Gradle使用指南

极客学院出版

前言

Gradle，这是一个基于 JVM 的富有突破性构建工具。Gradle 正迅速成为许多开源项目和前沿企业构建系统的选择，同时也在挑战遗留的自动化构建项目。本教程主要讲解了如何使用 Gradle 构建系统和构建系统过程中涉及的插件。

适合人群

适用于自动化地进行软件构建、测试、发布、部署、软件打包的项目。

学习前提

你需要有 Groovy 语言基础，对 Java 应用开发有一定的了解。

鸣谢：http://blog.csdn.net/column/details/gradle-translation.html?&page=2

版本信息

书中演示代码基于以下版本：

工具	版本信息
Gradle	1.5

目录

1 简介

# #

介绍

很高兴能向大家介绍 Gradle，这是一个基于 JVM 的富有突破性构建工具。

它为您提供了:

- 一个像 ant 一样，通用的灵活的构建工具
- 一种可切换的，像 maven 一样的基于约定约定优于配置的构建框架
- 强大的多工程构建支持
- 强大的依赖管理(基于 Apachelvy)
- 对已有的 maven 和 ivy 仓库的全面支持
- 支持传递性依赖管理，而不需要远程仓库或者 pom.xml 或者 ivy 配置文件
- ant 式的任务和构建是 gradle 的第一公民
- 基于 groovy，其 build 脚本使用 groovy dsl 编写
- 具有广泛的领域模型支持你的构建
- 在第二章节概述中，你会看到关于 Gradle 的详细介绍和指导

# #

关于本手册

与 Gradle 一样，本手册也在不停的更新中。许多部分并未完全进行描述。有些内容并未完全描述。我们需要你来一起帮助改进本手册。你可以在 Gradle 官方网站找到其余格式的文档。

2

概述

#

特性说明

下面是一些 Gradle 特性的列表。

#

基于声明的构建和基于约定的构建

Gradle 的核心在于基于 Groovy 的丰富而可扩展的域描述语言(DSL)。 Groovy 通过声明性的语言元素将基于声明的构建推向下层，你可以按你想要的方式进行组合。 这些元素同样也为支持 Java， Groovy，OSGi，Web 和 Scala 项目提供了基于约定的构建。 并且，这种声明性的语言是可以扩展的。你可以添加新的或增强现有的语言元素。 因此，它提供了简明、可维护和易理解的构建。

#

为以依赖为基础的编程方式提供语言支持

声明性语言优点在于通用任务图，你可以将其充分利用在构建中. 它提供了最大限度的灵活性，以让 Gradle 适应你的特殊需求。

#

构建结构化

Gradle 的灵活和丰富性最终能够支持在你的构建中应用通用的设计模式。 例如，它可以很容易地将你的构建拆分为多个可重用的模块，最后再进行组装，但不要强制地进行模块的拆分。 不要把原本在一起的东西强行分开（比如在你的项目结构里），从而避免让你的构建变成一场噩梦。 最后，你可以创建一个结构良好，易于维护，易于理解的构建。

#

深度 API

Gradle 允许你在构建执行的整个生命周期，对它的核心配置及执行行为进行监视并自定义。

#

Gradle 的扩展

Gradle 有非常良好的扩展性。 从简单的单项目构建，到庞大的多项目构建，它都能显著地提升你的效率。 这才是真正的结构化构建。通过最先进的增量构建功能，它可以解决许多大型企业所面临的性能瓶颈问题。

#

多项目构建

Gradle 对多项目构建的支持非常出色。项目依赖是首先需要考虑的问题。 我们允许你在多项目构建当中对项目依赖关系进行建模，因为它们才是你真正的问题域。 Gradle 遵守你的布局。

Gradle 提供了局部构建的功能。 如果你在构建一个单独的子项目，Gradle 也会帮你构建它所依赖的所有子项目。 你也可以选择重新构建依赖于特定子项目的子项目。 这种增量构建将使得在大型构建任务中省下大量时间。

#

多种方式管理依赖

不同的团队喜欢用不同的方式来管理他们的外部依赖。 从 Maven 和 Ivy 的远程仓库的传递依赖管理，到本地文件系统的 jar 包或目录，Gradle 对所有的管理策略都提供了方便的支持。

#

Gradle 是第一个构建集成工具

Ant tasks 是最重要的。而更有趣的是，Ant projects 也是最重要的。 Gradle 对任意的 Ant 项目提供了深度导入，并在运行时将 Ant 目标(target)转换为原生的 Gradle 任务(task)。 你可以从 Gradle 上依赖它们(Ant targets)，增强它们，甚至在你的 build.xml 上定义对 Gradle tasks 的依赖。Gradle 为属性、路径等等提供了同样的整合。

Gradle 完全支持用于发布或检索依赖的 Maven 或 Ivy 仓库。 Gradle 同样提供了一个转换器，用于将一个 Maven pom.xml 文件转换为一个 Gradle 脚本。Maven 项目的运行时导入的功能将很快会有。

#

易于移植

Gradle 能适应你已有的任何结构。因此，你总可以在你构建项目的同一个分支当中开发你的 Gradle 构建脚本，并且它们能够并行进行。 我们通常建议编写测试，以保证生成的文件是一样的。 这种移植方式会尽可能的可靠和减少破坏性。这也是重构的最佳做法。

#

Groovy

Gradle 的构建脚本是采用 Groovy 写的，而不是用 XML。 但与其他方法不同，它并不只是展示了由一种动态语言编写的原始脚本的强大。 那样将导致维护构建变得很困难。 Gradle 的整体设计是面向被作为一门语言，而不是一个僵化的框架。 并且 Groovy 是我们允许你通过抽象的 Gradle 描述你个人的 story 的黏合剂。 Gradle 提供了一些标准通用的 story。这是我们相比其他声明性构建系统的主要特点。 我们的 Groovy 支持也不是简单的糖衣层，整个 Gradle 的 API 都是完全 groovy 化的。只有通过 Groovy才能去运用它并对它提高效率。

#

The Gradle wrapper

Gradle Wrapper 允许你在没有安装 Gradle 的机器上执行 Gradle 构建。 这一点是非常有用的。比如，对一些持续集成服务来说。 它对一个开源项目保持低门槛构建也是非常有用的。 Wrapper 对企业来说也很有用，它使得对客户端计算机零配置。 它强制使用指定的版本，以减少兼容支持问题。

#

自由和开源

Gradle 是一个开源项目，并遵循 ASL 许可。

#

为什么使用 Groovy?

我们认为内部 DSL（基于一种动态语言）相比 XML 在构建脚本方面优势非常大。它们是一对动态语言。 为什么使用 Groovy? 答案在于 Gradle 内部的运行环境。 虽然 Gradle 核心目的是作为通用构建工具，但它还是主要面向 Java 项目。 这些项目的团队成员显然熟悉 Java。我们认为一个构建工具应该尽可能地对所有团队成员透明。

你可能会想说，为什么不能使用 Java 来作为构建脚本的语言。 我认为这是一个很有意义的问题。对你们的团队来讲，它确实会有最高的透明度和最低的学习曲线。 但由于 Java 本身的局限性，这种构建语言可能就不会那样友善、 富有表现力和强大。[1] 这也是为什么像 Python，Groovy 或者 Ruby 这样的语言在这方面表现得更好的原因。 我们选择了 Groovy，因为它向 Java 人员提供了目前为止最大的透明度。 其基本的语法，类型，包结构和其他方面都与 Java 一样，Groovy 在这之上又增加了许多东西。但是和 Java 也有着共同点。

对于那些分享和乐于去学习 Python 知识的 Java 团队而言，上述论点并不适用。 Gradle 的设计非常适合在 JRuby 或 Jython 中创建另一个构建脚本引擎。 那时候，对我们而言，它只是不再是最高优先级的了。我们很高兴去支持任何社区努力创建其他的构建脚本引擎。

3

安装

# #

先决条件

Gradle 需要 1.5 或更高版本的 JDK.Gradle 自带了 Groovy 库，所以不需要安装 Groovy。Gradle 会忽略已经安装的 Groovy。Gradle 会使用 ptah (这里的"path"应该是指 PATH 环境变量。[Rover12421]译注) 中的 JDK(可以使用 java -version 检查)。当然，你可以配置 JAVA_HOME 环境变量来指向 JDK 的安装目录。

# #

下载

从 Gralde 官方网站下载 Gradle 的最新发行包。

# #

解压

Gradle 发行包是一个 ZIP 文件。完整的发行包包括以下内容(官方发行包有 full 完整版，也有不带源码和文档的版本，可根据需求下载。[Rover12421]译注):

- Gradle 可执行文件
- 用户手册 (有 PDF 和 HTML 两种版本)
- DSL 参考指南
- API 手册(Javadoc 和 Groovydoc)
- 样例，包括用户手册中的例子，一些完整的构建样例和更加复杂的构建脚本
- 源代码。仅供参考使用,如果你想要自己来编译 Gradle 你需要从源代码仓库中检出发行版本源码，具体请查看 Gradle 官方主页。

#

配置环境变量

运行 gradle 必须将 GRADLE_HOME/bin 加入到你的 PATH 环境变量中。

#

测试安装

运行如下命令来检查是否安装成功.该命令会显示当前的 JVM 版本和 Gradle 版本。

```
gradle -v
```

#

JVM 参数配置

Gradle 运行时的 JVM 参数可以通过 GRADLE_OPTS 或 JAVA_OPTS 来设置.这些参数将会同时生效。 JAVA_OPTS 设置的参数将会同其它 JAVA 应用共享，一个典型的例子是可以在 JAVA_OPTS 中设置代理和 GRADLE_OPTS 设置内存参数。同时这些参数也可以在 gradle 或者 gradlew 脚本文件的开头进行设置。

4 问题反馈

当年使用 Gradle 或其它软件的时候或多或少都会遇到一些问题，或许是无法驾驭的新特性，或许是一些 bug，又抑或是关于 Gradle 一些常见问题。本章将给你一些解决问题的建议和如何获取帮助。

# #

解决问题

当你遇到问题时，首先确认一下是否用的最新版本的 Gradle。最新版本总是会更加的完善并且带有更多的新特性。或许你的问题在最新版本中已经得到的解决。

如果你采用守护模式运行，那么尝试用 --no-daemon 来停掉守护模式。

# #

获取帮助

你可以去 Gralde 官方论坛 http://forums.gradle.org 来寻求一些帮助。在这里你可以和 Gradle 的开发人员以及其他社区人员进行交流。

如果有什么搞不定了，去论坛发帖是解决问题的最佳方式。或许这对我们而言也是一些良好的改进建议。同时，开发团队也会周期性的在论坛发布一些帖子和发布最新版本。这样可以使你与 Gradle 开发团队一样时刻保持最新版本。

5

构建基础

#

Projects 和 tasks

projects 和 tasks是 Gradle 中最重要的两个概念。

任何一个 Gradle 构建都是由一个或多个 projects 组成。每个 project 包括许多可构建组成部分。 这完全取决于你要构建些什么。举个例子，每个 project 或许是一个 jar 包或者一个 web 应用，它也可以是一个由许多其他项目中产生的 jar 构成的 zip 压缩包。一个 project 不必描述它只能进行构建操作。它也可以部署你的应用或搭建你的环境。不要担心它像听上去的那样庞大。 Gradle 的 build-by-convention 可以让您来具体定义一个 project 到底该做什么。

每个 project 都由多个 tasks 组成。每个 task 都代表了构建执行过程中的一个原子性操作。如编译，打包，生成 javadoc，发布到某个仓库等操作。

到目前为止，可以发现我们可以在一个 project 中定义一些简单任务，后续章节将会阐述多项目构建和多项目多任务的内容。

# #

Hello world

你可以通过在命令行运行 gradle 命令来执行构建，gradle 命令会从当前目录下寻找 build.gradle 文件来执行构建。我们称 build.gradle 文件为构建脚本。严格来说这其实是一个构建配置脚本，后面你会了解到这个构建脚本定义了一个 project 和一些默认的 task。

你可以创建如下脚本到 build.gradle 中 To try this out，create the following build script named build.gradle。

#

第一个构建脚本

build.gradle

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

然后在该文件所在目录执行 gradle -q hello

> -q 参数的作用是什么？
>
> 该文档的示例中很多地方在调用 gradle 命令时都加了 -q 参数。该参数用来控制 gradle 的日志级别，可以保证只输出我们需要的内容。具体可参阅本文档第十八章日志来了解更多参数和信息。

#

执行脚本

```
Output of gradle -q hello
> gradle -q hello
Hello world!
```

上面的脚本定义了一个叫做 hello 的 task，并且给它添加了一个动作。当执行 gradle hello 的时候, Gralde 便会去调用 hello 这个任务来执行给定操作。这些操作其实就是一个用 groovy 书写的闭包。

如果你觉得它看上去跟 Ant 中的 targets 很像，那么是对的。Gradle 的 tasks 就相当于 Ant 中的 targets。不过你会发现他功能更加强大。我们只是换了一个比 target 更形象的另外一个术语。不幸的是这恰巧与 Ant 中的术语有些冲突。ant 命令中有诸如 javac、copy、tasks。所以当该文档中提及 tasks 时，除非特别指明 ant task。否则指的均是指 Gradle 中的 tasks。

#

快速定义任务

用一种更简洁的方式来定义上面的 hello 任务。

#

快速定义任务

build.gradle

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

上面的脚本又一次采用闭包的方式来定义了一个叫做 hello 的任务，本文档后续章节中我们将会更多的采用这种风格来定义任务。

#

代码即脚本

Gradle 脚本采用 Groovy 书写，作为开胃菜,看下下面这个例子。

#

在 gradle 任务中采用 groovy

build.gradle

```
task upper << {
    String someString = 'mY_nAmE'
    println "Original: " + someString
    println "Upper case: " + someString.toUpperCase()
}
Output of gradle -q upper
> gradle -q upper
```

```
Original: mY_nAmE
Upper case: MY_NAME
```

或者

#

在 gradle 任务中采用 groovy

build.gradle

```
task count << {
    4.times { print "$it " }
}
Output of gradle -q count
> gradle -q count
0 1 2 3
```

# #

任务依赖

你可以按如下方式创建任务间的依赖关系

#

在两个任务之间指明依赖关系

build.gradle

```
task hello << {
    println 'Hello world!'
}
task intro(dependsOn: hello) << {
    println "I'm Gradle"
}
```

gradle -q intro 的输出结果

```
Output of gradle -q intro
\> gradle -q intro
Hello world!
I'm Gradle
```

添加依赖 task 也可以不必首先声明被依赖的 task。

# #

延迟依赖

build.gradle

```
task taskX(dependsOn: 'taskY') << {
    println 'taskX'
}
task taskY << {
    println 'taskY'
}
```

Output of gradle -q taskX

```
 \> gradle -q taskX
taskY
taskX
```

可以看到，taskX 是 在 taskY 之前定义的，这在多项目构建中非常有用。

注意:当引用的任务尚未定义的时候不可使用短标记法来运行任务。

#

动态任务

借助 Groovy 的强大不仅可以定义简单任务还能做更多的事。例如，可以动态定义任务。

#

创建动态任务

build.gradle

```
4.times { counter ->
    task "task$counter" << {
        println "I'm task number $counter"
    }
}
```

gradle -q task1 的输出结果。

```
Output of gradle -q task1
\> gradle -q task1
I'm task number 1
```

#

任务操纵

一旦任务被创建后，任务之间可以通过 API 进行相互访问。这也是与 Ant 的不同之处。比如可以增加一些依赖。

#

通过 API 进行任务之间的通信 – 增加依赖

build.gradle

```
4.times { counter ->
    task "task$counter" << {
        println "I'm task number $counter"
    }
}
task0.dependsOn task2, task3
```

gradle -q task0的输出结果。

```
Output of gradle -q task0
\> gradle -q task0
I'm task number 2
I'm task number 3
I'm task number 0
```

为已存在的任务增加行为。

#

通过 API 进行任务之间的通信 – 增加任务行为

build.gradle

```
task hello << {
    println 'Hello Earth'
}
hello.doFirst {
    println 'Hello Venus'
```

```
}
hello.doLast {
    println 'Hello Mars'
}
hello << {
    println 'Hello Jupiter'
}
Output of gradle -q hello
> gradle -q hello
Hello Venus
Hello Earth
Hello Mars
Hello Jupiter
```

doFirst 和 doLast 可以进行多次调用。他们分别被添加在任务的开头和结尾。当任务开始执行时这些动作会按照既定顺序进行。其中 << 操作符 是 doLast 的简写方式。

# #

短标记法

你早就注意到了吧，没错，每个任务都是一个脚本的属性，你可以访问它:

#

以属性的方式访问任务

build.gradle

```
task hello << {
    println 'Hello world!'
}
hello.doLast {
    println "Greetings from the $hello.name task."
}
```

gradle -q hello 的输出结果

```
Output of gradle -q hello
\> gradle -q hello
Hello world!
Greetings from the hello task.
```

对于插件提供的内置任务。这尤其方便(例如:complie)

#

增加自定义属性

你可以为一个任务添加额外的属性。例如,新增一个叫做 myProperty 的属性，用 ext.myProperty 的方式给他一个初始值。这样便增加了一个自定义属性。

#

为任务增加自定义属性

build.gradle

```
task myTask {
    ext.myProperty = "myValue"
}

task printTaskProperties << {
    println myTask.myProperty
}
```

gradle -q printTaskProperties 的输出结果

```
Output of gradle -q printTaskProperties
\> gradle -q printTaskProperties
myValue
```

自定义属性不仅仅局限于任务上，还可以做更多事情。

# #

调用 Ant 任务

Ant 任务是 Gradle 中的一等公民。Gradle 借助 Groovy 对 Ant 任务进行了优秀的整合。Gradle 自带了一个 AntBuilder，在 Gradle 中调用 Ant 任务比在 build.xml 中调用更加的方便和强大。 通过下面的例子你可以学到如何调用一个 Ant 任务以及如何与 Ant 中的属性进行通信。

#

利用 AntBuilder 执行 ant.loadfile

build.gradle

```
task loadfile << {
    def files = file('../antLoadfileResources').listFiles().sort()
    files.each { File file ->
        if (file.isFile()) {
            ant.loadfile(srcFile: file, property: file.name)
            println " *** $file.name ***"
            println "${ant.properties[file.name]}"
        }
    }
}
```

gradle -q loadfile 的输出结果

```
Output of gradle -q loadfile
\> gradle -q loadfile
*** agile.manifesto.txt ***
Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration  over contract negotiation
Responding to change over following a plan
 *** gradle.manifesto.txt ***
Make the impossible possible, make the possible easy and make the easy elegant.
(inspired by Moshe Feldenkrais)
```

在你脚本里还可以利用 Ant 做更多的事情。想了解更多请参阅在 Gradle 中调用 Ant。

#

方法抽取

Gradle 的强大要看你如何编写脚本逻辑。针对上面的例子，首先要做的就是要抽取方法。

#

利用方法组织脚本逻辑

build.gradle

```
task checksum << {
    fileList('../antLoadfileResources').each {File file ->
        ant.checksum(file: file, property: "cs_$file.name")
        println "$file.name Checksum: ${ant.properties["cs_$file.name"]}"
    }
}
task loadfile << {
    fileList('../antLoadfileResources').each {File file ->
        ant.loadfile(srcFile: file, property: file.name)
        println "I'm fond of $file.name"
    }
}
File[] fileList(String dir) {
    file(dir).listFiles({file -> file.isFile() } as FileFilter).sort()
}
```

gradle -q loadfile 的输出结果

```
Output of gradle -q loadfile
\> gradle -q loadfile
I'm fond of agile.manifesto.txt
I'm fond of gradle.manifesto.txt
```

在后面的章节你会看到类似出去出来的方法可以在多项目构建中的子项目中调用。无论构建逻辑多复杂，Gradle 都可以提供给你一种简便的方式来组织它们。

#

定义默认任务

Gradle 允许在脚本中定义多个默认任务。

#

定义默认任务

build.gradle

```
defaultTasks 'clean', 'run'
task clean << {
    println 'Default Cleaning!'
}
task run << {
    println 'Default Running!'
}
task other << {
    println "I'm not a default task!"
}
```

gradle -q 的输出结果。

```
Output of gradle -q
\> gradle -q
Default Cleaning!
Default Running!
```

这与直接调用 gradle clean run 效果是一样的。在多项目构建中，每个子项目都可以指定单独的默认任务。如果子项目未进行指定将会调用父项目指定的的默认任务。

# #

Configure by DAG

稍后会对 Gradle 的配置阶段和运行阶段进行详细说明 配置阶段后，Gradle 会了解所有要执行的任务 Gradle 提供了一个钩子来捕获这些信息。一个例子就是可以检查已经执行的任务中有没有被释放。借由此，你可以为一些变量赋予不同的值。

在下面的例子中，为 distribution 和 release 任务赋予了不同的 version 值。

#

依赖任务的不同输出

build.gradle

```
task distribution << {
    println "We build the zip with version=$version"
}
task release(dependsOn: 'distribution') << {
    println 'We release now'
}
gradle.taskGraph.whenReady {taskGraph ->
    if (taskGraph.hasTask(release)) {
        version = '1.0'
    } else {
        version = '1.0-SNAPSHOT'
    }
}
```

gradle -q distribution 的输出结果

```
Output of gradle -q distribution
\> gradle -q distribution
We build the zip with version=1.0-SNAPSHOT
```

gradle -q release 的输出结果

```
Output of gradle -q release
\> gradle -q release
```

```
We build the zip with version=1.0
We release now
```

whenReady 会在已发布的任务之前影响到已发布任务的执行。即使已发布的任务不是主要任务(也就是说，即使这个任务不是通过命令行直接调用)

# #

下一步目标

在本章中，我们了解了什么是 task，但这还不够详细。欲知更多请参阅章节任务进阶。

另外，可以目录继续学习Java 构建入门。

6

Java 构建入门

# #

Java 插件

如你所见，Gradle 是一个通用工具。它可以通过脚本构建任何你想要实现的东西，真正实现开箱即用。但前提是你需要在脚本中编写好代码才行。

大部分 Java 项目基本流程都是相似的：编译源文件，进行单元测试，创建 jar 包。使用 Gradle 做这些工作不必为每个工程都编写代码。Gradle 已经提供了完美的插件来解决这些问题。插件就是 Gradle 的扩展，简而言之就是为你添加一些非常有用的默认配置。Gradle 自带了很多插件，并且你也可以很容易的编写和分享自己的插件。Java plugin 作为其中之一，为你提供了诸如编译，测试，打包等一些功能。

Java 插件为工程定义了许多默认值，如Java源文件位置。如果你遵循这些默认规则，那么你无需在你的脚本文件中书写太多代码。当然，Gradle 也允许你自定义项目中的一些规则，实际上，由于对 Java 工程的构建是基于插件的，那么你也可以完全不用插件自己编写代码来进行构建。

后面的章节我们通过许多深入的例子介绍了如何使用 Java 插件来进行以来管理和多项目构建等。但在这个章节我们需要先了解 Java 插件的基本用法。

#

一个基本 Java 项目

来看一下下面这个小例子，想用 Java 插件，只需增加如下代码到你的脚本里。

#

采用 Java 插件

build.gradle

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

备注:示例代码可以在 Gralde 发行包中的 samples/java/quickstart 下找到。

定义一个 Java 项目只需如此而已。这将会为你添加 Java 插件及其一些内置任务。

> 添加了哪些任务?
>
> 你可以运行 gradle tasks 列出任务列表。这样便可以看到 Java 插件为你添加了哪些任务。

标准目录结构如下:

```
project
    +build
    +src/main/java
    +src/main/resources
    +src/test/java
    +src/test/resources
```

Gradle 默认会从 `src/main/java` 搜寻打包源码，在 `src/test/java` 下搜寻测试源码。并且 `src/main/resources` 下的所有文件按都会被打包，所有 `src/test/resources` 下的文件 都会被添加到类路径用以执行测试。所有文件都输出到 build 下，打包的文件输出到 build/libs 下。

#

构建项目

Java 插件为你添加了众多任务。但是它们只是在你需要构建项目的时候才能发挥作用。最常用的就是 build 任务,这会构建整个项目。当你执行 gradle build 时，Gralde 会编译并执行单元测试，并且将 `src/main/*` 下面 class 和资源文件打包。

#

构建 Java 项目

运行 gradle build 的输出结果

```
Output of gradle build
> gradle build
:compileJava
:processResources
:classes
:jar
:assemble
:compileTestJava
:processTestResources
:testClasses
:test
:check
:build
BUILD SUCCESSFUL
Total time: 1 secs
```

其余一些较常用的任务有如下几个:

#

clean

删除 build 目录以及所有构建完成的文件。

#

assemble

编译并打包 jar 文件，但不会执行单元测试。一些其他插件可能会增强这个任务的功能。例如，如果采用了 War 插件，这个任务便会为你的项目打出 War 包。

#

check

编译并测试代码。一些其他插件也可能会增强这个任务的功能。例如，如果采用了 Code-quality 插件，这个任务会额外执行 Checkstyle。

#

外部依赖

通常，一个 Java 项目拥有许多外部依赖。你需要告诉 Gradle 如何找到并引用这些外部文件。在 Gradle 中通常 Jar 包都存在于仓库中。仓库可以用来搜寻依赖或发布项目产物。下面是一个采用 Maven 仓库的例子。

#

添加 Maven 仓库

build.gradle

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

添加依赖。这里声明了编译期所需依赖 commons-collections 和测试期所需依赖 junit。

#

添加依赖

build.gradle

```
dependencies {
    compile group: 'commons-collections', name: 'commons-collections', version: '3.2'
    testCompile group: 'junit', name: 'junit', version: '4.+'
}
```

了解更多可参阅依赖管理基础

#

自定义项目

Java 插件为你的项目添加了众多默认配置。这些默认值通常对于一个普通项目来说已经足够了。但如果你觉得不适用修改起来也很简单。看下面的例子，我们为 Java 项目指定了版本号以及所用的 JDK 版本，并且添加一些属性到 mainfest 中。

#

自定义 MANIFEST.MF

build.gradle

```
sourceCompatibility = 1.5
version = '1.0'
jar {
    manifest {
        attributes 'Implementation-Title': 'Gradle Quickstart', 'Implementation-Version': version
    }
}
```

> 都有哪些可用属性?
>
> 可以执行 gradle propertie s来得到项目的属性列表。用这条命令可以看到插件添加的属性以及默认值。

Java 插件添加的都是一些普通任务，如同他们写在 Build 文件中一样。这意味着前面章节展示的机制都可以用来修改这些任务的行为。例如，可以设置任务的属性，添加任务行为，更改任务依赖，甚至是重写覆盖整个任务。在下面的例子中，我们将修改 test 任务，这是一个 Test 类型任务。让我们来在它执行时为它添加一些系统属性。

#

为 test 添加系统属性

build.gradle

```
test {
    systemProperties 'property': 'value'
}
```

#

发布 jar 包

如何发布 jar 包?你需要告诉 Gradle 发布到到哪。在 Gradle 中 jar 包通常被发布到某个仓库中。在下面的例子中，我们会将 jar 包发布到本地目录。当然你也可以发布到远程仓库或多个远程仓库中。

#

发布 jar 包

build.gradle

```
uploadArchives {
    repositories {
       flatDir {
           dirs 'repos'
       }
    }
}
```

执行 gradle uploadArchives 以发布 jar 包。

#

创建 Eclipse 文件

若要把项目导入 Eclipse 中，你需要添加另外一个插件到你的脚本文件中。

#

Eclipse plugin

build.gradle

```
apply plugin: 'eclipse'
```

执行 gradle eclipse 来生成 Eclipse 项目文件。

#

示例汇总

这是示例代码汇总得到的一个完整脚本：

#

Java 示例 – 一个完整构建脚本

build.gradle

``` apply plugin: 'java' apply plugin: 'eclipse' sourceCompatibility = 1.5 version = '1.0' jar { manifest { attributes 'Implementation-Title': 'Gradle Quickstart', 'Implementation-Version': version } } repositories { mavenCentral() } dependencies { compile group: 'commons-collections', name: 'commons-collections', version: '3.2' testCompile group: 'junit', name: 'junit', version: '4.+' } test { systemProperties 'property': 'value' } uploadArchives { repositories { flatDir { dirs 'repos' } } }
```

#

多项目构建

现在来看一个典型的多项目构建的例子。项目结构如下：

#

多项目构建–项目结构

Build layout

multiproject/ api/ services/webservice/ shared/

备注: 本示例代码可在 Gradle 发行包的 samples/java/multiproject 位置找到

此处有三个工程。api 工程用来生成给客户端用的 jar 文件，这个 jar 文件可以为 XML webservice 提供 Java 客户端。webservice 是

#

多项目构建定义

定义一个多项目构建工程需要在根目录(本例中与 multiproject 同级)创建一个*setting* 配置文件来指明构建包含哪些项目。并且这个文件

#

多项目构建中的 settings.gradle

settings.gradle

include "shared", "api", "services:webservice", "services:shared"

#

公共配置

对多项目构建而言，总有一些共同的配置.在本例中，我们会在根项目上采用配置注入的方式定义一些公共配置。根项目就像一个容器，

#

多项目构建-公共配置

build.gradle

subprojects { apply plugin: 'java' apply plugin: 'eclipse-wtp' repositories { mavenCentral() } dependenc ies { testCompile 'junit:junit:4.11' } version = '1.0' jar { manifest.attributes provider: 'gradle' } }

值得注意的是我们为每个子项目都应用了 Java 插件。这意味着我们在前面章节学习的内容在子项目中也都是可用的。所以你可以在根

#

工程依赖

同一个构建中可以建立工程依赖，一个工程的 jar 包可以提供给另外一个工程使用。例如我们可以让 api 工程以依赖于 shared 工程的 j

#

多项目构建-工程依赖

api/build.gradle

dependencies { compile project(':shared') }

#

打包发布

如何发布，请看下文:

#

多项目构建-发布

api/build.gradle

task dist(type: Zip) { dependsOn spiJar from 'src/dist' into('libs') { from spiJar.archivePath from configurations.runtime } } artifacts { archives dist } ```

# #

下一步目标?

本章中，我们了解了如何构建一个基本 Java 工程。但这都是一小部分基础，用 Gradle 还可以做很多事。关于了解更多可参阅 Java 插件，The Java Plugin，并且你可以在 Gradle 发行包中的 samples/java 目录找到更多例子。

另外，不要忘了继续阅读依赖管理基础内容

7

依赖管理基础

本章节介绍如何使用 Gradle 进行基本的依赖管理.

# #

什么是依赖管理?

