Gradle

官方文档 中文翻译

Table of Contents

Gradle User Guide 中文版

- Gradle User Guide 中文版 正在翻译当中 欢迎大家一起加入 因为水平有限 也请大家指正翻译错误的地方
- https://github.com/DONGChuan/GradleUserGuide Github地址
- http://www.gradle.org/docs/current/userguide/userguide_single.html#N1012C 原文地址
- 我会开放权限给每一个加入的伙伴，请提前邮箱联系 dongchuan55@gmail.com

概述

- 特点
- 为什么用 Groovy?

特点

这里简述下 Gradle 的特点.

1. 声明式构建和合约构建

Gradle 的核心是基于 Groovy 的 领域特定语言 (DSL), 具有十分优秀的扩展性. Gradle 通过提供可以随意集成的声明式语言元素将声明性构建推到了一个新的高度. 这些元素还提供了对 Java, Groovy, OSGi, 网络和Scala 等项目的支持. 而且, 基于这种声明式语言的可扩展性. 你可以添加自己的语言元素或加强现有的语言元素, 从而提供简洁, 易于维护和易于理解的构建.

2. 基于依赖的编程语言

声明式语言位于通用任务图 (general purpose task graph) 的顶端， 它可以被充分利用在你的构建中. 它具有强大的灵活性, 可以满足使用者对 Gradle 的一些特别的需求.

3. 让构建结构化

Gradle 的易适应性和丰富性可让你在构建里直接套用通用的设计原则. 例如, 你可以非常容易容易的使用一些可重用的组件来构成你的构建. Inline stuff where unnecessary indirections would be inappropriate. 不要强行分离已经结合在一起的部分 (例如, 在你的项目层次结构中). 避免使构建难以维护. 总之, 你可以创建一个结构良好， 易于维护, 易于理解的构建.

4. API深化

你会非常乐意在整个构建执行的生命周期中使用 Gradle, 因为Gradle 允许你管理和定制它的配置和执行行为.

5. Gradle scales

Gradle scales very well. 不管是简单的独立项目还是大型的多项目构建, 它都能显著的提高效率. 它不仅可以提供最先进的构建功能， 还可以解决许多大公司碰到的构建 性能低下的问题.

6. 多项目构建

Gradle 对多项目的支持是非常出色的. 它允许你模拟在多项目构建中项目的关系， 这正是你所要关注的地方. Gradle 遵从你的布局而是去违反它.

Gradle 提供了局部构建的功能. 如果你构建一个单独的子项目, Gradle 会构建这个子项目依赖的所有子项目. 你也可以选择依赖于另一个特别的子项目重新构建这些子项目. 这样在一些大型项目里就可以节省非常多的时间.

7. 多种方式来管理你的依赖

不同的团队有不同的管理外部依赖的方法. Gradle 对于任何管理策略都提供了合适的支持. 从远程 Maven 和 Ivy 库的依赖管理到本地文件系统的 jars 或者 dirs.

8. Gradle 是第一个构建整合工具

Ant tasks are first class citizens. Even more interesting, Ant projects are first class citizens as well. Gradle provides a deep import for any Ant project, turning Ant targets into native Gradle tasks at runtime. You can depend on them from Gradle, you can enhance them from Gradle, you can even declare dependencies on Gradle tasks in your build.xml. The same integration is provided for properties, paths, etc ...

Gradle fully supports your existing Maven or Ivy repository infrastructure for publishing and retrieving dependencies. Gradle also provides a converter for turning a Maven pom.xml into a Gradle script. Runtime imports of Maven projects will come soon.

9. 易于迁移

Gradle can adapt to any structure you have. Therefore you can always develop your Gradle build in the same branch

where your production build lives and both can evolve in parallel. We usually recommend to write tests that make sure that the produced artifacts are similar. That way migration is as less disruptive and as reliable as possible. This is following the best-practices for refactoring by applying baby steps.

10. Groovy

Gradle's build scripts are written in Groovy, not XML. But unlike other approaches this is not for simply exposing the raw scripting power of a dynamic language. That would just lead to a very difficult to maintain build. The whole design of Gradle is oriented towards being used as a language, not as a rigid framework. And Groovy is our glue that allows you to tell your individual story with the abstractions Gradle (or you) provide. Gradle provides some standard stories but they are not privileged in any form. This is for us a major distinguishing features compared to other declarative build systems. Our Groovy support is also not just some simple coating sugar layer. The whole Gradle API is fully groovynized. Only by that using Groovy is the fun and productivity gain it can be.

10. The Gradle wrapper

The Gradle Wrapper allows you to execute Gradle builds on machines where Gradle is not installed. This is useful for example for some continuous integration servers. It is also useful for an open source project to keep the barrier low for building it. The wrapper is also very interesting for the enterprise. It is a zero administration approach for the client machines. It also enforces the usage of a particular Gradle version thus minimizing support issues.

11. 免费和开源

Gradle 是一个开源项目, 遵循 ASL 许可.

为什么用 **Groovy?**

我们认为在脚本构建时，内部基于XML的DSL（基于一个动态语言）优势是巨大的. 有许多动态语言在那里, 我们为什么选择 Groovy? 答案在于 Gradle 的运行环境. 虽然 Gradle 是以一个多用途的构建工具为核心，它的重点是Java项目. 在这样的项目中, 显然团队每个成员都了解Java. 我们认为构建应尽可能对所有团队成员都是透明的, 所以选择了 Groovy.

