
目錄

关于本书	1.1
Gradle介绍	1.2
Gradle是什么	1.2.1
Gradle的优势	1.2.2
约定优于配置	1.2.3
下载和安装	1.3
先决条件	1.3.1
安装包结构	1.3.2
下载	1.3.3
环境变量配置	1.3.4
测试安装	1.3.5
升级Gradle	1.3.6
快速入门	1.4
项目结构	1.4.1
脚本结构	1.4.2
第一个脚本	1.4.3
打包和运行	1.4.4
仓库	1.5
什么是仓库	1.5.1
为什么要用仓库	1.5.2
仓库的分类	1.5.3
私服nexus的安装与配置	1.5.4
依赖(构件)	1.6
什么是依赖管理	1.6.1
依赖的坐标和范围	1.6.2
依赖的分类与检查	1.6.3
依赖的传递与排除	1.6.4
依赖的更新与缓存	1.6.5
依赖的上传与发布	1.6.6
查看依赖报告和冲突的解决	1.6.7

任务	1.7
运行一个任务	1.7.1
任务的生命周期	1.7.2
任务的创建	1.7.3
依赖型任务	1.7.4
任务的操作	1.7.5
任务的执行和跳过	1.7.6
打包和发布	1.8
什么是构建	1.8.1
增量构建	1.8.2
代码混淆	1.8.3
包含和排除	1.8.4
如何发布	1.8.5
多项目构建	1.9
多模块项目结构	1.9.1
settings文件	1.9.2
build文件	1.9.3
分离化配置	1.9.4
水平布局 and 分层布局	1.9.5
质量保证	1.10
单元测试	1.10.1
findbugs	1.10.2
checkstyle	1.10.3
插件	1.11
如何使用插件	1.11.1
插件编写	1.11.2
参数化配置	1.12
实例解析	1.13
Spring多项目脚本的解读	1.13.1
安卓多渠道打包示例解读	1.13.2
效率优化	1.14
使用Gradle daemon	1.14.1
IDE集成	1.15
地球不会爆炸	1.15.1

与eclipse集成	1.15.2
与idea集成	1.15.3
团队协作	1.16
使用Gradle wrapper	1.16.1
与Jenkins搭配使用	1.17
关于中文编码	1.18
遇到问题怎么办？	1.19
提问技巧	1.19.1
在哪寻求帮助	1.19.2
附1、groovy	1.20
附2、常用插件	1.21
附3、版本管理	1.22

《跟我学Gradle》

本书适合所有程序猿阅读，你可以通过[点击传送门](#)进行在线阅读

在线阅读

关于本书

Gradle是新一代构建工具，从0.x版本一路走来虽然国内可寻的资料多了一些，但都是比较碎片化的知识。官方的Userguide虽然是业内良心之作，但无奈太长，且版本变化较快，又鉴于很多同学一看到英文内心便已认定无法读懂，遂打算利用业余时间攒此本《跟我学gradle》，希望通过此书可以降低学习曲线能让希望使用Gradle的同学更轻易地入门。

p.s：之前也有打算翻译过userguide，鉴于如下几个原因现在几乎已经处于停滞状态。

- gradle版本更新较快且多变化
- 参与人手不足
- 业余时间占用（主要是楼主工作上项目时间的占用）
- ...

书籍目录

你可以点击 [>> \[这里\]](#) [<<](#) 了解本书内容

如何参与

star/fork本项目，加入Gradle中文用户组109752483

授权许可



跟我学Gradle 由 [Shawn·PKAQ](#) 创作，采用 [知识共享 署名-非商业性使用-禁止演绎 4.0 国际许可协议](#)进行许可。

贡献人员

书籍校对

关于我们

欢迎你加入Gradle中文用户组109752483或者给我发送邮件 PKAQ#MSN.COM

捐助我们

很高兴我们能帮到您。无论金额多少，都足够表达您这份心意。谢谢！

捐赠方式

用 [手机支付宝](#) 扫描二维码，捐款到支付宝账号`PKAQ@MSN.COM`



捐赠记录

日期	施主	捐赠金额	寄语
2012-09-19	李施主	1.00	Thx

Gradle是新一代构建工具，从0.x版本一路走来虽然国内可寻的资料多了一些，但都是比较碎片化的知识。官方的Userguide虽然是业内良心之作，但无奈太长，且版本变化较快，又鉴于很多同学一看到英文内心便已认定无法读懂，遂打算利用业余时间攒此本《跟我学gradle》，希望通过此书可以降低学习曲线能让希望使用Gradle的同学更轻易地入门。