通俗来讲，依赖管理由如下两部分组成。首先，Gradle 需要知道项目构建或运行所需要的一些文件，以便于找到这些需要的文件。我们称这些输入的文件为项目的依赖。其次，你可能需要构建完成后自动上传到某个地方。我们称这些输出为发布。下面来仔细介绍一下这两部分：

大部分工程都不太可能完全自给自足，一般你都会用到其他工程的文件。比如我工程需要 Hibernate 就得把它的类库加进来，比如测试的时候可能需要某些额外 jar 包，例如 JDBC 驱动或 Ehcache 之类的 Jar 包。

这些文件就是工程的依赖。Gradle 需要你告诉它工程的依赖是什么，它们在哪，然后帮你加入构建中。依赖可能需要去远程库下载，比如 Maven 或者 Ivy 库。也可以是本地库，甚至可能是另一个工程。我们称这个过程叫*依赖解决*。

通常，依赖的自身也有依赖。例如，Hibernate 核心类库就依赖于一些其他的类库。所以，当 Gradle 构建你的工程时，会去找到这些依赖。我们称之为*依赖传递*。

大部分工程构建的主要目的是脱离工程使用。例如，生成 jar 包，包括源代码、文档等，然后发布出去。

这些输出的文件构成了项目的发布内容。Gralde 也会为你分担这些工作。你声明了发布到到哪，Gradle 就会发布到哪。“发布”的意思就是你想做什么。比如，复制到某个目录，上传到 Maven 或 Ivy 仓库。或者在其它项目里使用，这些都可以称之为*发行*。

#

依赖声明

来看一下这个脚本里声明依赖的部分：

#

声明依赖

build.gradle

```
apply plugin: 'java'
repositories {
    mavenCentral()
}
dependencies {
    compile group: 'org.hibernate', name: 'hibernate-core', version: '3.6.7.Final'
    testCompile group: 'junit', name: 'junit', version: '4.+'
}
```

这是什么意思呢？这段脚本是这么个意思。首先，Hibernate-core.3.6.7.final.jar 这货是编译期必需的依赖。并且这货相关的依赖也会一并被加载进来，该段脚本同时还声明项目测试阶段需要 4.0 版本以上的 Junit。同时告诉 Gradle 可以去 Maven 中央仓库去找这些依赖。下面的章节会进行更详细的描述。

#

依赖配置

Gradle 中依赖以组的形式来划分不同的*配置*。每个配置都只是一组指定的依赖。我们称之为*依赖配置* 。你也可以借由此声明外部依赖。后面我们会了解到，这也可用用来声明项目的发布。

Java 插件定义了许多标准配置项。这些配置项形成了插件本身的 classpath。比如下面罗列的这一些，并且你可以从 Table 23.5，“Java 插件 – 依赖配置”了解到更多详细内容.。

#

compile

编译范围依赖在所有的 classpath 中可用，同时它们也会被打包

#

runtime

runtime 依赖在运行和测试系统的时候需要，但在编译的时候不需要。比如，你可能在编译的时候只需要 JDBC API JAR，而只有在运行的时候才需要 JDBC 驱动实现

#

testCompile

测试期编译需要的附加依赖

#

testRuntime

测试运行期需要

不同的插件提供了不同的标准配置，你甚至也可以定义属于自己的配置项。

#

外部依赖

依赖的类型有很多种，其中有一种类型称之为*外部依赖*。这种依赖由外部构建或者在不同的仓库中，例如 Maven 中央仓库或 Ivy 仓库中抑或是本地文件系统的某个目录中。

定义外部依赖需要像下面这样进行依赖配置

#

定义外部依赖

build.gradle

```
dependencies {
    compile group: 'org.hibernate', name: 'hibernate-core', version: '3.6.7.Final'
}
```

外部依赖包含 group，name 和 version 几个属性。根据选取仓库的不同，group 和 version 也可能是可选的。

当然，也有一种更加简洁的方式来声明外部依赖。采用：将三个属性拼接在一起即可。"group:name:version"

#

快速定义外部依赖

build.gradle

```
dependencies {
    compile 'org.hibernate:hibernate-core:3.6.7.Final'
}
```

# #

仓库

Gradle 是在一个被称之为*仓库*的地方找寻所需的外部依赖。仓库即是一个按 group，name 和 version 规则进行存储的一些文件。Gradle 可以支持不同的仓库存储格式，如 Maven 和 Ivy，并且还提供多种与仓库进行通信的方式，如通过本地文件系统或 HTTP。

默认情况下，Gradle 没有定义任何仓库，你需要在使用外部依赖之前至少定义一个仓库，例如 Maven 中央仓库。

#

使用 Maven 中央仓库

build.gradle

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

或者其它远程 Maven 仓库：

#

使用 Maven 远程仓库

build.gradle

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

或者采用 Ivy 远程仓库

#

采用 Ivy 远程仓库

build.gradle

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

或者在指定本地文件系统构建的库。

#

采用本地 Ivy 目录

build.gradle

```
repositories {
    ivy {
        // URL can refer to a local directory
        url "../local-repo"
    }
}
```

一个项目可以采用多个库。Gradle 会按照顺序从各个库里寻找所需的依赖文件，并且一旦找到第一个便停止搜索。

了解更多库相关的信息请参阅章节 50.6，“仓库”。

# #

打包发布

依赖配置也被用于发布文件[3] (页 74)我们称之为*打包发布*或*发布*。

插件对于打包提供了完美的支持，所以通常而言无需特别告诉 Gradle 需要做什么。但是你需要告诉 Gradle 发布到哪里。这就需要在 uploadArchives 任务中添加一个仓库。下面的例子是如何发布到远程 Ivy 仓库的：

#

发布到 Ivy 仓库

build.gradle

```
uploadArchives {
    repositories {
        ivy {
            credentials {
                username "username"
                password "pw"
            }
            url "http://repo.mycompany.com"
        }
    }
}
```

执行 gradle uploadArchives，Gradle 便会构建并上传你的 jar 包，同时会生成一个 ivy.xml 一起上传到目标仓库。

当然你也可以发布到 Maven 仓库中。语法只需稍微一换就可以了。[4] (页 74)

p.s：发布到 Maven 仓库你需要 Maven 插件的支持，当然，Gradle 也会同时产生 pom.xml 一起上传到目标仓库。

#

发布到 Maven 仓库

build.gradle

```
apply plugin: 'maven'
uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: "file://localhost/tmp/myRepo/")
        }
    }
}
```

# #

下一步目标

若对 DSL 感兴趣，请看 Project.configurations{}，Project.repositories{} 和 Project.dependencies{}。

另外，继续顺着手册学习其它章节内容吧。~

[3] 这让人感觉有点囧，我们正在尝试逐步分离两种概念。

[4] 我们也在努力改进语法让发布到 Maven 仓库不那么费劲。

8

Groovy 快速入门

要构建一个 Groovy 项目，你需要使用 Groovy 插件。该插件扩展了 Java 插件，对你的项目增加了 Groovy 的编译功能. 你的项目可以包含 Groovy 源码，Java 源码，或者两者都包含。在其他各方面，Groovy 项目与我们在第七章 Java 快速入门中所看到的Java 项目几乎相同。

#

一个基本的 Groovy 项目

让我们来看一个例子。要使用 Groovy 插件，你需要在构建脚本文件当中添加以下内容：

例子 Groovy plugin

build.gradle

```
apply plugin: 'groovy'
```

注意： 此例子的代码可以在 Gradle 的二进制文件或源码中的 `samples/groovy/quickstart` 里看到。

这段代码同时会将 Java 插件应用到 project 中，如果 Java 插件还没被应用的话。Groovy 插件继承了 compile 任务 ，在 `src/main/groovy` 目录中查找源文件；且继承了 compileTest 任务，在 `src/test/groovy` 目录中查找测试的源文件。这些编译任务对这些目录使用了联合编译，这意味着它们可以同时包含 java 和 groovy 源文件。

要使用 groovy 编译任务，还必须声明要使用的 Groovy 版本以及从哪里获取 Groovy 库。你可以通过在 groovy 配置中添加依赖来完成。compile 配置继承了这个依赖,从而在编译 Groovy和 Java 源代码时，groovy 库也会被包含在类路径中。下面例子中，我们会使用 Maven 中央仓库中的 Groovy 2.2.0 版本。

例子 Dependency on Groovy 2.2.0

build.gradle

```
repositories {
    mavenCentral()
}
dependencies {
    compile 'org.codehaus.groovy:groovy-all:2.2.0'
}
```

这里是我们写好的构建文件：

例子 Groovy example - complete build file

build.gradle

```
apply plugin: 'eclipse'
apply plugin: 'groovy'
repositories {
```

```
    mavenCentral()
}
dependencies {
    compile 'org.codehaus.groovy:groovy-all:2.2.0'
    testCompile 'junit:junit:4.11'
}
```

运行 gradle build 将会对你的项目进行编译，测试和打成 jar 包。

# #

总结

这一章描述了一个很简单的 Groovy 项目。通常情况下，一个真实的项目所需要的不止于此。因为一个 Groovy 项目也 是一个 Java 项目， 由于 Groovy 工程也是一个 Java 工程，因此你能用 Java 做的事情 Groovy 也能做。

你可以参阅 Groovy 插件 去了解更多关于 Groovy 插件的内容，或在 Gradle 发行包的 samples/groovy 目录中找到更多的 Groovy 项目示例。

9

Web 工程构建

本章介绍了 Gradle 对 Web 工程的相关支持。Gradle 为 Web 开发提供了两个主要插件，War plugin 和 Jetty plugin。 其中 War plugin 继承自 Java plugin，可以用来打 war 包。jetty plugin 继承自 War plugin 作为工程部署的容器。

# #

打 War 包

需要打包 War 文件，需要在脚本中使用 War plugin：

#

War plugin

build.gradle

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

备注：本示例代码可以在 Gradle 发行包中的 samples/webApplication/quickstart 路径下找到。

由于继承自 Java 插件，当你执行 gradle build 时，将会编译、测试、打包你的工程。Gradle 会在 `src/main/webapp` 下寻找 Web 工程文件。编译后的 classes 文件以及运行时依赖也都会被包含在 War 包中。

> Groovy web构建
>
> 在一个工程中你可以采用多个插件。比如你可以在 web 工程中同时使用 War plugin 和 Groovy plugin。插件会将 Gradle 依赖添加到你的 War 包中。

# #

Web 工程启动

要启动 Web 工程，只需使用 Jetty plugin 即可：

#

采用 Jetty plugin 启动 web 工程

build.gradle

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

由于 Jetty plugin 继承自 War plugin。调用 gradle jettyRun 将会把你的工程启动部署到 jetty 容器中。调用 gradle jettyRunWar 会打包并启动部署到 jetty 容器中。

待添加：使用哪个 URL，配置端口，使用源文件的地方，可编辑你的文件，以及重新加载的内容。

10

Gradle 命令行的基本使用

本章介绍了命令行的基本使用。正如在前面的章节里你所见到的调用 gradle 命令来完成一些功能。

#

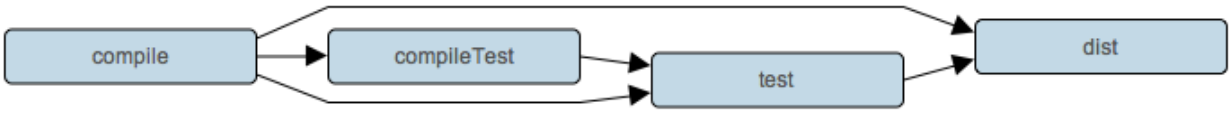
多任务调用

你可以以列表的形式在命令行中一次调用多个任务。例如 gradle compile test 命令会依次调用，并且每个任务仅会被调用一次。compile 和 test 任务以及它们的依赖任务。无论它们是否被包含在脚本中：即无论是以命令行的形式定义的任务还是依赖于其它任务都会被调用执行。来看下面的例子。

下面定义了四个任务。dist 和 test 都依赖于 compile，只用当 compile 被调用之后才会调用 gradle dist test 任务。

#

任务依赖



#

多任务调用

build.gradle

```
task compile << {
    println 'compiling source'
}
task compileTest(dependsOn: compile) << {
    println 'compiling unit tests'
}
task test(dependsOn: [compile, compileTest]) << {
    println 'running unit tests'
}
task dist(dependsOn: [compile, test]) << {
    println 'building the distribution'
}
```

gradle dist test 的输出结果。

```
\> gradle dist test
:compile
compiling source
:compileTest
compiling unit tests
:test
running unit tests
:dist
building the distribution
BUILD SUCCESSFUL
Total time: 1 secs
```

由于每个任务仅会被调用一次，所以调用 gradle test test 与调用 gradle test 效果是相同的。

# #

排除任务

你可以用命令行选项 -x 来排除某些任务，让我们用上面的例子来示范一下。

#

排除任务

gradle dist -x test 的输出结果。

```
\> gradle dist -x test
:compile
compiling source
:dist
building the distribution
BUILD SUCCESSFUL
Total time: 1 secs
```

可以看到，test 任务并没有被调用，即使他是 dist 任务的依赖。同时 test 任务的依赖任务 compileTest 也没有被调用，而像 compile 被 test 和其它任务同时依赖的任务仍然会被调用。

#

失败后继续执行

默认情况下只要有任务调用失败 Gradle 就是中断执行。这可能会使调用过程更快，但那些后面隐藏的错误不会被发现。所以你可以使用--continue 在一次调用中尽可能多的发现所有问题。

采用了--continue 选项，Gralde会调用每一个任务以及它们依赖的任务。而不是一旦出现错误就会中断执行。所有错误信息都会在最后被列出来。

一旦某个任务执行失败,那么所有依赖于该任务的子任务都不会被调用。例如由于 test 任务依赖于 complie 任务，所以如果 compile 调用出错，test 便不会被直接或间接调用。

# #

简化任务名

当你试图调用某个任务的时候，无需输入任务的全名。只需提供足够的可以唯一区分出该任务的字符即可。例如，上面的例子你也可以这么写。用 gradle di 来直接调用 dist 任务。

#

简化任务名

gradle di 的输出结果

```
\> gradle di
:compile
compiling source
:compileTest
compiling unit tests
:test
running unit tests
:dist
building the distribution
BUILD SUCCESSFUL
Total time: 1 secs
```

你也可以用驼峰命名的任务中每个单词的首字母进行调用。例如，可以执行 gradle compTest or even gradle cT 来调用 compileTest 任务。

#

简化驼峰任务名

gradle cT 的输出结果。

```
\> gradle cT
:compile
compiling source
:compileTest
compiling unit tests
```

```
BUILD SUCCESSFUL
Total time: 1 secs
```

简化后你仍然可以使用 -x 参数。

# #

选择构建位置

调用 gradle 时，默认情况下总是会构建当前目录下的文件，可以使用-b 参数选择构建的文件，并且当你使用此参数时settings。gradle 将不会生效,看下面的例子:

#

选择文件构建

subdir/myproject.gradle

```
task hello << {
    println "using build file '$buildFile.name' in '$buildFile.parentFile.name'."
}
```

gradle -q -b subdir/myproject.gradle hello 的输出结果

```
Output of gradle -q -b subdir/myproject.gradle hello
\> gradle -q -b subdir/myproject.gradle hello
using build file 'myproject.gradle' in 'subdir'.
```

另外，你可以使用 -p 参数来指定构建的目录，例如在多项目构建中你可以用 -p 来替代 -b 参数。

#

选择构建目录

gradle -q -p subdir hello 的输出结果

```
\> gradle -q -p subdir hello
using build file 'build.gradle' in 'subdir'.
```

# #

获取构建信息

Gradle 提供了许多内置任务来收集构建信息。这些内置任务对于了解依赖结构以及解决问题都是很有帮助的。

了解更多，可以参阅项目报告插件以为你的项目添加构建报告。

#

项目列表

执行 gradle projects 会为你列出子项目名称列表。如下例。

#

收集项目信息

gradle -q projects 的输出结果

```
\> gradle -q projects
\------------------------------------------------------------
Root project
\------------------------------------------------------------
Root project 'projectReports'
+--- Project ':api' - The shared API for the application
\--- Project ':webapp' - The Web application implementation
To see a list of the tasks of a project, run gradle <project-path>:tasks
For example, try running gradle :api:tasks
```

这份报告展示了每个项目的描述信息。当然你可以在项目中用 description 属性来指定这些描述信息。

#

为项目添加描述信息.

build.gradle

```
description = 'The shared API for the application'
```

#

任务列表

执行 gradle tasks 会列出项目中所有任务。这会显示项目中所有的默认任务以及每个任务的描述。如下例

#

获取任务信息

gradle -q tasks的输出结果：

```
\> gradle -q tasks
\------------------------------------------------------------
All tasks runnable from root project
\------------------------------------------------------------
Default tasks: dists
Build tasks
\-----------
clean - Deletes the build directory (build)
dists - Builds the distribution
libs - Builds the JAR
Build Setup tasks
\-----------------
init - Initializes a new Gradle build. [incubating]
wrapper - Generates Gradle wrapper files. [incubating]
Help tasks
\----------
dependencies - Displays all dependencies declared in root project 'projectReports'.
dependencyInsight - Displays the insight into a specific dependency in root project 'projectReports'.
help - Displays a help message
projects - Displays the sub-projects of root project 'projectReports'.
properties - Displays the properties of root project 'projectReports'.
tasks - Displays the tasks runnable from root project 'projectReports' (some of the displayed tasks may belong to subpro
To see all tasks and more detail, run with --all.
```

默认情况下，这只会显示那些被分组的任务。你可以通过为任务设置 group 属性和 description 来把 这些信息展示到结果中。

#

更改任务报告内容

build.gradle

```
dists {
    description = 'Builds the distribution'
    group = 'build'
}
```

当然你也可以用--all 参数来收集更多任务信息。这会列出项目中所有任务以及任务之间的依赖关系。

#

Obtaining more information about tasks

gradle -q tasks --all的输出结果：

```
\> gradle -q tasks --all
\------------------------------------------------------------
All tasks runnable from root project
\------------------------------------------------------------
Default tasks: dists
Build tasks
\-----------
clean - Deletes the build directory (build)
api:clean - Deletes the build directory (build)
webapp:clean - Deletes the build directory (build)
dists - Builds the distribution [api:libs, webapp:libs]
    docs - Builds the documentation
api:libs - Builds the JAR
    api:compile - Compiles the source files
webapp:libs - Builds the JAR [api:libs]
    webapp:compile - Compiles the source files
Build Setup tasks
\-----------------
init - Initializes a new Gradle build. [incubating]
wrapper - Generates Gradle wrapper files. [incubating]
Help tasks
\----------
dependencies - Displays all dependencies declared in root project 'projectReports'.
```

```
dependencyInsight - Displays the insight into a specific dependency in root project 'projectReports'.
help - Displays a help message
projects - Displays the sub-projects of root project 'projectReports'.
properties - Displays the properties of root project 'projectReports'.
tasks - Displays the tasks runnable from root project 'projectReports' (some of the displayed tasks may belong to subpro
```

#

获取任务帮助信息

执行 gradle help --task someTask 可以显示指定任务的详细信息。或者多项目构建中相同任务名称的所有任务的信息。如下例。

#

获取任务帮助

gradle -q help --task libs 的输出结果

```
\> gradle -q help --task libs
Detailed task information for libs
Paths
    :api:libs
    :webapp:libs
Type
    Task (org.gradle.api.Task)
Description
    Builds the JAR
```

这些结果包含了任务的路径、类型以及描述信息等。

#

获取依赖列表

执行 gradle dependencies 会列出项目的依赖列表，所有依赖会根据任务区分，以树型结构展示出来。如下例。

#

获取依赖信息

gradle -q dependencies api:dependencies webapp:dependencies 的输出结果

```
\> gradle -q dependencies api:dependencies webapp:dependencies
\------------------------------------------------------------
Root project
\------------------------------------------------------------
No configurations
\------------------------------------------------------------
Project :api - The shared API for the application
\------------------------------------------------------------
compile
\--- org.codehaus.groovy:groovy-all:2.2.0
testCompile
\--- junit:junit:4.11
    \--- org.hamcrest:hamcrest-core:1.3
\------------------------------------------------------------
Project :webapp - The Web application implementation
\------------------------------------------------------------
compile
+--- project :api
|    \--- org.codehaus.groovy:groovy-all:2.2.0
\--- commons-io:commons-io:1.2
testCompile
No dependencies
```

虽然输出结果很多，但这对于了解构建任务十分有用，当然你可以通过--configuration 参数来查看 指定构建任务的依赖情况。

#

过滤依赖信息

gradle -q api:dependencies --configuration testCompile 的输出结果

```
\> gradle -q api:dependencies --configuration testCompile
\------------------------------------------------------------
Project :api - The shared API for the application
\------------------------------------------------------------
```

```
testCompile
\--- junit:junit:4.11
     \--- org.hamcrest:hamcrest-core:1.3
```

#

查看特定依赖

执行 Running gradle dependencyInsight 可以查看指定的依赖情况。如下例。

#

获取特定依赖

gradle -q webapp:dependencyInsight --dependency groovy --configuration compile 的输出结果

```
\> gradle -q webapp:dependencyInsight --dependency groovy --configuration compile
org.codehaus.groovy:groovy-all:2.2.0
\--- project :api
     \--- compile
```

这对于了解依赖关系、了解为何选择此版本作为依赖十分有用。了解更多请参阅依赖检查报告。

dependencyInsight 任务是'Help'任务组中的一个。这项任务需要进行配置才可以。如果用了 Java 相关的插件，那么 dependencyInsight 任务已经预先被配置到'Compile'下了。你只需要通过'--dependency'参数来制定所需查看的依赖即可。如果你不想用默认配置的参数项你可以通过 '--configuration' 参数来进行指定。了解更多请参阅依赖检查报告。

#

获取项目属性列表

执行 gradle properties 可以获取项目所有属性列表。如下例。

#

属性信息

gradle -q api:properties 的输出结果

```
\> gradle -q api:properties
\------------------------------------------------------------
Project :api - The shared API for the application
\------------------------------------------------------------
allprojects: [project ':api']
ant: org.gradle.api.internal.project.DefaultAntBuilder@12345
antBuilderFactory: org.gradle.api.internal.project.DefaultAntBuilderFactory@12345
artifacts: org.gradle.api.internal.artifacts.dsl.DefaultArtifactHandler@12345
asDynamicObject: org.gradle.api.internal.ExtensibleDynamicObject@12345
buildDir: /home/user/gradle/samples/userguide/tutorial/projectReports/api/build
buildFile: /home/user/gradle/samples/userguide/tutorial/projectReports/api/build.gradle
```

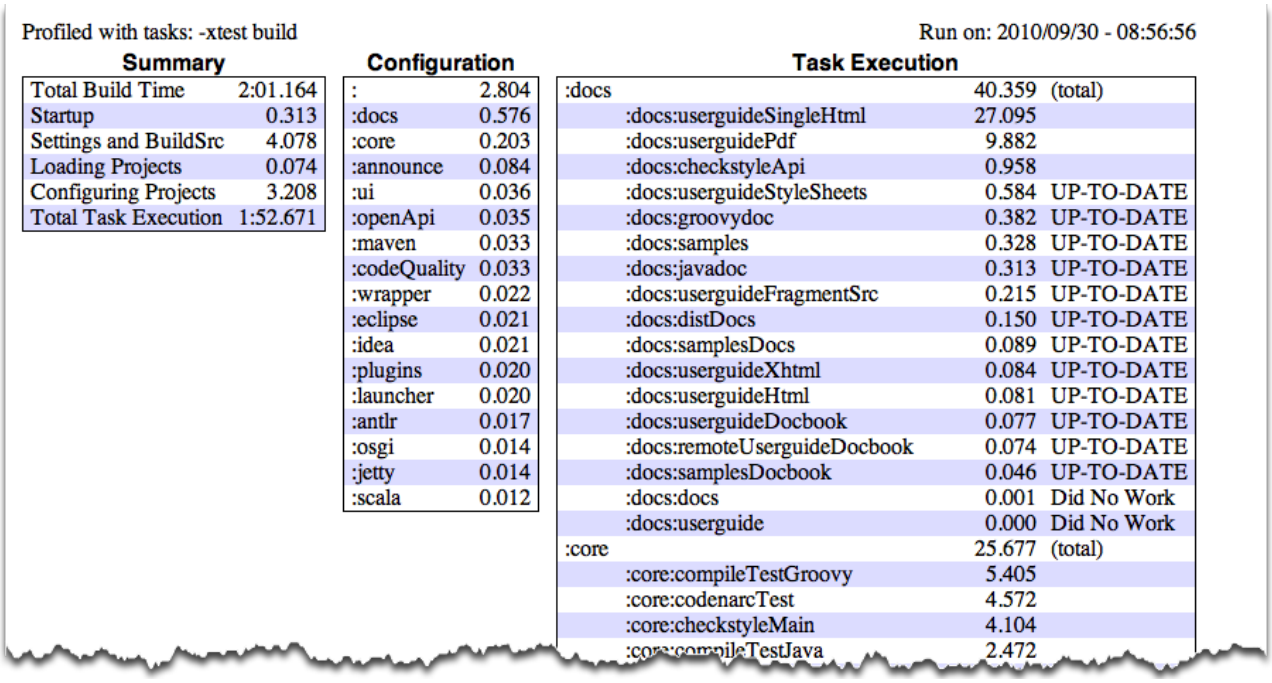
#

构建日志

--profile 参数可以收集一些构建期间的信息并保存到 build/reports/profile 目录下并且以构建时间命名这些文件。

下面这份日志记录了总体花费时间以及各过程花费的时间，并以时间大小倒序排列，并且记录了任务的执行情况。

如果采用了 buildSrc，那么在 buildSrc/build 下同时也会生成一份日志记录记录。



Profiled with tasks: -xtest build

Run on: 2010/09/30 - 08:56:56

Summary

Total Build Time	2:01.164
Startup	0.313
Settings and BuildSrc	4.078
Loading Projects	0.074
Configuring Projects	3.208
Total Task Execution	1:52.671

Configuration

:	2.804
:docs	0.576
:core	0.203
:announce	0.084
:ui	0.036
:openApi	0.035
:maven	0.033
:codeQuality	0.033
:wrapper	0.022
:eclipse	0.021
:idea	0.021
:plugins	0.020
:launcher	0.020
:antlr	0.017
:osgi	0.014
:jetty	0.014
:scala	0.012

Task Execution

:docs	40.359	(total)
:docs:userguideSingleHtml	27.095	
:docs:userguidePdf	9.882	
:docs:checkstyleApi	0.958	
:docs:userguideStyleSheets	0.584	UP-TO-DATE
:docs:groovydoc	0.382	UP-TO-DATE
:docs:samples	0.328	UP-TO-DATE
:docs:javadoc	0.313	UP-TO-DATE
:docs:userguideFragmentSrc	0.215	UP-TO-DATE
:docs:distDocs	0.150	UP-TO-DATE
:docs:samplesDocs	0.089	UP-TO-DATE
:docs:userguideXhtml	0.084	UP-TO-DATE
:docs:userguideHtml	0.081	UP-TO-DATE
:docs:userguideDocbook	0.077	UP-TO-DATE
:docs:remoteUserguideDocbook	0.074	UP-TO-DATE
:docs:samplesDocbook	0.046	UP-TO-DATE
:docs:docs	0.001	Did No Work
:docs:userguide	0.000	Did No Work
:core	25.677	(total)
:core:compileTestGroovy	5.405	
:core:codenarcTest	4.572	
:core:checkstyleMain	4.104	
:core:compileTestJava	2.472	

#

Dry Run

有时可能你只想知道某个任务在一个任务集中按顺序执行的结果，但并不想实际执行这些任务。那么你可以用-m参数。例如 gradle -m clean compile 会调用 clean 和 compile，这与 tasks 可以形成互补，让你知道哪些任务可以用于执行。

#

本章小结

在本章中你学到很多用命令行可以做的事情，了解更多你可以参阅 gradle command in 附录 D， Gradle 命令行。

11

使用 Gradle 图形用户界面

除了支持传统的命令行界面，Gradle 也提供了一个图形用户界面（GUI）。这是一个独立的用户界面，可以通过加上 --gui 参数来启动。

#

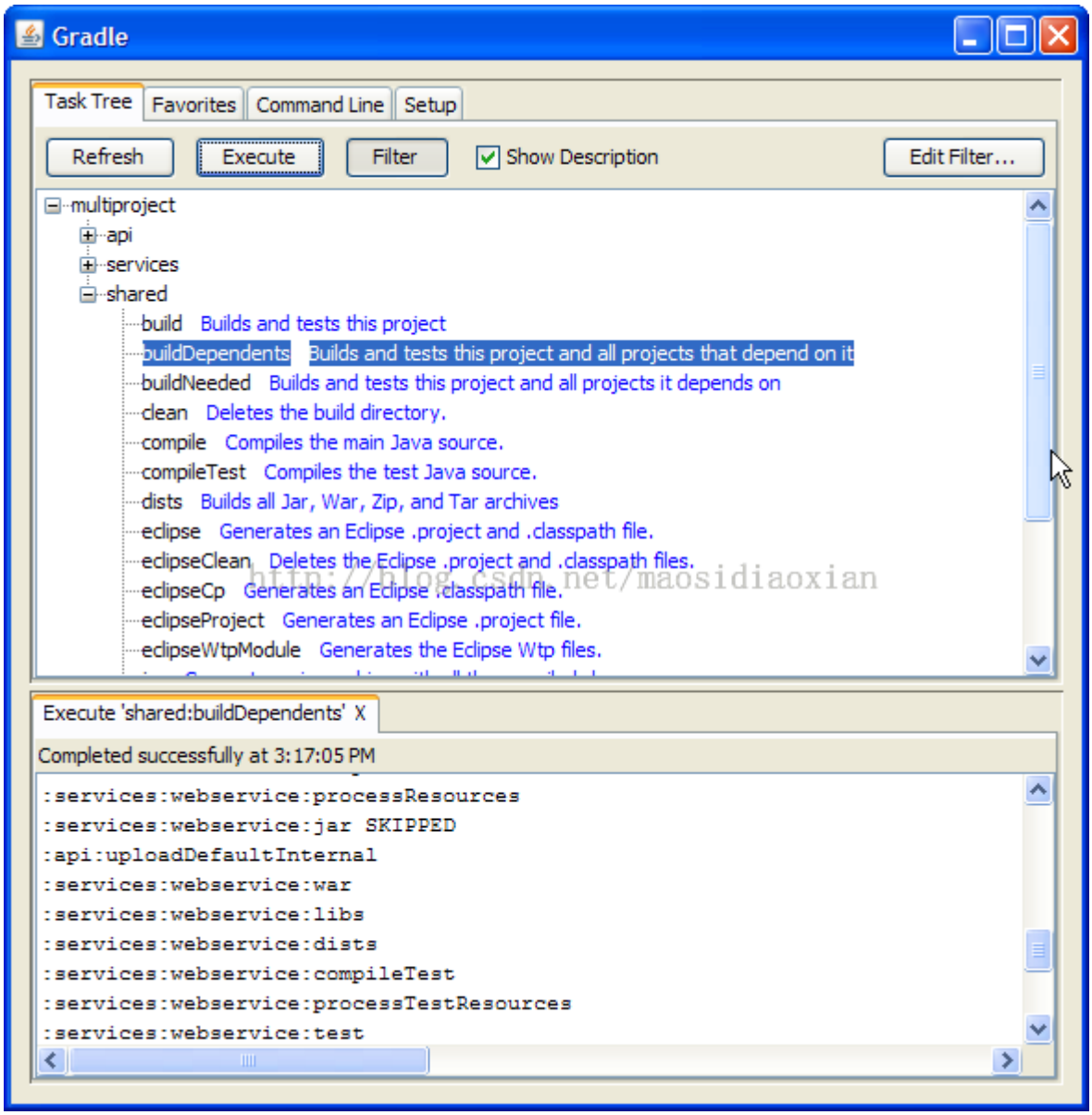
Launching the GUI

```
gradle --gui
```

注意：此命令行窗口被将锁定，直到 Gradle GUI 被关闭。如果是在 linux/unix 系统下，则可以通过(gradle --gui&)让它作为后台任务运行。