你可能会说，为什么不直接使用 Java 作为构建脚本的语言. 我们认为这是一个有效性的问题. 对于你的团队, 它要有最高的透明度和最低的学习曲线, 也就是说容易掌握. 但由于 Java 的限制, 这样的构建语言不会那么完美和强大. 如 Python，Groovy 或 Ruby 语言都可以有更高的效率. 我们选择了 Groovy 是因为它给 Java 开发人员提供了迄今为止最大的透明度. 其基本的符号和类型与 Java 是一样的，其封装结构和许多其他的地方也是如此.

对于那些同样分享 Python 或 Ruby 知识的 Java 团队将会很乐意学习它. Gradle 的设计非常适合在 JRuby 和 Jython 中创建另一个构建脚本引擎. 它只是目前开发的优先级里. 我们十分支持任何人来做贡献, 创建额外的构建脚本引擎.

教程

- 入门

入门

接下来的教程讲先介绍Gradle的基础知识

Chapter 3, 安装 Gradle

描述如何安装 Gradle.

Chapter 5, 脚本构建基础

介绍脚本构建的基础元素: projects 和 tasks.

Chapter 6, Java 快速入门

展示如何开始使用 Gradle 的合约构建来构建 Java 项目.

Chapter 7, 依赖管理基础

展示如何开始使用 Gradle 的依赖管理.

Chapter 8, Groovy 快速入门

使用 Gradle 的合约构建来构建 Groovy 项目.

Chapter 9, 网页应用快速入门

使用 Gradle 的合约构建来构建网页应用项目.

安装 Gradle

- 先决条件
- 下载
- 解压缩
- 环境变量
- 运行并测试您的安装
- JVM选项

先决条件

Gradle 需要安装一个 Java JDK 或者 JRE. Java 版本必须至少是6以上. Gradle 自带 Groovy 库, 所以没必要安装 Groovy. 任何已经安装的 Groovy 会被 Gradle 忽略.

Gradle使用任何存在在你的路径中的JDK (可以通过 **java -version**检查). 或者, 你可以设置 JAVA_HOME 环境参数来指定希望使用的JDK的安装目录.

下载

你可以从Gradle网站下载任意一个已经发布的版本

解压缩

Gradle发布的版本为ZIP格式. 所有文件包含:

- Gradle 二进制文件.
- 用户指南 (HTML 和 PDF).
- DSL参考指南.
- API文档 (Javadoc和 Groovydoc).
- 扩展的例子,包括用户指南中引用的实例，以及一些完整的以及更复杂的build来帮助用户构建自己的build.
- 二进制源码.此代码仅供参考.如果你想要构建Gradle你需要下载发布的源代码或者提取代码库中的源代码. 请参考官方网站获得具体的信息.

环境变量

为了运行Gradle, 添加 *GRADLE_HOME/bin* 到您的 *PATH* 环境变量中. 通常, 这样已经足够运行Gradle了.

这里的 *GRADLE_HOME* 是 Gradle 的安装路径.

运行并测试您的安装

您可以通过 **gradle** 命令来运行Gradle. 通过 **gradle -v** 命令来检测Gradle是否已经正确安装. 如果正确安装，会输出Gradle版本信息以及本地的配置环境 (groovy 和 JVM 版本等). 显示的版本信息应该与您所下载的gradle版本信息相匹配.

JVM 选项

JVM 选项可以通过设置环境变量来更改. 您可以使用 GRADLE_OPTS 或者 JAVA_OPTS. 根据惯例, JAVA_OPTS 是一个用于 JAVA 应用的环境变量. 一个典型的用例是在 JAVA_OPTS 里设置HTTP代理服务器(proxy) , 在 GRADLE_OPTS 这是内存选项. 这些变量也可以在 gradle 的一开始就设置或者通过 gradlew 脚本.

排除故障

当使用 Gradle 时, 你肯定会碰到许多问题. 你也许不知道如果使用一个特别的功能, 或者你碰到了一个 BUG. 或者, 你只是有一些关于使用y Gradle 的问题.

这一章给出了一些简单的建议并解释了如何解决你的问题.

解决遇到的问题

如果你碰到了问题, 首先要确定你使用的是最新版本的 Gradle. 我们会经常发布新版本, 解决一些 bug 并加入新的功能. 所以你遇到的问题可能就在新版本里解决了.

如果你正在使用 Gradle Daemon, 先暂时关闭 daemon (你可以使用 **switch --no-daemon** 命令). 在第19章我们可以了解到更多关于 daemon 的信息.

获得帮助

可以去 http://forums.gradle.org 获得相关的帮助. 在 Gradle 论坛里, 你可以提交问题, 当然也可以回答其他 Gradle 开发人员和使用者的问题.

如果你碰到不能解决的问题, 请在论坛里报告或者提出这个问题, 通常这是解决问题最快的方法. 您也可以提出建议或者一些新的想法. 开发团队会经常性的发布新的东西或者发布通知, 通过论坛, 您可以获得 Gradle 最新的开发信息.

构建脚本基础知识

- Projects 和 tasks
- Hello world
- 快捷的任务定义
- 构建脚本代码
- 任务依赖
- 动态任务
- 使用已经存在的任务
- 快捷注释
- 附加的 task 属性
- 使用 Ant 任务
- 使用方法
- 默认的任务
- 通过 DAG 配置

Projects 和 tasks

Gradle 里的任何东西都是基于这两个基础概念: *projects(项目)* 和 *tasks(任务)*.

