1. 背景

通常，一旦需要开发一个更改多个存储库的功能时，事情就会开始变得混乱。例如，可能有一个core存储库和一个module存储库，而正在为module开发的功能需要在module中更改 API。

如果是这样，可能会因为测试对module的更改是否见效而感到恼火。要测试功能，至少必须将本地快照发布到本地 Maven 存储库。虽然这可以在本地工作，但很容易忘记而错过发布，因此认为更改实际上依赖于过时的依赖项时有效。

一旦涉及 CI，或者想与同事分享正在进行的工作结果（例如进行审查），事情就会变得更加复杂：

- 你有没有抱怨过必须检查 core/some-branch，发布到本地 Maven，然后检查 module/some-feature-branch 并测试它？

- 你有没有意识到你忘记将更改推送到 master 以便他们可以尝试？

- 你有没有抱怨过，要在 CI 上实现这一点，你实际上必须急切地将你的feature分支合并到core，只是为了让feature分支上的另一个存储库可以看到它？

潜在的问题是使用 Maven SNAPSHOTs 来处理多存储库开发还不够好。它无法模拟多存储库开发的复杂性，在不同的分支上同时开发功能。使用 SNAPSHOT（二进制依赖）来协调项目会导致难以诊断错误、破坏集成过程。通常必须急切地推送更改，或等待在共享存储库上发布快照，这样才能真正验证与其他模块的集成是否有效。

我在使用 Micronaut 时遇到了同样的问题：我目前正在开发一项涉及同时更改多个存储库的功能：

- Micronaut Core，带有额外的公共 API

- Micronaut AOT，我正在积极开发的新模块

- Micronaut Gradle 插件，它集成了 AOT 模块

- 一个使用修改后的 Gradle 插件的本地测试项目

1. 简介

includegit-gradle-plugin（https://github.com/melix/includegit-gradle-plugin）

该插件允许将 Git 存储库作为源依赖项导入，而无需更改依赖项声明。

在上面的示例中，这意味着当我在处理module时，它需要针对 core/some-branch 进行构建：Gradle 将自动checkout项目，构建分支并替换对应于任何二进制依赖项core的源码依赖。



但它仍然需要将更改推送到远程存储库，以便可以在本地进行测试。好吧，一个好的多存储库开发故事必须整合本地和远程体验。这就是为什么这个插件实际上让支持这种模式变得轻而易举。

实际上有两种方法可以处理这个问题。第一个是向 Gradle 解释，它可以简单地使用本地副本，而不是检查源代码。在这种情况下，插件将简单地忽略在存储库的 gitRepositories 块中声明的任何内容，并使用本地可用的任何内容。为此需要设置一个指向本地副本的 local.git.<repoName> Gradle 属性（在 gradle.properties 文件中）。在上面的示例中，我将添加一个 local.git.core 属性，指向我的本地core副本。

或者，如果将内容组织到checkout目录中，那么可能会将所有与 micronaut 相关的项目都放在一个 micronaut-projects 目录中。在这种情况下，通过将 auto.include.git.dirs Gradle 属性设置为 micronaut-projects 目录，插件将自动将该 micronaut-projects 目录中的目录名称映射到包含的 Git 存储库名称。所以：



假设micronaut-projects 下有一个 micronaut-core 目录，然后它将自动使用，而不是从远程克隆。

这些选项使本地开发变得非常方便，并且只有在准备好时才推送更改。在 CI 上，构建会自动检查依赖项，无需配置任何内容。

完全相同的机制可用于在 CI 上创建“集成”构建。例如，它使得构建可以非常简单地针对 master 的最新状态自动构建，而不必等待 SNAPSHOT 发布，更重要的是，无需更改任何构建文件。作为奖励，它也适用于传递依赖：例如，如果有 A --depends on-→ B --depends on-→ C，那么可能需要确保如果 C 更改，A 仍然有效。如果 A 和 C 之间没有直接依赖关系，你如何处理快照？这个插件使测试变得非常简单：只需为 C 声明一个 Git 存储库，你就完成了！

1. 原理

多存储库开发虽然提供架构优势（减少库的范围、更快的开发周期……），但对开发人员来说往往是痛苦的，尤其是当涉及的模块数量增加时。

例如，要开发单个功能，开发人员可能必须一次处理多个存储库：一个core模块，位于core存储库中，以及一个依赖于core的library模块，位于library存储库中。

问题是特定功能的开发可能需要更改core和library。在这种情况下，Java 生态系统中的开发人员通常依赖于在其本地 Maven 存储库中发布快照：更改core，发布快照，然后更改library，发布快照，然后重复循环。

这种方法有几个问题：

- 它需要在项目之间切换，将中间工件发布到本地文件系统

- 使与他人协作变得困难，因为他们还需要checkout多个项目

- 它需要将 mavenLocal() 集成为本地构建的存储库，这被认为是一种不好的做法（因为它引入了不可重现性并使构建变得脆弱）

- 对于 CI，它需要发布到快照存储库以供下游构建获取更改，这意味着通常必须合并正在进行的工作才能测试更改

- 如果要使用的库不是一级依赖项，它根本就不起作用

为了改善这种情况，Gradle 用户可以使用复合构建来避免发布到本地存储库。通过避免 publishToMavenLocal 耗时，这已经使开发周期更快了。但是，复合构建有一些限制：