如果你在你的 Gradle 项目目录下运行 Gradle GUI，你应该会看到一个任务树。

GUI Task Tree



最好是从 Gradle 项目目录运行此命令，这样对 UI 的设置就可以存储在你的项目目录中。当然，你也可以先运行它，然后通过在 UI 中的设置（Setup）选项卡，改变工作目录。

在 Gradle 的用户界面（UI）中，上面是 4 个选项卡，下面则是输出窗口。

# #

任务树

任务树显示了所有项目和它们的任务的层次结构。双击一个任务可以执行它。

这里还提供了一个过滤器，可以把比较少用的任务隐藏。你可以通过过滤器（Filter）按钮切换是否进行过滤。通过编辑过滤器，你可以对哪些任务和项目要显示进行配置。隐藏的任务显示为红色。注意：新创建的任务默认情况下是显示状态（而不是隐藏状态）

任务树的上下文菜单会提供以下选项：

- 执行忽略依赖关系。这使得重新构建时不去依赖项目（与 -a 选项一样）
- 将任务添加到收藏夹（见收藏夹（Favourites）选项卡）
- 隐藏选择的任务。这将会把它们添加到过滤器中。
- 编辑 build.gradle 文件。注意：该操作需要 Java 1.6 或更高的版本，并且要求在你的操作系统中关联 gradle 文件。

#

收藏夹

收藏夹选项卡用来储存经常执行的命令。这些命令可以是复杂的命令（只要它们符合 Gradle 的语法），你可以给它们设置一个显示名称。它用于创建一个自定义的命令，来显示地跳过测试，文档，例子。你可以称之为“快速构建”。

你可以根据自己的喜好，对收藏夹进行排序，甚至可以把它们导出到磁盘，并在其他地方导入。如果你在编辑它们的时候，选上“始终显示实时输出”，它只有在你选上“当发生错误时才显示输出”时有效。它会始终强制显示输出。

#

命令行

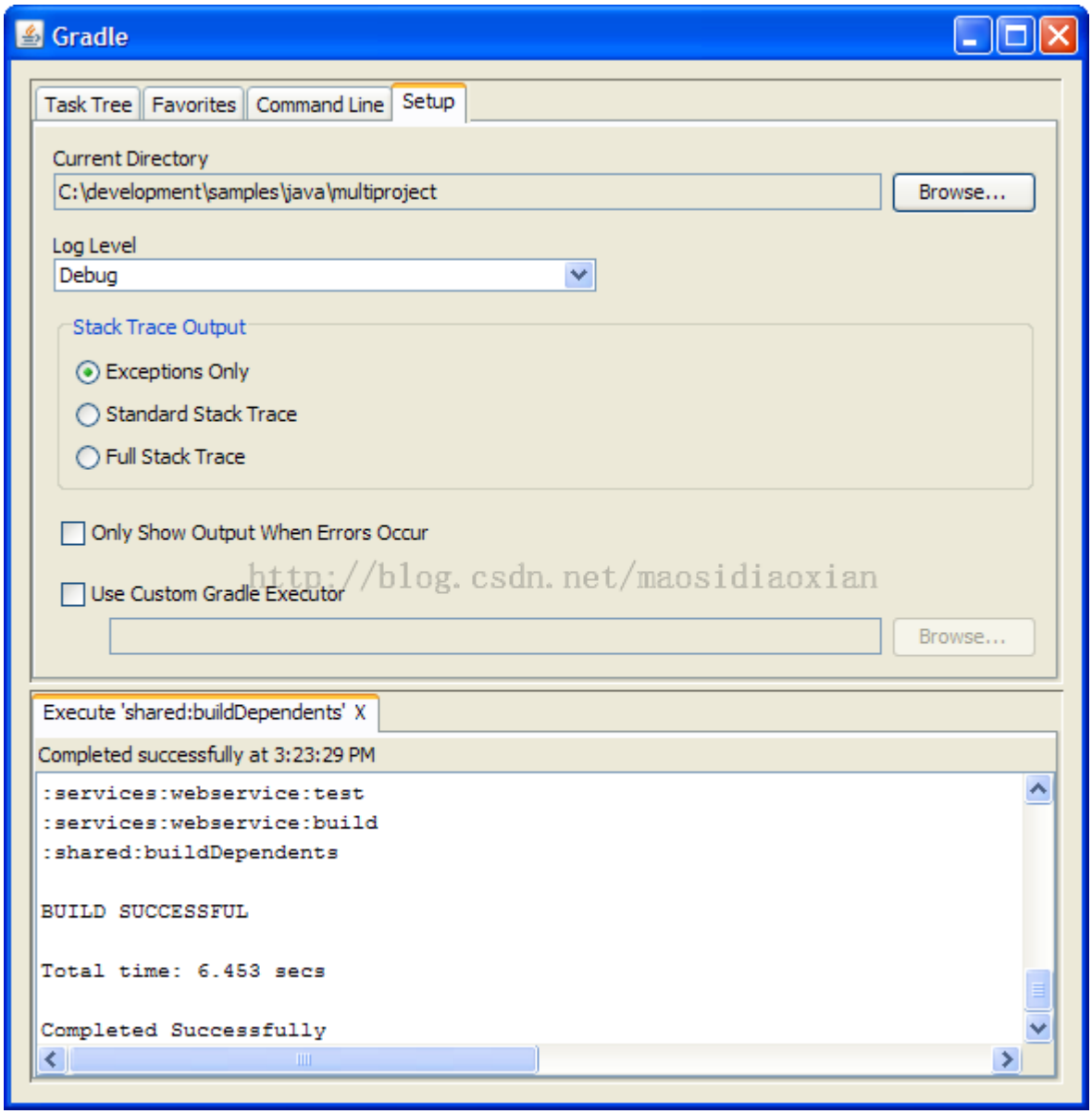
命令行选项卡是直接执行单个的 Gradle 命令的地方。你只需要输入命令行中你经常在“Gradle”后面输入的命令即可。它也对要添加到收藏夹的命令提供了先去尝试的地方。

#

设置

设置（Setup）选项卡允许你配置一些常规的设置

GUI Setup



- 当前目录 定义了你的 Gradle 项目（通常是 build.gradle 所在的位置）的根目录。
- 堆栈跟踪输出 这决定了当出现错误时，有多少信息定到堆栈跟踪。注意：如果你在命令行或收藏夹选项卡上指定了堆栈跟踪级别，将会覆盖这里的设置。

- 只在出现错误时显示输出 启用此选项将在任务执行时隐藏任何输出，除非构建失败。
- 使用自定义的 Gradle 执行器 – 高级功能 这为你提供了启动 Gradle 命令行的替代方法。这是很有用的。如果你的项目需要在另一个批处理文件或 shell 脚本中做一些额外的配置（比如指定一个初始化脚本）。

12

编写构建脚本

这一章着眼于一些编写构建脚本的详细信息。

# #

Gradle 构建语言

Gradle 提供一种领域特定语言或者说是 DSL，来描述构建。这种构建语言基于 Groovy 中，并进行了一些补充，使其易于描述构建。

#

Project API

在Java 构建入门的教程中，我们使用了 apply ()方法。这方法从何而来？我们之前说在 Gradle 中构建脚本定义了一个项目（ project ）。在构建的每一个项目中，Gradle 创建了一个 Project 类型的实例，并在构建脚本中关联此 Project 对象。当构建脚本执行时，它会配置此 Project 对象：

- 在构建脚本中，你所调用的任何一个方法，如果在构建脚本中未定义，它将被委托给 Project 对象。
- 在构建脚本中，你所访问的任何一个属性，如果在构建脚本里未定义，它也会被委托给 Project 对象。

下面我们来试试这个，试试访问 Project 对象的 name 属性。

访问 Project 对象的属性

build.gradle

```
println name
println project.name
```

gradle -q check 的输出结果

```
> gradle -q check
projectApi
projectApi
```

这两个 println 语句打印出相同的属性。在生成脚本中未定义的属性，第一次使用时自动委托到 Project 对象。其他语句使用了在任何构建脚本中可以访问的 project 属性，则返回关联的 Project 对象。只有当您定义的属性或方法 Project 对象的一个成员相同名字时，你才需要使用 project 属性。

# #

标准 project 属性

Project对象提供了一些在构建脚本中可用的标准的属性。下表列出了常用的几个属性。

表 13.1. Project 属性

名称	类型	默认值
project	Project	Project 实例
name	String	项目目录的名称。
path	String	项目的绝对路径。
description	String	项目的描述。
projectDir	File	包含生成脚本的目录。
buildDir	File	projectDir /build
group	Object	未指定
version	Object	未指定
ant	AntBuilder	AntBuilder 实例

# #

Script API

当 Gradle 执行一个脚本时，它将脚本编译为一个实现了 Script 接口的类。这意味着所有由该Script 接口声明的属性和方法在您的脚本中是可用的。

#

声明变量

有两类可以在生成脚本中声明的变量： 局部变量和额外属性。

#

局部变量局部

局部变量是用 def 关键字声明的。它们只在定义它们的范围内可以被访问。局部变量是 Groovy 语言底层的一个特征。

示例 13.2. 使用局部变量

build.gradle

```
def dest = "dest"
task copy(type: Copy) {
   from "source"
   into dest
}
```

#

额外属性

Gradle 的域模型中，所有增强的对象都可以容纳用户定义的额外的属性。这包括但并不限于项目（ projec t ）、任务（ task ）和源码集（ source set ）。额外的属性可以通过所属对象的 ext 属性进行添加，读取和设置。或者，可以使用 ext 块同时添加多个属性。

13.3 例子. 使用额外属性

build.gradle

```
apply plugin: "java"
ext {
   springVersion = "3.1.0.RELEASE"
   emailNotification = "build@master.org"
}
```

```
sourceSets.all { ext.purpose = null }
sourceSets {
    main {
        purpose = "production"
    }
    test {
        purpose = "test"
    }
    plugin {
        purpose = "production"
    }
}
task printProperties << {
    println springVersion
    println emailNotification
    sourceSets.matching { it.purpose == "production" }.each { println it.name }
}
```

gradle -q printProperties的输出结果

```
> gradle -q printProperties
3.1.0.RELEASE
build@master.org
main
plugin
```

在此示例中， 一个 ext 代码块将两个额外属性添加到 project 对象中。此外，通过将ext.purpose 设置为 null（null是一个允许的值），一个名为 purpose 的属性被添加到每个源码集（source set）。一旦属性被添加，他们就可以像预定的属性一样被读取和设置。

通过添加属性所要求特殊的语法，Gradle 可以在你试图设置（预定义的或额外的）的属性，但该属性拼写错误或不存在时 fail fast。[5]额外属性在任何能够访问它们所属的对象的地方都可以被访问，这使它们有着比局部变量更广泛的作用域。父项目上的额外属性，在子项目中也可以访问。

有关额外属性和它们的 API 的详细信息，请参阅 ExtraPropertiesExtension。

# #

一些 Groovy 的基础知识

Groovy 提供了用于创建 DSL 的大量特点，并且 Gradle 构建语言利用了这些特点。了解构建语言是如何工作的，将有助于你编写构建脚本，特别是当你开始写自定义插件和任务的时候。

#

Groovy JDK

Groovy 对 JVM 的类增加了很多有用的方法。例如， iterable 新增的 each 方法，会对iterable 的元素进行遍历：

** Groovy JDK 的方法**

build.gradle

```
// Iterable gets an each() method
configurations.runtime.each { File f -> println f }
```

可以看看http://groovy.codehaus.org/groovy-jdk/，了解更多详细信息。

#

属性访问器

Groovy 会自动地把一个属性的引用转换为对适当的 getter 或 setter 方法的调用。

属性访问器

build.gradle

```
// Using a getter method
println project.buildDir
println getProject().getBuildDir()
// Using a setter method
project.buildDir = 'target'
getProject().setBuildDir('target')
```

#

括号可选的方法调用

调用方法时括号是可选的。

不带括号的方法调用

build.gradle

```
test.systemProperty 'some.prop', 'value'
test.systemProperty('some.prop', 'value')
```

#

List 和 Map

Groovy 提供了一些定义 List 和 Map 实例的快捷写法。

List and map

build.gradle

```
// List literal
test.includes = ['org/gradle/api/**', 'org/gradle/internal/**']
List<String> list = new ArrayList<String>()
list.add('org/gradle/api/**')
list.add('org/gradle/internal/**')
test.includes = list
// Map literal
apply plugin: 'java'
Map<String, String> map = new HashMap<String, String>()
map.put('plugin', 'java')
apply(map)
```

#

作为方法最后一个参数的闭包

Gradle DSL 在很多地方使用闭包。你可以在这里查看更多有关闭包的资料。当方法的最后一个参数是一个闭包时，你可以把闭包放在方法调用之后：

作为方法参数的闭包

build.gradle

```
repositories {
    println "in a closure"
}
repositories() { println "in a closure" }
repositories({ println "in a closure" })
```

#

闭包委托（delegate）

每个闭包都有一个委托对象，Groovy 使用它来查找变量和方法的引用，而不是作为闭包的局部变量或参数。Gradle 在配置闭包中使用到它，把委托对象设置为被配置的对象。

闭包委托

build.gradle

```
dependencies {
    assert delegate == project.dependencies
    compile('junit:junit:4.11')
    delegate.compile('junit:junit:4.11')
}
```

13

教程—杂七杂八

# #

创建目录

有一个常见的情况是，多个任务都依赖于某个目录的存在。当然，你可以在这些任务的开始加入 mkdir 来解决这个问题。但这是种臃肿的解决方法。这里有一个更好的解决方案 (仅适用于这些需要这个目录的任务有着 dependsOn 的关系的情况)：

使用 mkdir 创建目录

build.gradle

```
classesDir = new File('build/classes')
task resources << {
    classesDir.mkdirs()
    // do something
}
task compile(dependsOn: 'resources') << {
    if (classesDir.isDirectory()) {
        println 'The class directory exists. I can operate'
    }
    // do something
}
```

gradle -q compile的输出结果

```
> gradle -q compile
The class directory exists. I can operate
```

#

Gradle 属性和系统属性

Gradle 提供了许多方式将属性添加到您的构建中。 从Gradle 启动的 JVM，你可以使用 -D 命令行选项向它传入一个系统属性。 Gradle 命令的-D选项和 java 命令的 -D 选项有着同样的效果。

此外，您也可以通过属性文件向您的 project 对象添加属性。您可以把一个 gradle.properties 文件放在 Gradle 的用户主目录（默认为 USER_HOME/.gradle），或您的项目目录中。对于多项目构建，您可以将 gradle.properties 文件放在任何子项目的目录中。通过 project 对象，可以访问到 gradle.properties 里的属性。用户的主目录中的属性文件比项目目录中的属性文件更先被访问到。

你也可以通过使用-P命令行选项来直接向您的项目对象添加属性。在更多的用法中，您甚至可以通过系统和环境属性把属性直接传给项目对象。例如，如果你在一个持续集成服务器上运行构建，但你没有这台机器的管理员权限，而你的构建脚本需要一些不能让其他人知道的属性值，那么，您就不能使用 -P 选项。在这种情况下，您可以在项目管理部分（对普通用户不可见）添加一个环境属性。如果环境属性遵循 ORG_GRADLE_PROJECT_propertyName= somevalue 的模式，这里的 propertyName 会被添加到您的项目对象中。对系统属性我们也支持相同的机制。唯一的区别是，它是 org.gradle.projectpropertyName 的模式。

通过 gradle.properties 文件，你还可以设置系统属性。如果此类文件中的属性有一个systemProp.的前缀，该属性和它的值会被添加到系统属性，且不带此前缀。在多项目构建中，除了在根项目之外的任何项目里的 systemProp. 属性集都将被忽略。也就是，只有根项目gradle.properties 文件里的 systemProp. 属性会被作为系统属性。

使用 gradle.properties 文件设置属性

gradle.properties

```
gradlePropertiesProp=gradlePropertiesValue
systemProjectProp=shouldBeOverWrittenBySystemProp
envProjectProp=shouldBeOverWrittenByEnvProp
systemProp.system=systemValue
```

build.gradle

```
task printProps << {
    println commandLineProjectProp
    println gradlePropertiesProp
    println systemProjectProp
    println envProjectProp
```

```
    println System.properties['system']
}
```

`gradle -q -PcommandLineProjectProp=commandLineProjectPropValue -Dorg.gradle.project.systemProjectProp=systemPropertyValue printProps` 的输出结果

```
> gradle -q -PcommandLineProjectProp=commandLineProjectPropValue -Dorg.gradle.project.systemProjectProp=syst
commandLineProjectPropValue
gradlePropertiesValue
systemPropertyValue
envPropertyValue
systemValue
```

#

检查项目的属性

当你要使用一个变量时，你可以仅通过其名称在构建脚本中访问一个项目的属性。如果此属性不存在，则会引发异常，并且构建失败。如果您的构建脚本依赖于一些可选属性，而这些属性用户可能在比如 gradle.properties 文件中设置，您就需要在访问它们之前先检查它们是否存在。你可以通过使用方法 hasProperty('propertyName') 来进行检查，它返回 true 或 false。

# #

使用外部构建脚本配置项目

您可以使用外部构建脚本来配置当前项目。Gradle 构建语言的所有内容在外部脚本中也可以使用。您甚至可以在外部脚本中应用其他脚本。

使用外部构建脚本配置项目

build.gradle

```
apply from: 'other.gradle'
```

other.gradle

```
println "configuring $project"
task hello << {
    println 'hello from other script'
}
```

gradle -q hello的输出结果

```
> gradle -q hello
configuring root project 'configureProjectUsingScript'
hello from other script
```

#

配置任意对象

您可以用以下非常易理解的方式配置任意对象。

配置任意对象

build.gradle

```
task configure << {
    pos = configure(new java.text.FieldPosition(10)) {
        beginIndex = 1
        endIndex = 5
    }
    println pos.beginIndex
    println pos.endIndex
}
```

gradle -q configure 的输出结果

```
> gradle -q configure
1
5
```

# #

使用外部脚本配置任意对象

你还可以使用外部脚本配置任意对象

使用脚本配置任意对象

build.gradle

```
task configure << {
    pos = new java.text.FieldPosition(10)
    // Apply the script
    apply from: 'other.gradle', to: pos
    println pos.beginIndex
    println pos.endIndex
}
```

other.gradle

```
beginIndex = 1;
endIndex = 5;
```

gradle -q configure 的输出结果

```
> gradle -q configure
1
5
```

# #

缓存

为了提高响应速度，默认情况下 Gradle 会缓存所有已编译的脚本。这包括所有构建脚本，初始化脚本和其他脚本。你第一次运行一个项目构建时， Gradle 会创建 .gradle 目录，用于存放已编译的脚本。下次你运行此构建时， 如果该脚本自它编译后没有被修改，Gradle 会使用这个已编译的脚本。否则该脚本会重新编译，并把最新版本存在缓存中。如果您通过 --recompile-scripts 选项运行 Gradle ，会丢弃缓存的脚本，然后重新编译此脚本并将其存在缓存中。通过这种方式，您可以强制 Gradle 重新生成缓存。

14

任务详述

在入门教程构建基础中，你已经学习了如何创建简单的任务。之后您还学习了如何将其他行为添加到这些任务中。并且你已经学会了如何创建任务之间的依赖。这都是简单的任务。但 Gradle 让任务的概念更深远。Gradle 支持增强的任务，也就是，有自己的属性和方法的任务。这是真正的与你所使用的 Ant 目标（target）的不同之处。这种增强的任务可以由你提供，或由 Gradle 提供。

#

定义任务

在构建基础中我们已经看到如何通过关键字这种风格来定义任务。在某些情况中，你可能需要使用这种关键字风格的几种不同的变式。例如，在表达式中不能用这种关键字风格。

定义任务

build.gradle

```
task(hello) << {
    println "hello"
}
task(copy, type: Copy) {
    from(file('srcDir'))
    into(buildDir)
}
```

您还可以使用字符串作为任务名称：

定义任务 — — 使用字符串作为任务名称

build.gradle

```
task('hello') <<
{
    println "hello"
}
task('copy', type: Copy) {
    from(file('srcDir'))
    into(buildDir)
}
```

对于定义任务，有一种替代的语法你可能更愿意使用：

使用替代语法定义任务

build.gradle

```
tasks.create(name: 'hello') << {
    println "hello"
}
tasks.create(name: 'copy', type: Copy) {
```

```
    from(file('srcDir'))
    into(buildDir)
}
```

在这里我们将任务添加到 tasks 集合。关于 create() 方法的更多变化可以看看 TaskContainer。

#

定位任务

你经常需要在构建文件中查找你所定义的任务，例如，为了去配置或是依赖它们。对这样的情况，有很多种方法。首先，每个任务都可作为项目的一个属性，并且使用任务名称作为这个属性名称：

以属性方式访问任务

build.gradle

```
task hello
println hello.name
println project.hello.name
```

任务也可以通过 tasks 集合来访问。

通过 tasks 集合访问任务

build.gradle

```
task hello
println tasks.hello.name
println tasks['hello'].name
```

您可以从任何项目中，使用 tasks.getByPath() 方法获取任务路径并且通过这个路径来访问任务。你可以用任务名称，相对路径或者是绝对路径作为参数调用 getByPath() 方法。

通过路径访问任务

build.gradle

```
project(':projectA') {
    task hello
}
task hello
println tasks.getByPath('hello').path
println tasks.getByPath(':hello').path
println tasks.getByPath('projectA:hello').path
println tasks.getByPath(':projectA:hello').path
```

gradle -q hello的输出结果

```
> gradle -q hello
:hello
:hello
:projectA:hello
:projectA:hello
```

有关查找任务的更多选项，可以看一下 TaskContainer。

#

配置任务

作为一个例子，让我们看看由 Gradle 提供的 Copy 任务。若要创建 Copy 任务，您可以在构建脚本中声明：

创建一个复制任务

build.gradle

```
task myCopy(type: Copy)
```

上面的代码创建了一个什么都没做的复制任务。可以使用它的 API 来配置这个任务（见 Copy）。下面的示例演示了几种不同的方式来实现相同的配置。

配置任务的几种方式

build.gradle

```
Copy myCopy = task(myCopy, type: Copy)
myCopy.from 'resources'
myCopy.into 'target'
myCopy.include('**/*.txt', '**/*.xml', '**/*.properties')
```

这类似于我们通常在 Java 中配置对象的方式。您必须在每一次的配置语句重复上下文（myCopy）。这显得很冗余并且很不好读。

还有另一种配置任务的方式。它也保留了上下文，且可以说是可读性最强的。它是我们通常最喜欢的方式。

配置任务–使用闭包

build.gradle

```
task myCopy(type: Copy)
myCopy {
  from 'resources'
  into 'target'
  include('**/*.txt', '**/*.xml', '**/*.properties')
}
```

这种方式适用于任何任务。该例子的第 3 行只是 tasks.getByName() 方法的简洁写法。特别要注意的是，如果您向 getByName() 方法传入一个闭包，这个闭包的应用是在配置这个任务的时候，而不是任务执行的时候。

您也可以在定义一个任务的时候使用一个配置闭包。

使用闭包定义任务

build.gradle

```
task copy(type: Copy) {
  from 'resources'
  into 'target'
  include('**/*.txt', '**/*.xml', '**/*.properties')
}
```

#

对任务添加依赖

定义任务的依赖关系有几种方法。在任务依赖中，已经向你介绍了使用任务名称来定义依赖。任务的名称可以指向同一个项目中的任务，或者其他项目中的任务。要引用另一个项目中的任务，你需要把它所属的项目的路径作为前缀加到它的名字中。下面是一个示例，添加了从 projectA:taskX 到 projectB:taskY 的依赖关系：

从另一个项目的任务上添加依赖

build.gradle

```
project('projectA') {
    task taskX(dependsOn: ':projectB:taskY') << {
        println 'taskX'
    }
}
project('projectB') {
    task taskY << {
        println 'taskY'
    }
}
```

gradle -q taskX 的输出结果

```
> gradle -q taskX
taskY
taskX
```

您可以使用一个 Task 对象而不是任务名称来定义依赖，如下：

使用 task 对象添加依赖

build.gradle

```
task taskX << {
    println 'taskX'
}
task taskY << {
    println 'taskY'
}
taskX.dependsOn taskY
```

gradle -q taskX的输出结果

```
> gradle -q taskX
taskY
taskX
```

对于更高级的用法，您可以使用闭包来定义任务依赖。在计算依赖时，闭包会被传入正在计算依赖的任务。这个闭包应该返回一个 Task 对象或是 Task 对象的集合，返回值会被作为这个任务的依赖项。下面的示例是从 task X 加入了项目中所有名称以 lib 开头的任务的依赖：

使用闭包添加依赖

build.gradle

```
task taskX << {
    println 'taskX'
}
taskX.dependsOn {
    tasks.findAll { task -> task.name.startsWith('lib') }
}
task lib1 << {
    println 'lib1'
}
task lib2 << {
    println 'lib2'
}
task notALib << {
    println 'notALib'
}
```

gradle -q taskX 的输出结果

```
> gradle -q taskX
lib1
lib2
taskX
```

有关任务依赖的详细信息，请参阅 Task 的 API。

#

任务排序

任务排序还是一个孵化中的功能。请注意此功能在以后的 Gradle 版本中可能会改变。

在某些情况下，控制两个任务的执行的顺序，而不引入这些任务之间的显式依赖，是很有用的。任务排序和任务依赖之间的主要区别是，排序规则不会影响那些任务的执行，而仅将执行的顺序。

任务排序在许多情况下可能很有用：

- 强制任务顺序执行： 如，'build' 永远不会在 'clean' 前面执行。
- 在构建中尽早进行构建验证：如，验证在开始发布的工作前有一个正确的证书。
- 通过在长久验证前运行快速验证以得到更快的反馈：如，单元测试应在集成测试之前运行。
- 一个任务聚合了某一特定类型的所有任务的结果：如，测试报告任务结合了所有执行的测试任务的输出。

有两种排序规则是可用的："必须在之后运行"和"应该在之后运行"。

通过使用 “ 必须在之后运行”的排序规则，您可以指定 taskB 必须总是运行在 taskA 之后，无论 taskA 和 taskB 这两个任务在什么时候被调度执行。这被表示为 taskB.mustRunAfter(taskA) 。“应该在之后运行”的排序规则与其类似，但没有那么严格，因为它在两种情况下会被忽略。首先是如果使用这一规则引入了一个排序循环。其次，当使用并行执行，并且一个任务的所有依赖项除了任务应该在之后运行之外所有条件已满足，那么这个任务将会运行，不管它的“应该在之后运行”的依赖项是否已经运行了。当倾向于更快的反馈时，会使用“应该在之后运行”的规则，因为这种排序很有帮助但要求不严格。

目前使用这些规则仍有可能出现 taskA 执行而 taskB 没有执行，或者 taskB 执行而 taskA 没有执行。

添加 '必须在之后运行 ' 的任务排序

build.gradle

```
task taskX << {
    println 'taskX'
}
task taskY << {
    println 'taskY'
}
taskY.mustRunAfter taskX
```

gradle -q taskY taskX 的输出结果

```
> gradle -q taskY taskX
taskX
taskY
```

添加 '应该在之后运行 ' 的任务排序

build.gradle

```
task taskX << {
    println 'taskX'
}
task taskY << {
    println 'taskY'
}
taskY.shouldRunAfter taskX
```

gradle -q taskY taskX 的输出结果

```
> gradle -q taskY taskX
taskX
taskY
```

在上面的例子中，它仍有可能执行 taskY 而不会导致 taskX 也运行：

任务排序并不意味着任务执行

gradle -q taskY 的输出结果

```
> gradle -q taskY
taskY
```

如果想指定两个任务之间的“必须在之后运行”和“应该在之后运行”排序，可以使用 Task.mustRunAfter() 和 Task.shouldRunAfter() 方法。这些方法接受一个任务实例、 任务名称或 Task.dependsOn()所接受的任何其他输入作为参数。

请注意"B.mustRunAfter(A)"或"B.shouldRunAfter(A)"并不意味着这些任务之间的任何执行上的依赖关系：

- 它是可以独立地执行任务 A 和 B 的。排序规则仅在这两项任务计划执行时起作用。
- 当--continue 参数运行时，可能会是 A 执行失败后B执行了。

如之前所述，如果“应该在之后运行”的排序规则引入了排序循环，那么它将会被忽略。

当引入循环时，“应该在其之后运行”的任务排序会被忽略

build.gradle

```
task taskX << {
    println 'taskX'
}
task taskY << {
    println 'taskY'
}
task taskZ << {
    println 'taskZ'
}
taskX.dependsOn taskY
taskY.dependsOn taskZ
taskZ.shouldRunAfter taskX
```

gradle -q taskX 的输出结果

```
> gradle -q taskX
taskZ
taskY
taskX
```

#

向任务添加描述

你可以向你的任务添加描述。例如，当执行 gradle tasks 时显示这个描述。

向任务添加描述

build.gradle

```
task copy(type: Copy) {
  description 'Copies the resource directory to the target directory.'
  from 'resources'
  into 'target'
  include('**/*.txt', '**/*.xml', '**/*.properties')
}
```

#

替换任务

有时您想要替换一个任务。例如，您想要把通过 Java 插件添加的一个任务与不同类型的一个自定义任务进行交换。你可以这样实现：

重写任务

build.gradle

```
task copy(type: Copy)
task copy(overwrite: true) << {
    println('I am the new one.')
}
```

gradle -q copy 的输出结果

```
> gradle -q copy
I am the new one.
```

在这里我们用一个简单的任务替换 Copy 类型的任务。当创建这个简单的任务时，您必须将 overwrite 属性设置为 true。否则 Gradle 将抛出异常，说这种名称的任务已经存在。