每一个 Gradle 构建都是由一个或多个projects 构成的. 一个 project 到底代表什么依赖于你想用 Gradle 做什么. 举个例子, 一个 project 也许代表一个 JAR 或者一个网页应用. 它也可能代表一个发布的 ZIP 压缩包, 这个 ZIP 可能是由许多其他项目的 JARs 构成的. 一个 project 不必要代表要被构建的某个东西. 它可以代表一件要做的事, 比如布署你的应用. 不要担心现在这些说明看上去有一点模糊. Gradle 的合约构建的支持加入了一个更加具体的关于 project 的定义.

每一个 project 是由一个或多个 tasks 构成的. 一个 task 代表一些更加细化的构建. 可能是编译一些 classes, 创建一个 JAR,生成 javadoc, 或者生成莫个目录的压缩文件.

目前, 我们将关注定义构建里的一些简单的 tasks . 以后的章节会关注与多项目构建以及如果通过projects 和 tasks 工作.

Hello world

您通过 **gradle** 命令运行一个 Gradle 构建. **gradle** 命令会在当前目录查找一个叫 build.gradle 的文件. 我们称这个 build.gradle 文件为一个构建脚本 (build script), 虽然严格来说它是一个构建配置脚本 (build configuration script). 这个脚本定义了一个 project 和它的 tasks.

让我们来试一试，创建一个名为build.gradle的构建脚本.

Example 6.1. 第一个构建脚本

build.gradle

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

在命令行里, 进入包含的文件夹然后通过 **gradle -q hello** 执行构建脚本:

gradle -q hello 的输出

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

这里发生了什么? 这个构建脚本定义了一个单独的 task, 叫做 hello, 并且加入了一个 action. 当你运行 **gradle hello**, Gradle 执行叫做 hello 的 task, 也就是执行了你所提供的 action . 这个 action 是一个包含一些 Groovy 代码的闭包(closure 这个概念不清楚的同学好好谷歌下).

如果你认为这些看上去和 Ant 的 targets 很想象, 好吧, 你是对的. Gradle tasks 和 Ant 的 targets 是对等的. 但是你也会看到, 他们是更加强力的. 我们使用一个不同于 Ant 的术语 task , 看上去比 target 更加能直白. 不幸的是这个带来了一个术语冲突, 因为 Ant 称它的命令, 比如 javac 或者 copy, 叫 tasks. 所以当我们谈论 tasks, 是指 Gradle 的 tasks. 如果我们能讨论 Ant 的 tasks (Ant 命令), 我们会直接称呼 ant task.

补充一点命令里的 **-q** 是干什么的?

这个指南里绝大多说的例子会在命令里加入 **-q**. 它取缔了 Gradle 的日志信息 (log messages), 所以用户只能看到 tasks 的输出. 它例子的输出更加清晰. 你并不一定需要加入这个选项. 参考第 18 章, 日志的 Gradle 影响输出的详细信息.

快捷的任务定义

有一种比我们之前定义的 hello 任务更简明的方法

*Example 6.3. 快捷的任务定义

build.gradle*

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

再一次, 它定义一个叫做 hello 的任务, 这个任务是一个可以执行的闭包. 我们将使用这种方式来定义这本指南里所有的任务.

构建脚本代码

Gradle 的构建脚本展示给你 Groovy 的所有能力. 作为开胃菜, 来看看这个:

Example 6.4. 在 Gradle 任务里使用 Groovy

build.gradle

```
task upper << {
    String someString = 'mY_nAmE'
    println "Original: " + someString
    println "Upper case: " + someString.toUpperCase()
}
```

gradle -q upper 命令的输出

```
> gradle -q upper
Original: mY_nAmE
Upper case: MY_NAME
```

或者

Example 6.5. 在 Gradle 任务里使用 Groovy

build.gradle

```
task count << {
    4.times { print "$it " }
}
```

gradle -q count 命令的输出

```
> gradle -q count
0 1 2 3
```

任务依赖

就像你所猜想的那样, 你可以申明任务之间的依赖关系.

Example 6.6. 申明任务之间的依赖关系

build.gradle

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

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

gradle -q intro 命令的输出

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

再加入一个依赖之前, 这个依赖的任务不需要提前定义了,来看下面的例子.

Example 6.7. Lazy dependsOn - 其他的任务还没有存在

build.gradle

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

gradle -q taskX 命令的输出

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

taskX 到 taskY 的依赖在 taskY 被定义之前就已经申明了. 对于我们之后讲到的多任务构建是非常重要的. 任务依赖将会在 14.4 具体讨论.

请注意你不能使用快捷注释 (参考 5.8, “快捷注释”) 当所关联的任务还没有被定义.

动态任务

Groovy 不仅仅被用来定义一个任务可以做什么. 举个例子, 你可以使用它来动态的创建任务.

Example 6.8. 动态的创建一个任务

build.gradle

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

gradle -q task1 命令的输出

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

使用已经存在的任务

当任务创建之后, 它可以通过API来访问. 这个和 Ant 不一样. 举个例子, 你可以创建额外的依赖.

Example 6.9. 通过*API*访问一个任务 - 加入一个依赖

build.gradle

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

gradle -q task0 命令的输出

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

或者你可以给一个已经存在的任务加入行为.

Example 6.10. 通过*API*访问一个任务 - 加入行为

build.gradle

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

gradle -q hello 命令的输出

```
> gradle -q hello
Hello Venus
Hello Earth
Hello Mars
Hello Jupiter
```

doFirst 和 doLast 可以被执行许多次. 他们可以在任务动作列表的开始和结束加入动作. 当任务执行的时候, 在动作列表里的动作将被按顺序执行. << 操作符是 doLast 的简单别称.