- 包含的构建必须在本地目录中可用

- 除非创建一个硬编码checkout的“集成”项目，否则很难甚至不可能在 CI 服务器上进行设置

- 它强制更改构建的配置以使用复合构建

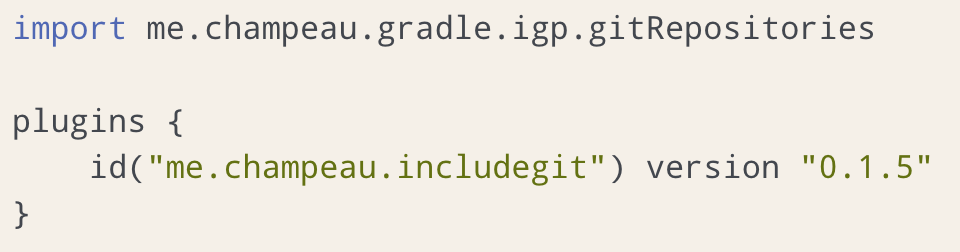
此外，在多存储库环境中开发存在流程问题：如果该功能需要更改多个模块，为了能够集成更改，特别是在 CI 上，必须发布快照或预发布.问题是这不一定是可以接受的：例如，可能希望在每个存储库的分支中开发一个功能，并且只有在完整功能准备好后才合并。

Gradle 还为源依赖项提供实验性支持，但解决了不同的问题。特别是，源依赖是常规依赖的替代品：它们需要更改具有“源依赖”的构建中的依赖表示法，包括分支。相反，我们想要做的是保持我们的构建文件不变，并用源代码替换二进制依赖项。

这个插件通过允许包含（在 Gradle 包含的构建的意义上）Git 存储库并指定应该使用哪些分支/标签来提供此问题的解决方案。

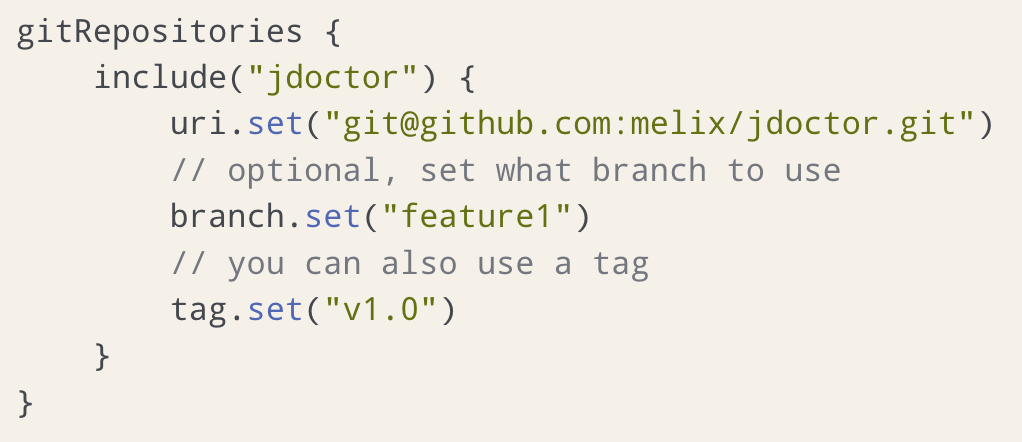
1. 配置

此插件是一个设置插件，必须应用于 settings.gradle(.kts) 文件，而不是项目 build.gradle(.kts) 文件。



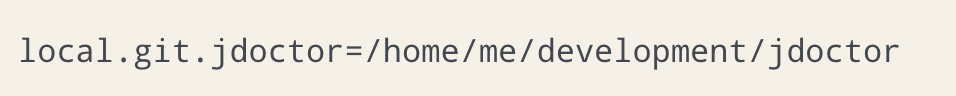
该插件定义了一个 gitRepositories 扩展，用于声明包含的 Git 存储库。

例如，假设想要包含托管在 https://github.com/melix/jdoctor/ 的 jdoctor，那么应该编写：



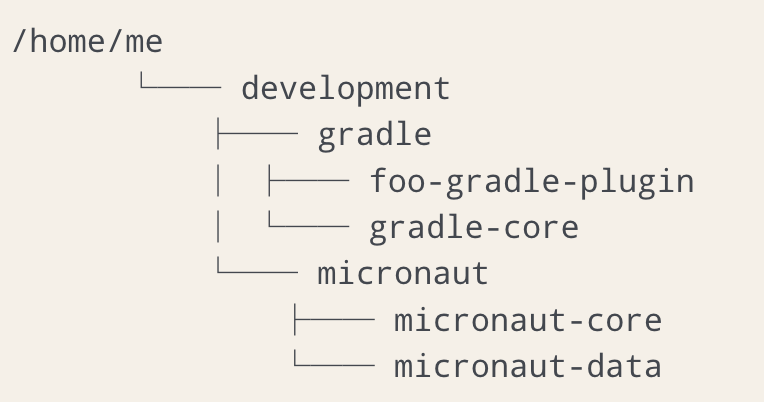
默认情况下，该插件将克隆包含的 Git 存储库到项目的 checkouts 目录中。如果存储库已被克隆，插件将每 24 小时自动执行一次更新。或者，可以通过将 -Drefresh.git.repositories 添加到 Gradle 命令行来强制它更新。

声明包含的 Git 存储库将自动使插件克隆远程存储库。这使得在 CI 上使用非常方便，因为现在可以拥有使用来自其他存储库的其他工作分支的分支。但是，可能已经拥有已经在修改并且想要用于开发的本地克隆。在这种情况下，可以在 gradle.properties 文件中设置 local.git.XXXX Gradle 属性，其中 XXXX 是包含的存储库名称，以指向本地副本。在这种情况下，建议使用位于用户主目录中的 gradle.properties 文件：