#

跳过任务

Gradle 提供多种方式来跳过任务的执行。

#

使用断言

你可以使用 onlyIf()方法将断言附加到一项任务中。如果断言结果为 true，才会执行任务的操作。你可以用一个闭包来实现断言。闭包会作为一个参数传给任务，并且任务应该执行时返回 true，或任务应该跳过时返回 false。断言只在任务要执行前才计算。

使用断言跳过一个任务

build.gradle

```
task hello << {
    println 'hello world'
}
hello.onlyIf { !project.hasProperty('skipHello') }
```

gradle hello -PskipHello 的输出结果

```
> gradle hello -PskipHello
:hello SKIPPED
BUILD SUCCESSFUL
Total time: 1 secs
```

#

使用 StopExecutionException

如果跳过任务的规则不能与断言同时表达，您可以使用 StopExecutionException。如果一个操作（action）抛出了此异常，那么这个操作（action）接下来的行为和这个任务的其他 操作（action）都会被跳过。构建会继续执行下一个任务。

使用 StopExecutionException 跳过任务

build.gradle

```
task compile << {
    println 'We are doing the compile.'
}
compile.doFirst {
    // Here you would put arbitrary conditions in real life. But we use this as an integration test, so we want defined behavior
    if (true) { throw new StopExecutionException() }
}
task myTask(dependsOn: 'compile') << {
   println 'I am not affected'
}
```

gradle -q myTask 的输出结果

```
> gradle -q myTask
I am not affected
```

如果您使用由 Gradle 提供的任务，那么此功能将非常有用。它允许您向一个任务的内置操作中添加执行条件。

#

启用和禁用任务

每一项任务有一个默认值为 true 的 enabled 标记。将它设置为 false，可以不让这个任务的任何操作执行。

启用和禁用任务

build.gradle

```
task disableMe << {
    println 'This should not be printed if the task is disabled.'
}
disableMe.enabled = false
```

Gradle disableMe 的输出结果

```
> gradle disableMe
:disableMe SKIPPED
BUILD SUCCESSFUL
Total time: 1 secs
```

#

跳过处于最新状态的任务

如果您使用 Gradle 自带的任务，如 Java 插件所添加的任务的话，你可能已经注意到 Gradle 将跳过处于最新状态的任务。这种行在您自己定义的任务上也有效，而不仅仅是内置任务。

#

声明一个任务的输入和输出

让我们来看一个例子。在这里我们的任务从一个 XML 源文件生成多个输出文件。让我们运行它几次。

一个生成任务

build.gradle

```
task transform {
    ext.srcFile = file('mountains.xml')
    ext.destDir = new File(buildDir, 'generated')
    doLast {
        println "Transforming source file."
        destDir.mkdirs()
        def mountains = new XmlParser().parse(srcFile)
        mountains.mountain.each { mountain ->
            def name = mountain.name[0].text()
            def height = mountain.height[0].text()
            def destFile = new File(destDir, "${name}.txt")
            destFile.text = "$name -> ${height}\n"
        }
    }
}
```

gradle transform 的输出结果

```
> gradle transform
:transform
Transforming source file.
```

gradle transform的输出结果

```
> gradle transform
:transform
Transforming source file.
```

请注意 Gradle 第二次执行执行这项任务时，即使什么都未作改变，也没有跳过该任务。我们的示例任务被用一个操作（action）闭包来定义。Gradle 不知道这个闭包做了什么，也无法自动判断这个任务是否为最新状态。若要使用 Gradle 的最新状态（up-to-date）检查，您需要声明这个任务的输入和输出。

每个任务都有一个 inputs 和 outputs 的属性，用来声明任务的输入和输出。下面，我们修改了我们的示例，声明它将 XML 源文件作为输入，并产生输出到一个目标目录。让我们运行它几次。

声明一个任务的输入和输出

build.gradle

```
task transform {
    ext.srcFile = file('mountains.xml')
    ext.destDir = new File(buildDir, 'generated')
    inputs.file srcFile
    outputs.dir destDir
    doLast {
        println "Transforming source file."
        destDir.mkdirs()
        def mountains = new XmlParser().parse(srcFile)
        mountains.mountain.each { mountain ->
            def name = mountain.name[0].text()
            def height = mountain.height[0].text()
            def destFile = new File(destDir, "${name}.txt")
            destFile.text = "$name -> ${height}\n"
        }
    }
}
```

gradle transform 的输出结果

```
> gradle transform
:transform
Transforming source file.
```

gradle transform 的输出结果

```
> gradle transform
:transform UP-TO-DATE
```

现在，Gradle 知道哪些文件要检查以确定任务是否为最新状态。

任务的 inputs 属性是 TaskInputs 类型。任务的 outputs 属性是 TaskOutputs 类型。

一个没有定义输出的任务将永远不会被当作是最新的。对于任务的输出并不是文件的场景，或者是更复杂的场景， TaskOutputs.upToDateWhen() 方法允许您以编程方式计算任务的输出是否应该被判断为最新状态。

一个只定义了输出的任务，如果自上一次构建以来它的输出没有改变，那么它会被判定为最新状态。

#

它是怎么实现的?

在第一次执行任务之前，Gradle 对输入进行一次快照。这个快照包含了输入文件集和每个文件的内容的哈希值。然后 Gradle 执行该任务。如果任务成功完成，Gradle 将对输出进行一次快照。该快照包含输出文件集和每个文件的内容的哈希值。Gradle 会保存这两个快照，直到任务的下一次执行。

之后每一次，在执行任务之前，Gradle 会对输入和输出进行一次新的快照。如果新的快照和前一次的快照一样，Gradle 会假定这些输出是最新状态的并跳过该任务。如果它们不一则， Gradle 则会执行该任务。Gradle 会保存这两个快照，直到任务的下一次执行。

请注意，如果一个任务有一个指定的输出目录，在它上一次执行之后添加到该目录的所有文件都将被忽略，并且不会使这个任务成为过时状态。这是不相关的任务可以在不互相干扰的情况下共用一个输出目录。如果你因为一些理由而不想这样，请考虑使用 TaskOutputs.upToDateWhen()

#

任务规则

有时你想要有这样一项任务，它的行为依赖于参数数值范围的一个大数或是无限的数字。任务规则是提供此类任务的一个很好的表达方式：

任务规则

build.gradle

```
tasks.addRule("Pattern: ping<ID>") { String taskName ->
    if (taskName.startsWith("ping")) {
        task(taskName) << {
            println "Pinging: " + (taskName - 'ping')
        }
    }
}
```

Gradle q pingServer1 的输出结果

```
> gradle -q pingServer1
Pinging: Server1
```

这个字符串参数被用作这条规则的描述。当对这个例子运行 gradle tasks 的时候，这个描述会被显示。

规则不只是从命令行调用任务才起作用。你也可以对基于规则的任务创建依赖关系：

基于规则的任务依赖

build.gradle

```
tasks.addRule("Pattern: ping<ID>") { String taskName ->
    if (taskName.startsWith("ping")) {
        task(taskName) << {
            println "Pinging: " + (taskName - 'ping')
        }
    }
}
task groupPing {
    dependsOn pingServer1, pingServer2
}
```

Gradle q groupPing 的输出结果

```
> gradle -q groupPing
Pinging: Server1
Pinging: Server2
```

#

析构器任务

析构器任务是一个孵化中的功能 。当最终的任务准备运行时，析构器任务会自动地添加到任务图中。

添加一个析构器任务

build.gradle

```
task taskX << {
    println 'taskX'
}
task taskY << {
    println 'taskY'
}
taskX.finalizedBy taskY
```

gradle -q taskX 的输出结果

```
> gradle -q taskX
taskX
taskY
```

即使最终的任务执行失败，析构器任务也会被执行。

执行失败的任务的任务析构器

build.gradle

```
task taskX << {
    println 'taskX'
    throw new RuntimeException()
}
task taskY << {
    println 'taskY'
}
taskX.finalizedBy taskY
```

gradle -q taskX 的输出结果

```
> gradle -q taskX
taskX
taskY
```

另一方面，如果最终的任务什么都不做的话，比如由于失败的任务依赖项或如果它被认为是最新的状态，析构任务不会执行。

在不管构建成功或是失败，都必须清理创建的资源的情况下，析构认为是很有用的。这样的资源的一个例子是，一个 web 容器会在集成测试任务前开始，并且在之后关闭，即使有些测试失败。

你可以使用 Task.finalizedBy()方法指定一个析构器任务。这个方法接受一个任务实例、任务名称或 <a4><c5>Task.dependsOn()</c5></a4> 所接受的任何其他输入作为参数。

#

总结

如果你是从 Ant 转过来的，像 Copy 这种增强的 Gradle 任务，看起来就像是一个 Ant 目标（target）和一个 Ant 任务（task）之间的混合物。实际上确实是这样子。Gradle 没有像 Ant 那样对任务和目标进行分离。简单的 Gradle 任务就像 Ant 的目标，而增强的 Gradle 任务还包括 Ant 任务方面的内容。Gradle 的所有任务共享一个公共 API，您可以创建它们之间的依赖性。这样的一个任务可能会比一个 Ant 任务更好配置。它充分利用了类型系统，更具有表现力而且易于维护。

15

使用文件

大多数构建工作都要使用到文件。Gradle 添加了一些概念和 API 来帮助您实现这一目标。

# #

定位文件

你可以使用 Project.file()方法来找到一个相对于项目目录的文件 。

查找文件

build.gradle

```
// Using a relative path
File configFile = file('src/config.xml')
// Using an absolute path
configFile = file(configFile.absolutePath)
// Using a File object with a relative path
configFile = file(new File('src/config.xml'))
```

您可以把任何对象传递给 file()方法，而它将尝试将其转换为一个绝对路径的 File 对象。通常情况下，你会传给它一个 String 或 File 的实例。而所提供的这个对象的 tostring() 方法的值会作为文件路径。如果这个路径是一个绝对路径，它会用于构构一个 File 实例。否则，会通过先计算所提供的路径相对于项目目录的相对路径来构造 File 实例。这个 file()方法也可以识别URL，例如是 `file:/some/path.xml` 。

这是把一些用户提供的值转换为一个相对路径的 File 对象的有用方法。由于 file()方法总是去计算所提供的路径相对于项目目录的路径，最好是使用 new File(somePath)，因为它是一个固定的路径，而不会因为用户运行 Gradle 的具体工作目录而改变。

#

文件集合

一个文件集合只是表示一组文件。它通过 FileCollection 接口来表示。Gradle API 中的许多对象都实现了此接口。比如，依赖配置 就实现了 FileCollection 这一接口。

使用 Project.files()方法是获取一个 FileCollection 实例的其中一个方法。你可以向这个方法传入任意个对象，而它们会被转换为一组 File 对象。这个 Files() 方法接受任何类型的对象作为其参数。根据16.1 章节 “定位文件”里对 file() 方法的描述，它的结果会被计算为相对于项目目录的相对路径。你也可以将集合，迭代变量，map 和数组传递给 files() 方法。它们会被展开，并且内容会转换为 File 实例。

创建一个文件集合

build.gradle

```
FileCollection collection = files('src/file1.txt', new File('src/file2.txt'), ['src/file3.txt', 'src/file4.txt'])
```

一个文件集合是可迭代的，并且可以使用 as 操作符转换为其他类型的对象集合。您还可以使用+运算符把两个文件集合相加，或使用-运算符减去一个文件集合。这里是一些使用文件集合的例子。

使用一个文件集合

build.gradle

```
// Iterate over the files in the collection
collection.each {File file ->
    println file.name
}
// Convert the collection to various types
Set set = collection.files
Set set2 = collection as Set
List list = collection as List
String path = collection.asPath
File file = collection.singleFile
File file2 = collection as File
// Add and subtract collections
def union = collection + files('src/file3.txt')
def different = collection - files('src/file3.txt')
```

你也可以向 files()方法传一个闭包或一个 Callable 实例。它会在查询集合内容，并且它的返回值被转换为一组文件实例时被调用。这个闭包或 Callable 实例的返回值可以是 files() 方法所支持的任何类型的对象。这是 “实现” FileCollection 接口的简单方法。

实现一个文件集合

build.gradle

```
task list << {
    File srcDir
    // Create a file collection using a closure
    collection = files { srcDir.listFiles() }
    srcDir = file('src')
    println "Contents of $srcDir.name"
    collection.collect { relativePath(it) }.sort().each { println it }
    srcDir = file('src2')
    println "Contents of $srcDir.name"
    collection.collect { relativePath(it) }.sort().each { println it }
}
```

gradle -q list 的输出结果

```
> gradle -q list
Contents of src
src/dir1
src/file1.txt
Contents of src2
src2/dir1
src2/dir2
```

你可以向 files() 传入一些其他类型的对象：

FileCollection

它们会被展开，并且内容会被包含在文件集合内。

Task

任务的输出文件会被包含在文件集合内。

TaskOutputs

TaskOutputs 的输出文件会被包含在文件集合内。

要注意的一个地方是，一个文件集合的内容是缓计算的，它只在需要的时候才计算。这意味着您可以，比如创建一个 FileCollection 对象而里面的文件会在以后才创建，比方说在一些任务中才创建。

#

文件树

文件树是按层次结构排序的文件集合。例如，文件树可能表示一个目录树或 ZIP 文件的内容。它通过 FileTree 接口表示。FileTree 接口继承自 FileCollection，所以你可以用对待文件集合一样的方式来对待文件树。Gradle 中的几个对象都实现了 FileTree 接口，例如 source sets。

使用 Project.fileTree()方法是获取一个 FileTree 实例的其中一种方法。它将定义一个基目录创建 FileTree 对象，并可以选择加上一些 Ant 风格的包含与排除模式。

创建一个文件树

build.gradle

```
// Create a file tree with a base directory
FileTree tree = fileTree(dir: 'src/main')
// Add include and exclude patterns to the tree
tree.include '**/*.java'
tree.exclude '**/Abstract*'
// Create a tree using path
tree = fileTree('src').include('**/*.java')
// Create a tree using closure
tree = fileTree('src') {
    include '**/*.java'
}
// Create a tree using a map
tree = fileTree(dir: 'src', include: '**/*.java')
tree = fileTree(dir: 'src', includes: ['**/*.java', '**/*.xml'])
tree = fileTree(dir: 'src', include: '**/*.java', exclude: '**/*test*/**')
```

你可以像使用一个文件集合的方式一样来使用一个文件树。你也可以使用 Ant 风格的模式来访问文件树的内容或选择一个子树：

使用文件树

build.gradle

```
// Iterate over the contents of a tree
tree.each {File file ->
    println file
}
```

```
// Filter a tree
FileTree filtered = tree.matching {
    include 'org/gradle/api/**'
}
// Add trees together
FileTree sum = tree + fileTree(dir: 'src/test')
// Visit the elements of the tree
tree.visit {element ->
    println "$element.relativePath => $element.file"
}
```

# #

使用归档文件的内容作为文件树

您可以使用档案的内容，如 ZIP 或者 TAR 文件，作为一个文件树。你可以通过使用 Project.zipTree() 或 Project.tarTree()方法来实现这一过程。这些方法返回一个 FileTree 实例，您可以像使用任何其他文件树或文件集合一样使用它。例如，您可以用它来通过复制内容扩大归档，或把一些档案合并到另一个归档文件中。

使用归档文件作为文件树

build.gradle

```
// Create a ZIP file tree using path
FileTree zip = zipTree('someFile.zip')
// Create a TAR file tree using path
FileTree tar = tarTree('someFile.tar')
//tar tree attempts to guess the compression based on the file extension
//however if you must specify the compression explicitly you can:
FileTree someTar = tarTree(resources.gzip('someTar.ext'))
```

#

指定一组输入文件

Gradle 中的许多对象都有一个接受一组输入文件的属性。例如，JavaCompile 任务有一个 source 属性，定义了要编译的源代码文件。你可以使用上面所示的 files()方法所支持的任意类型的对象设置此属性。这意味着您可以通过如 File、String、集合、FileCollection 对象，或甚至是一个闭包来设置此属性。这里有一些例子：

指定一组文件

build.gradle

```
// Use a File object to specify the source directory
compile {
    source = file('src/main/java')
}
// Use a String path to specify the source directory
compile {
    source = 'src/main/java'
}
// Use a collection to specify multiple source directories
compile {
    source = ['src/main/java', '../shared/java']
}
// Use a FileCollection (or FileTree in this case) to specify the source files
compile {
    source = fileTree(dir: 'src/main/java').matching { include 'org/gradle/api/**' }
}
// Using a closure to specify the source files.
compile {
    source = {
        // Use the contents of each zip file in the src dir
        file('src').listFiles().findAll {it.name.endsWith('.zip')}.collect { zipTree(it) }
    }
}
```

通常情况下，有一个与属性相同名称的方法，可以追加这个文件集合。再者，这个方法接受 files()方法所支持的任何类型的参数。

指定一组文件

build.gradle

```
compile {
    // Add some source directories use String paths
    source 'src/main/java', 'src/main/groovy'
    // Add a source directory using a File object
    source file('../shared/java')
    // Add some source directories using a closure
    source { file('src/test/').listFiles() }
}
```

#

复制文件

你可以使用 Copy 任务来复制文件。复制任务非常灵活，并允许您进行，比如筛选要复制的文件的内容，或映射文件的名称。

若要使用 Copy 任务，您必须提供用于复制的源文件和目标目录。您还可以在复制文件的时候指定如何转换文件。你可以使用一个复制规范来做这些。一个复制规范通过 CopySpec 接口来表示。Copy 任务实现了此接口。你可以使用 CopySpec.from()方法指定源文件，使用 CopySpec.into()方法使用目标目录。

使用 copy 任务复制文件

build.gradle

```
task copyTask(type: Copy) {
    from 'src/main/webapp'
    into 'build/explodedWar'
}
```

from() 方法接受和 files() 方法一样的任何参数。当参数解析为一个目录时，该目录下的所有文件（不包含目录本身）都会递归复制到目标目录。当参数解析为一个文件时，该文件会复制到目标目录中。当参数解析为一个不存在的文件时，参数会被忽略。如果参数是一个任务，那么任务的输出文件（即该任务创建的文件）会被复制，并且该任务会自动添加为 Copy 任务的依赖项。into() 方法接受和 files() 方法一样的任何参数。这里是另一个示例：

示例 16.11. 指定复制任务的源文件和目标目录

build.gradle

```
task anotherCopyTask(type: Copy) {
    // Copy everything under src/main/webapp
    from 'src/main/webapp'
    // Copy a single file
    from 'src/staging/index.html'
    // Copy the output of a task
    from copyTask
    // Copy the output of a task using Task outputs explicitly.
    from copyTaskWithPatterns.outputs
    // Copy the contents of a Zip file
    from zipTree('src/main/assets.zip')
    // Determine the destination directory later
```

```
    into { getDestDir() }
}
```

您可以使用 Ant 风格的包含或排除模式，或使用一个闭包，来选择要复制的文件：

选择要复制的文件

build.gradle

```
task copyTaskWithPatterns(type: Copy) {
    from 'src/main/webapp'
    into 'build/explodedWar'
    include '**/*.html'
    include '**/*.jsp'
    exclude { details -> details.file.name.endsWith('.html') && details.file.text.contains('staging') }
}
```

此外，你也可以使用 Project.copy()方法来复制文件。它是与任务一样的工作方式，尽管它有一些主要的限制。首先， copy()不能进行增量操作。

使用没有最新状态检查的 copy() 方法复制文件

build.gradle

```
task copyMethod << {
    copy {
        from 'src/main/webapp'
        into 'build/explodedWar'
        include '**/*.html'
        include '**/*.jsp'
    }
}
```

第二，当一个任务用作复制源（即作为 from() 的参数）的时候，copy()方法不能建立任务依赖性，因为它是一个方法，而不是一个任务。因此，如果您在任务的 action 里面使用 copy()方法，必须显式声明所有的输入和输出以得到正确的行为。

使用有最新状态检查的 copy() 方法复制文件

build.gradle

```
task copyMethodWithExplicitDependencies{
    inputs.file copyTask // up-to-date check for inputs, plus add copyTask as dependency
    outputs.dir 'some-dir' // up-to-date check for outputs
    doLast{
        copy {
```

```
        // Copy the output of copyTask
        from copyTask
        into 'some-dir'
    }
  }
}
```

在可能的情况下，最好是使用 Copy 任务，因为它支持增量构建和任务依赖关系推理，而不需要你额外付出。copy()方法可以作为一个任务执行的部分来复制文件。即，这个 copy() 方法旨在用于自定义任务中，需要文件复制作为其一部分功能的时候。在这种情况下，自定义任务应充分声明与复制操作有关的输入/输出。

#

重命名文件

重命名复制的文件

build.gradle

```
task rename(type: Copy) {
  from 'src/main/webapp'
  into 'build/explodedWar'
  // Use a closure to map the file name
  rename { String fileName ->
    fileName.replace('-staging-', '')
  }
  // Use a regular expression to map the file name
  rename '(.+)-staging-(.+)', '$1$2'
  rename(/(.+)-staging-(.+)/, '$1$2')
}
```

#

过滤文件

过滤要复制的文件

build.gradle

```
import org.apache.tools.ant.filters.FixCrLfFilter
import org.apache.tools.ant.filters.ReplaceTokens
task filter(type: Copy) {
  from 'src/main/webapp'
```

```
    into 'build/explodedWar'
    // Substitute property references in files
    expand(copyright: '2009', version: '2.3.1')
    expand(project.properties)
    // Use some of the filters provided by Ant
    filter(FixCrLfFilter)
    filter(ReplaceTokens, tokens: [copyright: '2009', version: '2.3.1'])
    // Use a closure to filter each line
    filter { String line ->
        "[$line]"
    }
}
```

#

使用 CopySpec 类

复制规范用来组织一个层次结构。一个复制规范继承其目标路径，包含模式，排除模式，复制操作，名称映射和过滤器。

嵌套的复制规范

build.gradle

```
task nestedSpecs(type: Copy) {
    into 'build/explodedWar'
    exclude '**/*staging*'
    from('src/dist') {
        include '**/*.html'
    }
    into('libs') {
        from configurations.runtime
    }
}
```

#

使用 Sync 任务

Sync 任务继承了 Copy 任务。当它执行时，它会将源文件复制到目标目录中，然后从目标目录移除所有不是它复制的文件。这可以用来做一些事情，比如安装你的应用程序、 创建你的归档文件的 exploded 副本，或维护项目的依赖项的副本。

这里是一个例子，维护在 build/libs 目录中的项目运行时依赖的副本。

使用同步任务复制依赖项

build.gradle

```
task libs(type: Sync) {
    from configurations.runtime
    into "$buildDir/libs"
}
```

# #

创建归档文件

一个项目可以有你所想要的一样多的 JAR 文件。您也可以将 WAR、 ZIP 和 TAG 文件添加到您的项目。使用各种归档任务可以创建以下的归档文件： Zip, Tar, Jar, War, and Ear. 他们的工作方式都一样，所以让我们看看如何创建一个 ZIP 文件。

创建一个 ZIP 文件

build.gradle

```
apply plugin: 'java'
task zip(type: Zip) {
   from 'src/dist'
   into('libs') {
      from configurations.runtime
   }
}
```

为什么要用 Java 插件?

Java 插件对归档任务添加了一些默认值。如果你愿意，使用归档任务时可以不需要 Java 插件。您需要提供一些值给附加的属性。

归档任务与 Copy 任务的工作方式一样，并且实现了相同的 CopySpec 接口。像使用 Copy 任务一样，你需要使用 from() 的方法指定输入的文件，并可以选择是否通过 into() 方法指定最终在存档中的位置。您可以通过一个复制规范来筛选文件的内容、 重命名文件和进行其他你可以做的事情。

#

归档命名

生成的归档的默认名称是 projectName-version.type。举个例子：

创建 ZIP 文件

build.gradle

```
apply plugin: 'java'
version = 1.0
```

```
task myZip(type: Zip) {
    from 'somedir'
}
println myZip.archiveName
println relativePath(myZip.destinationDir)
println relativePath(myZip.archivePath)
```

gradle -q myZip 的输出结果

```
> gradle -q myZip
zipProject-1.0.zip
build/distributions
build/distributions/zipProject-1.0.zip
```

它添加了一个名称为 myZip 的ZIP归档任务，产生 ZIP 文件 zipProject 1.0.zip。区分归档任务的名称和归档任务生成的归档文件的名称是很重要的。归档的默认名称可以通过项目属性 archivesBaseName 来更改。还可以在以后的任何时候更改归档文件的名称。

这里有很多你可以在归档任务中设置的属性。它们在以下的表 16.1，"存档任务-命名属性"中列出。你可以，比方说，更改归档文件的名称：

配置归档任务-自定义归档名称

build.gradle

```
apply plugin: 'java'
version = 1.0
task myZip(type: Zip) {
    from 'somedir'
    baseName = 'customName'
}
println myZip.archiveName
```

gradle -q myZip 的输出结果

```
gradle -q myZip customName-1.0.zip
```

您可以进一步自定义存档名称：

配置归档任务 - appendix & classifier

build.gradle

```

apply plugin: 'java'
archivesBaseName = 'gradle'
```

```
version = 1.0
task myZip(type: Zip) {
    appendix = 'wrapper'
    classifier = 'src'
    from 'somedir'
}
println myZip.archiveName
```

gradle -q myZip 的输出结果

```
> gradle -q myZip
gradle-wrapper-1.0-src.zip
```

表 16.1. 归档任务-命名属性

属性名称	类型	默认值	描述
archiveName	String	extension 如果这些属性中的任何一个为空，那后面的 - 不会被添加到该名称中。	生成的归档文件的基本文件名
archivePath	File	archiveName	生成的归档文件的绝对路径。
destinationDir	File	依赖于归档类型。JAR包和 WAR包会生成到 project.buildDir /libraries中。ZIP文件和 TAR文件会生成到 project.buildDir /distributions中。	存放生成的归档文件的目录
baseName	String	project.name	归档文件的名称中的基本名称部分。
appendix	String	null	归档文件的名称中的附录部分。
version	String	project.version	归档文件的名称中的版本部分。
classifier	String	null	归档文件的名称中的分类部分。
extension	String	依赖于归档的类型，用于TAR文件，可以是以下压缩类型： tbz2 .	归档文件的名称中的扩展名称部分。

#

共享多个归档之间的内容

你可以使用 Project.copySpec()方法在归档之间共享内容。

你经常会想要发布一个归档文件，这样就可从另一个项目中使用它。

16

从 Gradle 中调用 Ant

Gradle 提供了对 Ant 的优秀集成您可以在你的 Gradle 构建中，使用单独的 Ant 任务或整个 Ant 构建。事实上，你会发现在 Gradle 中使用 Ant 任务比使用 Ant 的 XML 格式更容易也更强大。你甚至可以只把 Gradle 当作一个强大的 Ant 任务脚本的工具。

Ant 可以分为两层。第一层是 Ant 的语言。它提供了用于 build.xml，处理的目标，特殊的构造方法比如宏，还有其他等等的语法。换句话说，除了 Ant 任务和类型之外全部都有。Gradle 理解这种语言，并允许您直接导入你的 Ant build.xml 到 Gradle 项目中。然后你可以使用你的 Ant 构建中的 target，就好像它们是 Gradle 任务一样。

Ant 的第二层是其丰富的 Ant 任务和类型，如 javac、copy 或 jar。这一层 Gradle 单靠 Groovy 和不可思议的 AntBuilder，对其提供了集成。

最后，由于构建脚本是 Groovy 脚本，所以您始终可以作为一个外部进程来执行 Ant 构建。你的构建脚本可能包含有类似这样的语句："ant clean compile".execute()。[8]

你可以把 Gradle 的 Ant 集成当成一个路径，将你的构建从 Ant 迁移至 Gradle 。例如，你可以通过导入您现有的 Ant 构建来开始。然后，可以将您的依赖声明从 Ant 脚本移到您的构建文件。最后，您可以将整个任务移动到您的构建文件，或者把它们替换为一些 Gradle 插件。这个过程可以随着时间一点点完成，并且在这整个过程当中你的 Gradle 构建都可以使用用。

#

在构建中使用 Ant 任务和类型

在构建脚本中，Gradle 提供了一个名为 ant 的属性。它指向一个 AntBuilder 实例。AntBuilder 用于从你的构建脚本中访问 Ant 任务、 类型和属性。从 Ant 的 build.xml 格式到 Groovy 之间有一个非常简单的映射，下面解释。

通过调用 AntBuilder实例上的一个方法，可以执行一个 Ant 任务。你可以把任务名称当作方法名称使用。例如，你可以通过调用ant.echo()方法执行 Ant 的 echo 任务。Ant 任务的属性会作为 Map 参数传给该方法。下面是执行 echo 任务的例子。请注意我们还可以混合使用 Groovy 代码和 Ant 任务标记。这将会非常强大。

使用 Ant 任务

build.gradle

```
task hello << {
    String greeting = 'hello from Ant'
    ant.echo(message: greeting)
}
```

gradle hello 的输出结果

```
> gradle hello
:hello
[ant:echo] hello from Ant
BUILD SUCCESSFUL
Total time: 1 secs
```

你可以把一个嵌套文本，通过作为任务方法调用的参数，把它传给一个 Ant 任务。在此示例中，我们将把作为嵌套文本的消息传给 echo 任务：

向 Ant 任务传入嵌套文本

build.gradle

```
task hello << {
    ant.echo('hello from Ant')
}
```

gradle hello 的输出结果

```
> gradle hello
:hello
[ant:echo] hello from Ant
BUILD SUCCESSFUL
Total time: 1 secs
```

你可以在一个闭包里把嵌套的元素传给一个 Ant 任务。嵌套元素的定义方式与任务相同，通过调用与我们要定义的元素一样的名字的方法。

向 Ant 任务传入嵌套元素

build.gradle

```
task zip << {
    ant.zip(destfile: 'archive.zip') {
        fileset(dir: 'src') {
            include(name: '**.xml')
            exclude(name: '**.java')
        }
    }
}
```

您可以用访问任务同样的方法，把类型名字作为方法名称，访问 Ant 类型。方法调用返回 Ant 数据类型，然后可以在构建脚本中直接使用。在以下示例中，我们创建一个 Ant 的 path 对象，然后循环访问它的内容。

使用 Ant 类型

build.gradle

```
task list << {
    def path = ant.path {
        fileset(dir: 'libs', includes: '*.jar')
    }
    path.list().each {
        println it
    }
}
```