快捷注释

正如同你已经在之前的示例里看到的, 有一个方便的注释可以访问一个存在的任务. 每个任务可以作为构建脚本的属性:

Example 6.11. 当成构建脚本的属性来访问一个任务

build.gradle

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

gradle -q hello 命令的输出

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

这里的name是任务的默认属性, 代表当前任务的名字, 这里是 hello

这使得代码易于读取，特别是当使用了由插件（如编译）提供的任务时尤其如此.

附加的 **task** 属性

你可以给任务加入你自己的属性. 为了加入一个 myProperty 属性, 设置一个初始值给 ext.myProperty. 从这一点上来说，该属性可以读取和设置像一个预定义的任务属性.

Example 6.12. 给任务加入额外的属性

build.gradle

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

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

gradle -q printTaskProperties 命令的输出

```
> gradle -q printTaskProperties
myValue
```

给任务加额外的属性是没有限制的. 你可以在 13.4.2, “额外属性” 里获得更多的信息.

使用 **Ant** 任务

Ant 任务是 Gradle 的一等公民. Gradle 通过 Groovy 出色的集成了 Ant 任务. Groovy 和 AntBuilder 装在在一起. 相比于使用 Ant任务从一个build.xml文件, 在 Gradle 里使用 Ant 任务是为方便和强大. 从下面的例子中，你可以学习如何执行 Ant 任务以及如何访问 ant 属性：

Example 6.13. 使用 *AntBuilder* 来执行 *ant.loadfile* 任务

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 命令的输出

```
> 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 ***
```

使不可能成为可能, 使可能更加简单，使简单更加优雅.

使用方法

Gradle 能很好地衡量你规划构建逻辑的能力. 首先衡量的是如何提取一个方法.

*Example 6.14. 使用方法规划你的构建逻辑

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()
}
```

adle -q loadfile 命令的输出

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

稍后你看到，这种方法可以在多项目构建的子项目之间共享. 如果你的构建逻辑变得更加复杂,Gradle 为你提供了其他非常方便的方法. 请参见第59章，组织构建逻辑。

默认任务

Gradle 允许你定义一个或多个默认任务.

Example 6.15. 定义一个默认任务

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 命令的输出

```
> gradle -q
Default Cleaning!
Default Running!
```

等价于 **gradle clean run**. 在一个多项目构建中, 每一个子项目都可以有它特别的默认任务. 如果一个子项目没有特别的默认任务, 父项目的默认任务将会被执行.

通过 DAG 配置

正如我们之后的详细描述(参见第55章， 构建的生命周期), Gradle 有一个配置阶段和执行阶段. 在配置阶段后, Gradle 将会知道应执行的所有任务. Gradle 为你提供一个"钩子", 以便利用这些信息. 举个例子, 判断发布的任务是否在要被执行的任务当中. 根据这一点, 你可以给一些变量指定不同的值.

在接着的例子中, distribution 任务和 release 任务 将根据变量的版本产生不同的值.

Example 6.16. 根据选择的任务产生不同的输出

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 命令的输出

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

最重要的是 whenReady 在 release 任务执行之前就已经影响了 release 任务. 甚至 release 任务不是首要任务 (i.e., 首要任务是指通过 gradle 命令的任务).

Java 快速入门

- Java 插件
- 一个基础的 Java 项目
- 多项目的 Java 构建

Java 插件

正如我们已经看到的, Gradle 是一种多种用途的构建工具 . 它可以在你的构建脚本里构建你在意的许多东西. 然而, 除非你在构建脚本里加入代码, 不然它什么都不会执行.

大都数 Java 项目是非常相像的 : 你需要编译你的 Java 源文件, 运行一些单元测试, 同时创建一个包含你类文件的 JAR. 如果你不需要为每一个项目重复编写这些, 我想你会非常乐意的. 幸运的是, 你现在不再需要了. Gradle 通过使用插件解决了这个问题. 一个插件是 Gradle 的一个扩展, 它会通过某种方式配置你的项目, 典型的有加入一些预配置任务. Gradle 装载了许多插件, 你也可以很简单地编写自己的插件并和其他开发者分享它. Java 插件就是一个这样的插件. 这个插件在你的项目里加入了许多任务， 这些任务会编译和Th单元测试你的源文件, 并且把它们集成都一个 JAR 文件里.

Java 插件是基于合约的. 这意味着插件给项目的许多方面定义了默认的值, 比如 Java 源文件在哪里. 如果你在项目里遵从这些合约, 你通常不需要在你的构建脚本里加入太多东西. 如果你不想要或者是你不能遵循合约, Gradle 允许你自己定制你的项目. 事实上, 因为对 Java 项目的支持是通过插件实现的, 如果你不想要的话你一点也不需要使用这个插件来构建你的 .

在后面的章节, 我们有许多能让你深入了解 Java 插件, 依赖管理和多项目构建的例子很多例子. 在本章中， 我们想给你一个如何使用Java插件来构建一个Java项目的初步认识。

一个基础的 **Java** 项目

让我们来看一个简单的例子. 加入下面的代码来使用 Java 插件:

例子 *6.1.* 使用 *Java* 插件

build.gradle

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

这个例子的代码可以在 samples/java/quickstart 里找到, 二进制代码和源代码里都包含这些文件. 它将会把 Java 插件加入到你的项目中, 这意味着许多任务被自动的加入到了你的项目里.

Gradle 希望能在 src/main/java 找到你的源代码, 在 src/test/java 找到你的测试代码. 另外, 任何在 src/main/resources 的文件都将被包含在 JAR 文件里, 同时任何在 src/test/resources 的文件会被加入到 classpath 中以运行测试代码. 所有的输出文件将会被创建在构建目录里, JAR 文件 存放在 build/libs 文件夹里.

都有什么可以执行的任务呢**?**

你可以使用 ****gradle tasks** 来列出项目的所有任务to. 通过这个命令来尝试看看 Java 插件都在你的项目里加入了哪些命令吧.

建立项目

Java 插件在你的项目里加入了许多任务. 然而, 你只会用到其中的一小部分任务. 最常用的任务是 build 任务, 它会建立你的项目. 当你运行 **gradle build** 命令时, Gradle 将编译和测试你的代码, 并且创建一个包含类和资源的 JAR 文件:

Example 7.2. 建立一个 *Java* 项目

gradle build 命令的输出