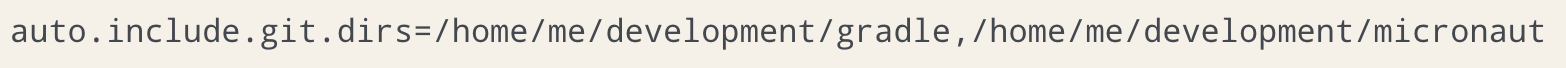


如果找到该属性，则分支或标签配置将被忽略。

或者，可能有一个或多个包含已checkouts 项目的目录。在这种情况下，该插件提供了一种便利，它将自动将目录映射到 Git 存储库名称。为此，需要将 auto.include.git.dirs 设置为要扫描的目录列表。例如：



然后你可以在你的 gradle.properties 文件中设置它：



该插件将自动扫描 gradle 和 micronaut 目录，并将 foo-gradle-plugin、gradle-core、micronaut-core 和 micronaut-data 目录映射到可能包含的 Git 存储库。如果构建包含名为 micronaut-core 的存储库，那么它将自动从 micronaut-core 目录中选择它。

这种机制使得处理具有多个 Git 存储库的复杂代码库变得非常方便。

默认情况下，将自动包含克隆存储库的根目录。您可以通过调用 gitRepositories 扩展的 includeBuild 方法来调整包含构建的配置：



在某些情况下，克隆项目的根目录可能不是想要包含的目录，或者可能想要包含几个子目录作为单独的包含构建。为此，可以使用与 Gradle 的 includeBuild 完全相同的 includeBuild 语句，只是根目录是checkouts 目录：