Gradle是继Maven之后的新一代构建工具，它采用基于groovy的DSL语言作为脚本，相比传统构建工具通过XML来配置而言，最直观上的感受就是脚本更加的简洁、优雅。如果你之前对Maven有所了解，那么可以很轻易的转换到Gradle，它采用了同Maven一致的目录结构，可以与Maven一样使用Maven中央仓库以及各类仓库的资源，并且Gradle默认也内置了脚本转换命令可以方便的将POM转换为gradle.build。

Gradle的优势

- 依赖管理:即将你项目中的jar包管理起来，你可以使用Maven或者Ivy的远程仓库、或者本地文件系统等
- 编译打包:可以通过脚本实现花样打包，包括修改文件、添加抑或排除某些类或资源、采用指定JDK版本构建、打包后自动上传等
- 多项目支持: Gradle对多项目有着良好的支持，比如一个很具有代表性的实践就是spring framework
- 多语言支持：无论是java、groovy、scala、c++都有良好的支持
- 跨平台支持：gradle是基于jvm的，只要有jvm你就可以让gradle运行
- 灵活的脚本：你可以使用groovy灵活的编写任务完成你想要做的任何事情

约定优于配置(convention over configuration),简单而言就是遵循一定的固定规则从而可以避免额外的配置。虽然这一定程度上降低了灵活性,但却能减少重复的额外配置,同时也可以帮助开发人员遵守一定的规则。当然,约定并不是强制性约束,Gradle提供了各种灵活的途径可以让你更改默认的配置。

本章主要介绍gradle的安装以及环境配置。

.0. 先决条件

- * 翻墙
- * 翻墙
- * 翻墙

1.5以上版本的JDK,Gradle会采用你环境变量中设置的JDK目录(可以用`java -version`进行检查),你需要配置环境变量 `JAVA_HOME` 并将它指向你的JDK安装目录.

p.s:Gradle自带了Groovy库,所以无需事先安装Groovy,所有已经安装的Groovy也将被Gradle忽略.

.2.安装包结构

- 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。

.1.下载

从[Gralde官方网站](#)下载Gradle的最新发行包,下载后解压到任意目录即可（最好不要放在中文以及带有空格的目录中）

.3.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.

.4.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
```

.5.Gradle 升级

如果你采用的非windows系统，那么可以通过 `ln -s gradle-version gradle` 来创建一个软连接，之后直接解压新版本的Gradle替换目录即可，windows下建议初次安装时，将gradle-version文件夹重命名为gradle，后续升级直接清空此目录解压新版本文件到此目录即可，而无需更改环境变量。

欢迎来到第三章,这一章节你将会了解gradle的答题结构以及如何书写第一个gradle脚本并且把它打成war包

本文基于gradle2.12版本

标准结构

Gradle遵循COC(convention over configuration约定优于配置)的理念,默认情况下提供了与maven相同的项目结构配置 大体结构如下

- project root
 - src/main/java(测试)
 - src/main/resources
 - src/test/java(测试源码目录)
 - src/test/resources(测试资源目录)
 - src/main/webapp(web工程)

创建标准结构

好在,Gradle提供了一些[内置初始化任务](#),可以方便的为我们生成默认的目录结构以及示例代码,如下命令会产生如下效果

- 应用 java 插件
- 应用 jcenter() 仓库
- 采用JUnit测试框架
- 创建标准目录结构
- 包含一份示例代码

“java-library使用示例”