```
> gradle build
:compileJava
:processResources
:classes
:jar
:assemble
:compileTestJava
:processTestResources
:testClasses
:test
:check
:build

BUILD SUCCESSFUL

Total time: 1 secs
```

其余一些有用的任务是:

clean

删除 build 生成的目录和所有生成的文件.

assemble

编译并打包你的代码, 但是并不运行单元测试. 其他插件会在这个任务里加入更多的东西. 举个例子, 如果你使用 War 插件, 这个任务将根据你的项目生成一个 WAR 文件.

check

编译并测试你的代码. 其他的插件会加入更多的检查步骤. 举个例子, 如果你使用 checkstyle 插件, 这个任务将会运行 Checkstyle 来检查你的代码.

外部的依赖

通常, 一个 Java 项目将有许多外部的依赖, 既是指外部的 JAR 文件. 为了在项目里引用这些 JAR 文件, 你需要告诉 Gradle 去哪里找它们. 在 Gradle 中, JAR 文件位于一个仓库中， 这里的仓库类似于 MAVEN 的仓库. 仓库可以被用来提取依赖, 或者放入一个依赖, 或者两者皆可. 举个例子, 我们将使用开放的 Maven 仓库:

例子 *6.3.* 加入 *Maven* 仓库

build.gradle

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

接下来让我们加入一些依赖. 这里, 我们假设我们的项目在编译阶段有一些依赖:

例子 *6.4.* 加入依赖

build.gradle

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

你可以在第7章里看到更多这方面的内容.

定制项目

Java 插件给项目加入了一些属性 (propertiy) . 这些属性已经被定义了默认的值, 已经足够来开始构建项目了. 如果你认为不合适, 改变它们的值也是很简单的. 让我们看下这个例子. 这里我们将指定 Java 项目的版本号, 以及我们所使用的 Java 的版本. 我们同样也加入了一些属性在 jar 的清单里.

例子 *6.5.* 定制 *MANIFEST.MF* 文件

build.gradle

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

Java 插件加入的任务是常规性的任务, 准确地说, 就如同它们在构建文件里声明地一样. 这意味着你可以使用任务之前的章节提到的方法来定制这些任务T. 举个例子, 你可以设置一个任务的属性, 在任务里加入行为, 改变任务的依赖, 或者完全重写一个任务, 我们将配置一个测试任务, 当测试执行的时候它会加入一个系统属性:

例子 *6.6.* 测试阶段加入一个系统属性

build.gradle

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

哪些属性是可用的?

你可以使用 **gradle properties** 命令来列出项目的所有属性. 这样你就可以看到 Java 插件加入的属性以及它们的默认值.

发布 **JAR** 文件

通常 JAR 文件需要在某个地方发布. 为了完成这一步, 你需要告诉 Gradle 哪里发布 JAR 文件. 在 Gradle 里, 生成的文件比如 JAR 文件将被发布到仓库里. 在我们的例子里, 我们将发布到一个本地的目录. 你也可以发布到一个或多个远程的地点.

Example 7.7. 发布 JAR 文件

build.gradle

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

运行 **gradle uploadArchives** 命令来发布 JAR 文件.

创建 **Eclipse** 项目

为了把你的项目导入到 Eclipse, 你需要加入另外一个插件:

Example 7.8. Eclipse 插件

build.gradle

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

现在运行 **gradle eclipse** 命令来生成 Eclipse 的项目文件. Eclipse 任务将在第 38 章, Eclipse 插件里详细讨论.

总结

下面是一个完整的构建文件的样本:

Example 7.9. Java 例子 - 完整的构建文件

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'
       }
    }
}
```

多项目的 **Java** 构建

现在让我们看一个典型的多项目构建. 下面是项目的布局:

Example 7.10. 多项目构建 - 分层布局

构建布局

```
multiproject/
  api/
  services/webservice/
  shared/
```

注意: 这个例子的代码可以在 samples/java/multiproject 里找到.

现在我们能有三个项目. 项目的应用程序接口 (API) 产生一个 JAR 文件, 这个文件将提供给用户, 给用户提供基于 XML 的网络服务. 项目的网络服务是一个网络应用, 它返回 XML. shared 目录包含被 api 和 webservice 共享的代码.

定义一个多项目构建

为了定义一个多项目构建, 你需要创建一个设置文件 (settings file). 设置文件放在源代码的根目录, 它指定要包含哪个项目. 它的名字必须叫做 **settings.gradle**. 在这个例子中, 我们使用一个简单的分层布局. 下面是对应的设置文件:

Example 7.11. 多项目构建 - settings.gradle file

settings.gradle

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

在第56章. 多项目构建, 你可以找到更多关于设置文件的信息.