可以使用来自单个存储库的多个 includeBuild 语句。

该插件支持 3 种不同的身份验证机制：

- 基本认证（用户名+密码）

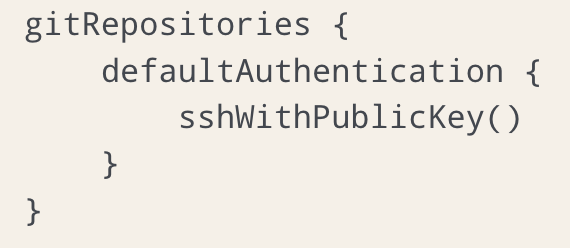
- 使用公钥的 ssh

- ssh 带密码

可以为每个存储库配置身份验证：

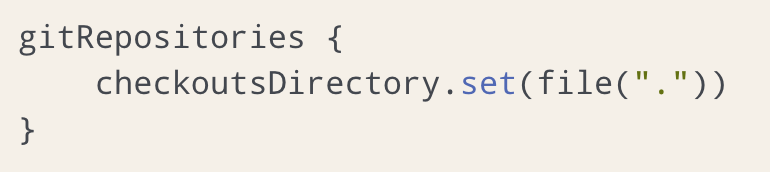


还可以配置默认身份验证机制，当没有在存储库上专门配置身份验证时将使用该机制：



该插件支持 2 种不同的方式来配置checkouts 目录：

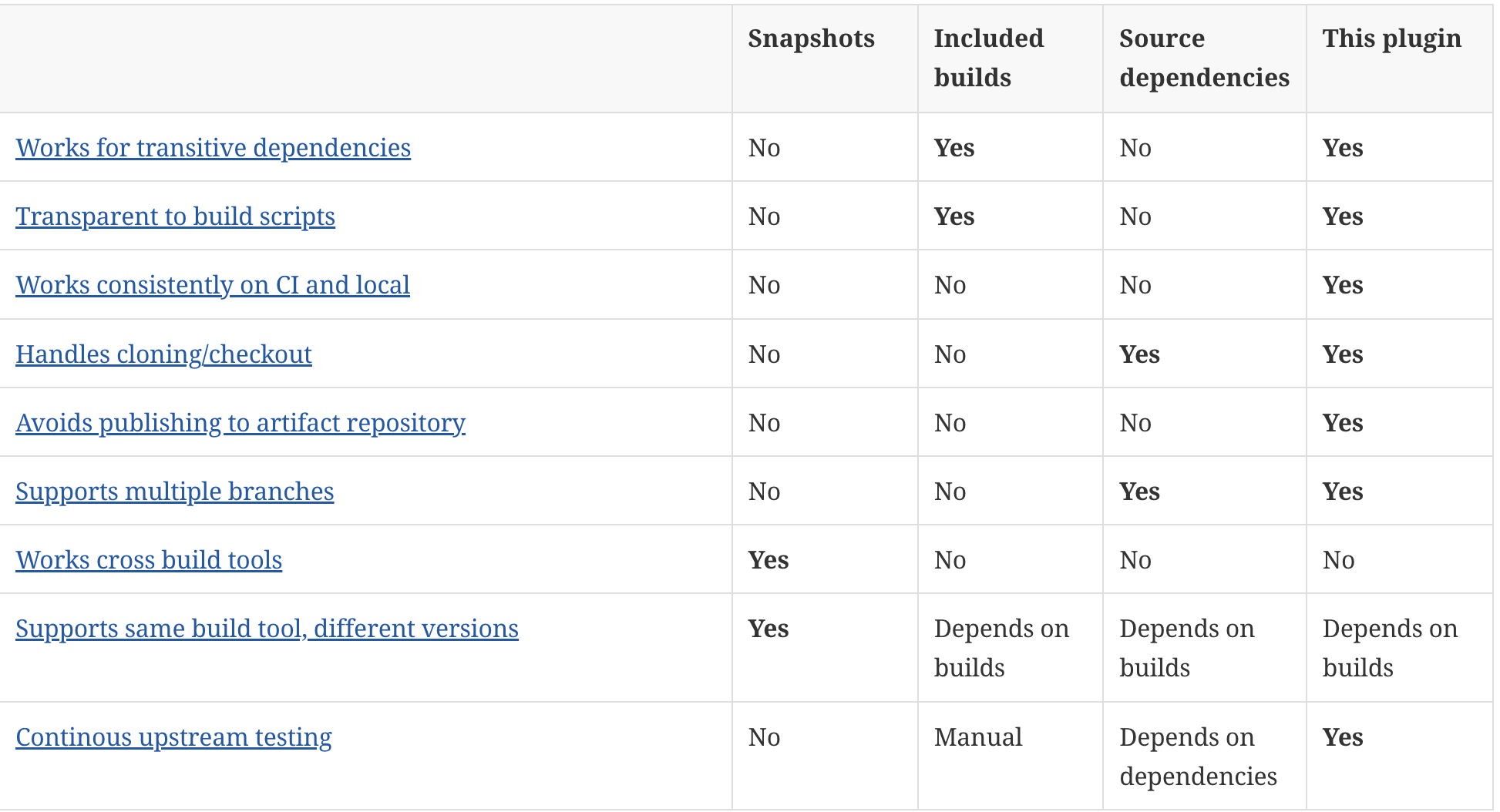
- 通过配置将检出所有存储库的根目录（默认情况下，检出）



- 通过为每个包含的存储库配置一个checkouts 目录



此表总结了每种解决方案的一些优缺点，以便您做出正确的决定。



此比较是针对多存储库设置进行的。这并不意味着它会是相同的，例如，对于生活在单个存储库中的 Gradle 复合构建：

- 适用于传递依赖：构建定义“直接”依赖，通常直接在源代码中使用，但通常需要测试的是传递依赖。此列指示解决方案是否可以透明地用源替换传递依赖项

- 对构建脚本透明：某些解决方案（通常是 SNAPSHOTS）需要更改构建脚本，因为需要引入 mavenLocal、放置特定版本或引入第一级依赖项，以便更改可见。像这个插件这样的其他解决方案只需要应用插件，但保持你的依赖声明不变。

- 在 CI 和本地一致地工作：技术解决方案是否在本地和 CI 上一致地工作？快照是难以推理的典型示例，因为本地 Maven 存储库可能包含与远程快照存储库不同的依赖项。它还需要同步和刷新依赖项。其他解决方案（如复合构建）适用于本地开发，但一旦推送 CI，就会中断，因为本地存储库不可用。

- 处理cloning/checkout：解决方案是否处理checkout（或用 Git 术语cloning）依赖项？它会使依赖项在 IDE 中作为源可见吗

- 避免发布到工件存储库：快照通常需要将工件发布到二进制存储库或本地文件系统，以便其他构建“看到”更改。某些解决方案（例如包含的构建）不会，因为它们使用源来处理依赖关系。

- 支持多个分支：快照工作得很好，除非需要集成来自不同分支的更改：必须发布具有不同坐标或版本的不同工件才能在下游项目中测试它们，或者必须合并更改并推送快照。相反，源依赖可以优雅地处理分支，因为它们不需要任何发布到二进制存储库。

- 工作交叉构建工具：快照可以从不同的构建工具中使用，通常是 Maven 和 Gradle。源依赖项、包含的构建和此插件要求所有参与的构建都使用 Gradle，因此如果混合使用构建工具，则不适合。

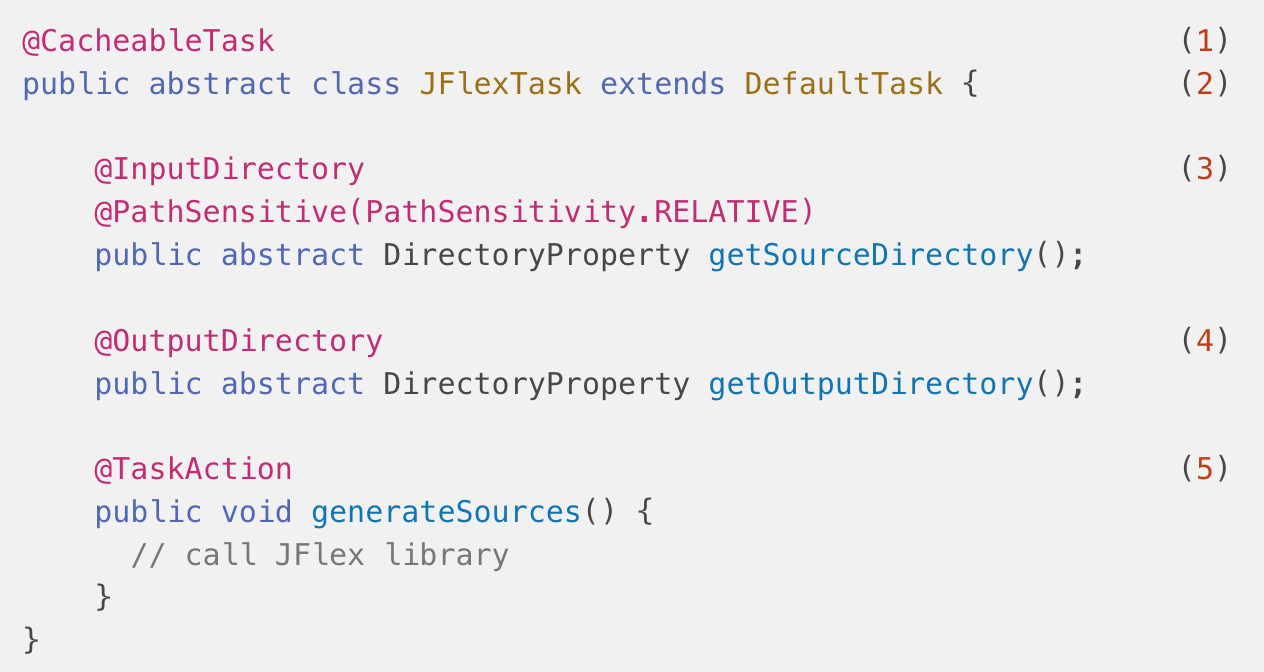
- 支持相同的构建工具，不同的版本：快照是二进制依赖项，因此使用的构建工具无关紧要。包含的构建和源依赖项将使用包含其他构建作为“驱动程序”的构建工具的版本。如果主构建版本和包含的版本之间存在不兼容，构建可能会失败。