#

在您的构建中使用自定义 Ant 任务

要使自定义任务在您的构建中可用，你可以使用 Ant 任务 taskdef（通常更容易）或typedef，就像在 build.xml 文件中一样。然后，您可以像引用内置的 Ant 任务一样引用自定义 Ant 任务。

使用自定义 Ant 任务

build.gradle

```
task check << {
   ant.taskdef(resource: 'checkstyletask.properties') {
      classpath {
         fileset(dir: 'libs', includes: '*.jar')
      }
   }
   ant.checkstyle(config: 'checkstyle.xml') {
      fileset(dir: 'src')
   }
}
```

你可以使用 Gradle 的依赖管理组合类路径，以用于自定义任务。要做到这一点，你需要定义一个自定义配置的类路径中，然后将一些依赖项添加到配置中。

声明用于自定义 Ant 任务的类路径

build.gradle

```
configurations {
   pmd
}
dependencies {
   pmd group: 'pmd', name: 'pmd', version: '4.2.5'
}
```

若要使用类路径配置，请使用自定义配置里的 asPath 属性。

同时使用自定义 Ant 任务和依赖管理

build.gradle

```
task check << {
   ant.taskdef(name: 'pmd', classname: 'net.sourceforge.pmd.ant.PMDTask', classpath: configurations.pmd.asPath)
   ant.pmd(shortFilenames: 'true', failonruleviolation: 'true', rulesetfiles: file('pmd-rules.xml').toURI().toString()) {
      formatter(type: 'text', toConsole: 'true')
      fileset(dir: 'src')
   }
}
```

#

导入 Ant 构建

你可以使用 ant.importBuild()方法来向 Gradle 项目导入一个 Ant 构建。当您导入一个 Ant 构建时，每个 Ant 目标被视为一个 Gradle 任务。这意味着你可以用与 Gradle 任务完全相机的方式操纵和执行 Ant 目标。

导入 Ant 构建

build.gradle

```
ant.importBuild 'build.xml'
```

build.xml

```
<project>
    <target name="hello">
        <echo>Hello, from Ant</echo>
    </target>
</project>
```

gradle hello 的输出结果

```
> gradle hello
:hello
[ant:echo] Hello, from Ant
BUILD SUCCESSFUL
Total time: 1 secs
```

您可以添加一个依赖于 Ant 目标的任务：

依赖于 Ant 目标的任务

build.gradle

```
ant.importBuild 'build.xml'
task intro(dependsOn: hello) << {
    println 'Hello, from Gradle'
}
```

gradle intro 的输出结果

```
> gradle intro
:hello
[ant:echo] Hello, from Ant
:intro
```

```
Hello, from Gradle
BUILD SUCCESSFUL
Total time: 1 secs
```

或者，您可以将行为添加到 Ant 目标中：

将行为添加到 Ant 目标

build.gradle

```
ant.importBuild 'build.xml'
hello << {
    println 'Hello, from Gradle'
}
```

gradle hello 的输出结果

```
> gradle hello
:hello
[ant:echo] Hello, from Ant
Hello, from Gradle
BUILD SUCCESSFUL
Total time: 1 secs
```

它也可以用于一个依赖于 Gradle 任务的 Ant 目标：

依赖于 Gradle 任务的 Ant 目标

build.gradle

```
ant.importBuild 'build.xml'
task intro << {
    println 'Hello, from Gradle'
}
build.xml
<project>
    <target name="hello" depends="intro">
        <echo>Hello, from Ant</echo>
    </target>
</project>
```

gradle hello 的输出结果

```
> gradle hello
:intro
Hello, from Gradle
:hello
```

```
[ant:echo] Hello, from Ant
BUILD SUCCESSFUL
Total time: 1 secs
```

#

Ant 属性和引用

有几种方法来设置 Ant 属性，以便使该属性被 Ant 任务使用。你可以直接在 AntBuilder 实例上设置属性。Ant 属性也可以从一个你可以修改的 Map 中获得。您还可以使用 Ant property 任务。下面是一些如何做到这一点的例子。

Ant 属性设置

build.gradle

```
ant.buildDir = buildDir
ant.properties.buildDir = buildDir
ant.properties['buildDir'] = buildDir
ant.property(name: 'buildDir', location: buildDir)
build.xml
<echo>buildDir = ${buildDir}</echo>
```

许多 Ant 任务在执行时会设置一些属性。有几种方法来获取这些属性值。你可以直接从AntBuilder 实例获得属性。Ant 属性也可作为一个 Map。下面是一些例子。

获取 Ant 属性

build.xml

```
<property name="antProp" value="a property defined in an Ant build"/>
build.gradle
println ant.antProp
println ant.properties.antProp
println ant.properties['antProp']
```

有几种方法可以设置 Ant 引用：

Ant 引用设置

build.gradle

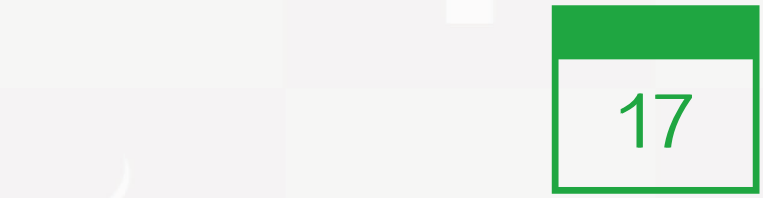
```
ant.path(id: 'classpath', location: 'libs')
ant.references.classpath = ant.path(location: 'libs')
ant.references['classpath'] = ant.path(location: 'libs')
build.xml
<path refid="classpath"/>
```

有几种方法可以获取 Ant 引用：

获取 Ant 引用

build.xml

```
<path id="antPath" location="libs"/>
build.gradle
println ant.references.antPath
println ant.references['antPath']
```



17

日志

日志是构建工具的主要界面。如果日志太多，真正的警告和问题容易被隐藏。另一方面，如果出了错，你需要找出相关的信息。Gradle 定义了6个日志级别，如表 18.1，“日志级别”所示。除了那些您通过可能会看到的日志级别之外，有两个 Gradle 特定日志级别。这两个级别分别是 QUIET 和 LIFECYCLE. 默认使用后面的这个日志级别，用于报告构建进度。

表 18.1. 日志级别

Level	用于
ERROR	错误消息
QUIET	重要的信息消息
WARNING	警告消息
LIFECYCLE	进度信息消息
INFO	信息性消息
DEBUG	调试消息

#

选择一个日志级别

您可以使用表 18.2，“日志级别的命令行选项”中所示的命令行开关来选择不同的日志级别。在表 18.3，“栈跟踪的命令行选项”中，你可以看到影响栈跟踪日志的命令行开关。

表 18.2. 日志级别的命令行选项

选项	输出日志级别
没有日志选项	LIFECYCLE 及更高
`--quiet`	QUIET 及更高
`--info`	INFO 及更高
`--debug`	DEBUG 及更高

表 18.3. 栈跟踪的命令行选项

选项	意义
没有栈跟踪选项	构建错误（如编译错误）时没有栈跟踪打印到控制台。只有在内部异常的情况下才打印栈跟踪。如果选择 `DEBUG` 日志级别，则总是输出截取后的栈跟踪信息。
`--stacktrace`	输出截断的栈跟踪。我们推荐使用这一个选项而不是打印全栈的跟踪信息。Groovy 的全栈跟踪非常冗长（由于其潜在的动态调用机制，然而他们通常不包含你的的代码中哪里错了的相关信息。）
`--full-stacktrace`	打印全栈的跟踪信息。

#

编写自己的日志消息

在构建文件，打印日志的一个简单方法是把消息写到标准输出中。Gradle 会把写到标准输出的所有内容重定向到它的日志系统的 QUIET 级别中。

使用标准输出写日志

build.gradle

```
println 'A message which is logged at QUIET level'
```

Gradle 还提供了一个 logger 属性给构建脚本，它是一个 Logger 实例。该接口扩展自 SLF4J的 Logger 接口，并添加了几个 Gradle 的特有方法。下面是关于如何在构建脚本中使用它的示例：

编写自己的日志消息

build.gradle

```
logger.quiet('An info log message which is always logged.')
logger.error('An error log message.')
logger.warn('A warning log message.')
logger.lifecycle('A lifecycle info log message.')
logger.info('An info log message.')
logger.debug('A debug log message.')
logger.trace('A trace log message.')
```

您也可以在构建脚本中通过其他使用的类挂钩到 Gradle 的日志系统中（例如 buildSrc 目录中的类）。只需使用一个 SLF4J 的 logger 对象。你可以在构建脚本中，用与内置的 logger 同样的方式使用这个 logger。

使用 SLF4J 编写日志消息

build.gradle

```
import org.slf4j.Logger
import org.slf4j.LoggerFactory
Logger slf4jLogger = LoggerFactory.getLogger('some-logger')
slf4jLogger.info('An info log message logged using SLF4j')
```

#

使用外部工具和库记录日志

Gradle 内部使用 Ant 和 Ivy。它们都有自己的日志系统。Gradle 将其日志输出重定向到 Gradle 的日志系统。从 Ant/Ivy 的日志级别到 Gradle 的日志级别是一对一的映射，除了 Ant/Ivy 的 TRACE 级别之外，它是映射到 Gradle 的 DEBUG 级别的。这意味着默认情况下， Gradle 日志级别将不会显示任何 Ant/Ivy 的输出，除非是错误或警告信息。

有很多的工具仍然在使用标准输出日志记录。默认情况下，Gradle 将标准输出重定向到QUIET日志级别，把标准错误输出重写向到 ERROR 级别。这种行为是可配置的。Project 对象提供了一个LoggingManager，它允许您在计算构建脚本时，修改标准输出和错误重定向的日志级别。

配置标准输出捕获

build.gradle

```
logging.captureStandardOutput LogLevel.INFO
println 'A message which is logged at INFO level'
```

为能在任务执行过程中更改标准输出或错误的日志级别，task 也提供了一个 LoggingManager。

对任务配置标准输出捕获

build.gradle

```
task logInfo {
    logging.captureStandardOutput LogLevel.INFO
    doFirst {
        println 'A task message which is logged at INFO level'
    }
}
```

Gradle 还提供了对 Java Util Logging，Jakarta Commons Logging 和 Log4j 的日志工具的集成。你生成的类使用这些日志记录工具输出的任何日志消息，都将被重定向到 Gradle 的日志系统。

#

改变 Gradle 日志

您可以用您自己的 logging UI 大量地替换 Gradle 的。你可以这样做，例如，如果您想要以某种方式自定义 UI——以输出更多或更少的信息，或修改日志格式您可以使用 Gradle.useLogger() 方法替换这个 logging。它可以在构建脚本，或 init 脚本，或通过内嵌的 API 访问。请注意它完全禁用 Gradle 的默认输出。下面是一个示例，在 init 脚本中修改任务执行和构建完成的日志打印。

自定义 Gradle 日志

init.gradle

```
useLogger(new CustomEventLogger())
class CustomEventLogger extends BuildAdapter implements TaskExecutionListener {
    public void beforeExecute(Task task) {
        println "[$task.name]"
    }
    public void afterExecute(Task task, TaskState state) {
        println()
    }
    public void buildFinished(BuildResult result) {
        println 'build completed'
        if (result.failure != null) {
            result.failure.printStackTrace()
        }
    }
}
```

gradle -I init.gradle build 的输出结果

```
> gradle -I init.gradle build
[compile]
compiling source
[testCompile]
compiling test source
[test]
running unit tests
[build]
build completed
```

你的 logger 可以实现下面列出的任何监听器接口。当你注册一个 logger 时，只能替换它实现的接口的日志记录。其他接口的日志记录是不变的。

- BuildListener
- ProjectEvaluationListener
- TaskExecutionGraphListener
- TaskExecutionListener
- TaskActionListener

18

Gradle 守护进程

# #

走进守护进程

Gradle 守护进程（有时也称为构建守护进程）的目的是改善 Gradle 的启动和执行时间。

我们准备了几个守护进程非常有用的用例。对于一些工作流，用户会多次调用 Gradle，以执行少量的相对快速的任务。举个例子：

- 当使用测试驱动开发时，单元测试会被执行多次。
- 当开发一个 web 应用程序中，应用程序会被组装多次。
- 当发现构建能做什么，在 gradle tasks 在哪里会执行多次。

对以上各种工作流来说，让调用 Gradle 的启动成本尽可能小会很重要。

此外，如果可以相对较快地建立 Gradle 模型，用户界面可以提供一些有趣的功能。例如，该守护进程可能用于以下情形：

- 在 IDE 中的内容帮助
- 在 GUI 中的实时可视化构建
- 在 CLI 中的 tab 键完成

一般情况下，构建工具的敏捷行为总是可以派上用场。如果你尝试在你的本地构建中使用守护进程的话，它会变得让你很难回到正常的 Gradle 使用。

Tooling API 在整个过程当中都使用守护进程。如，你无法在没有守护进程时正式地使用 Tooling API。这意味着当您在 Eclipse 中使用 STS Gradle 或在 Intellij IDEA 中使用 Gradle 支持时，您已经在使用 Gradle 守护进程。

未来，该守护进程还会提供更多的功能：

- 敏捷的 up-to-date 检查：使用本地文件系统修改通知（例如，通过 jdk7 nio.2）预先执行 up-to-date 分析。
- 更快的构建： 预评估项目，这样当用户接下来调用 Gradle 时，模型就准备好了。
- 我们提到了更快的构建吗？守护进程可以预先下载依赖项或进行快照依赖的新版本检查。
- 使用可用于编译和测试的一个可复用线程池。例如，Groovy 和 Scala 的编译器启动开销都很大。构建守护进程可以维持一个已下载的 Groovy 和（或）Scala 进程。

- 预先执行某些任务，比如编译。更快的反馈。
- 快速、 准确的 bash 的 tab 键完成。
- Gradle 缓存的定期垃圾收集。

#

重用和失效的守护程序

基本的思想是， gradle 命令会 fork 一个守护进程，用于执行实际的构建。Gradle 命令的后续调用将重用该守护进程，以避免启动开销。有时我们不能使用现有的守护进程，是因为它正忙或其 java 版本或 jvm 参数不同。关于 fork 一个完全新的守护进程的具体细节，请阅读下面的专题。守护进程将在空闲3小时后自动失效。

以下是我们 fork 一个新的守护进程的所有情况：

- 如果该守护进程当前正忙于运行一些作业，将启动一个全新的守护进程。
- 对每个 java home，我们会 fork 一个单独的守护进程。所以即使有一些闲置的守护进程等待构建请求，但你碰巧通过不同的 java HOME 运行构建，那么一个全新的守护进程将会被 fork。
- 如果用于构建的jvm的参数足够不同，我们会 fork 一个单独的守护进程。例如，如果某些系统属性已经更改，我们不会 fork 一个新的守护进程。然而，如果 -Xmx 内存设置更改了，或一些基本的不变的系统属性更改了（ 例如 file.encoding ），那么将 fork 新的守护进程。
- 在这一刻，守护进程会被加上 Gradle 的特定版本号。这意味着即使一些守护进程处于空闲状态，但您正在运行的构建与 Gradle 不同版本，也将启动一个新的守护进程。这也有一种 --stop 命令行指令的结果： 当运行 --stop 时，您仅可以停止以你的 Gradle 版本启动的守护进程。

我们计划在将来改进守护进程的 managing / pooling 的方法。

# #

用法和故障排除

关于命令行的用法，可以看一下专题附录 D， Gradle 命令行。如果你已经厌倦反复使用相同的命令行选项，可以看看构建环境。这一章节包含了有关如何以一种“持久化”的方式配置某些行为（包括在默认情况下打开守护进程）的信息。

以下是有关 Gradle 守护进程的故障排除的一些方面：

- 如果你的构建有问题，请尝试暂时禁用守护进程（您可以通过使用命令行开关--no-daemon)。
- 有时候，您可能想要通过--stop命令行选项或更有力的方式停止守护程序。
- 默认情况下位于 Gradle 用户主目录有一个守护进程的日志文件。
- 你可能想要以--foreground 模式启动守护程序，以观察构建是怎么执行的。

# #

配置守护进程

可以配置一些守护进程的设置，例如 JVM 参数、内存设置或 Java home 目录。有关更多信息请参阅构建环境。

构建环境

#

通过 gradle.properties 配置构建环境

Gradle 提供了几个选项，可以很容易地配置将用于执行您的构建的 Java 进程。当可以通过 GRADLE_OPTS 或 JAVA_OPTS 在你的本地环境中配置这些选项时，如果某些设置如 JVM 内存设置，Java home，守护进程的开/关，它们可以和你的项目在你的版本控制系统中被版本化的话，将会更有用，这样整个团队就可以使用一致的环境了。在你的构建当中，建立一致的环境，就和把这些配置放进 gradle.properties 文件一样简单。这些配置将会按以下顺序被应用（以防在多个地方都有配置时只有最后一个 生效）：

- 位于项目构建目录的gradle.properties。
- 位于gradle 用户主目录的gradle.properties。
- 系统属性，例如当在命令行中使用 -Dsome.property 时。

下面的属性可以用于配置 Gradle 构建环境：

org.gradle.daemon
当设置为 true 时，Gradle 守护进程会运行构建。对于本地开发者的构建而言，这是我们最喜欢的属性。开发人员的环境在速度和反馈上会优化，所以我们几乎总是使用守护进程运行 Gradle 作业。由于 CI 环境在一致性和可靠性上的优化，我们不通过守护进程运行 CI 构建（即长时间运行进程）。

org.gradle.java.home 为 Gradle 构建进程指定 java home 目录。这个值可以设置为 jdk 或 jre 的位置，不过，根据你的构建所做的，选择 jdk 会更安全。如果该设置未指定，将使用合理的默认值。

org.gradle.jvmargs 指定用于该守护进程的 jvmargs。该设置对调整内存设置特别有用。目前的内存上的默认设置很大方。

org.gradle.configureondemand
启用新的孵化模式，可以在配置项目时使得 Gradle 具有选择性。只适用于相关的项目被配置为在大型多项目中更快地构建。

org.gradle.parallel
如果配置了这一个，Gradle 将在孵化的并行模式下运行。

#

Forked java 进程

许多设置（如 java 版本和最大堆大小）可以在启动一个新的 JVM 构建进程时指定。这意味着 Gradle 在分析了各种 gradle.properties 文件之后，必须启动一个单独的 JVM 进程，以执行构建操作。当通过守护进程运行时，带有正确参数的 JVM 会启动一次，并在每次的守护进程构建执行时复用。当不通过守护进程执行 Gradle 时，在每次构建执行中都必须启动一个新的 JVM ，除非 JVM 是由 Gradle 启动脚本启动的，并且恰好具有相同的参数。

在执行每个构建时运行一个额外的 JVM 的代价是非常昂贵的，这就是为什么我们强烈推荐您使用 Gradle 守护进程，如果你指定了 org.gradle.java.home 或 org.gradle.jvmargs。更多详细信息，请参阅Gradle 守护进程。

#

通过代理访问网站

配置 HTTP 代理服务器（例如用于下载依赖）是通过标准的 JVM 系统属性来做的。这些属性可以直接在构建脚本中设置；例如设置代理主机为 System.setProperty（'http.proxyHost', 'www.somehost.org'）。或者，可以在构建的根目录或 Gradle 主目录中的 gradle.properties 文件中指定这些属性。

配置 HTTP 代理服务器

gradle.properties

```
systemProp.http.proxyHost=www.somehost.org
systemProp.http.proxyPort=8080
systemProp.http.proxyUser=userid
systemProp.http.proxyPassword=password
systemProp.http.nonProxyHosts=*.nonproxyrepos.com|localhost
```

对于 HTTPS 有单独的设置。

配置 HTTPS 代理服务器

gradle.properties

```
systemProp.https.proxyHost=www.somehost.org
systemProp.https.proxyPort=8080
systemProp.https.proxyUser=userid
systemProp.https.proxyPassword=password
systemProp.https.nonProxyHosts=*.nonproxyrepos.com|localhost
```

我们无法很好地概述所有可能的代理服务器设置。其中可以去看的一个地方是 Ant 项目的一个文件中的常量。这里是SVN的视图的链接。另一个地方是 JDK 文档的网络属性页。如果有人知道更好的概述，请发邮件让我们知道。

#

NTLM 身份验证

如果您的代理服务器需要 NTLM 身份验证，您可能需要提供验证域，以及用户名和密码。有两种方法可以向 NTLM 代理提供验证域：

- 将 http.proxyUser 系统属性设置为一个这样的值：域/用户名。
- 通过 http.auth.ntlm.domain 系统属性提供验证域。

20

Gradle 插件

Gradle 在它的核心中有意地提供了一些小但有用的功能，用于在真实世界中的自动化。所有有用的功能，例如以能够编译 Java 代码为例，都是通过插件进行添加的。插件添加了新任务（例如JavaCompile），域对象（例如SourceSet），约定（例如主要的 Java 源代码是位于 src/main/java），以及扩展的核心对象和其他插件的对象。

在这一章中，我们将讨论如何使用插件以及术语和插件相关的概念。

#

应用插件

插件都认为是被应用，通过 Project.apply() 方法来完成。

应用插件

build.gradle

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

插件都有表示它们自己的一个短名称。. 在上述例子中，我们使用短名称 java 去应用 JavaPlugin。

我们还可以使用下面的语法：

通过类型应用插件

build.gradle

```
apply plugin: org.gradle.api.plugins.JavaPlugin
```

由于 Gradle 的默认导入，您还可以这样写：

通过类型应用插件

build.gradle

```
apply plugin: JavaPlugin
```

插件的应用是幂等的。也就是说，一个插件可以被应用多次。如果以前已应用了该插件，任何进一步的应用都不会再有任何效果。

一个插件是任何实现了 Plugin 接口的简单的类。Gradle 提供了核心插件作为其发行包的一部分，所以简单地应用如上插件是你所需要做的。然而，对于第三方插件，你需要进行配置以使插件在构建类路径中可用。有关如何进行此操作的详细信息。

#

插件都做了什么

把插件应用到项目中可以让插件来扩展项目的功能。它可以做的事情如：

- 将任务添加到项目（如编译、测试）
- 使用有用的默认设置对已添加的任务进行预配置。
- 向项目中添加依赖配置（见“依赖配置”）。
- 通过扩展对现有类型添加新的属性和方法。

让我们来看看：

通过插件添加任务

build.gradle

```
apply plugin: 'java'
task show << {
    println relativePath(compileJava.destinationDir)
    println relativePath(processResources.destinationDir)
}
```

gradle -q show 的输出结果

```
> gradle -q show
build/classes/main
build/resources/main
```

Java 插件已经向项目添加了 compileJava 任务和 processResources 任务，并且配置了这两个任务的 destinationDir 属性。

#

约定

插件可以通过智能的方法对项目进行预配置以支持约定优于配置。Gradle 对此提供了机制和完善的支持，而它是强大-然而-简洁的构建脚本中的一个关键因素。

在上面的示例中我们看到，Java 插件添加了一个任务，名字为 compileJava ，有一个名为 destinationDir 的属性（即配置编译的 Java 代码存放的地方）。Java 插件默认此属性指向项目目录中的 build/classes/main。这是通过一个合理的默认的约定优于配置的例子。

我们可以简单地通过给它一个新的值来更改此属性。

更改插件的默认设置

build.gradle

```
apply plugin: 'java'
compileJava.destinationDir = file("$buildDir/output/classes")
task show << {
    println relativePath(compileJava.destinationDir)
}
```

gradle -q show 的输出结果

```
> gradle -q show
build/output/classes
```

然而，compileJava 任务很可能不是唯 一需要知道类文件在哪里的任务。

Java 插件添加了 source sets 的概念（见SourceSet）来描述的源文件集的各个方面，其中一个方面是在编译的时候这些类文件应该被写到哪个地方。Java 插件将 compileJava 任务的 destinationDir 属性映射到源文件集的这一个方面。

我们可以通过这个源码集修改写入类文件的位置。

插件中的约定对象

build.gradle

```
apply plugin: 'java'
sourceSets.main.output.classesDir = file("$buildDir/output/classes")
task show << {
```

```
    println relativePath(compileJava.destinationDir)
}
```

gradle -q show 的输出结果

```
> gradle -q show
build/output/classes
```

在上面的示例中，我们应用 Java 插件，除其他外，还做了下列操作：

- 添加了一个新的域对象类型： SourceSet
- 通过属性的默认（即常规）配置了 main 源码集
- 配置支持使用这些属性来执行工作的任务

所有这一切都发生在 apply plugin: "java" 这一步过程中。在上面例子中，我们在约定配置被执行之后，修改了类文件所需的位置。在上面的示例中可以注意到，compileJava.destinationDir 的值也被修改了，以反映出配置的修改。

考虑一下另一种消费类文件的任务的情况。如果这个任务使用 sourceSets.main.output.classesDir 的值来配置，那么修改了这个位置的值，无论它是什么时候被修改，将同时更新 compileJava 任务和这一个消费者任务。

这种配置对象的属性以在所有时间内（甚至当它更改的时候）反映另一个对象的任务的值的能力被称为“映射约定”。它可以令 Gradle 通过约定优于配置及合理的默认值来实现简洁的配置方式。而且，如果默认约定需要进行修改时，也不需要进行完全的重新配置。如果没有这一点，在上面的例子中，我们将不得不重新配置需要使用类文件的每个对象。

21

标准的 Gradle 插件

Gradle 的发行包中有大量的插件。如下列所示：

# #

语言插件

这些插件添加了让各种语言可以被编译和在 JVM 执行的支持。

语言插件

插件 Id	自动应用	与什么插件一起使用	描述
java	`java-base`	-	向一个项目添加 Java 编译、 测试和捆绑的能力。它是很多其他 Gradle 插件的基础服务。
groovy	`groovy-base`	-	添加对 Groovy 项目构建的支持。
scala	`scala-base`	-	添加对 Scala 项目构建的支持。
antlr	`java`	-	添加对使用Antlr作为生成解析器的支持。

# #

正在孵化的语言插件

这些插件添加了对多种语言的支持：

语言插件

插件 Id	自动应用	与什么插件一起使用	描述
assembler	–	–	向项目添加本机汇编语言的功能。
c	–	–	向项目添加 C语言源代码编译功能。
cpp	–	–	向项目添加 c++ 源代码编译功能。
objective-c	–	–	向项目中添加 Objective-C 源代码编译功能。
objective-cpp	–	–	向项目中添加 Objective-C++ 源代码编译功能。
windows-resources	–	–	添加对在本地bin文件中包含 Windows 资源的支持。

#

集成插件

以下这些插件提供了一些与各种运行时技术的集成。

集成插件

插件 Id	自动应用	与什么插件一起使用	描述
application	java	-	添加了一些任务，用于运行和捆绑一个Java项目作为命令行应用程序。
ear	-	java	添加用于构建 J2EE 应用程序的支持。
jetty	war	-	在构建中部署你的web程序到一个内嵌的Jetty web容器中。
maven	-	war	添加用于将项目发布到 Maven 仓库的支持。
osgi	java-base	java	添加构建 OSGi 捆绑包的支持。
war	java	-	添加用于组装 web 应用程序的 WAR 文件的支持。

#

孵化中的集成插件

以下这些插件提供了一些与各种运行时技术的集成。

孵化中的集成插件

插件 Id	自动应用	与什么插件一起使用	描述
distribution	-	-	添加构建 ZIP 和 TAR 分发包的支持。
java-library-distribution	distribution	-	添加构建一个Java类库的 ZIP 和 TAR 分发包的支持。
ivy-publish	-	war	这个插件提供了新的 DSL，用于支持发布文件到 Ivy 存储库，改善了现有的 DSL。
maven-publish	-	war	这个插件提供了新的 DSL，用于支持发布文件到 Maven 存储库，改善了现有的 DSL。

#

软件开发插件

这些插件提供一些软件开发过程上的帮助。

软件开发插件

插件 Id	自动应用	与什么插件一起使用	描述
announce	–	–	将消息发布到你所喜爱的平台，如 Twitter 或 Growl。
build-announcements	announce	–	在构建的生命周期中，把本地公告中有关你感兴趣的事件发送到你的桌面。
checkstyle	java-base	–	使用Checkstyle对您的项目的 Java 源文件执行质量检查并生成报告。
codenarc	groovy-base	–	使用CodeNarc对您的项目的 Groovy 源文件执行质量检查并生成报告。
eclipse	–	scala	生成Eclipse IDE所用到的文件，从而使项目能够导入到 Eclipse。
eclipse-wtp	–	war	与 eclipse 插件一样，但它还生成 eclipse WTP（Web 工具平台）的配置文件。你的war/ear项目在导入eclipse 后，应配置为能在 WTP 中使用。
findbugs	java-base	–	使用FindBugs对您的项目的 Java 源文件执行质量检查并生成报告。
idea	–	java	生成Intellij IDEA IDE所用到的文件，从而使项目能够导入到 IDEA。
jdepend	java-base	–	使用JDepend对您的项目的源文件执行质量检查并生成报告。
pmd	java-base	–	使用PMD对您的项目的 Java 源文件执行质量检查并生成报告。
project-report	reporting-base	–	生成关于Gradle构建中有用的信息的报告。
signing	base	–	添加对生成的文件或构件进行数字签名的功能。

插件 Id	自动应用	与什么插件一起使用	描述
sonar	–	java-base, java, jacoco	提供对sonar-runner插件取代。

#

孵化中的软件开发插件

这些插件提供一些软件开发过程上的帮助。

软件开发插件

插件 Id	自动应用	与什么插件一起使用	描述
build-dashboard	reporting-base	–	生成构建的主控面板的报表。
build-init	wrapper	–	添加用于初始化一个新 Gradle 构建的支持。处理转换 Maven 构建为 Gradle 构建。
cunit	–	–	添加用于运行CUnit测试的支持。
jacoco	reporting-base	java	提供对 Java 的JaCoCo代码覆盖率库的集成。
sonar-runner	–	java-base, java, jacoco	提供对sonar插件取代。
visual-studio	–	本机语言插件	添加对 Visual Studio 的集成。
wrapper	–	–	添加一个用于生成 Gradle wrapper 文件的Wrapper任务。

# #

基本插件

这些插件组成了基本的构建块，其他插件都由此组装而来。它们可供你在你的构建文件中使用，并在此处完整列出。然而，请注意它们都不被认为是 Gradle 公共 API 的一部分。因此，这些插件都不在用户指南中记录。您可能会引用他们的 API 文档，以了解更多关于它们的信息。

基本插件

插件 Id	描述
base	添加标准的生命周期任务，并为归档任务默认进行合理的配置： • 添加构建 添加上传 为所有归档任务配置合适的默认值（比如从version属性被预先配置了默认值，这是非常有用的，因为它促进了跨项目的一致性；完成了有关构件命名规范及构建之后的位置上的一致。）
java-base	对项目添加source set 的概念。不会添加任何特定的soruce sets。
groovy-base	向项目中添加Groovy 的source set概念。
scala-base	向项目中添加Scala 的source set概念。
reporting-base	将一些共享的公约属性添加到项目中，它们与报告的生成有关。