通用配置

对于绝大多数多项目构建, 有一些配置对所有项目都是常见的或者说是通用的. 在我们的例子里, 我们将在根项目里定义一个这样的通用配置, 使用一种叫做配置注入的技术 (configuration injection). 这里, 根项目就像一个容器, subprojects 方法遍历这个容器的所有元素并且注入指定的配置 . 通过这种方法, 我们可以很容易的定义所有档案和通用依赖的内容清单:

Example 7.12. 多项目构建 - 通用配置

build.gradle

```
subprojects {
    apply plugin: 'java'
    apply plugin: 'eclipse-wtp'

    repositories {
       mavenCentral()
    }

    dependencies {
        testCompile 'junit:junit:4.11'
    }

    version = '1.0'

    jar {
        manifest.attributes provider: 'gradle'
    }
}
```

注意我们例子中, Java 插件被应用到了每一个子项目中plugin to each. 这意味着我们前几章看到的任务和属性都可以在子项目里被调用. 所以, 你可以通过在根目录里运行 **gradle build** 命令编译, 测试, 和 JAR 所有的项目.

项目之间的依赖

你可以在同一个构建里加入项目之间的依赖, 举个例子, 一个项目的 JAR 文件被用来编译另外一个项目. 在 api 构建文件里我们将加入一个由 shared 项目产生的 JAR 文件的依赖. 由于这个依赖, Gradle 将确保 shared 项目总是在 api 之前被构建.

Example 7.13. 多项目构建 - 项目之间的依赖

api/build.gradle

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

创建一个发行版本

(该章需加入更多内容。。。原稿写的太简单了)我们同时也加入了一个发行版本, 将会送到客户端:

Example 7.14. 多项目构建 - 发行文件

api/build.gradle

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

artifacts {
   archives dist
}
```

依赖管理的基础知识

- 什么是依赖管理
- 声明你的依赖
- 依赖配置
- 外部的依赖
- 仓库
- 发布 artifacts

什么是依赖管理?

Very roughly, dependency management is made up of two pieces. Firstly, Gradle needs to know about the things that your project needs to build or run, in order to find them. We call these incoming files the dependencies of the project. Secondly, Gradle needs to build and upload the things that your project produces. We call these outgoing files the publications of the project. Let's look at these two pieces in more detail:

Most projects are not completely self-contained. They need files built by other projects in order to be compiled or tested and so on. For example, in order to use Hibernate in my project, I need to include some Hibernate jars in the classpath when I compile my source. To run my tests, I might also need to include some additional jars in the test classpath, such as a particular JDBC driver or the Ehcache jars.

These incoming files form the dependencies of the project. Gradle allows you to tell it what the dependencies of your project are, so that it can take care of finding these dependencies, and making them available in your build. The dependencies might need to be downloaded from a remote Maven or Ivy repository, or located in a local directory, or may need to be built by another project in the same multi-project build. We call this process dependency resolution.

Often, the dependencies of a project will themselves have dependencies. For example, Hibernate core requires several other libraries to be present on the classpath with it runs. So, when Gradle runs the tests for your project, it also needs to find these dependencies and make them available. We call these transitive dependencies.

The main purpose of most projects is to build some files that are to be used outside the project. For example, if your project produces a java library, you need to build a jar, and maybe a source jar and some documentation, and publish them somewhere.

These outgoing files form the publications of the project. Gradle also takes care of this important work for you. You declare the publications of your project, and Gradle take care of building them and publishing them somewhere. Exactly what "publishing" means depends on what you want to do. You might want to copy the files to a local directory, or upload them to a remote Maven or Ivy repository. Or you might use the files in another project in the same multi-project build. We call this process publication.

声明你的依赖

让我们看一下一些依赖的声明. 下面是一个基础的构建脚本:

例子 *7.1.* 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.+'
}
```

这里发生了什么? 这个构建脚本声明 Hibernate core 3.6.7.Final 被用来编译项目的源代码. By implication, 在运行阶段同样也需要 Hibernate core 和它的依赖. 构建脚本同样声明了需要 junit >= 4.0 的版本来编译项目测试. 它告诉 Gradle 到 Maven 库里找任何需要的依赖. 接下来的部分会具体说明.

依赖配置

在 Gradle 里, 依赖可以组合成配置. 一个配置简单地说就是一系列的依赖. 我们称它们为依赖配置. 你可以使用它们声明项目的外部依赖. 正如我们将在后面看到, 它们也被用来声明项目的发布.

Java 插件定义了许多标准的配置. 下面列出了一些, 你也可以在表格 23.5, “Java 插件 - 依赖配置”里发现更多具体的信息.

compile

用来编译项目源代码的依赖.

runtime

在运行时被生成的类使用的依赖. 默认的, 也包含了编译时的依赖.

testCompile

编译测试代码的依赖. 默认的, 包含生成的类运行所需的依赖和编译源代码的依赖.

testRuntime

运行测试所需要的依赖. 默认的, 包含上面三个依赖.

各种各样的插件加入许多标准的配置. 你也可以定义你自己的配置. 参考 50.3, “配置依赖” 可以找到更加具体的定义和定制一个自己的依赖配置.

外部的依赖

你可以声明许多种依赖. 其中一种是外部依赖. 这是一种在当前构建之外的一种依赖, 它被存放在远程或本地的仓库里, 比如 Maven 的库, 或者 Ivy 库, 甚至是一个本地的目录.

下面的例子讲展示如何加入外部依赖

例子 *7.2.* 定义一个外部依赖

build.gradle

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

引用一个外部依赖需要使用 group, name 和 version 属性. 根据你想要使用的库, group 和 version 可能会有所差别.