```
//创建一个java项目,默认使用Junit测试框架
gradle init --type java-library
//使用spock替代junit
gradle init --type java-library --test-framework spock
//使用testng替代junit
gradle init --type java-library --test-framework testng
```

“scala-library使用示例” `gradle init --type scala-library`

- 应用 scala 插件
- 应用 jcenter() 仓库
- 采用scala2.10
- 应用ScalaTest测试框架
- 采用JUnit测试框架
- 创建标准目录结构
- 包含一份示例代码
- 使用 Zinc Scala 编译器

“groovy-library使用示例” `gradle init --type groovy-library`

- 应用 `groovy` 插件
- 应用 `jcenter()` 仓库
- - 采用`scala2.x`
- 采用`Spock`测试框架
- 创建标准目录结构
- 包含一份示例代码

非标准结构配置

在一些老项目上,可能目录结构并不是标准结构,然而一般开发人员又不好进行结构调整,此时可以通过配置`sourceSet`来指定目录结构

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

或者采用如下写法也是可以的

```
sourceSets {  
    main.java.srcDirs = ['src/java']  
    main.resources.srcDirs = ['src/resources']  
}
```

在android中

```
android {
    sourceSets {
        main {
            manifest.srcFile 'AndroidManifest.xml'
            java.srcDirs = ['src']
            resources.srcDirs = ['src']
            aidl.srcDirs = ['src']
            renderscript.srcDirs = ['src']
            res.srcDirs = ['res']
            assets.srcDirs = ['assets']
        }

        androidTest.setRoot('tests')
    }
}
```

当然如果你的资源目录与源码目录相同这样就比较....了,但你仍然可以按照如下方式搭配 `include` 和 `exclude` 进行指定

```
sourceSets {
    main {
        java {
            //your java source paths and exclusions go here...
        }

        resources {
            srcDir 'main/resources'
            include '**/*.properties'
            include '**/*.png'

            srcDir 'src'
            include '**/Messages*.properties'
            exclude '**/*.java'
        }
    }
}
```

一个简单的Gradle脚本,或许包含如下内容,其中标明可选的都是可以删掉的部分

- 插件引入:声明你所需的插件
- 属性定义(可选):定义扩展属性
- 局部变量(可选):定义局部变量
- 属性修改(可选):指定project自带属性
- 仓库定义:指明要从哪个仓库下载jar包
- 依赖声明:声明项目中需要哪些依赖
- 自定义任务(可选):自定义一些任务

```
//应用插件,这里引入了Gradle的Java插件,此插件提供了Java构建和测试所需的一切。
apply plugin: 'java'
//定义扩展属性(可选)
ext {
    foo="foo"
}
//定义局部变量(可选)
def bar="bar"

//修改项目属性(可选)
group 'pkaq'
version '1.0-SNAPSHOT'

//定义仓库,当然gradle也可以使用各maven库 ivy库 私服 本地文件等,后续章节会详细介绍
repositories {
    jcenter()
}

//定义依赖,这里采用了g:a:v简写方式,加号代表了最新版本
dependencies {
    compile "cn.pkaq:ptj.tiger:+"
}

//自定义任务(可选)
task printFoobar {
    println "${foo}__${bar}"
}
```

一个基础脚本,与上一小节大致相同,为项目加载Tiger依赖

```
//应用插件,这里引入了Gradle的Java插件,此插件提供了Java构建和测试所需的一切。
apply plugin: 'java'

group 'pkaq'
version '1.0-SNAPSHOT'

//定义仓库,当然gradle也可以使用各maven库 ivy库 私服 本地文件等,后续章节会详细介绍
repositories {
    jcenter()
}

//定义依赖,这里采用了g:a:v简写方式,加号代表了最新版本
dependencies {
    def version = "+";
    compile "cn.pkaq:ptj.tiger:+"
}
```

执行下面的命令,可以将你的项目打成一个jar包