# #

第三方插件

你可以在维基上找到外部插件的列表。

22

Java 插件

Java 插件向一个项目添加了 Java 编译、 测试和 bundling 的能力。它是很多其他 Gradle 插件的基础服务。

# #

用法

要使用 Java 插件，请在构建脚本中加入：

使用 Java 插件

build.gradle

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

#

源集

Java 插件引入了一个源集的概念。一个源集只是一组用于编译并一起执行的源文件。这些源文件可能包括 Java 源代码文件和资源文件。其他有一些插件添加了在源集里包含 Groovy 和 Scala 的源代码文件的能力。一个源集有一个相关联的编译类路径和运行时类路径。

源集的一个用途是，把源文件进行逻辑上的分组，以描述它们的目的。例如，你可能会使用一个源集来定义一个集成测试套件，或者你可能会使用单独的源集来定义你的项目的 API 和实现类。

Java 插件定义了两个标准的源集，分别是 main 和 test。main 源集包含你产品的源代码，它们将被编译并组装成一个 JAR 文件。test 源集包含你的单元测试的源代码，它们将被编译并使用 JUnit 或 TestNG 来执行。

#

任务

Java 插件向你的项目添加了大量的任务，如下所示。

表 23.1. Java 插件–任务

任务名称	依赖于	类型	描述
compileJava	产生编译类路径中的所有任务。这包括了用于 jar 任务。	JavaCompile	使用 javac 编译产品中的 Java 源文件。
processResources	–	Copy	把生产资源文件拷贝到生产的类目录中。
classes	processResources 。一些插件添加了额外的编译任务。	Task	组装生产的类目录。
compileTestJava	compile ，再加上所有能产生测试编译类路径的任务。	JavaCompile	使用 javac 编译 Java 的测试源文件。
processTestResources	–	Copy	把测试的资源文件拷贝到测试的类目录中。
testClasses	processTestResources 。一些插件添加了额外的测试编译任务。	Task	组装测试的类目录。
jar	compile	Jar	组装 JAR 文件
javadoc	compile	Javadoc	使用 Javadoc 生成生产的 Java 源代码的API文档
test	compileTest ，再加上所有产生测试运行时类路径的任务。	Test	使用 JUnit 或 TestNG运行单元测试。
uploadArchives	使用 jar 。	Upload	使用 archives 配置上传包括 JAR 文件的构件。
clean	–	Delete	删除项目的 build 目录。
TaskName	–	Delete	删除由指定的任务所产生的输出文件。例如， jar 任务中所创建的 JAR 文件， test 任务所创建的测试结果。

对于每个你添加到该项目中的源集，Java 插件将添加以下的编译任务：

表 23.2. Java 插件–源集任务

任务名称	依赖于	类型	描述
sourceSet Java	所有产生源集编译类路径的任务。	JavaCompile	使用 javac 编译给定的源集中的 Java 源文件。
sourceSet Resources	–	Copy	把给定的源集的资源文件拷贝到类目录中。
sourceSet Classes	sourceSet Resources。某些插件还为源集添加了额外的编译任务。	Task	组装给定源集的类目录。

Java 插件还增加了大量的任务构成该项目的生命周期：

表 23.3. Java 插件–生命周期任务

任务名称	依赖于	类型	描述
assemble	项目中的所有归档项目，包括 jar 任务。某些插件还向项目添加额外的归档任务。	Task	组装项目中所有的归类文件。
check	项目中的所有核查项目，包括 test 任务。某些插件还向项目添加额外的核查任务。	Task	执行项目中所有的核查任务。
build	assemble	Task	执行项目的完事构建。
buildNeeded	build 任务。	Task	执行项目本身及它所依赖的其他所有项目的完整构建。
buildDependents	build 任务。	Task	执行项目本身及依赖它的其他所有项目的完整构建。
ConfigurationName	使用配置ConfigurationName生成构件的任务。	Task	组装指定配置的构件。该任务由Base插件添加，并由Java插件隐式实现。
ConfigurationName	使用配置ConfigurationName上传构件的任务。	Upload	组装并上传指定配置的构件。该任务由Base插件添加，并由Java插件隐式实现。

uploadConfigurationName 使用配置 ConfigurationName 上传构件的任务。 Upload 组装并上传指定配置的构件。该任务由 Base 插件添加，并由 Java 插件隐式实现。 下图显示了这些任务之间的关系。

图23.1. Java 插件 – 任务

Java 插件 – 任务

#

项目布局

Java 插件会假定如下所示的项目布局。这些目录都不需要一定存在，或者是里面有什么内容。Java 插件将会进行编译，不管它发现什么，并处理缺少的任何东西。

表 23.4. Java 插件–默认项目布局

目录	意义
src/main/java	产品的Java源代码
src/main/resources	产品的资源
src/test/java	Java 测试源代码
src/test/resources	测试资源
sourceSet /java	给定的源集的Java源代码
sourceSet /resources	给定的源集的资源

#

更改项目布局

你可以通过配置适当的源集，来配置项目的布局。这一点将在以下各节中详细讨论。这里是如何更改 main Java 和资源源目录的一个简短的例子。

自定义 Java 源代码布局

build.gradle

```
sourceSets {
    main {
        java {
            srcDir 'src/java'
        }
        resources {
            srcDir 'src/resources'
        }
    }
}
```

#

依赖管理

Java 插件向项目添加了许多依赖配置，如下图所示。它对一些任务指定了这些配置，如 compileJava 和 test。

表23.5. Java插件 – 依赖配置

名称	继承自	在哪些任务中使用	意义
compile	–	compileJava	编译时依赖
runtime	compile	–	运行时依赖
testCompile	compile	compileTestJava	用于编译测试的其他依赖
testRuntime	runtime, testCompile	test	只用于运行测试的其他依赖
archives	–	uploadArchives	由本项目生产的构件（如jar包）。
default	runtime	–	本项目上的默认项目依赖配置。包含本项目运行时所需要的构件和依赖。

图23.2. Java 插件 – 依赖配置

Java 插件 – 依赖配置

对于每个你添加到项目中的源集，Java 插件都会添加以下的依赖配置：

表23.6. Java 插件 – 源集依赖配置

名称	继承自	在哪些任务中使用	意义
sourceSetCompile	–	compileSourceSetJava	给定源集的编译时依赖
sourceSetRuntime	sourceSetCompile	–	给定源集的运行时依赖

#

常规属性

Java 插件向项目添加了许多常规属性，如下图所示。您可以在构建脚本中使用这些属性，就像它们是 project 对象的属性一样。

表23.7. Java 插件 – 目录属性

属性名称	类型	默认值	描述
reportsDirName	String	reports	相对于build目录的目录名称，报告将生成到此目录。
reportsDir	File (read-only)	reportsDirName	报告将生成到此目录。
testResultsDirName	String	test-results	相对于build目录的目录名称，测试报告的.xml文件将生成到此目录。
testResultsDir	File (read-only)	testResultsDirName	测试报告的.xml文件将生成到此目录。
testReportDirName	String	tests	相对于build目录的目录名称，测试报告将生成到此目录。
testReportDir	File (read-only)	testReportDirName	测试报告生成到此目录。
libsDirName	String	libs	相对于build目录的目录名称，类库将生成到此目录中。
libsDir	File (read-only)	libsDirName	类库将生成到此目录中。
distsDirName	String	distributions	相对于build目录的目录名称，发布的文件将生成到此目录中。
distsDir	File (read-only)	distsDirName	要发布的文件将生成到此目录。
docsDirName	String	docs	相对于build目录的目录名称，文档将生成到此目录。
docsDir	File (read-only)	docsDirName	要生成文档的目录。
dependencyCacheDirName	String	dependency-cache	相对于build目录的目录名称，该目录用于缓存源代码的依赖信息。
dependencyCacheDir	File (read-only)	dependencyCacheDirName	该目录用于缓存源代码的依赖信息。

表 23.8. Java 插件 – 其他属性

属性名称	类型	默认值	描述
sourceSets	SourceSetContainer (read-only)	非空	包含项目的源集。
sourceCompatibility	JavaVersion. 可以使用字符串或数字来设置，例如 1.5 。	当前JVM所使用的值	当编译Java源代码时所使用的Java版本兼容性。
targetCompatibility	JavaVersion. 可以使用字符串或数字来设置，例如 1.5 。	sourceCompatibility	要生成的类的 Java 版本。
archivesBaseName	String	projectName	像JAR或ZIP文件这样的构件的basename
manifest	Manifest	一个空的清单	要包括的所有 JAR 文件的清单。

这些属性由 JavaPluginConvention， BasePluginConvention 和 ReportingBasePluginConvention 这些类型的常规对象提供。

#

使用源集

你可以使用 sourceSets 属性访问项目的源集。这是项目的源集的容器，它的类型是 SourceSetContainer。除此之后，还有一个 sourceSets {}的脚本块，可以传入一个闭包来配置源集容器。源集容器的使用方式几乎与其他容器一样，例如 tasks。

访问源集

build.gradle

```
// Various ways to access the main source set
println sourceSets.main.output.classesDir
println sourceSets['main'].output.classesDir
sourceSets {
    println main.output.classesDir
}
sourceSets {
    main {
        println output.classesDir
    }
}
// Iterate over the source sets
sourceSets.all {
    println name
}
```

要配置一个现有的源集，你只需使用上面的其中一种访问方法来设置源集的属性。这些属性将在下文中进行介绍。下面是一个配置 main 的 Java 和资源目录的例子：

配置源集的源代码目录

build.gradle

```
sourceSets {
    main {
        java {
            srcDir 'src/java'
        }
        resources {
            srcDir 'src/resources'
        }
```

```
    }
}
```

#

源集属性

下表列出了一些重要的源集属性。你可以在 SourceSet 的 API 文档中查看更多的详细信息。

表 23.9. Java 插件 – 源集属性

属性名称	类型	默认值	描述
name	String (read-only)	非空	用来确定一个源集的源集名称。
output	SourceSetOutput (read-only)	非空	源集的输出文件，包含它编译过的类和资源。
output.classesDir	File	name	要生成的该源集的类的目录。
output.resourcesDir	File	name	要生成的该源集的资源的目录。
compileClasspath	FileCollection	SourceSet 配置。	该类路径在编译该源集的源文件时使用。
runtimeClasspath	FileCollection	SourceSet 配置。	该类路径在执行该源集的类时使用。
java	SourceDirectorySet (read-only)	非空	该源集的Java源文件。仅包含Java源文件目录里的 .java 文件，并排除其他所有文件。
java.srcDirs	Set<File>	name /java]	该源目录包含了此源集的所有Java源文件。
resources	SourceDirectorySet (read-only)	非空	此源集的资源文件。仅包含资源文件，并且排除在资源源目录中找到的所有 .java 文件。其他插件，如Groovy 插件，会从该集合中排除其他类型的文件。
resources.srcDirs	Set<File>	name /resources]	该源目录包含了此源集的资源文件。
allJava	SourceDirectorySet (read-only)	java	该源集的所有 .java 文件。有些插件，如Groovy 插件，会从该集合中增加其他的Java源文件。

属性名称	类型	默认值	描述
allSource	SourceDirectorySet (read-only)	resources + java	该源集的所有源文件。包含所有的资源文件和Java源文件。有些插件，如Groovy 插件，会从该集合中增加其他的源文件。

#

定义新的源集

要定义一个新的源集，你只需在 sourceSets {}块中引用它。下面是一个示例：

定义一个源集

build.gradle

```
sourceSets {
   intTest
}
```

当您定义一个新的源集时，Java 插件会为该源集添加一些依赖配置，如表 23.6，“Java 插件 – 源集依赖项配置”所示。你可以使用这些配置来定义源集的编译和运行时的依赖。

定义源集依赖

build.gradle

```
sourceSets {
   intTest
}
dependencies {
   intTestCompile 'junit:junit:4.11'
   intTestRuntime 'org.ow2.asm:asm-all:4.0'
}
```

Java 插件还添加了大量的任务，用于组装源集的类，如表 23.2，“Java 插件 – 源设置任务”所示。例如，对于一个被叫做 intTest 的源集，你可以运行 gradle intTestClasses 来编译 int 测试类。

编译源集

gradle intTestClasses的输出结果

```
> gradle intTestClasses
:compileIntTestJava
:processIntTestResources
```

```
:intTestClasses
BUILD SUCCESSFUL
Total time: 1 secs
```

#

一些源集的范例

添加一个包含了源集的类的 JAR 包

示例 23.8. 为一个源集装配一个JAR文件

build.gradle

```
task intTestJar(type: Jar) {
    from sourceSets.intTest.output
}
```

为一个源集生成 Javadoc：

示例 23.9. 为一个源集生成 Javadoc：

build.gradle

```
task intTestJavadoc(type: Javadoc) {
    source sourceSets.intTest.allJava
}
```

添加一个测试套件以运行一个源集里的测试

示例 23.10. 运行源集里的测试

build.gradle

```
task intTest(type: Test) {
    testClassesDir = sourceSets.intTest.output.classesDir
    classpath = sourceSets.intTest.runtimeClasspath
}
```

#

Javadoc

Javadoc 任务是 Javadoc 的一个实例。它支持核心的 javadoc 参数选项，以及在 Javadoc 可执行文件的参考文档中描述的标准 doclet 参数选项。对于支持的 Javadoc 参数选项的完整列表，请参考下面的类的 API 文档：CoreJavadocOptions 和StandardJavadocDocletOptions。

表 23.10. Java 插件 – Javadoc 属性

任务属性	类型	默认值
classpath	FileCollection	sourceSets.main.output + sourceSets.main.compileClasspath
source	FileTree.	sourceSets.main.allJava
destinationDir	File	docsDir /javadoc
title	String	project的名称和版本

#

清理

clean 任务是 Delete 的一个实例。它只是删除由其 dir 属性表示的目录。

表 23.11. Java 插件 – Clean 性能

任务属性	类型	默认值
`dir`	`File`	`buildDir`

#

资源

Java 插件使用 Copy 任务进行资源的处理。它为该 project 里的每个源集添加一个实例。你可以在16.6章节，“复制文件”中找到关于 copy 任务的更多信息。

表 23.12. Java 插件–ProcessResources 属性

任务属性	类型	默认值
srcDirs	Object .	sourceSet .resources
destinationDir	16.1 章节，“查找文件”中所讲到的任何一种方式来设置。	sourceSet .output.resourcesDir

#

CompileJava

Java 插件为该 project 里的每个源集添加一个 JavaCompile 实例。一些最常见的配置选项如下所示。

表 23.13. Java 插件– Compile 属性

任务属性	类型	默认值
classpath	FileCollection	sourceSet.compileClasspath
source	FileTree	sourceSet.java
destinationDir	File.	sourceSet.output.classesDir

compile 任务委派了 Ant 的 javac 任务。将 options.useAnt 设置为 false 将绕过 Ant 任务，而激活 Gradle 的直接编译器集成。在未来的 Gradle 版本中，将把它作为默认设置。

默认情况下，Java 编译器在 Gradle 过程中运行。将 options.fork 设置为 true 将会使编译出现在一个单独的进程中。在 Ant javac 任务中，这意味着将会为每一个 compile 任务fork 一个新的进程，而这将会使编译变慢。相反，Gradle 的直接编译器集成（见上文）将尽可能多地重用相同的编译器进程。在这两种情况下，使用 options.forkOptions 指定的所有fork 选项都将得到重视。

# #

Test

test 任务是 Test 的一个实例。它会自动检测和执行 test 源集中的所有单元测试。测试执行完成后，它还会生成一份报告。同时支持 JUnit 和 TestNG。可以看一看Test的完整的 API。

#

测试执行

测试在单独的 JVM 中执行，与 main 构建进程隔离。Test 任务的 API 可以让你控制什么时候开始。

有大量的属性用于控制测试进程的启动。这包括系统属性、 JVM 参数和使用的 Java 可执行文件。

你可以指定是否要并行运行你的测试。Gradle 通过同时运行多个测试进程来提供并行测试的执行。每个测试进程会依次执行一个单一的测试，所以你一般不需要对你的测试做任何的配置来利用这一点。 MaxParallelForks 属性指定在给定的时间可以运行的测试进程的最大数。它的默认值是 1，也就是说，不并行执行测试。

测试进程程会为其将 org.gradle.test.worker 系统属性设置为一个唯一标识符，这个标识符可以用于文件名称或其他资源标识符。

你可以指定在执行了一定数量的测试类之后，重启那个测试进程。这是一个很有用的替代方案，让你的测试进程可以有很大的堆内存。forkEvery 属性指定了要在测试进程中执行的测试类的最大数。默认是每个测试进程中执行的测试数量不限。

该任务有一个 ignoreFailures 属性，用以控制测试失败时的行为。test 会始终执行它检测到的每一个测试。如果 ignoreFailures 为 false，并且有测试不通过，那它会停止继续构建。IgnoreFailures 的默认值为 false。

testLogging 属性可以配置哪些测试事件需要记录，并且使用什么样的日志级别。默认情况下，对于每个失败的测试都只会打印一个简洁的消息。请参阅 TestLoggingContainer，查看如何把你的测试日志打印调整为你的偏好设置。

#

调试

test 任务提供了一个 Test.getDebug()属性，可以设置为启动，使 JVM 在执行测试之前，等待一个 debugger 连接到它的 5005 端口上。

这也可以在调用时通过--debug-vm task 选项进行启用。

#

测试过滤

从 Gradle 1.10 开始，可以根据测试的名称模式，只包含指定的测试。过滤，相对于测试类的包含或排除，是一个不同的机制。它将在接下来的段落中进行描述（-Dtest.single， test.include 和 friends ）。后者基于文件，比如测试实现类的物理位置。file-level 的测试选择不支持的很多有趣的情况，都可以用 test-level 过滤来做到。以下这些场景中，有一些 Gradle 现在就可以处理，而有一些则将在未来得到实现：

- 在指定的测试的方法级别上进行过滤；执行单个测试方法
- 基于自定义注解（以后实现）进行过滤
- 基于测试层次结构进行过滤； 执行所有继承了某一基类（以后实现） 的测试
- 基于一些自定义的运行时的规则进行过滤，例如某个系统属性或一些静态的特定值（以后实现）

测试过滤功能具有以下的特征：

- 支持完整的限定类名称或完整的限定的方法名称，例如“org.gradle.SomeTest”、“org.gradle.SomeTest.someMethod”
- 通配符 “*” 支付匹配任何字符
- 提供了“--tests”的命令行选项，以方便地设置测试过滤器。这对“单一测试方法的执行”的经典用例特别有用。当使用该命令行选项选项时，在构建脚本中声明的列入过滤器的测试将会被忽略。
- Gradle 尽最大努力对有着特定的测试框架 API 的局限的测试进行过滤。一些高级的、 综合的测试可能不完全符合过滤条件。然而，绝大多数的测试和用例都会被很好地处理。
- 测试过滤将会取代基于文件的测试选择。后者可能在将来会被完全地取代掉。我们将会继续改进测试过滤的 API，并添加更多种类的过滤器。

在构建脚本中过滤测试

build.gradle

```
test {
    filter {
```

```
        //include specific method in any of the tests
        includeTestsMatching "*UiCheck"
        //include all tests from package
        includeTestsMatching "org.gradle.internal.*"
        //include all integration tests
        includeTestsMatching "*IntegTest"
    }
}
```

有关更多的详细信息和示例，请参阅 TestFilter 的文档。

使用命令行选项的一些示例：

- gradle test --tests org.gradle.SomeTest.someSpecificFeature
- gradle test --tests *SomeTest.someSpecificFeature
- gradle test --tests *SomeSpecificTest
- gradle test --tests all.in.specific.package*
- gradle test --tests *IntegTest
- gradle test --tests *IntegTest*ui*
- gradle someTestTask --tests *UiTest someOtherTestTask --tests *WebTest*ui

#

通过系统属性执行单一的测试

这种机制已经被上述的“测试过滤”所取代。 设置一个 taskName.single = testNamePattern 的系统属性将会只执行匹配 testNamePattern 的那些测试。这个 taskName 可以是一个完整的多项目路径，比如“sub1:sub2:test”，或者仅是一个任务名称。testNamePattern 将用于形成一个“**/testNamePattern*.class” 的包含模式。如果这个模式无法找到任何测试，那么将会抛出一个异常。这是为了使你不会误以为测试通过。如果执行了一个以上的子项目的测试，该模式会被应用于每一个子项目中。如果在一个特定的子项目中，找不到测试用例，那么将会抛出异常。在这种情况下你可以使用路径标记法的模式，这样该模式就会只应用于特定的子项目的测试任务中。或者你可以指定要执行的任务的完整限定名称。你还可以指定多个模式。示例：

- gradle -Dtest.single=ThisUniquelyNamedTest test
- gradle -Dtest.single=a/b/ test
- gradle -DintegTest.single=*IntegrationTest integTest
- gradle -Dtest.single=:proj1:test:Customer build

- gradle -DintegTest.single=c/d/ :proj1:integTest

#

测试检测

Test 任务通过检查编译过的测试类来检测哪些类是测试类。默认情况下它会扫描所有的.class 文件。您可以设置自定义的 includes 或 excludes，这样就只有这些类才会被扫描。根据所使用的测试框架（ JUnit 或 TestNG ），会使用不同的标准进行测试类的检测。

当使用 JUnit 时，我们扫描 JUnit 3 和 4 的测试类。如果满足以下的任何一个条件，这个类就被认为是一个 JUnit 测试类:

- 类或超类继承自 TestCase 或 GroovyTestCase
- 类或超类使用了 @RunWith 进行注解
- 类或超类含有一个带 @Test 注解的方法

当使用 TestNG 时，我们扫描所有带有 @Test 注解的方法。

请注意，抽象类不会被执行。Gradle 还将扫描测试类路径中的 jar 文件里的继承树。

如果你不想要使用测试类检测，可以通过设置 scanForTestClasses 为 false 来禁用它。这将使test任务只使用 includes / excludes 来找到测试类。如果 scanForTestClasses 为disabled，并且没有指定 include 或 exclude 模式，则使用各自的默认值。对于 include 的默认值是 "/*Tests.class"， "/*Test.class"，而对于exclude它的默认值是 "**/Abstract*.class"。

#

测试分组

JUnit 和 TestNG 可以对测试方法进行复杂的分组。

为对 Junit 测试类和方法进行分组，JUnit 4.8 引入了类别的概念。test 任务可以实现一个规范，让你 include 和 exclude 想要的 JUnit 类别。

JUnit 类别

build.gradle

```
test {
    useJUnit {
        includeCategories 'org.gradle.junit.CategoryA'
        excludeCategories 'org.gradle.junit.CategoryB'
    }
}
```

TestNG 框架有一个非常相似的概念。在 TestNG 中你可以指定不同的测试组。应从测试执行中 include 或 exclude 的测试组，可以在 test 任务中配置。

对 TestNG 测试分组

build.gradle

```
test {
    useTestNG {
        excludeGroups 'integrationTests'
        includeGroups 'unitTests'
    }
}
```

#

测试报告

test 任务默认情况下会生成以下结果。

- 一个 HTML 测试报告。
- 与 Ant Junit report 任务兼容的 XML 格式的结果。这种格式可以被许多工具所支持，比如CI服务器。
- 有效二进制格式的结果。这个任务会从这些二进制结果生成其他的结果。

您可以使用 Test.setTestReport()方法来禁用 HTML 测试报告。目前不能禁用其他的结果。

这里还有一个独立的 TestReport 任务类型，它可以从一个或多个 Test 任务实例生成的二进制结果中生成 HTML 测试报告。要使用这个任务类型，你需要定义一个 destinationDir 和要包含到报告的测试结果。这里是一个范例，从子项目的单元测试中生成一个联合报告：

为多个子项目创建一个单元测试报告

build.gradle

```
subprojects {
    apply plugin: 'java'
```

```
    // Disable the test report for the individual test task
    test {
        reports.html.enabled = false
    }
}
task testReport(type: TestReport) {
    destinationDir = file("$buildDir/reports/allTests")
    // Include the results from the `test` task in all subprojects
    reportOn subprojects*.test
}
```

你应该注意到，TestReport 类型组合了多个测试任务的结果，并且需要聚合各个测试类的结果。这意味着，如果一个给定的测试类被多个 test 任务所执行，那么测试报告将包括那个类的所有执行结果，但它难以区分那个类的每次执行和它们的输出。

#

TestNG 参数化方法和报告

TestNG 支持参数化测试方法，允许一个特定的测试方法使用不同的输入执行多次。Gradle 会在这个方法的执行报告中包含进它的参数值。

给定一个带有两个参数，名为 aParameterizedTestMethod 参数化测试方法，它将使用名称这个名称进行报告：aParameterizedTestMethod(toStringValueOfParam1, toStringValueOfParam2)。这使得在特定的迭代过程，很容易确定参数值。

#

常规值

表 23.14. Java 插件 – test 属性

任务属性	类型	默认值
testClassesDir	File	sourceSets.test.output.classesDir
classpath	FileCollection	sourceSets.test.runtimeClasspath
testResultsDir	File	testResultsDir
testReportDir	File	testReportDir
testSrcDirs	List<File>	sourceSets.test.java.srcDirs

#

Jar

Jar 任务创建一个包含类文件和项目资源的 JAR 文件。JAR 文件在 archives 依赖配置中被声明为一个构件。这意味着这个 JAR 文件被包含在一个依赖它的项目的类路径中。如果你把你的项目上传到仓库，这个 JAR 文件会被声明为依赖描述符的一部分。你可以在第16.8节，“创建档案”和第51章， 发布 artifact 中了解如何使用 archives 和配置 artifact。

#

Manifest

每个 jar 或 war 对象都有一个单独的 Manifest 实例的 manifest 属性。当生成 archive 时，相应的 MANIFEST.MF 文件也会被写入进去。

自定义的 MANIFEST.MF

build.gradle

```
jar {
    manifest {
        attributes("Implementation-Title": "Gradle", "Implementation-Version": version)
    }
}
```

您可以创建一个单独的 Manifest 实例。它可以用于共享两个 jar 包的 manifest 信息。

创建一个 manifest 对象。

build.gradle

```
ext.sharedManifest = manifest {
    attributes("Implementation-Title": "Gradle", "Implementation-Version": version)
}
task fooJar(type: Jar) {
    manifest = project.manifest {
        from sharedManifest
    }
}
```

你可以把其他的 manifest 合并到任何一个 Manifest 对象中。其他的 manifest 可能使用文件路径来描述，像上面的例子，使用对另一个 Manifest 对象的引用。

指定 archive 的单独 MANIFEST.MF

build.gradle

```
task barJar(type: Jar) {
   manifest {
      attributes key1: 'value1'
      from sharedManifest, 'src/config/basemanifest.txt'
      from('src/config/javabasemanifest.txt', 'src/config/libbasemanifest.txt') {
         eachEntry { details ->
            if (details.baseValue != details.mergeValue) {
               details.value = baseValue
            }
            if (details.key == 'foo') {
               details.exclude()
            }
         }
      }
   }
}
```

Manifest 会根据在 from 语句中所声明的顺序进行合并。如果基本的 manifest 和要合并的 manifest 都定义了同一个 key 的值，那么默认情况下会采用要合并的 manifest 的值。你可以通过添加 eachEntry action 来完全自定义合并行为，它可以让你对每一项生成的 manifest 访问它的一个 ManifestMergeDetails 实例。这个合并操作不会在 from 语句中就马上被触发。它是懒执行的，不管是对于生成一个 jar 包，还是调用了 writeTo 或者 effectiveManifest

你可以很轻松地把一个 manifest 写入磁盘中。

指定 archive 的单独 MANIFEST.MF

build.gradle

```
jar.manifest.writeTo("$buildDir/mymanifest.mf")
```

23

Groovy 插件

Groovy 的插件继承自 Java 插件并添加了对 Groovy 项目的支持。它可以处理 Groovy 代码，以及混合的 Groovy 和 Java 代码，甚至是纯 Java 代码（尽管我们不一定推荐使用）。该插件支持联合编译，可以任意地混合及匹配 Groovy 和 Java 代码各自的依赖。例如，一个 Groovy 类可以继承自一个 Java 类，而这个 Java 类也可以继承自一个 Groovy 类。这样一来，我们就能够在项目中使用最适合的语言，并且在有需要的情况下用其他的语言重写其中的任何类。

# #

用法

要使用 Groovy 的插件，请在构建脚本中包含以下语句：

使用 Groovy 插件

build.gradle

```
apply plugin: 'groovy'
```

# #

任务

Groovy 的插件向 project 中添加了以下任务。

表 24.1. Groovy 插件 – 任务

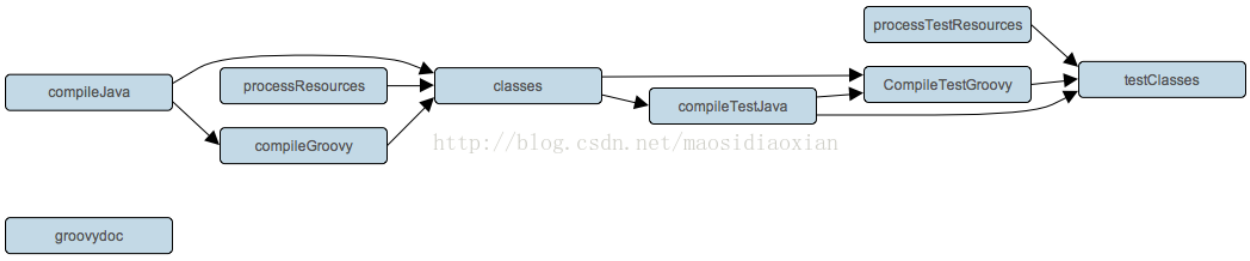
任务名称	依赖于	类型	描述
compileGroovy	compileJava	GroovyCompile	编译production 的 Groovy 源文件。
compileTestGroovy	compileTestJava	GroovyCompile	编译test 的 Groovy 的源文件。
SourceSet Groovy	SourceSet Java	GroovyCompile	编译给定的 source set 里的 Groovy 源文件。
groovydoc	–	Groovydoc	为 production 里的 Groovy 源文件生成 API 文档。

Groovy 的插件向 Java 插件所加入的 tasks 添加了以下的依赖。

表 24.2. Groovy 插件 – 额外的 task 依赖

任务名称	依赖于
classes	compileGroovy
testClasses	compileTestGroovy
sourceSetClasses	compileSourceSetGroovy