有一种简写形式, 只使用一串字符串 "group:name:version".

例子 *7.3.* 外部依赖的简写形式

build.gradle

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

仓库

Gradle 是怎样找到那些外部依赖的文件的呢? Gradle 会在一个仓库里找这些文件. 仓库其实就是文件的集合, 通过 group, name 和 version 整理分类. Gradle 能解析好几种不同的仓库形式, 比如 Maven 和 Ivy, 同时可以理解各种进入仓库的方法, 比如使用本地文件系统或者 HTTP.

默认地, Gradle 不提前定义任何仓库. 在使用外部依赖之前, 你需要自己至少定义一个库. 比如使用下面例子中的 Maven central 仓库:

例子 *7.4. Maven central* 仓库

build.gradle

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

或者使用一个远程的 Maven 仓库:

例子 *7.5.* 使用远程的 *Maven* 仓库

build.gradle

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

或者一个远程的 Ivy 仓库:

例子 *7.6.* 使用远程的 *Ivy* 仓库

build.gradle

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

你也可以使用本地的文件系统里的库. Mqven 和 Ivy 都支持下载的本地.

例子 *7.7.* 使用本地的 *Ivy* 目录

build.gradle

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

一个项目可以有好几个库. Gradle 会根据依赖定义的顺序在各个库里寻找它们, 在第一个库里找到了就不会再在第二个库里找它了.

可以在第 50.6 章, “仓库”里找到更详细的信息.

发布 artifacts

依赖配置也可以用来发布文件. 我们称这些文件publication artifacts, 或者就叫 artifacts.

插件可以很好的定义一个项目的 artifacts, 所以你并不需要做一些特别的工作来让 Gradle 需要发布什么. 你可以通过在 uploadArchives 任务里加入仓库来完成. 下面是一个发布远程 Ivy 库的例子:

例子 *7.8.* 发布一个 *Ivy* 库

build.gradle

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

现在, 当你运 **gradle uploadArchives**, Gradle 将构建和上传你的 Jar. Gradle 也会生成和上传 ivy.xml .

你也可以发布到 Maven 库. 请注意你需要加入 Maven 插件来发布一个 Maven 库. 在下面的例子里, Gradle 将生成和上传 pom.xml.

例子 *7.9.* 发布 *Maven* 库

build.gradle

```
apply plugin: 'maven'

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

在第 51 章, 发布 artifacts 里有更加具体的介绍.

Groovy 快速入门

创建 Groovy 项目时, 你需要使用 Groovy 插件. 这个插件扩展了 Java 插件, 加入了编译 Groovy 的依赖. 你的项目可以包含 Groovy 的源代码, Java 源代码, 或者它们的混合.

一个基本的 **Groovy** 项目

让我们看一个例子. 为了使用 Groovy 插件, 加入下面的代码:

例子 *8.1. Groovy* 插件

build.gradle

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

它也会同时把 Java 插件加入到你的项目里. Groovy 插件扩展了编译任务, 这个任务会在 src/main/groovy 目录里寻找源代码文件, 并且加入了编译测试任务来寻找 src/test/groovy 目录里的测试源代码. 编译任务使用 联合编译 (joint compilation) 来编译这些目录, 这里的联合指的是它们混合有 java 和 groovy 的源文件.

使用 groovy 编译任务, 你必须声明 Groovy 的版本和 Groovy 库的位置. 你可以在配置文件里加入依赖, 编译配置会继承这个依赖, 然后 groovy 库将被包含在 classpath 里.

例子 *8.2. Groovy 2.2.0*

build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    compile 'org.codehaus.groovy:groovy-all:2.3.3'
}
```

下面是完整的构建文件:

例子 *8.3.* 完整的构建文件

build.gradle

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

repositories {
    mavenCentral()
}

dependencies {
    compile 'org.codehaus.groovy:groovy-all:2.3.3'
    testCompile 'junit:junit:4.11'
}
```

运行 **gradle build** 命令将会开始编译, 测试和创建 JAR 文件.

总结

这一章描述了一个非常简单的 Groovy 项目. 通常, 一个真正的项目要比这个复杂的多. 因为 Groovy 项目是一个 Java 项目, 任何你可以对 Java 项目做的配置也可以对 Groovy 项目做.

第 24 章, Groovy 插件有更加详细的描述, 你也可以在 samples/groovy 目录里找到更多的例子.

网页应用快速入门

这一章官网正在补充.还没有完成.

Gradle 提供了两个插件用来支持网页应用: War 插件和 Jetty 插件. War 插件是在 Java 插件的基础上扩充的用来构建 WAR 文件. Jetty 插件是在 War 插件的基础上扩充的, 允许用户将网页应用发布到一个介入的 Jetty 容器里.

构建一个 **WAR** 文件

为了构建一个 WAR 文件, 需要在项目中加入 War 插件:

例子 *9.1. War* 插件

build.gradle

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

这个插件也会在你的项目里加入 Java 插件. 运行 **gradle build** 将会编译, 测试和创建项目的 WAR 文件. Gradle 将会把源文件包含在 WAR 文件的 src/main/webapp 目录里. 编译后的 classe , 和它们运行所需要的依赖也会被包含在 WAR 文件里.

Running your web application

要启动Web工程,在项目中加入Jetty plugin即可:

例 *9.2.* 采用*Jetty plugin*启动*web*工程

build.gradle

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

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

TODO: which url, configure port, uses source files in place and can edit your files and reload.

总结

了解更多关于War plugin和Jetty plugin的应用请参阅第 26 章, War Plugin以及第 28 章, Jetty Plugin.

你可以在发行包的samples/webApplication下找到更多示例.

使用 **Gradle** 命令行

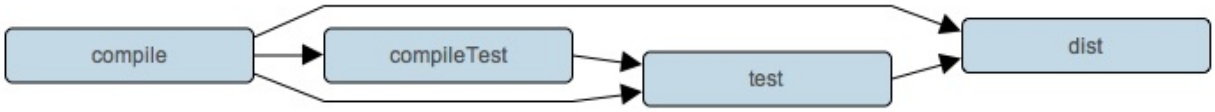
本章介绍了命令行的一些基本功能.正如在前面的章节里你所见到的调用 **gradle** 命令来运行一个构建.

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 compile 和 test 任务, 它们所依赖的任务也会被调用. 这些任务只会被调用一次, 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

示例图 10.1. 任务依赖



例子 *10.1.* 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 命令的输出