```
gradle -q jar
```

这个jar包默认会存放到你项目目录下的buildlibs里

gradle默认是采用增量打包的,每次打包后被打包的文件都会记载一个快照,如果下次打包时没有更改是不会触发重新编译的

欢迎来到第四章,本章将为你讲解仓库相关知识

什么是仓库

顾名思义,仓库就是一个进行集中存储东西的地方,放到这里可以理解为集中管理构件(jar包)的地方.仓库包含了绝大多数流行的开源Java构件,以及源码、作者信息、SCM、信息、许可证信息等.

为什么要用仓库

在采用传统方式管理的项目中,通常会把第三方依赖jar包放到 `./lib` 或者 `./web-inf/lib` 下,这种情况会产生如下几个弊端:

1. 侵占硬盘空间:虽然现在硬盘的容量已经越来越大,但采用传统方式管理,假设100个项目都用到了log4j,那么硬盘上会存在100个log4j.jar,这种重复对于一个正常的程序猿而言显然是不可容忍的.
2. 依赖识别困难:时间久了,往往搞不清楚哪个jar是干嘛的.
3. 依赖管理困难:需要人工维护第三方jar依赖的其它三方jar.
4. 版本升级困难:如果想升级版本,除了要浪费时间手动去各jar的官网寻找文件下载,还要搞清楚找到的jar和它所依赖的三方jar版本是否兼容.

然而现在有了仓库,可以对jar进行集中管理,通过书写简单的脚本,可以让构建工具主动去下载对应版本的jar包,并且可以解析所需的三方依赖.从而可以让你从无意义的体力劳动中解放出来.并且采用仓库之后,即便有1W个项目都使用到了同一个版本的log4j,那么所有的项目引用的都是同一个log4j而不会出现存在1W份相同文件的情况.

这里可以理解为如果用传统方式,那么是通过相对路径引用的当前项目某个目录下的依赖,而采用构建工具后,会使用绝对路径引用固定目录下的同一份文件.

仓库的分类

仓库大体可以分为:本地仓库、中央仓库、本地目录点击后面的连接可以查看,[关于这几种仓库的使用方式](#)

- 本地仓库 即本地硬盘存储目录,默认情况下会在系统的用户目录下创建一个`.gradle`文件夹,所有的依赖都会缓存到此,在依赖检查时,会先检索本地缓存,如果存在那么会直接采用本地缓存目录中存放的依赖,从而节约带宽和时间。 如果想要更改本地缓存目录位置,可以参考[Gradle指定/修改缓存目录](#)通过如下几种方式修改,
 - 法1.通过添加系统变量 `GRADLE_USER_HOME`
 - 法2.设置虚拟机参数 `org.gradle.user.home` 属性
 - 法3.通过命令行`-g`或者 `--gradle-user-home` 参数设置
- 远程仓库 远程仓库又分为以下三种
 - 中央仓库 最核心的中央仓库,其中包含了绝大多数构件(jar包),一般而言java项目的依赖都可以在这找到,比较常用的有 `mavenCenter()` , `jcenter()`
 - 公共仓库 有好多良心的公司搭建了一些公共库可以使用,比如osc的 `osc repo`
 - 私有仓库 这里通常指的是公司内部或者个人搭建的私服,可以通过[Nexus](#)或者[Artifactory](#),搭建私有服务器,采用这种方式不仅可以有效降低外网带宽,还可以加快构建速度,增强稳定性.并且当诸如 `oracle jdbc`驱动这种不存在于中央仓库的jar可以上传到私服而不用采用本地目录的方式进行加载.特别是在条件有限的情况下,诸如开发人员连接外网有限制,公司没有提供VPN且自己又没有梯子的情况下,搭建一台配置VPN的私服绝对是居家旅行,码字构建的必需.要知道在天朝很多情况下没有VPN要么很慢,要么连不上.