图 24.1. Groovy 插件 – tasks



#

项目布局

Groovy 的插件会假定项目的布局如表 24.3，“Groovy 插件 – 项目布局”中所示。所有 Groovy 的源目录都可以包含 Groovy 和 Java 代码。Java 源目录只能包含 Java 源代码。这些目录不一定得存在或是里面包含有什么内容；Groovy 的插件只会进行编译，而不管它发现什么。

表 24.3. Groovy 插件 – 项目布局

目录	意义
src/main/java	产品的Java源代码
src/main/resources	产品的资源
src/main/groovy	Production Groovy 源代码。此外可能包含联合编译的 Java 源代码。
src/test/java	Java 测试源代码
src/test/resources	测试资源
src/test/groovy	Test Groovy 源代码。此外可能包含联合编译的 Java 源代码。
sourceSet /java	给定的源集的Java源代码
sourceSet /resources	给定的源集的资源
sourceSet /groovy	给定的source set 的 Groovy 源代码。此外可能包含联合编译的 Java 源代码。

#

更改项目布局

和 Java 插件一样，Groovy 插件允许把 Groovy 的 production 和 test 的源文件配置为自定义的位置。

自定义 Groovy 自定义源文件布局

build.gradle

```
sourceSets {
   main {
      groovy {
         srcDirs = ['src/groovy']
      }
   }
   test {
      groovy {
```

```
        srcDirs = ['test/groovy']
      }
    }
}
```

#

依赖管理

由于 Gradle 的构建语言基于 Groovy 的，且部分的 Groovy 使用 Groovy 实现，因此 Gradle 已经带有了一个 Groovy 库（截至 Gradle 1.6 所带的 Groovy 库的版本是 1.8.6）。然而，Groovy 项目需要显式地声明一个 Groovy 依赖。这个依赖会在编译和运行的类路径时用到。它还将用于分别获取 Groovy 编译器及 Groovydoc 工具。

如果 Groovy 用于 production 代码，Groovy 依赖应该添加到 compile 配置中：

Groovy 的依赖配置

```
build.gradle
repositories {
    mavenCentral()
}
dependencies {
    compile 'org.codehaus.groovy:groovy-all:2.2.0'
}
```

如果 Groovy 仅用于测试代码，Groovy 的依赖应该被添加到 testCompile 配置中：

配置 Groovy 测试依赖

build.gradle

```
dependencies {
    testCompile "org.codehaus.groovy:groovy-all:2.2.0"
}
```

如果要使用 Gradle 所带的 Groovy 库，请声明 localGroovy()依赖。注意，不同 Gradle 版本附带的 Groovy 版本不同；因此，声明一个固定的 Groovy 依赖要比使用 localGroovy()更安全一些。

配置捆绑的 Groovy 依赖

build.gradle

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

Groovy 库不一定得从远程仓库中获取。它也可以获取自本地中可能检入版本控制的 lib 目录：

配置 Groovy 文件依赖

build.gradle

```
repositories {
    flatDir { dirs 'lib' }
}
dependencies {
    compile module('org.codehaus.groovy:groovy:1.6.0') {
        dependency('asm:asm-all:2.2.3')
        dependency('antlr:antlr:2.7.7')
        dependency('commons-cli:commons-cli:1.2')
        module('org.apache.ant:ant:1.9.3') {
            dependencies('org.apache.ant:ant-junit:1.9.3@jar', 'org.apache.ant:ant-launcher:1.9.3')
        }
    }
}
```

#

groovyClasspath 的自动配置

GroovyCompile 和 Groovydoc tasks 会以两种方式使用 Groovy： 在它们的 classpath 以及它们的groovyClasspath上。前者用于在源代码中查找类的引用，通常会包含 Groovy 库和其他库。后者用来分别加载和执行 Groovy 编译器和 Groovydoc 工具，并且应该只包含 Groovy 库及其依赖项。

除非显式配置了一个 task 的 groovyClasspath ，否则 Groovy（基础）插件会尝试推断该 task 的 classpath。以如下方式进行：

- 如果在在 classpath 中找到 groovy-all(-indy) Jar，相同的 Jar 将添加到groovyClasspath 中。
- 如果在 classpath 中找到 groovy(-indy) Jar ，并且该项目已经在至少一个仓库中声明了它，那么相应的 groovy(-indy)的仓库依赖将添加到 groovyClasspath 中。
- 其他情况，该 task 将执行失败，并提示无法推断 groovyClasspath。

#

常规属性

Groovy 的插件没有向 project 添加任何的常规属性。

#

source set 属性

Groovy 的插件向 project 的每一个source set 添加了下列的常规属性。你可以在你的构建脚本中，把这些属性当成是 source set 对象中的属性一样使用。

表 24.4. Groovy 插件 - source set 属性

属性名称	类型	默认值	描述
groovy	SourceDirectorySet (read-only)	非空	该source set 中的 Groovy 源文件。包含在 Groovy 源目录中找到的所有的 .java 文件，并排除所有其他类型的文件。
groovy.srcDirs	Set<File>.	name /groovy]	源目录包含该 source set 中的 Groovy 源文件。此外可能还包含用于联合编译的 Java 源文件。
allGroovy	FileTree (read-only)	非空	该source set 中的所有 Groovy 源文件。包含在 Groovy 源目录中找到的所有的 .groovy 文件。

这些属性由一个 GroovySourceSet 的约定对象提供。

Groovy 的插件还修改了一些 source set 的属性：

表 24.5. Groovy 的插件 - source set 属性

属性名称	修改的内容
allJava	添加在 Groovy 源目录中找到的所有 .java 文件。
allSource	添加在 Groovy 的源目录中找到的所有源文件。

#

GroovyCompile

Java 插件向 project 里的每个 source set 添加了一个 JavaCompile task。这个 task 的类型继承自 JavaCompile task。除非 groovyOptions.useAnt 设置为 true，否则将使用 Gradle 集成的本地的 Groovy 编译器。对于大多数项目而言，这相比基于 Ant 编译器，是个更好的选择。GroovyCompile task 支持官方的 Groovy 编译器的大多数配置选项。

表 24.6. Groovy 插件 – GroovyCompile 属性

任务属性	类型	默认值
classpath	FileCollection	sourceSet.compileClasspath
source	FileTree	sourceSet.groovy
destinationDir	File.	sourceSet.output.classesDir
groovyClasspath	FileCollection	如果 classpath 中找到的 Groovy 库

24

Scala 插件

Scala 的插件继承自 Java 插件并添加了对 Scala 项目的支持。它可以处理 Scala 代码，以及混合的 Scala 和 Java 代码，甚至是纯 Java 代码（尽管我们不一定推荐使用）。该插件支持联合编译，联合编译可以通过 Scala 及 Java 的各自的依赖任意地混合及匹配它们的代码。例如，一个 Scala 类可以继承自一个 Java 类，而这个 Java 类也可以继承自一个 Scala 类。这样一来，我们就能够在项目中使用最适合的语言，并且在有需要的情况下用其他的语言重写其中的任何类。

# #

用法

要使用 Scala 插件，请在构建脚本中包含以下语句：

使用 Scala 插件

build.gradle

```
apply plugin: 'scala'
```

# #

任务

Scala 的插件向 project 中添加了以下任务。

表 25.1. Scala 插件 – 任务

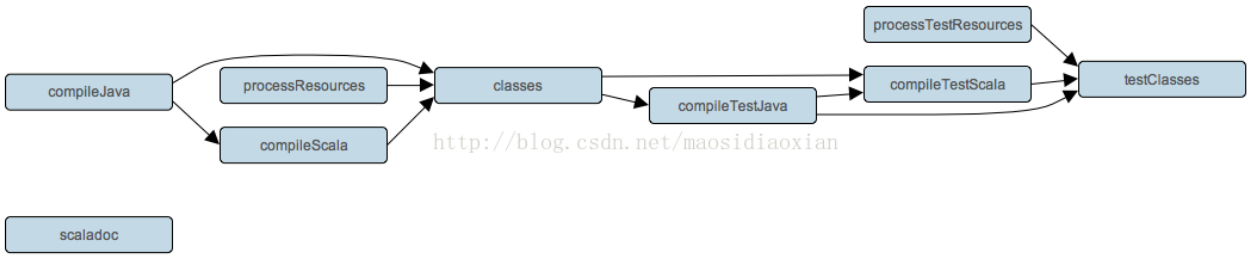
任务名称	依赖于	类型	描述
compileScala	compileJava	ScalaCompile	编译production 的 Scala 源文件。
compileTestScala	compileTestJava	ScalaCompile	编译test 的 Scala 的源文件。
SourceSet Scala	SourceSet Java	ScalaCompile	编译给定的source set 里的 Scala 源文件。
scaladoc	–	scaladoc	为production 里的 Scala 源文件生成 API 文档。

Scala 插件向 Java 插件所加入的 tasks 添加了以下的依赖。

表 24.2. Scala 感觉 插件 – 额外的 task 依赖

任务名称	依赖于
classes	compileScala
testClasses	compileTestScala
sourceSet Classes	SourceSet Scala

图 25.1. Scala 插件–任务



#

项目布局

Scala 插件会假定如下所示的项目布局。所有 Scala 的源目录都可以包含 Scala和Java 代码。Java 源目录只能包含 Java 源代码。这些目录不一定是存在的，或是里面包含有什么内容；Scala 插件只会进行编译，而不管它发现什么。

表 25.3. Scala 插件 – 项目布局

目录	意义
src/main/java	产品的Java源代码
src/main/resources	产品的资源
src/main/scala	Production Scala 源代码。此外可能包含联合编译的 Java 源代码。
src/test/java	Java 测试源代码
src/test/resources	测试资源
src/test/scala	Test Scala 源代码。此外可能包含联合编译的 Java 源代码。
sourceSet /java	给定的源集的Java源代码
sourceSet /resources	给定的源集的资源
sourceSet /scala	给定的source set 的 Scala 源代码。此外可能包含联合编译的 Java 源代码。

#

更改项目布局

和 Java 插件一样，Scala 插件允许把 Scala 的 production 和 test 的源文件配置为自定义的位置。

自定义 Scala 源文件布局

build.gradle

```
sourceSets {
   main {
      scala
         srcDirs = ['src/scala']
      }
   }
   test {
      scala
```

```
        srcDirs = ['test/scala']
      }
    }
}
```

# #

依赖管理

Scala 项目需要声明一个 scala-library 依赖项。这个依赖会在编译和运行的类路径时用到。它还将用于分别获取 Scala 编译器及 Scaladoc 工具。

如果 Scala 用于 production 代码， scala-library 依赖应该添加到 compile 的配置中：

为production 代码定义一个Scala 依赖

build.gradle

```
repositories {
    mavenCentral()
}
dependencies {
    compile 'org.scala-lang:scala-library:2.9.1'
}
```

如果 Scala 仅用于测试代码， scala-library 依赖应被添加到 testCompile 配置中：

为 test 代码定义一个Scala 依赖

build.gradle

```
dependencies {
    testCompile "org.scala-lang:scala-library:2.9.2"
}
```

#

scalaClasspath 的自动配置

ScalaCompile 和 ScalaDoc tasks 会以两种方式使用 Scala： 在它们的 classpath 以及scalaClasspath 上。前者用于在源代码中查找类的引用，通常会包含 scala-library 和其他库。后者用来分别加载和执行 Scala 编译器和 Scala 工具，并且应该只包含 scala-library及其依赖项。

除非显式配置了一个 task 的 scalaClasspath ，否则 Scala（基础）插件会尝试推断该 task 的 classpath。以如下方式进行：

- 如果在 classpath 中找到 scala-library Jar ，并且该项目已经在至少一个仓库中声明了它，那么相应的 scala-compiler 的仓库依赖将添加到 scalaClasspath中。
- 其他情况，该 task 将执行失败，并提示无法推断 scalaClasspath。

#

公约属性

Scala 插件没有向 project 添加任何的公约属性。

# #

source set 属性

Scala 的插件向 project 的每一个 source set 添加了下列的公约属性。你可以在你的构建脚本中，把这些属性当成是 source set 对象中的属性一样使用。

表 25.4. Scala 插件 - source set 属性

属性名称	类型	默认值	描述
scala	SourceDirectory Set (read-only)	非空	该source set 中的 Scala 源文件。包含在 Scala 源目录中找到的所有的 .java 文件，并排除所有其他类型的文件。
scala.srcDirs	Set<File> .	name /scala]	源目录包含该 source set 中的 Scala 源文件。此外可能还包含用于联合编译的 Java 源文件。
allScala	FileTree (read-only)	非空	该source set 中的所有 Scala 源文件。包含在 Scala 源目录中找到的所有的 .scala 文件。

这些属性由一个 ScalaSourceSet 的约定对象提供。

Scala 的插件还修改了一些 source set 的属性：

表 25.5. Scala 插件 - source set 属性

属性名称	修改的内容
allJava	添加在 Scala 源目录中找到的所有 .java 文件。
allSource	添加在 Scala 的源目录中找到的所有源文件。

# #

Fast Scala Compiler

Scala 插件包含了对 fsc，即 Fast Scala Compiler 的支持。fsc 运行在一个单独的进程中，并且可以显著地提高编译速度。

启用 Fast Scala Compiler

build.gradle

```
compileScala
    scalaCompileOptions.useCompileDaemon = true
    // optionally specify host and port of the daemon:
    scalaCompileOptions.daemonServer = "localhost:4243"
}
```

注意，每当 fsc 的编译类路径的内容发生变化时，它都需要重新启动。(它本身不会去检测编译类路径的更改。）这使得它不太适合于多项目的构建。

#

在外部进程中编译

当 scalaCompileOptions.fork 设置为 true 时，编译会在外部进程中进行。fork 的详细情况依赖于所使用的编译器。基于 Ant 的编译器 (scalaCompileOptions.useAnt = true) 将为每个 ScalaCompile 任务 fork 一个新进程，而默认情况下它不进行 fork。基于 Zinc 的编译器 (scalaCompileOptions.useAnt = false) 将利用 Gradle 编译器守护进程，且默认情况下也是这样。

外部过程默认使用JVM 的的默认内存设置。如果要调整内存设置，请根据需要配置scalaCompileOptions.fork Options：

调整内存设置

build.gradle

```
tasks.withType(ScalaCompile) {
   configure(scalaCompileOptions.forkOptions) {
      memoryMaximumSize = '1g'
      jvmArgs = ['-XX:MaxPermSize=512m']
   }
}
```

# #

增量编译

增量编译是只编译那些源代码在上一次编译之后有修改的类，及那些受这些修改影响到的类，它可以大大减少 Scala 的编译时间。频繁编译代码的增量部分是非常有用的，因为在开发时我们经常要这样做。

Scala 插件现在通过集成 Zinc 来支持增量编译， 它是 sbt 增量 Scala 编译器的一个单机版本。若要把 ScalaCompile 任务从默认的基于 Ant 的编译器切换为新的基于 Zinc 的编译器，需要将 scalaCompileOptions.useAnt 设置为 false：

激活基于 Zinc 编译器

build.gradlev

```
tasks.withType(ScalaCompile) {
    scalaCompileOptions.useAnt = false
}
```

除非在 API 文档中另有说明，否则基于 Zinc 的据编译器支持与基于 Ant 的编译器完全相同的配置选项。但是，要注意的是，Zinc 编译器需要 Java 6 或其以上版本来运行。这意味着 Gradle 本身要使用 Java 6 或其以上版本。

Scala 插件添加了一个名为 zinc 的配置，以解析 Zinc 库及其依赖。如果要重写 Gradle 默认情况下使用的 Zinc 版本，请添加一个显式的 Zinc 依赖项（ 例如zinc "com.typesafe.zinc:zinc:0.1.4" ）。无论使用哪一个 Zinc 版本，Zinc 都是使用在scalaTools配置上找到的 Scala 编译器。

就像 Gradle 上基于Ant 的编译器一样，基于 Zinc 的编译器支持 Java 和 Scala 代码的联合编译。默认情况下，在 src/main/scala 下的所有 Java 和 Scala 代码都会进行联合编译。使用基于 Zinc 的编译器时，即使是 Java 代码也将会进行增量编译。

增量编译需要源代码的相关性分析。解析结果进入由 scalaCompileOptions.incrementalOptions.analysisFile 所指定的文件（ 它有一个合理的默认值 ）。在多项目构建中，分析文件被传递给下游的 ScalaCompile 任务，以启用跨项目的增量编译。对于由 Scala 插件添加的 ScalaCompile 任务，无需对这一点进行配置。对于其他的 ScalaCompile 任务，需要根据类文件夹或 Jar archive 的代码中，是哪一个的代码被传递给 ScalaCompile 任务的下游类路径，把ScalaCompileOptions.incrementalOptions.publishedCode 配置为指向它们。注意，如果 publishedCode 设置不正确，上游代码发生变化时下游任务可能不会重新编译代码，导致编译结果不正确。

由于依赖分析的系统开销，一次干净的编译或在代码有了较大的更改之后的编译，可能花费的时间要长于使用基于 Ant 的编译器。对于 CI 构建和版本的构建中，我们目前推荐使用基于 Ant 的编译器。

注意现在 Zinc 基于守护进程模式的 Nailgun 还不支持。相反，我们打算加强 Gradle 自己的编译器守护进程，使得在跨 Gradle 调用时继续存活，利用同一个 Scala 编译器。这将会为 Scala 编译带来另一个方面上的明显加速。

# #

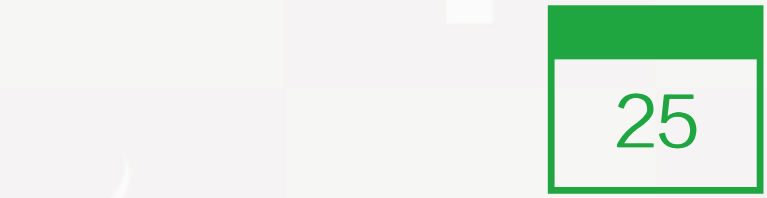
eclipse 集成

当 Eclipse 插件遇到 Scala 项目时，它将添加额外的配置，使得项目能够在使用 Scala IDE 时开箱即用。具体而言，该插件添加一个 Scala 性质和依赖的容器。

# #

IntelliJ 集成

当 IDEA 插件遇到 Scala 项目时，它将添加额外的配置，使得项目能够在使用 IDEA 时开箱即用。具体而言，该插件添加了一个 Scala facet 和一个匹配项目的类路径上的 Scala 版本的 Scala 编译器类库。



25

War 插件

War 的插件继承自 Java 插件并添加了对组装 web 应用程序的 WAR 文件的支持。它禁用了 Java 插件生成默认的 JAR archive，并添加了一个默认的 WAR archive 任务。

# #

用法

要使用 War 的插件，请在构建脚本中包含以下语句：

使用 War 插件

build.gradle

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

# #

任务

War 插件向 project 中添加了以下任务。

表 26.1. War 插件 – 任务

任务名称	依赖于	类型	描述
war	compile	War	组装应用程序 WAR 文件。

War 插件向 Java 插件所加入的 tasks 添加了以下的依赖。

表 26.2. War 插件 – 额外的 task 依赖

任务名称	依赖于
assemble	war

图 26.1. War 插件 – tasks

#

项目布局

表 26.3. War 插件 – 项目布局

目录	意义
from <s1>'src/main/webapp'</s1>	Web 应用程序源代码

#

依赖管理

War 插件添加了两个依赖配置： providedCompile 和 providedRuntime。虽然它们有各自的compile 和 runtime 配置，但这些配置有相同的作用域，只是它们不会添加到 WAR 文件中。要特别注意，这些 provided 配置的传递使用。假设你添加 commons-httpclient:commons-httpclient:3.0 依赖到任何一个 provided 配置。这个依赖又依赖于 commons-codec。这意味着 httpclient 和 commons-codec 都不会添加到你的 WAR 中，即使 commons-codec 是 compile 配置上的一个显示依赖。如果你不想要这种传递行为，只是把 provided 依赖声明成和commons-httpclient:commons-httpclient:3.0@jar 一样。

#

公约属性

表26.4. War 插件 – 目录属性

属性名称	类型	默认值	描述
webAppDirName	String	from <s1>'src/main/webapp'</s1>	web 应用程序源目录的名称，是一个相对于项目目录的目录名称。
webAppDir	File (read-only)	webAppDirName	Web 应用程序的源目录。

这些属性由一个 WarPluginConvention 公约对象提供。

# #

War

War task 的默认行为是将 src/main/webapp 的内容复制到 archive 的根目录下。你的 webapp 目录自然可能包含一个 WEB-INF 子目录，这个子目录可能还再包含一个 web.xml 文件。已编译的类被编译进 WEB-INF/classes。所有 runtime 配置的依赖被复制到 WEB-INF/lib。

#

自定义

下面是一个示例，展示了最重要的自定义选项：

war 插件的自定义

build.gradle

```
configurations {
  moreLibs
}
repositories {
  flatDir { dirs "lib" }
  mavenCentral()
}
dependencies {
   compile module(":compile:1.0") {
      dependency ":compile-transitive-1.0@jar"
      dependency ":providedCompile-transitive:1.0@jar"
   }
   providedCompile "javax.servlet:servlet-api:2.5"
   providedCompile module(":providedCompile:1.0") {
      dependency ":providedCompile-transitive:1.0@jar"
   }
   runtime ":runtime:1.0"
   providedRuntime ":providedRuntime:1.0@jar"
   testCompile 'junit:junit:4.11'
}
   moreLibs ":otherLib:1.0"
}
war {
   from 'src/rootContent' // adds a file-set to the root of the archive
   webInf { from 'src/additionalWebInf' } // adds a file-set to the WEB-INF dir.
   classpath fileTree('additionalLibs') // adds a file-set to the WEB-INF/lib dir.
   classpath configurations.moreLibs // adds a configuration to the WEB-INF/lib dir.
   webXml = file('src/someWeb.xml') // copies a file to WEB-INF/web.xml
}
```

当然，你可以用一个定义了 excludes 和 includes 的闭包来配置不同的文件集。

26

Ear 插件

Ear 插件添加了用于组装 web 应用程序的 EAR 文件的支持。它添加了一个默认的 EAR archive task。它不需要 Java 插件，但是对于使用了 Java 插件的项目，它将禁用默认的 JAR archive 的生成。

# #

用法

要使用 Ear 的插件，请在构建脚本中包含以下语句：

使用 Ear 插件

build.gradle

```
apply plugin: 'ear'
```

# #

Tasks

Ear 插件向 project 中添加了以下任务。

表 27.1. Ear 插件 - tasks

任务名称	依赖于	类型	描述
ear	compile（仅在也配置了使用 Java 插件的时候）	ear	组装应用程序 EAR 文件。

Ear 插件向基础插件所加入的 tasks 添加了以下的依赖。

表 27.2. Ear 插件 - 额外的 task 依赖

任务名称	依赖于
assemble	ear

#

项目布局

表 27.3. Ear 插件 – 项目布局

目录	意义
src/main/application	Ear 资源，如 META-INF 目录

# #

依赖管理

Ear 插件添加了两个依赖配置：deploy和earlib。所有在 deploy 配置中的依赖项都放在 EAR 文件的根目录中，并且是不可传递的。所有在 earlib 配置的依赖都放在 EAR 文件的“lib”目录中，并且是可传递的。

#

公约属性

表27.4. Ear 插件 – 目录属性

属性名称	类型	默认值	描述
appDirName	String	src/main/application	相对于项目目录的应用程序源目录名称。
libDirName	String	into(<s2>'libs'</s2>) {	生成的 EAR 文件里的 lib 目录名称。
deploymentDescriptor	org.gradle.plugins.ear.descriptor.DeploymentDescriptor	部署描述符，它有一个合理的名为 application.xml 的默认值	用于生成部署描述符文件的元数据，例如 ear.deploymentDescriptor 中的显式配置将被忽略。

这些属性由一个 EarPluginConvention 公约对象提供。

# #

Ear

Ear task 的默认行为是将 src/main/application 的内容复制到 archive 的根目录下。如果你的 application 目录没有包含 META-INF/application.xml 部署描述符，那么将会为你生成一个。

另请参阅 Ear。

#

自定义

下面是一个示例，展示了最重要的自定义选项：

ear 插件的自定义

build.gradle

```
apply plugin: 'ear'
apply plugin: 'java'
repositories { mavenCentral() }
dependencies {
   //following dependencies will become the ear modules and placed in the ear root
   deploy project(':war')
   //following dependencies will become ear libs and placed in a dir configured via libDirName property
   earlib group: 'log4j', name: 'log4j', version: '1.2.15', ext: 'jar'
}
ear {
   appDirName 'src/main/app'  // use application metadata found in this folder
   libDirName 'APP-INF/lib'  // put dependency libraries into APP-INF/lib inside the generated EAR;
                             // also modify the generated deployment descriptor accordingly
   deploymentDescriptor {  // custom entries for application.xml:
//     fileName = "application.xml"  // same as the default value
//     version = "6"  // same as the default value
       applicationName = "customear"
       initializeInOrder = true
       displayName = "Custom Ear"  // defaults to project.name
       description = "My customized EAR for the Gradle documentation"  // defaults to project.description
//     libraryDirectory = "APP-INF/lib"  // not needed, because setting libDirName above did this for us
//     module("my.jar", "java")  // wouldn't deploy since my.jar isn't a deploy dependency
//     webModule("my.war", "/")  // wouldn't deploy since my.war isn't a deploy dependency
       securityRole "admin"
       securityRole "superadmin"
       withXml { provider -> // add a custom node to the XML
          provider.asNode().appendNode("data-source", "my/data/source")
       }
   }
}
```

你还可以使用 Ear 任务提供的自定义选项，如 from 和 metaInf。

#

使用自定义的描述符文件

假设你已经有了 application.xml，并且想要使用它而不是去配置 ear.deploymentDescriptor 代码段。去把 META-INF/application.xml 放在你的源文件夹里的正确的位置（请查看 appDirName 属性）。这个已存在的文件的内容将会被使用，而 ear.deploymentDescriptor 里的显示配置则会被忽略。

27

Jetty 插件

Jetty 插件继承自 War 插件，并添加一些任务，这些任务可以让你在构建时部署你的 web 应用程序到一个 Jetty 的 web 嵌入式容器中。

# #

用法

要使用 Jetty 的插件，请在构建脚本中包含以下语句：

使用 Jetty 插件

build.gradle

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

# #

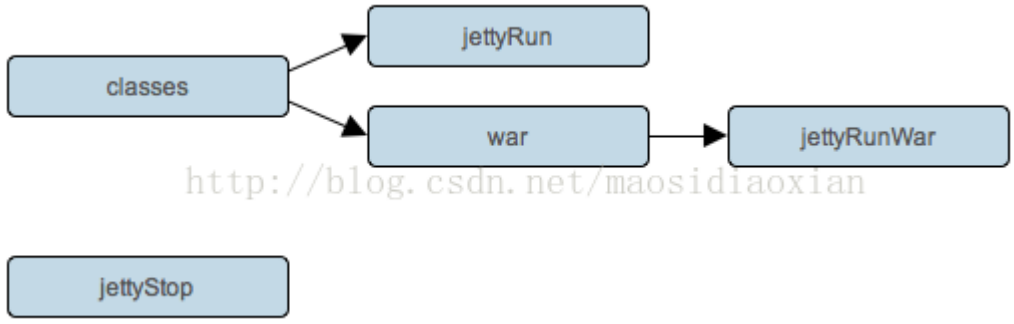
任务

Jetty 插件定义了以下任务：

表 28.1. Jetty 插件 – 任务

任务名称	依赖于	类型	描述
jettyRun	compile	jettyRun	启动 Jetty 实例并将部署上 exploded web 应用程序。
jettyRunWar	war	jettyRunWar	启动 Jetty 实例并将部署上 WAR 包。
jettyStop	–	jettyStop	停止 Jetty 实例。

图 28.1. Jetty 插件 – tasks



#

项目布局

Jetty 插件使用 和 War 插件相同的布局。

# #

依赖管理

Jetty 插件并不定义任何依赖配置。

# #

公约属性

Jetty 插件定义了下列公约属性：

表 28.2. Jetty 插件 – 属性

属性名称	类型	默认值	描述
contextPath	String	WAR 文件的base name	在 Jetty 容器里面的应用程序部署位置。
httpPort	Integer	8080	Jetty 监听 HTTP 请求的 TCP 端口。
stopPort	Integer	null	Jetty 监听 admin 请求的 TCP 端口。
stopKey	String	null	当需要请求停止时，传递给 Jetty 的key。

这些属性都由一个 JettyPluginConvention 公约对象提供。

28

Checkstyle 插件

Checkstyle 插件使用 Checkstyle 对你的项目的 Java 源文件执行质量检查，并从检查结果中生成报告。

# #

用法

要使用 Checkstyle 插件，请在构建脚本中包含以下语句：

使用 Checkstyle 插件

build.gradle

```
apply plugin: 'checkstyle'
```

该插件向你的项目添加了大量的执行质量检查的任务。你可以通过运行 gradle check 执行检查。

#

Tasks

Checkstyle 插件向 project 中添加了以下 tasks：

表 29.1. Checkstyle 插件 – tasks

任务名称	依赖于	类型	描述
checkstyleMain	classes	checkstyle	针对生产Java 源文件运行 Checkstyle。
checkstyleTest	testClasses	checkstyle	针对测试 Java 源文件运行 Checkstyle。
SourceSet	sourceSet Classes	checkstyle	针对source set 的 Java 源文件运行 Checkstyle。

Checkstyle 插件向 Java 插件所加入的 tasks 添加了以下的依赖。

表 29.2. Checkstyle 插件 – 额外的 task 依赖

任务名称	依赖于
check	所有 Checkstyle tasks，包括 checkstyleTest 。

#

项目布局

Checkstyle 插件预计是以下的项目布局：

表 29.3. Checkstyle 插件 – 项目布局

File	意义
config/checkstyle/checkstyle.xml	Checkstyle 配置文件

# #

依赖管理

Checkstyle 插件添加了下列的依赖配置：

表29.4. Checkstyle 插件 – 依赖配置

名称	意义
`checkstyle`	用到的 Checkstyle 库

29

CodeNarc 插件

CodeNarc 插件使用 CodeNarc 对项目的 Groovy 源文件执行质量检查并生成报告。

# #

用法

要使用 CodeNarc 插件，请在构建脚本中包含以下语句：

使用 CodeNarc 插件

build.gradle

```
apply plugin: 'codenarc'
```

该插件向你的项目添加了大量的执行质量检查的任务。你可以通过运行 gradle check 执行检查。

# #

任务

CodeNarc 插件向project 中添加了以下任务：

表 30.1. CodeNarc 插件 – 任务

任务名称	依赖于	类型	描述
codenarcMain	–	codenarc	针对生产 Groovy 源文件运行 CodeNarc。
codenarcTest	–	codenarc	针对测试 Groovy 源文件运行 CodeNarc。
SourceSet	–	codenarc	针对给定的source set 的 Groovy 源文件运行 CodeNarc。