```
> 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.

例子 *10.2.* 排除任务

gradle dist -x test 命令的输出

```
> 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,即使它是 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.

失败后继续执行构建

默认情况下, 只要有任务调用失败, Gradle就会中断执行. 这可能会使调用过程更快, 但那些后面隐藏的错误就没有办法发现了. 所以你可以使用 **--continue** 选项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, 你并不需要输入任务的全名. 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:

例 10.3. 简化任务名

gradle di 命令的输出

```
> gradle di
:compile
compiling source
:compileTest
compiling unit tests
:test
running unit tests
:dist
building the distribution

BUILD SUCCESSFUL

Total time: 1 secs
```

你也可以用在驼峰命名方式 (通俗的说就是每个单词的第一个字母大写,除了第一个单词) 的任务中每个单词的首字母进行调用. 例如, 可以执行 **gradle compTest** 或者 **gradle cT** 来调用 compileTest 任务

例 11.4. 简化驼峰方式的任务名

gradle cT 命令的输出

```
> 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.

选择文件构建

调用 gradle 命令时, 默认情况下总是会构建当前目录下的文件, 可以使用 **-b** 参数选择其他目录的构建文件, 并且当你使用此参数时 settings.gradle 将不会生效, 看下面的例子:

例 10.5. 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

```
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

例 10.6. 选择构建目录

gradle -q -p subdir hello 命令的输出

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

-b 参数用以指定脚本具体所在位置, 格式为 dirpwd/build.gradle.

-p 参数用以指定脚本目录即可.

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** 命令会为你列出子项目名称列表.

例 10.7. 收集项目信息

gradle -q projects 命令的输出结果

```
> 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.

例 10.8. 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** 命令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.

例 10.9 获取任务信息

gradle -q tasks 命令的输出

```
> 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 subpr

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.

例 10.10. 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.

例 10.11 获得更多的任务信息

gradle -q tasks --all 命令的输出

```
> 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'.
api:dependencies - Displays all dependencies declared in project ':api'.
webapp:dependencies - Displays all dependencies declared in project ':webapp'.
dependencyInsight - Displays the insight into a specific dependency in root project 'projectReports'.
api:dependencyInsight - Displays the insight into a specific dependency in project ':api'.
webapp:dependencyInsight - Displays the insight into a specific dependency in project ':webapp'.
help - Displays a help message
api:help - Displays a help message
webapp:help - Displays a help message
projects - Displays the sub-projects of root project 'projectReports'.
api:projects - Displays the sub-projects of project ':api'.
webapp:projects - Displays the sub-projects of project ':webapp'.
properties - Displays the properties of root project 'projectReports'.
api:properties - Displays the properties of project ':api'.
webapp:properties - Displays the properties of project ':webapp'.
tasks - Displays the tasks runnable from root project 'projectReports' (some of the displayed tasks may belong to subpr
api:tasks - Displays the tasks runnable from project ':api'.
webapp:tasks - Displays the tasks runnable from project ':webapp'.
```

获取任务具体信息

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.

例 *10.12.* 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** 命令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:

例 *10.13.* 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:2.3.3

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:2.3.3
\--- 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:

例 *10.14.* 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** 命令可以查看指定的依赖. 如下面的例子.

例 *10.15.* 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:2.3.3
\--- project :api
     \--- compile
```

这个 task 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 DependencyInsightReportTask.

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使用. Report 查找那些在指定的配置里特定的依赖.

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 DependencyInsightReportTask.

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:

例 *10.16.* 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
baseClassLoaderScope: org.gradle.api.internal.initialization.DefaultClassLoaderScope@12345
buildDir: /home/user/gradle/samples/userguide/tutorial/projectReports/api/build
buildFile: /home/user/gradle/samples/userguide/tutorial/projectReports/api/build.gradle
```

Using the Gradle Graphical User Interface

Writing Build Scripts

Tutorial - 'This and That'

More about Tasks

Working With Files

Locating files

You can locate a file relative to the project directory using the `Project.file()` method.

Example 16.1. Locating files

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'))`

You can pass any object to the file() method, and it will attempt to convert the value to an absolute File object. Usually, you would pass it a String or File instance. If this path is an absolute path, it is used to construct a File instance. Otherwise, a File instance is constructed by prepending the project directory path to the supplied path. The file() method also understands URLs, such as file:/some/path.xml.

Using this method is a useful way to convert some user provided value into an absolute File. It is preferable to using new File(somePath), as file() always evaluates the supplied path relative to the project directory, which is fixed, rather than the current working directory, which can change depending on how the user runs Gradle.

File collections