- 本地目录 `gradle`可以直接采用本地目录作为仓库加载所需构件,诸如上文提到的想 `oracle jdbc`驱动这种在中央仓库不存在的流氓构件,如果你恰好又没有条件搭建私服的话,可以像传统方式一样放到 `lib` 下直接从本地加载.至于`oracle jdbc`驱动为何不在中央仓库的问题不在本文讨论范围,欲知详情的,自行搜索

Nexus的安装与配置

仅以此文,献给陷入懒癌晚期的小伙伴们.

本文基于nexus 3.xx

.0. What?Why?When?Who?Where?

Sonatype Nexus是一款maven仓库管理软件,有了它,你可以方便的搭建属于自己的maven私服。而通过搭建私服,可以带来几个显而易见的好处。

- 节省外网带宽:以一个30人的项目小队为例,如果没有私服,所有的人在更新依赖时,都将从中央仓库去检查更新,这无疑增加了外网负担。
- 方便内网协作:很多Team工作中会出现没有外网、不允许连接外网或者外网访问受限的情况。这时候总有那么1-2台机器是可以顺畅的连接外网的,如果在外网机器上搭建私服,再配以梯子。无疑方便又高效。
- 组件共享:开发的公共组件,又不想上传到三方仓库怎么办,上传到私服,设定一个坐标便可以像其它依赖一样随时引用了。

.1.下载

下载地址:<http://www.sonatype.com/download-oss-sonatype>

一般官方会提供两个包,一个安装板和解压版。可以根据自己的喜好选择喜欢的版本下载。这里我选择的是解压版的进行安装。 本文采用的是3.X版本,3.x版本增加了对docker、npm等仓库的支持,但遗憾的是暂时不支持手工upload构件,据说这个功能会在3.1之后支持。

.2.安装&配置

[官方文档] (<http://books.sonatype.com/nexus-book/3.0/reference/install.html>) 先决条件:配置好了JDK,JDK的配置不在本文讨论范围,不赘述。启动 解压下载的压缩包,打开命令行跳到指定解压目录下的bin目录下,执行 `./nexus /run` (windows执行 `nexus.exe /run`),然后经过一段时间的等待,当你发现控制台打印出 `Started Nexus Repository Manager 3.0.1-01` 这句话时,恭喜,Nexus已经安装成功。 访问 : <http://localhost:8081>,可以进入仓库页面。默认的用户名密码是admin/admin123,运行后会自动生成一个nexus工作目录sonatype-work,nexus下载的jar包会存放在sonatype-work/nexus/storage中。

停止 `Ctrl+c` 即可

修改存储路径

可以通过修改 `bin/nexus.vmoptions` 文件修改数据存储路径、缓存路径等。

修改端口、ip

如果要修改IP、端口等可以通过修改 `etc/org.sonatype.nexus.cfg` 文件进行修改。

仓库 nexus的仓库类型分为以下几种：

- group: 仓库组
- hosted：宿主仓库
- proxy：代理仓库

点击页面顶端的齿轮按钮->点击repositories,可以看到nexus已经预置了7个仓库，可以点击create repositories添加其他的仓库,比如

- JBOSS的两个：
<http://repository.jboss.org/maven2>
<http://repository.jboss.org/nexus/content/repositories/releases>
- spring的：
<https://repo.spring.io/libs-release>

.3.使用

使用私服 点击页面顶端的齿轮按钮->点击repositories,在列表中,找到需要的仓库,复制url内容,添加到gradle脚本中.