CodeNarc 插件向 Groovy 插件所加入的任务添加了以下的依赖。

表 30.2. CodeNarc 插件 – 附加的任务依赖

任务名称	依赖于
check	所有的 CodeNarc 任务，包括 codenarcTest 。

# #

项目布局

CodeNarc 插件预计是以下的项目布局：

表 30.3. CodeNarc 插件 – 项目布局

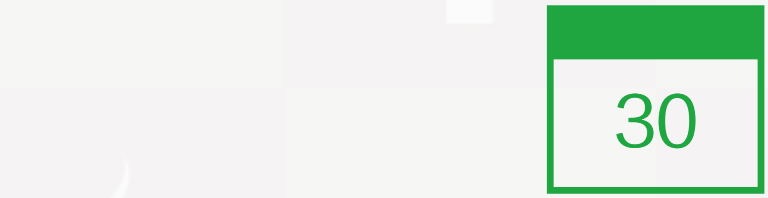
File	意义
config/codenarc/codenarc.xml	CodeNarc 配置文件

#

依赖管理 CodeNarc 插件添加了下列的依赖配置：

表30.4. CodeNarc 插件 – 依赖配置

名称	意义
codenarc	使用的 CodeNarc 库



30

FindBugs 插件

FindBugs 插件使用 FindBugs 对项目的 Java 源文件执行质量检查，并从检查结果中生成报告。

# #

用法

要使用 FindBugs 插件，请在构建脚本中包含以下语句：

使用 FindBugs 插件

build.gradle

```
apply plugin: 'findbugs'
```

该插件向你的项目添加了大量的执行质量检查的任务。你可以通过运行 gradle check 执行检查。

# #

任务

FindBugs 插件向 project 中添加了以下任务：

表 31.1. FindBugs 插件 – 任务

任务名称	依赖于	类型	描述
findbugsMain	classes	findbugs	针对生产Java 源文件运行 FindBugs。
findbugsTest	testClasses	findbugs	针对测试 Java 源文件运行 FindBugs。
SourceSet	sourceSet Classes	findbugs	针对source set 的 Java 源文件运行 FindBugs。

FindBugs 插件向 Java 插件所加入的任务添加了以下的依赖。

表 31.2. FindBugs 插件 – 附加的任务依赖

任务名称	依赖于
check	所有 FindBugs 任务，包括 findbugsTest 。

# #

依赖管理

FindBugs 插件增加了下列的依赖配置：

表 31.3. FindBugs 插件 – 依赖配置

名称	意义
findbugs	使用的 FindBugs 库

31

JDepend 插件

JDepend 插件使用 JDepend 对项目的源文件执行质量检查，并从检查结果中生成报告。

#

用法

要使用 JDepend 插件，请在构建脚本中包含以下语句：

使用 JDepend 插件

build.gradle

```
apply plugin: 'jdepend'
```

该插件向你的项目添加了大量的执行质量检查的任务。你可以通过运行 gradle check 执行检查。

# #

任务

JDepend 插件向 project 中添加了以下任务：

表 32.1. JDepend 插件 – 任务

任务名称	依赖于	类型	描述
jdependMain	classes	jdepend	针对生产Java 源文件运行 JDepend。
jdependTest	testClasses	jdepend	针对测试Java 源文件运行 JDepend。
SourceSet	sourceSet Classes	jdepend	针对source set 的 Java 源文件运行 JDepend。

JDepend 插件向 Java 插件所加入的任务添加了以下的依赖。

表 32.2. JDepend 插件 – 附加的任务依赖

任务名称	依赖于
check	所有 JDepend 任务，包括 jdependTest 。

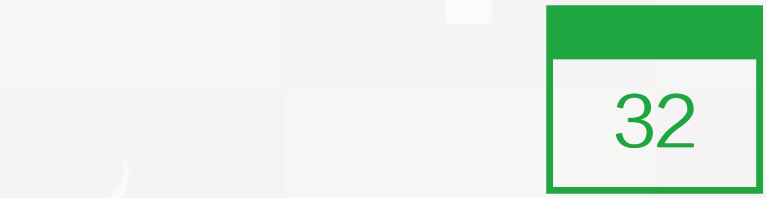
# #

依赖管理

JDepend 插件添加了下列的依赖配置：

表32.3. JDepend 插件 – 依赖配置

名称	意义
jdepend	使用的 JDepend 库



32

PMD 插件

PMD 插件使用 PMD 对项目的 Java 源文件执行质量检查，并从检查结果中生成报告。

# #

用法

要使用 PMD 插件，请在构建脚本中包含以下语句：

使用 PMD 插件

build.gradle

```
apply plugin: 'pmd'
```

该插件向你的项目添加了大量的执行质量检查的任务。你可以通过运行 gradle check 执行检查。

# #

任务

PMD 插件向 project 中添加了以下任务：

表 33.1. PMD 插件 – 任务

任务名称	依赖于	类型	描述
`pmdMain`	–	pmd	针对生产Java 源文件运行 PMD。
`pmdTest`	–	pmd	针对测试 Java 源文件运行 PMD。
`SourceSet`	–	pmd	针对source set 的 Java 源文件运行 PMD。

PMD 插件向 Java 插件所加入的任务添加了以下的依赖。

表 33.2. PMD 插件 – 附加的任务依赖

任务名称	依赖于
`check`	所有的 PMD 任务，包括 `pmdTest` 。

# #

依赖管理

PMD 插件添加了下列的依赖配置：

表33.3. PMD 插件 – 依赖配置

名称	意义
`pmd`	使用的 PMD 库

33

JaCoCo 插件

JaCoCo 插件目前还是孵化中状态。请务必注意，在以后的 Gradle 版本中，DSL 和其他配置可能会有所改变。

JaCoCo 插件通过集成 JaCoCo为 Java 代码提供了代码覆盖率指标。

#

入门

要想开始，请将 JaCoCo 插件应用于你想要计算代码覆盖率的项目中。

应用 JaCoCo 插件

build.gradle

```
apply plugin: "jacoco"
```

如果 Java 插件也被应用于你的项目，那么会创建一个名为 jacocoTestReport 的新任务，该新任务依赖于 test 任务。该报告可以在 $buildDir/reports/jacoco/test 中看到。默认情况下，会生成一个 HTML 报告。

# #

配置 JaCoCo 插件

JaCoCo 插件添加一个名为 jacoco 类型为 JacocoPluginExtension 的 project 扩展，这个扩展允许在你的构建中配置 JaCoCo 所使用的默认值。

配置 JaCoCo 插件设置

build.gradle

```
jacoco
    toolVersion = "0.6.2.201302030002"
    reportsDir = file("$buildDir/customJacocoReportDir")
}
```

表 34.1. JaCoCo 属性的 Gradle 默认值

Property	Gradle 默认值
reportsDir	"*$buildDir*/reports/jacoco"

#

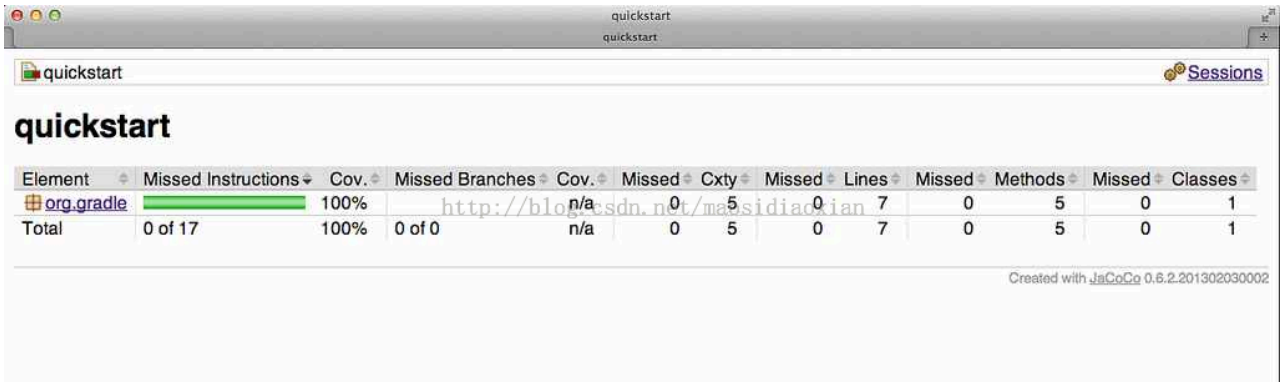
JaCoCo 报告配置

JacocoReport 任务可以用于生成不同格式的代码覆盖率报告。它实现了标准的 Gradle 类型 Reporting，并呈现了一个 JacocoReportsContainer 类型的报告容器。

配置测试任务

build.gradle

```
jacocoTestReport {
    reports
        xml.enabled false
        csv.enabled false
        html.destination "${buildDir}/jacocoHtml"
    }
}
```



#

JaCoCo 的特定任务配置

JaCoCo 插件添加了一个 JacocoTaskExtension 扩展到 Test 类型的所有任务中。该扩展允许配置 JaCoCo 中的测试任务的一些特定属性。

配置测试任务

build.gradle

```
test {
    jacoco
        append = false
        destinationFile = file("$buildDir/jacoco/jacocoTest.exec")
        classDumpFile = file("$buildDir/jacoco/classpathdumps")
    }
}
```

表 34.2. JaCoCo 任务扩展的默认值

Property	Gradle 默认值
enabled	true
destPath	*$buildDir*/jacoco
append	true
includes	[]
excludes	[]
excludeClassLoaders	[]
sessionId	auto-generated
dumpOnExit	true
output	Output.FILE
address	–
port	–
classDumpPath	–
jmx	false

虽然 Test 的所有任务会在 java 插件被配置使用时会自动增强以提供覆盖率信息，但是任何实现了 JavaForkOptions 的任务都可以通过 JaCoCo 插件得到增强。也就意味着，任何 fork Java 进程的任务都可以用于生成覆盖率信息。

例如，你可以配置您的构建使用 application 插件来生成代码覆盖率。

使用 application 插件来生成代码覆盖率数据

build.gradle

```
apply plugin: "application"
apply plugin: "jacoco"
mainClassName = "org.gradle.MyMain"
jacoco {
    applyTo run
}
task applicationCodeCoverageReport(type:JacocoReport){
    executionData run
    sourceSets sourceSets.main
}
```

注: 此示例中的代码可以在 Gradle 的二进制分发包及源代码分发包中的 samples/testing/jacoco/application 中找到。

由 applicationCodeCoverageReport 生成的覆盖率报告

构建布局

```
application
  build
    jacoco
      run.exec
    reports/jacoco/applicationCodeCoverageReport/html/
      index.html
```

# #

任务

对于同时也配置使用了 Java 插件的项目，JaCoCo 插件会自动添加以下任务：

表 34.3. JaCoCo 插件 – 任务

任务名称	依赖于	类型	描述
jacocoTestReport	–	JacocoReport	为测试任务生成代码覆盖率报告。

# #

依赖管理

JaCoCo 插件添加了下列的依赖配置：

表 34.4. JaCoCo 插件 – 依赖配置

名称	意义
jacocoAnt	用于运行 JacocoMerge 任务的 JaCoCo Ant 库。
jacocoAgent	用于测试位于test下的代码的 JaCoCo 客户端库。

34

Sonar 插件

你可能会想使用新的 Sonar Runner 插件来代替现在这个插件。尤其是因为只有 Sonar Runner 插件支持 Sonar 3.4 及更高的版本。

Sonar 插件提供了对 Sonar，一个基于 web 的代码质量监测平台的集成。该插件添加了sonarAnalyze task，用来分析一个 project 及子 project 都应用了哪个插件。分析结果存储于 Sonar 数据库中。该插件基于 Sonar Runner，并要求是 Sonar 2.11 或更高的版本。

SonarAnalyze task 是一项需要显式执行的独立任务，不依赖于任何其他 task。除了源代码之外，该 task 还分析了类文件和测试结果文件（如果有）。为获得最佳结果，建议在分析前运行一次完整的构建。在典型的设置中，会每天在构建服务器上运行一次分析。

# #

用法

最低要求是必须配置 Sonar 插件应用于该 project。

配置使用 Sonar 插件

build.gradle

```
apply plugin: "sonar"
```

除非 Sonar 是在本地上运行，并且有默认的配置，否则有必要配置 Sonar 服务器及数据库的连接设置。

配置 Sonar 连接设置

build.gradle

```
sonar
   server {
      url = "http://my.server.com"
   }
   database {
      url = "jdbc:mysql://my.server.com/sonar"
      driverClassName = "com.mysql.jdbc.Driver"
      username = "Fred Flintstone"
      password = "very clever"
   }
}
```

或者，可以从命令行设置某些或全部的连接设置。

Project 设置会决定这个项目将如何进行分析。默认配置非常适合于分析标准 Java 项目，并可以在许多方面进行自定义。

配置 Sonar project 设置

build.gradle

```
sonar
   project
      coberturaReportPath = file("$buildDir/cobertura.xml")
   }
}
```

在上面的例子中，sonar，server，database 和 project 块分别配置的是SonarRootModel， SonarServer， SonarDatabase 及 SonarProject 类型的对象。可以查阅它们的 API 文档以了解更多信息。

# #

分析多项目构建

Sonar 插件能够一次分析整个项目的层次结构。它能够在 Sonar 的 web 界面生成一个层次图，该层次图包含了综合的指标且能够深入到子项目中。同时，它比单独分析每个项目更快。

要分析项目的层次结构， 需要把 Sonar 插件应用于层次结构的最顶层项目。通常（但不是一定）会是根项目。在该 project 中的 sonar 块配置的是一个 SonarRootModel 类型的对象。它拥有所有全局配置，最重要的服务器和数据库的连接设置。

在多项目构建中的全局配置

build.gradle

```
apply plugin: "sonar"
sonar {
    server {
        url = "http://my.server.com"
    }
    database {
        url = "jdbc:mysql://my.server.com/sonar"
        driverClassName = "com.mysql.jdbc.Driver"
        username = "Fred Flintstone"
        password = "very clever"
    }
}
```

层次结构中的每个项目都有其自身的项目配置。共同的值可以在父构建脚本中进行设置。

多项目构建中的共同项目配置

build.gradle

```
subprojects {
    sonar
        project
            sourceEncoding = "UTF-8"
        }
    }
}
```

在子项目中的 sonar 块配置的是一个 SonarProjectModel 类型的对象。

这些 Projects 也可以单独配置。例如，设置 skip 属性为 true 以防止一个项目（和它的子项目）被分析。跳过的项目将不会显示在 Sonar 的 web 界面中。

多项目构建中的单独项目配置

build.gradle

```
project
   sonar
      project
         skip = true
      }
   }
}
```

另一种典型的各个项目配置是配置要分析的编程语言。注意，Sonar 只能分析每个项目的一种语言。

配置语言分析

build.gradle

```
project
   sonar
      project
         language = "groovy"
      }
   }
}
```

当一次只设置一个属性时，等效属性的语法更加简洁：

使用属性语法

build.gradle

```
project(":project2").sonar.project.language = "groovy"
```

# #

分析自定义的 Source Sets

默认情况下，Sonar 插件将分析 main source set 里的生产源文件，以及 test source sets 里的测试源文件。它的分析独立于项目的源目录布局。根据需要，可以添加额外的 source sets。

分析自定义的 Source Sets

build.gradle

```
sonar.project {
    sourceDirs += sourceSets.custom.allSource.srcDirs
    testDirs += sourceSets.integTest.allSource.srcDirs
}
```

#

分析非 Java 语言

要分析非 Java 语言编写的代码，请安装相应的 Sonar 插件，并相应地设置 sonar.project.language：

分析非 Java 语言

build.gradle

```
sonar.project {
    language = "grvy" // set language to Groovy
}
```

截至 Sonar 3.4，每个项目只可以分析一种语言。不过，在多项目构建中你可以为不同的项目设置不同的语言。

#

设置自定义的 Sonar 属性

最终，大多数配置都会以被称为 Sonar 属性的键-值对的形式传递给 Sonar 的代码分析器。在 API 文档中的 SonarProperty 注解显示了插件的对象模型的属性是如何映射到相应的 Sonar 属性中的。Sonar 插件提供了 hooks，用于 Sonar 属性传给代码分析器前的后置处理。相同的 hook 可以用来添加额外的属性，并且不会被插件的对象模型所覆盖。

对于全局的 Sonar 属性，可以使用 SonarRootModel 上的 withGlobalProperties hook：

设置自定义的全局属性

build.gradle

```
sonar.withGlobalProperties { props ->
    props["some.global.property"] = "some value"
    // non-String values are automatically converted to Strings
    props["other.global.property"] = ["foo", "bar", "baz"]
}
```

对于每个项目的 Sonar 属性，使用 SonarProject 上的 withProjectProperties hook：

设置自定义的项目属性

build.gradle

```
sonar.project.withProjectProperties { props ->
    props["some.project.property"] = "some value"
    // non-String values are automatically converted to Strings
    props["other.global.property"] = ["foo", "bar", "baz"]
}
```

Sonar 的可用属性的列表可以在 Sonar 文档中找到。注意，对于大多数的这些属性，Sonar 插件的对象模型具有等效的属性，且没有必要使用 withGlobalProperties 或withProjectProperties 的 hook。对于第三方 Sonar 插件的配置，请参阅插件的文档。

#

从命令行配置 Sonar 的设置

下面的属性或者可以从命令行中或者是作为 sonarAnalyze 任务的任务参数这两种方式之一来设置。任务参数将覆盖任何在构建脚本中设置的相应值。

- server.url
- database.url
- database.driverClassName
- database.username
- database.password
- showSql
- showSqlResults
- verbose
- forceAnalysis

下面是一个完整的例子：

```
gradle sonarAnalyze --server.url=http://sonar.mycompany.com --database.password=myPassword --verbose
```

如果你需要从命令行设置其他属性，你可以使用系统属性来做：

实现自定义命令行属性

build.gradle

```
sonar.project {
    language = System.getProperty("sonar.language", "java")
}
```

然而，请记住，通常最好是配置在构建脚本中，并在代码控制下。

# #

任务

Sonar 插件向 project 中添加了以下任务。

表 35.1. 声纳插件 – 任务

任务名称	依赖于	类型	描述
sonarAnalyze	–	sonarAnalyze	分析项目层次结构，并将结果存储在 Sonar 数据库。

35

Sonar Runner 插件

Sonar runner 插件是目前仍是孵化状态。请务必注意，在以后的 Gradle 版本中，DSL 和其他配置可能会有所改变。

Sonar Runner 插件提供了对 Sonar，一个基于 web 的代码质量监测平台的集成。它基于 Sonar Runner，一个分析源代码及构建输出，并将所有收集的信息储存在 Sonar 数据库的 Sonar 客户端组件。相比单独使用 Sonar Runner，Sonar Runner 插件提供了以下便利：

自动配置 Sonar Runner

可以通过一个正规的 Gradle 任务来执行 Sonar Runner，这使得在任何 Gradle 可用的地方，它都可以用（开发人员构建，CI 服务器等），而无需下载，安装，和维护 Sonar Runner 的安装。

通过 Gradle 构建脚本动态配置

根据需要，可以利用 Gradle 脚本的所有特性去配置 Sonar Runner。

提供了广泛范围的默认配置

Gradle 已经有很多 Sonar Runner 成功分析一个项目所需的信息。基于这些信息对 Sonar Runner 进行预配置，减少了许多手动配置的需要。

# #

插件状态和兼容性

Sonar Runner 插件是 Sonar 插件的继任者。目前它还在孵化中的状态。该插件基于 Sonar Runner 2.0，这使它与 Sonar 2.11 或更高的版本相兼容。不同于 Sonar 插件，Sonar Runner 插件与 Sonar 3.4 或更高的版本一起使用时也表现正常。

# #

入门

若要开始，请对要分析的项目配置使用 Sonar Runner 插件。

配置使用 Sonar Runner 插件

build.gradle

```
apply plugin: "sonar-runner"
```

假设一个本地的 Sonar 服务使用开箱即用的设置启动和运行，则不需要进一步的强制性的配置。执行 gradle sonarRunner 并等待构建完成，然后打开 Sonar Runner 输出结果的底部所指示的网页。你现在应该能够看到分析结果了。

在执行 sonarRunner 任务前，所有产生输出以用于 Sonar 分析的需要都需要被执行。通常情况下，它们是编译任务、测试任务和代码覆盖任务。为了满足这些需要，如果应用了 java 插件，Sonar Runner 插件将从 sonarRunner 添加一个对 test 的任务依赖。根据需要，可以添加更多的任务依赖。

#

配置 Sonar Runner

Sonar Runner 插件向 project 添加了一个 SonarRunner 扩展，它允许通过被称为 Sonar 属性 的键/值对配置 Sonar Runner。一个典型的基线配置包括了 Sonar 服务器和数据库的连接设置。

配置 Sonar 连接设置

build.gradle

```
sonarRunner {
  sonarProperties {
    property "sonar.host.url", "http://my.server.com"
    property "sonar.jdbc.url", "jdbc:mysql://my.server.com/sonar"
    property "sonar.jdbc.driverClassName", "com.mysql.jdbc.Driver"
    property "sonar.jdbc.username", "Fred Flintstone"
    property "sonar.jdbc.password", "very clever"
  }
}
```

对于标准的 Sonar 属性的完整列表，请参阅 Sonar 文档。如果你碰巧使用另外的 Sonar 插件，请参考它们的文档。

或者，可以从命令行设置 Sonar 属性。有关更多信息，请参见第35.6节，“从命令行配置 Sonar 设置” 。

Sonar Runner 插件利用 Gradle 的对象模型所包含的信息，提供了许多标准的 Sonar 属性的智能默认值。下表总结了这些默认值。注意，对于配置使用了 java-base 或 java 插件的project，有提供另外的默认值。对于一些属性（尤其是服务器和数据库的连接配置），确定留给 Sonar Runner 一个合适的默认值。

表 36.1. 标准 Sonar 属性的 Gradle 默认值

Property	Gradle 默认值
sonar.projectKey	"$project.group:$project.name"（所分析的层次结构的根项目，否则留给 Sonar Runner 处理）
sonar.projectName	project.name
sonar.projectDescription	project.description
sonar.projectVersion	project.version
sonar.projectBaseDir	project.projectDir

Property	Gradle 默认值
sonar.working.directory	"$project.buildDir/sonar"
sonar.dynamicAnalysis	"reuseReports"

表 36.2. 配置使用 java-base 插件时另外添加的默认值

Property	Gradle 默认值
sonar.java.source	project.sourceCompatibility
sonar.java.target	project.targetCompatibility

表 36.2. 配置使用 java 插件时另外添加的默认值

Property	Gradle 默认值
sonar.sources	sourceSets.main.allSource.srcDirs（过滤为只包含存在的目录）
sonar.tests	sourceSets.test.allSource.srcDirs（过滤为只包含存在的目录）
sonar.binaries	sourceSets.main.runtimeClasspath（过滤为只包含存在的目录）
sonar.libraries	sourceSets.main.runtimeClasspath（过滤为仅包括文件； 如果有必要会加上 `rt.jar` ）
sonar.surefire.reportsPath	test.testResultsDir（如果该目录存在）
sonar.junit.reportsPath	test.testResultsDir（如果该目录存在）

#

分析多项目构建

Sonar Runner 插件能够一次分析整个项目的层次结构。它能够在 Sonar 的 web 界面生成一个层次图，该层次图包含了综合的指标且能够深入到子项目中。分析一个项目的层次结果还可以比单独分析每个项目花费更省时间。

要分析一个项目的层次结构， 需要把 Sonar Runner 插件应用于层次结构的最顶层项目。通常（但不是一定）会是这个 Gradle 构建的根项目。与分析有关的信息作为一个整体，比如服务器和数据库的连接设置，必须在这一个 project 的 sonarRunner 块中进行配置。在命令行上设置的任何 Sonar 属性也会应用到这个 project 中。

全局配置设置

build.gradle

```
sonarRunner {
    sonarProperties {
        property "sonar.host.url", "http://my.server.com"
        property "sonar.jdbc.url", "jdbc:mysql://my.server.com/sonar"
        property "sonar.jdbc.driverClassName", "com.mysql.jdbc.Driver"
        property "sonar.jdbc.username", "Fred Flintstone"
        property "sonar.jdbc.password", "very clever"
    }
}
```

在 subprojects 块中，可以配置共享子项目之间的配置。

共享的配置设置

build.gradle

```
subprojects {
    sonarRunner {
        sonarProperties {
            property "sonar.sourceEncoding", "UTF-8"
        }
    }
}
```

特定项目的信息在对应的 project 的 sonarRunner 块中配置。

个别配置设置

build.gradle

```
project
   sonarRunner {
      sonarProperties {
         property "sonar.language", "grvy"
      }
   }
}
```

对于一个特定的子项目，要跳过 Sonar 分析，可以设置 sonarRunner.skipProject。

跳过项目分析

build.gradle

```
project
   sonarRunner {
      skipProject = true
   }
}
```

# #

分析自定义的 Source Sets

默认情况下， Sonar Runner 插件传给 project 的 main source set 将作为生产源文件，传给 project 的 test source sets 将作为测试源文件。这个过程与 project 的源目录布局无关。根据需要，可以添加额外的 source sets。

分析自定义的Source Sets

build.gradle

```
sonarRunner {
    sonarProperties {
        properties["sonar.sources"] += sourceSets.custom.allSource.srcDirs
        properties["sonar.tests"] += sourceSets.integTest.allSource.srcDirs
    }
}
```

#

分析非 Java 语言

要分析非 Java 语言编写的代码，请安装相应的 Sonar 插件，并相应地设置 sonar.project.language：

分析非 Java 语言

build.gradle

```
sonarRunner {
  sonarProperties {
    property "sonar.language", "grvy" // set language to Groovy
  }
}
```

截至 Sonar 3.4，每个项目只可以分析一种语言。不过，在多项目构建中你可以为每一个项目分析一种不同的语言。

# #

更多关于配置 Sonar 的属性

让我们再详细看看 sonarRunner.sonarProperties {}块。正如我们在示例中已经看到的， property()方法允许设置新属性或重写现有的属性。此外，所有已配置到这一点的属性，包括通过 Gradle 预配置的所有属性，还可通过 properties 访问器进行使用。

在 properties map 的条目可以使用常见的 Groovy 语法来读取和写入。为了方便它们的操作，这些值值仍然使用它们惯用的类型（ File，List等 ）。SonarProperties 块在经过评估后，这些值值被转换为字符串，如下所示： 集合的值（ 递归 ）转换为以逗号分隔的字符串，其他所有的值通过调用其tostring ()方法进行转换。

因为 sonarProperties 块的评估是惰性的，Gradle 的对象模型的属性可以在块中被安全地引用，而无需担心它们还没有被赋值。

# #

从命令行设置 Sonar 属性

Sonar 属性也可以从命令行中设置，通过设置一个系统属性，名称就像正在考虑中的 Sonar 属性。当处理敏感信息（例如证件），环境信息，或点对点配置时，这会非常有用。

```
gradle sonarRunner -Dsonar.host.url=http://sonar.mycompany.com -Dsonar.jdbc.password=myPassword -Dsonar.verb
```

虽然有时当然很有用，但我们建议在（版本控制的）构建脚本中，能够方便地让每个人都保持大部分的配置。

通过一个系统属性设置的 Sonar 属性值将覆盖构建脚本中设置的任何值（同样的属性名称）。当分析项目的层次结构时，通过系统属性设置的值应用于所分析层次结构的根项目。

#

在一个单独的进程中执行 Sonar Runner

根据项目大小，Sonar Runner 可能需要大量的内存。由于这个和其他（主要是隔离）的原因，最好在一个独立的进程中执行 Sonar Runner。一旦 Sonar Runner 2.1 发布，将提供这个功能，并由 Sonar Runner 插件采用。到那时，Sonar Runner 会在 Gradle 主进程中执行。

# #

任务

Sonar Runner 插件向 project 中添加了以下任务。

表 36.4. Sonnar Runner 插件 – 任务

任务名称	依赖于	类型	描述
sonarRunner {	–	sonarRunner {	分析项目层次结构，并将结果存储在 Sonar 数据库。

36

OSGi 插件

OSGi 插件提供了工厂方法来创建一个 OsgiManifest 对象。OsgiManifest 继承自 Manifest。如果应用了 Java 插件，OSGi 插件将把默认 jar 的 manifest 对象替换为一个 OsgiManifest 对象。被替换的 manifest 会被合并到新的对象单中。

OSGi 插件使 Peter Kriens BND tool 大量使用。

# #

用法

要使用 OSGi 插件，请在构建脚本中包含以下语句：

** 使用 OSGi 插件**

build.gradle

```
apply plugin: 'osgi'
```

# #

隐式应用插件

适用于 Java 基础插件。

#

任务

此插件不会添加任何任务。

# #

依赖管理

待决定

#

约定对象

OSGi 插件添加了下列约定对象： OsgiPluginConvention

#

约定属性

OSGi 插件没有向 project 添加任何的公约属性。

#

约定方法

OSGi 插件添加了以下方法。有关更多详细信息，请参见约定对象的 API 文档。

表 37.1. OSGi 方法

方法	返回类型	描述
osgiManifest()	OsgiManifest	返回一个 OsgiManifest 对象。
osgiManifest(Closure cl)	OsgiManifest	返回一个通过闭包配置的 OsgiManifest 对象。

在 classes 目录下的类文件会被分析出关于它们的包的依赖，以及它们所公布的包名。并基于此计算 OSGi Manifest 中 Import-Package 和 Export-Package 的值。如果 classpath 中包含了 jar 包和 OSGi bundle，bundle 信息会被用来指定 Import-Package 的值的版本信息。在 OsgiManifest 对象的显式属性旁边，你可以添加 instructions。

OSGi MANIFEST.MF 文件配置

build.gradle

```
jar {
   manifest { // the manifest of the default jar is of type OsgiManifest
      name = 'overwrittenSpecialOsgiName'
      instruction 'Private-Package',
            'org.mycomp.package1',
            'org.mycomp.package2'
```

```
        instruction 'Bundle-Vendor', 'MyCompany'
        instruction 'Bundle-Description', 'Platform2: Metrics 2 Measures Framework'
        instruction 'Bundle-DocURL', 'http://www.mycompany.com'
    }
}
task fooJar(type: Jar) {
    manifest = osgiManifest {
        ~instruction 'Bundle-Vendor', 'MyCompany'
    }
}
```

instruction 调用的第一个参数是属性的键。其他参数构成了它的值。他们由 Gradle 使用,分隔符连接。要了解更多关于 instructions 的信息，可以看看 BND tool。

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台

更多信息请访问

http://wiki.jikexueyuan.com/project/gradle/