- 持续上游测试：该解决方案是否可以持续测试上游依赖项？通常，在不更改构建脚本的情况下，如果可以测试项目是否与依赖项的最新主分支兼容，那就太好了。该插件使其实现起来非常简单，而包含的构建需要一些手动设置。快照无济于事。

1. 插件剖析

在 Gradle 中，插件本质上包括注册许多任务（执行action的东西，如编译源代码），或所谓的extension（以 DSL 的形式向用户公开，用于配置构建）。插件可以做什么和你可以在构建脚本中做什么没有区别，但是，一旦你有超出配置的东西，把东西移到插件中是个好主意。好处是在 Gradle 中创建插件非常简单，甚至不需要在 Maven Central 或 Gradle Plugin Repository 上发布插件：一切都可以在项目中本地化。

在 Gradle 中，负责执行action的单元称为task。在我们的例子中，我们需要一个读取 JFlex 文件并生成源代码的任务。这是它的骨架：



1.这个注解告诉Gradle可以缓存这个任务的执行结果

2.每个 Gradle 任务都需要扩展 DefaultTask 类型

3.我们任务的输入是一个包含 JFlex 文件的目录

4.我们任务的输出将是一个包含生成的 Java 文件的目录

5.这是主要task action，它将调用 JFlex

首先，注意我们的task是定义抽象的：这是因为我们将让 Gradle 为我们生成样板代码，特别是如何注入输入和输出属性。稍后我们将看到更多让 Gradle 为我们服务的有趣原因，但现在显而易见的原因是它减少了必须编写的代码量：不需要知道如何创建 DirectoryProperty： Gradle 会为我们做这件事。

其次，请注意我们如何使用 DirectoryProperty 作为输入和输出属性的类型。这是一个非常重要的 Gradle 类型，属于所谓的“provider API”，或者，你有时可以读到，“lazy API”。早期版本的 Gradle 的大部分排序问题都已通过此 API 修复，因此请使用它！

我们可以注意到的一件事是它是强类型的：为了声明一个输入目录，我们没有将属性定义为File或Path：它是一个目录，它可以帮助 Gradle 和用户理解你应该作为输入提供什么：如果属性设置为常规文件，则 Gradle 可以提供合理的错误消息，说明它需要一个目录。

如何在构建脚本中使用这种类型？



1.将输入目录设置为 src/main/jflex

2.将输出目录设置为 build/generated/jflex

或者这样使用：



register 是新的 create：你不应该再使用 create，因为它会急切地创建任务，这意味着即使它们不会参与任务图，也要配置它们，而 register 是惰性的：如果需要执行任务，并且仅如果，它将被配置。

outputDirectory = file("build/generated/jflex') 对我们的属性仍然有效，并且如果执行会导致相同的结果。要理解这一点，让我们关注输出目录，这使得发生的事情更加明显：比较 build/generated/jflex 和 layout.buildDirectory.dir('generated/jflex')。

在第一种情况下，输出目录被硬编码到 build/generated/jflex 目录。在第 2 种情况下，输出目录派生自构建目录的位置。这意味着，如果由于某种原因，构建被配置为使用与传统构建目录不同的输出目录，例如 target（如在 Maven 中）。在第一种情况下，任务的输出目录是 build/generated/jflex，所以它会写入错误的目录。在 2d 情况下，输出将正确连接到 target/generated/jflex。

或许可能认为可以通过使用 file("$buildDir/generated/jflex") 来解决这个问题。这更好，但还不够，因为结果取决于何时调用：如果在配置任务后更改构建目录，那么会得到错误的结果，这就是为什么很多用户开始随机添加 afterEvaluate 到解决此类问题。

provider API 解决了排序问题并避免了对 afterEvaluate 的虚假调用。

创建插件类：



1.声明一个project范围的插件

2.这个插件会贡献Java源码，所以依赖于Java插件，我们来应用它

3.Java 插件定义了我们需要的 Java 扩展

4.注册我们的 generateLexer 任务

5.定义 JFlex 源文件的常规（默认）位置

6.定义生成源文件的常规（默认）位置

7.定义任务的输出是需要编译的Java文件

我建议使用纯 Java 编写插件，但可以使用 Groovy 或 Kotlin。它使事情变得更加冗长，但它们更清晰并且没有“DSL 魔法”。让我们探索插件在做什么。首先，它是一个项目插件，这基本上意味着它是一个应该应用于项目构建文件的插件，所以通常是一个 build.gradle 文件。

在大多数情况下，该插件与我们在构建脚本中所做的完全一样：它注册一个task，给它一个描述，但更重要的是，它设置了输入和输出的常规值。请注意我如何使用约定方法来设置输入目录，而不是我们在构建脚本中使用的 set 方法：虽然两者都可以使用，但两者之间存在语义差异：在插件中，很可能想要设置约定值，这是默认使用的值，如果用户什么都没说。

我们的插件还做了一件我们还没有涉及的事情：正如我的同事所问的那样，在“生命周期”中连接task。 “生命周期”的概念在 Gradle 中并没有真正的意义，而且很可能来自 Maven 思维模式，其中事物是通过“生命周期”来定义的。但主要区别在于：在 Gradle 中，所有内容都声明其输入，并且该工具负责正确连接事物，因此不必执行多余的工作。