```
repositories {  
    maven { url "http://localhost:8081/repository/maven-public/" }  
    maven { url "http://localhost:8081/repository/spring-public/" }  
    jcenter()  
    mavenCentral()  
}
```

发布到私服 修改 `gradle.properties`

```
nexusUrl=http://localhost:8081  
nexusUsername=admin  
nexusPassword=admin123
```

使用maven插件进行上传

```
uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: "${nexusUrl}/repository/maven-releases/") {
                authentication(userName: nexusUsername, password: nexusPassword)
            }
            snapshotRepository(url: "${nexusUrl}/repository/maven-snapshots") {
                authentication(userName: nexusUsername, password: nexusPassword)
            }
        }
    }
}
```

手动发布

以下是2.x版本的上传方式，3.x版本据说要等到3.1以后才支持。

在nexus的仓库列表中点击要部署的目标仓库，然后点击Artifact Upload选项卡,在Gav definition选择,Gav parameters，然后在下面的三个输入框里输入group、artifactID、version等相关信息，然后点击select artifact to upload上传即可。

本章针对依赖进行讲解，主要包括如下内容

- 什么是依赖
- 依赖的坐标与范围
- 依赖的分类
- 依赖的传递与排除
- 依赖的更新与缓存
- 构件的上传与发布
- 依赖冲突的解决

什么是依赖管理

通常而言，依赖管理包括两部分，对依赖的管理以及发布物的管理；依赖是指构建项目所需的构件(jar包等)。例如，对于一个应用了spring普通的java web项目而言，spring相关jar包即项目所需的依赖。发布物，则是指项目产出的需要上传的项目产物。

传统依赖管理的痛点

毋庸置疑，几乎绝大多数项目都会或多或少的依赖于第三方库，尤其像log4j、dom4j、common-lang、guava这类的三方库。传统情况下我们通常会去网络上自己搜寻这些依赖并把他们下载到WEB-INF/lib下来维护和管理这些三方类库。然而随着项目的推进和时间的推移以及人员的更迭，维护和管理这些依赖就会逐渐变成一件令人头疼的事情。

- 想要知道每个jar文件的作用，以及每个jar文件依赖的其它类库，或者升级更新某个jar文件的版本。
- jar文件会随着项目源代码一起提交到版本控制系统，如果n个项目同时用了log4j那么自己的磁盘以及版本控制系统即存在n份相同的文件，额外占据了大量存储空间。

自动化依赖管理

不得不说maven的出现使得开发人员从头疼的依赖管理工作中解放了出来，它通过XML的形式来声明项目所需的依赖以及依赖生效的范围，只需一个简单的命令便会自动去帮你寻找和下载所需的依赖以及依赖所需的依赖（即传递性依赖），然而如果一个项目依赖众多并且需要一些额外的处理工作，那么一份冗长的XML配置文件仍然会让你觉得头疼不已。

Gradle可以说很大程度上完成了maven可以做的所有工作，并且通过灵活的DSL配置还可以完成更多。当然不同于XML的繁琐，Gradle采用更简洁的方式来配置所需的依赖，如以下列举的maven与gradle声明依赖的代码即可看出Gradle是有多么的优雅和简洁。采用maven配置所需依赖

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-core</artifactId>
  <version>4.3.0.RELEASE</version>
</dependency>
```

采用gradle配置所需依赖

```
compile 'org.springframework:spring-core:4.3.0.RELEASE'
```

是的，你只需要一行代码即可完成maven用3行代码完成的事情（当然如果你要说去掉换行也可以达到一行效果的话，那我就无言以对了）

自动化依赖管理带来的优势

可以看到，采用自动化依赖管理后，我们只需要在脚本文件里按照相应的规范书写依赖标识即可轻松完成依赖的管理。

- 构建工具会依据你所配置的仓库去搜寻所需依赖，并将这些依赖缓存到本地便于其它项目使用，对，便于其它项目使用，也就是说完全相同的依赖你的磁盘只需要存在一份就可以了。
- 采用自动化依赖管理带来的好处就是，无需关心依赖所需的其它依赖（即传递性依赖），比如hibernate-core又依赖了dom4j，那么无需配置dom4j，构建工具会自动发现传递性依赖并将这些传递性依赖一并下载到本地磁盘。
- 采用自动化依赖管理，可以方便的对依赖版本进行管理，需要升级的时候只需要改一下脚本中的版本号即可，也可以通过定义版本号变量的形式，使多个依赖引用这一变量，做到版本号的统一控制。

比如下面两个示例 仅需很少的代码即可替代传统方式大量的工作量。采用变量统一控制版本号

```
dependencies {
    def bootVersion = "1.3.5.RELEASE"
    compile      "org.springframework.boot:spring-boot-starter-web:${bootVersion}",
                 "org.springframework.boot:spring-boot-starter-data-jpa:${bootVersion}"
    ,
                 "org.springframework.boot:spring-boot-starter-tomcat:${bootVersion}"
}
```

自动获取最新版本依赖 如果你想某个库每次构建时都检查是否有新版本，那么可以采用 + 来让Gradle在每次构建时都检查并应用最新版本的依赖。当然也可以采用 1.x , 2.x 的方式来获取某个大版本下的最新版本。

```
dependencies {
    compile      "org.springframework.boot:spring-boot-starter-web:+"
}
```

