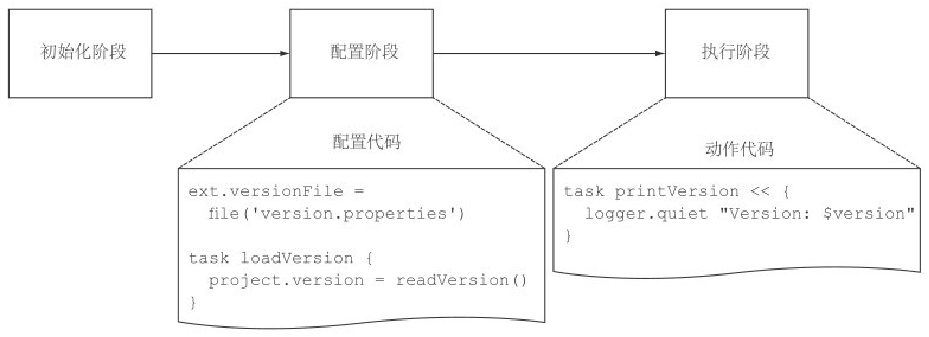
现在运行printVersion，你会看到新创建的loadVersion先被执行。尽管task的名字没有被打印出来，但是你仍然知道这种情况，因为打印的日志是你自己添加的。



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。下图可视化了构建阶段的运行顺序和它们执行的代码。



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。

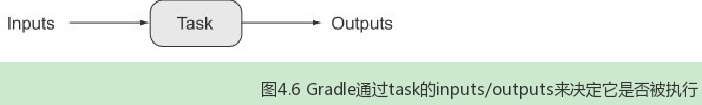
注意：项目每一次构建的任何配置代码都可以被执行——即使你只执行gradle tasks。

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。

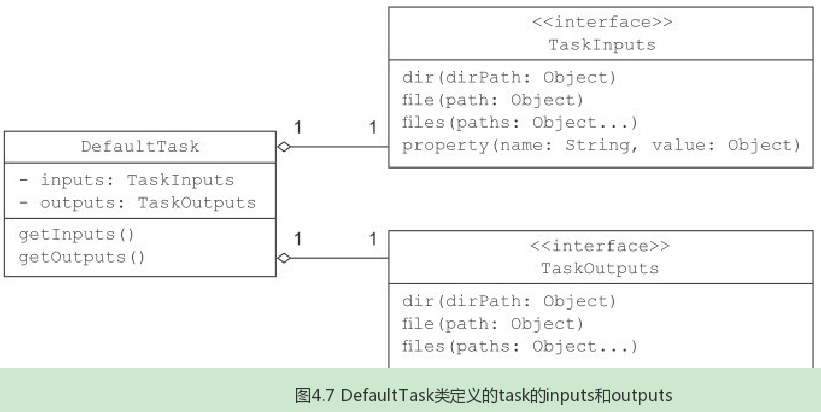
正如你所见，Gradle的增量式构建特性紧紧地与生命周期相结合。在第3章中，你已经看到Java插件大量地使用了这个特性。如果Java源文件与最后一次运行的构建不同的话，则只运行compileJava task。最终，这个特性将充分提高构建的性能。在下一节中，我们将介绍针对自己的task如何使用增量式构建特性。

#### 4.2.8 声明task的inputs和outputs

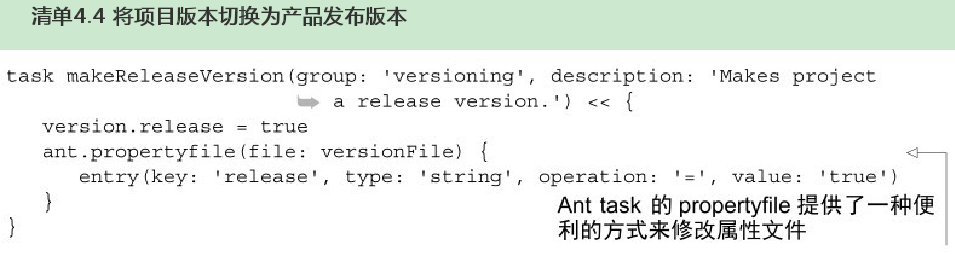
Gradle通过比较两个构建task的inputs和outputs来决定task是否是最新的，如图4.6所示。自从最后一个task执行以来，如果inputs和outputs没有发生变化，则认为task是最新的。因此，只有当inputs和outputs



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。



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。



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：