在这里，合理的问题是：我的插件生成了一些源文件，但它们需要编译，因此可用于我的生产代码，我该怎么做？这就是 JavaPluginExtension 发挥作用的地方。事实上，我们的插件并不是独立工作的：它假设我们正在编写 Java，并且假设我们可以编译 Java 源代码。为此，我们实际上可以通过要求应用 JavaPlugin 来明确假设。应用此插件时，它定义了一个 JavaPluginExtension，它声明了源集。特别是，它定义了 Java 源集（main 和 test），它们是被编译的源。

因此，思维方式的转变不是想知道如何编译生成的源代码并将它们放在“类路径上”，就像在 Maven 中所做的那样，而是简单地解释还有另一个源代码目录需要考虑。



这表示“请将 generateLexer task的输出添加为源目录”。这在语义上更强大。神奇的是因为generateLexer任务定义了一个输出目录，现在我们刚才说这个输出目录包含Java类。任何需要 Java 源代码的任务都会自动触发我们的 generateLexer 任务的执行：我们不必明确定义关系！

Gradle 使用在插件之间共享的域对象对关系进行建模。由于这些对象，它可以推断依赖关系。

换句话说，因为Java编译任务的输入是一个源集，而那个源集定义了作为输入，它有一个由generateLexer任务生成的目录，Gradle知道在编译之前，它需要调用那个生成Lexer 任务。使用源集作为输入的任何其他任务都将执行相同的操作：它避免了重复代码和硬接线！

在这个阶段，我们几乎完成了布线，但我们仍然错过了任务的实际实现。

JFlex API 使用静态和实例状态的混合来配置自身。这不好，因为它基本上意味着 API 不是线程安全的：如果由于某种原因，我们有多个任务生成源（例如，如果我们有多个 jflex 目录，或者不同的项目具有 jflex 源），那么我们不能安全地并行生成源！

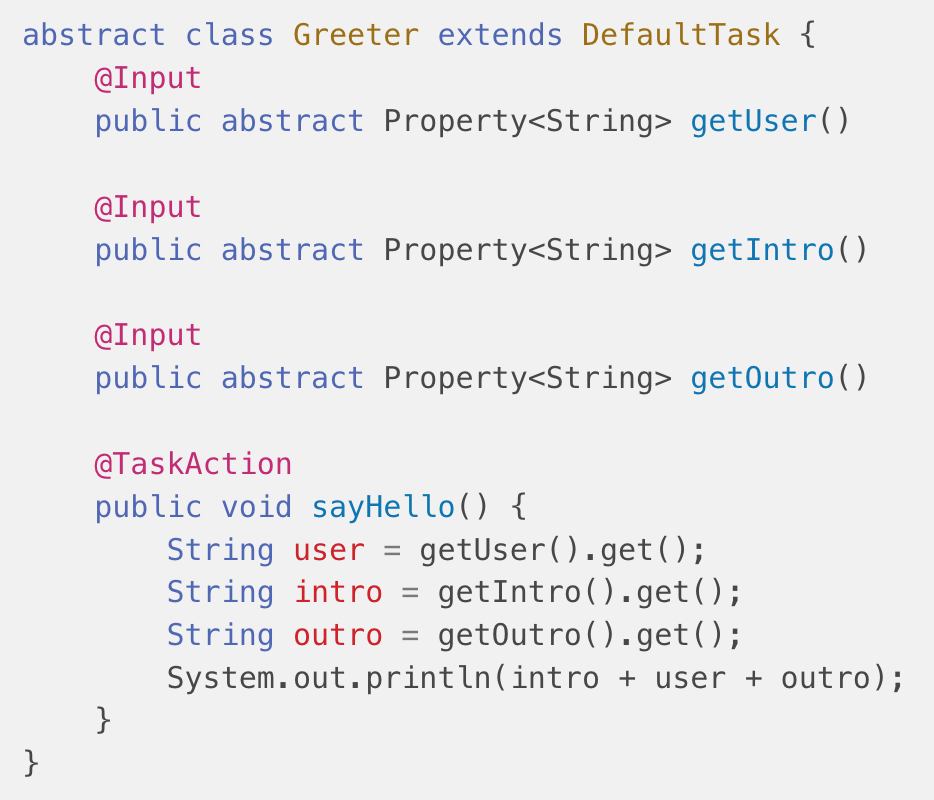
Gradle 提供了一个简单的解决方法：worker API。 worker API 允许做很多事情，但特别是它允许在不同的进程中执行代码，或者更轻量级，在隔离的类加载器中执行代码。第二个选项对我们有好处，因为 Java 中的静态状态仅与给定类加载器中的静态状态一样：如果在 2 个不同的类加载器中加载了 2 个“相同”类，那么它们都有各自独立的静态状态。我们将使用它来正确隔离代码的执行。

要使用 worker API，我们需要在我们的任务中注入所谓的 WorkerExecuter：

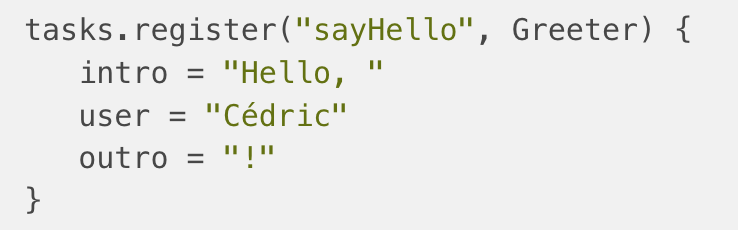


不需要关心如何获得 WorkerExecuter：只需告诉 Gradle 您需要它，瞧！使用 worker API 时，任务动作基本上变成了一个空壳，它只是配置了实际执行应该如何发生。在这种情况下，我们声明类加载器隔离，以及将隔离执行的操作的输入。

Provider API 最有趣的方面之一是值推导。为了理解这个概念，让我们想象一个负责打招呼的 Greeter 任务：



我们可以通过这样做注册一个用英语打招呼的任务：

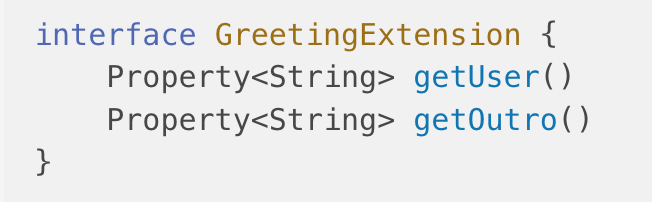


还有一个用法语打招呼：

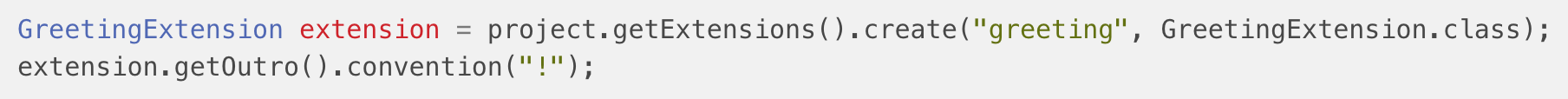


有点烦人的是我们必须在两个任务中重复用户声明，并且“在法语中，感叹号必须以空格开头”的规则不需要用户知道。为了避免这种冗余，我们将编写一个插件，让这一切变得更加方便。

首先，我们将创建一个扩展，它将包含与用户配置相关的内容：要打招呼的人的姓名以及我们想要使用的 outro。



同样，我们不必为此提供实现，Gradle 知道如何创建 Property<String>。这个扩展只需要由我们的插件创建：

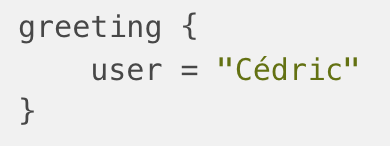


我们的 DSL 只会暴露“user”和“outro”，而不会暴露 intro，这实际上是依赖于语言的。我们还可以在扩展本身上设置一个常规值。然后插件可以为我们注册这两个任务：



现在看到了使用提供程序 API 的好处：对于英文版，任务将直接使用 outro 值，而对于法文版，默认情况下，它将计算派生值。

用户将通过扩展配置任务：

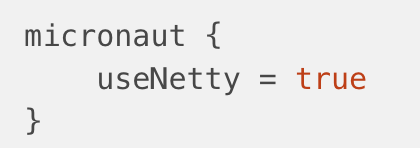


provider API 允许精确定义如何以惰性方式从另一个属性计算值，并提供一种优雅的方式来提供默认值或常规值。

1. Gradle 的条件依赖

如果曾经为框架（例如 Micronaut）编写过 Gradle 插件，或者在用户配置特定标志时需要添加依赖项的插件，那么可能遇到了一些排序问题。

例如，假设你有这个 DSL：



显然，在某个时间点，必须弄清楚是否设置了属性 useNetty 以便透明地添加依赖项。一个天真的解决方案是使用旧的 afterEvaluate 块。许多插件都是这样做的：

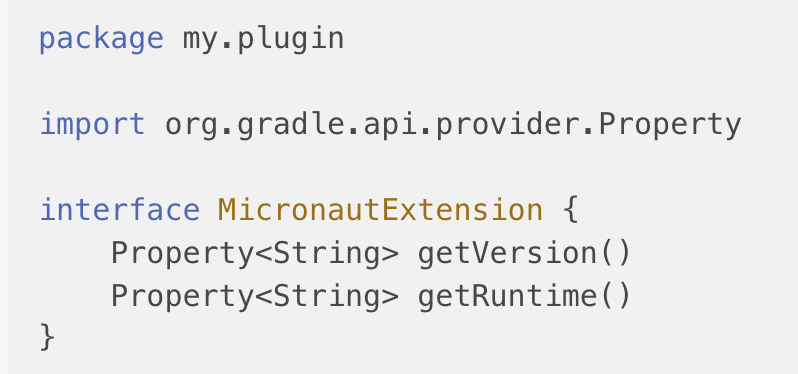


问题是，虽然 afterEvaluate 似乎可以解决问题，但这只是一个肮脏的解决方法，将问题推迟到稍后阶段：取决于应用的插件，它们本身可以使用 afterEvaluate，这个代码块可能会或可能不会看到“最终”配置状态。

为依赖项使用provider

提供覆盖运行时版本的能力是插件的常见要求。例如，默认情况下，checkstyle 插件会按照约定使用 checkstyle 的版本，但如果想使用不同的版本，它仍然允许覆盖该版本。

如果我们想在属性设置为特定值的情况下添加依赖项怎么办？为此，假设我们定义了一个运行时属性，它将告诉使用哪个运行时。让我们将此属性添加到我们的扩展中：



现在让我们更新我们的插件以使用该属性，并根据运行时属性的值添加一个依赖项：



1.如果运行时设置为 netty，则添加依赖项

2.如果运行时设置为 tomcat，则添加依赖项

3.但是如果没有设置运行时什么也不做

我们已经展示了如何添加依赖项并从扩展中定义的版本派生依赖项表示法。然后，我们看到了如何根据扩展的值添加或不添加依赖项：返回支持的依赖项表示法，或者如果不需要添加任何内容，则返回 null。

Gradle 实际上支持更复杂的情况。例如：

- 添加依赖提供程序并配置其丰富版本（请参阅 DependencyHandler#addProvider）。

- 添加依赖项列表，而不是单个依赖项（请参阅 Configuration#getDependencies 和 DependencySet#addAllLater）。

- 计算来自两个或多个提供者的依赖关系（参见 Provider#zip）

1. 插件实现

插件的原理基于Gradle的组合构建原理。什么是组合构建方式？组合构建简单来说就是包含其他构建的构建。通常情况下，组合构建与Gradle多项目构建相同，不同之处在于他不包含单个projects，而是包含完整的builds。

组合构建的好处：

- 结合通常独立开发的构建，例如在应用程序使用的库中尝试修复错误时

- 将大型多项目构建分解为更小、更孤立的块，可以根据需要独立或一起工作

包含在组合构建中的构建很自然地称为包含的构建(“included build”)。包含的构建不与组合构建或其他包含的构建共享任何配置。每个包含的构建都是独立配置和执行的。

包含的构建通过依赖替换（“dependency substitution”）与其他构建交互。如果组合中的任何构建具有包含构建可以满足的依赖项，则该依赖项将被包含构建的项目依赖项替换。由于对依赖替换的依赖，组合构建可能会在组合任务执行图时强制更早地解析配置。这会对整体构建性能产生负面影响，因为这些配置不是并行解决的。

默认情况下，Gradle 将尝试确定可以被包含的构建替换的依赖项。但是，为了获得更大的灵活性，如果 Gradle 确定的默认替换对组合不正确，则可以显式声明这些替换。

除了通过项目依赖使用输出外，复合构建还可以直接在包含的构建上声明任务依赖。包含的构建是隔离的，并且无法声明对复合构建或其他包含构建的任务依赖关系。

实现组合构建：

- --include-build 命令行参数将执行的构建转换为组合，将包含的构建中的依赖项替换为执行的构建。

- 通过使用 Settings.includeBuild(java.lang.Object) 在 settings.gradle（或 Kotlin 中的 settings.gradle.kts）文件中声明包含的构建。设置文件可用于同时添加子项目和包含的构建。包含的构建按位置添加。

- 上述方法的一个缺点是它需要修改现有构建，使其作为独立构建不太有用。避免这种情况的一种方法是定义一个单独的复合构建，其唯一目的是组合其他单独的构建。复合构建必须定义一个委派任务。



包含构建的一个特例是定义 Gradle 插件的构建。这些构建应使用设置文件的 pluginManagement {} 块内的 includeBuild 语句包含在内。使用这种机制，包含的构建还可以提供一个设置插件，该插件可以应用于设置文件本身。



大多数构建都可以包含在组合中，包括其他组合构建。但是有一些限制。每个包含的构建：

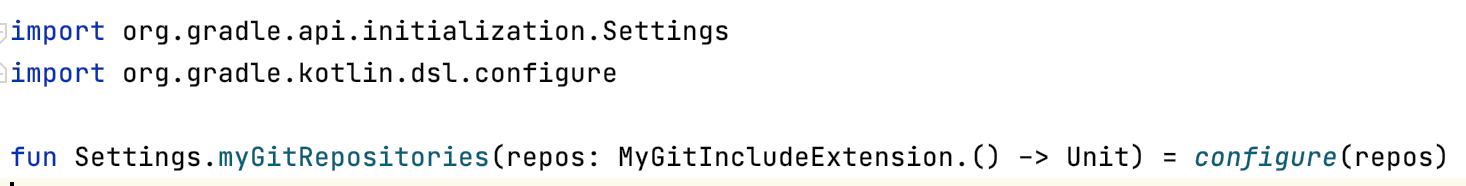
- rootProject.name 不能与另一个包含的构建相同。

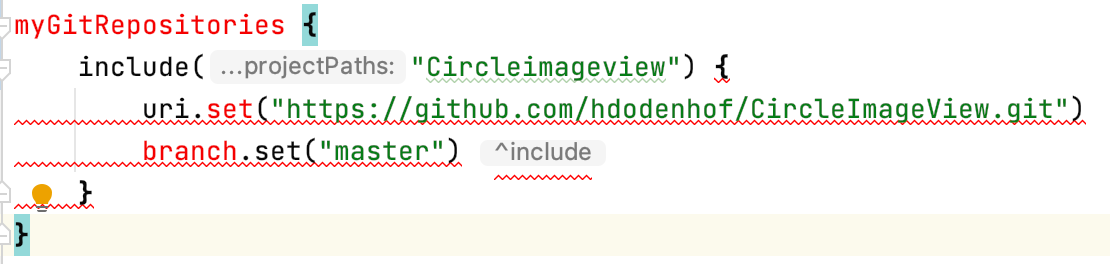
- 不得具有与复合构建的顶级项目相同的 rootProject.name。

- 不得具有与复合构建 rootProject.name 相同的 rootProject.name。

通过组合构建就可以在不改变项目依赖配置的情况下实现远程仓库与本地代码的切换插件。一般情况下，我们在多仓库开发的时候，会在业务模块里面依赖本地仓库的功能模块aar包，如果同时要求改业务模块和功能模块，正常的开发流程是先修改功能模块的代码，然后提交远程仓库，然后本地提交共嗯那个模块的aar包，然后在业务模块中进行功能模块的aar版本更新，这样子其实很繁琐。通过组合构建可以在不改变业务模块依赖配置的情况下，将远程的功能模块最新代码checkout然后做到实时修改实时生效。

如果要开发这种插件，首先要做的是让用户配置需要被组合构建的模块是哪些，因为涉及到源码的修改，肯定需要将远程代码下载到本地，这边通过eclipse git模块在代码中动态将配置的仓库地址、分支、TAG、commit来进行远程代码的拉取。配置的地方就在seetings.gradle文件中设置自定义的Extensions。通过插件里面动态给Settings对象设置伴生方法来调用插件的方法。





include方法里面主要是判断本地源码是否存在，不存在就到git上面clone，存在的话可以根据配置进行拉取更新。