依赖的坐标

仓库中构件（jar包）的坐标是由 `configurationName` `"group:name:version:classifier@extension"` 组成的字符串构成，如同Maven中的 GAV 坐标，Gradle可借由此来定位你想搜寻的 jar 包。在gradle中可以通过以下方式来说明依赖：

```
testCompile group: 'junit', name: 'junit', version: '4.0'
```

项目	描述
configurationName	依赖的作用范围，具体介绍看本章第二小节
group	通常用来描述组织、公司、团队或者其它有象征代表意义的名字，比如阿里就是 <code>com.alibaba</code> ，一个group下一般会有多个 artifact
name	依赖的名称，或者更直接来讲叫包名、模块、构件名、发布物名以及随便你怎么称呼。 <code>druid</code> 就是 <code>com.alibaba</code> 下的一个连接池库的名称
version	见名知意，无它，版本号。
classifier	类库版本，在前三项相同的情况下，如果目标依赖还存在对应不同JDK版本的版本，可以通过此属性指明
extension	依赖的归档类型，如 <code>aar</code> 、 <code>jar</code> 等，默认不指定的话是 <code>jar</code>

这是由于Gradle依赖配置支持多种书写方式，采用map或者字符串。

```
// 采用map描述依赖
testCompile group: 'junit', name: 'junit', version: '4.0'
```

```
// 采用字符串方式描述依赖
testCompile 'junit:junit:4.0'
```

显然采用字符串的方式更加简单直观，当然借助 `groovy` 语言强大的 `GString` 还可以对版本号进行抽离。如下面的示例，这里需要注意的是如果要用 `GString` 的话，依赖描述的字符串要用 `""` 双引号包起来才会生效。

```
def ver = "4.0"
testCompile "junit:junit:${ver}"
```

依赖的范围

上面的例子中采用的 `testCompile` 是声明依赖的作用范围，关于各种作用范围的功效可见下表。

名称	说明
compileOnly	gradle2.12之后版本新添加的,2.12版本时期曾短暂的叫provided,后续版本已经改成了compileOnly,由java插件提供,适用于编译期需要而不需要打包的情况
providedCompile	war插件提供的范围类型:与compile作用类似,但不会被添加到最终的war包中这是由于编译、测试阶段代码需要依赖此类jar包,而运行阶段容器已经提供了相应的支持,所以无需将这些文件打入到war包中了;例如Servlet API就是一个很明显的例子.
compile	编译范围依赖在所有的classpath中可用,同时它们也会被打包。
providedRuntime	同proiveCompile类似。
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
archives	-
default	配置默认依赖范围

依赖的分类

类型	描述
外部依赖	依赖存放于外部仓库中,如 <code>jcenter</code> , <code>mavenCentral</code> 等仓库提供的依赖
项目依赖	依赖于其它项目(模块)的依赖
文件依赖	依赖存放在本地文件系统中,基于本地文件系统获取依赖
内置依赖	跟随Gradle发行包或者基于Gradle API的一些依赖,通常在插件开发时使用
子模块依赖	还没搞清楚是什么鬼

外部依赖

可以通过如下方式声明外部依赖，Gradle支持通过`map`方式或者 `g:a:v` 的简写方式传入依赖描述，这些声明依赖会去配置的 `repository` 查找。

```
dependencies {  
    // 采用map方式传入单个  
    compile group: 'commons-lang', name: 'commons-lang', version: '2.6'  
    // 采用map方式传入多个  
    compile(  
        [group: 'org.springframework', name: 'spring-core', version: '2.5'],  
        [group: 'org.springframework', name: 'spring-aop', version: '2.5']  
    )  
    // 采用简写方式声明  
    compile 'org.projectlombok:lombok:1.16.10'  
    // 采用简写方式传入多个  
    compile 'org.springframework:spring-core:2.5',  
            'org.springframework:spring-aop:2.5'  
}
```

项目依赖

此类依赖多见于多模块项目，书写方式如下，其中 `:` 是基于跟项目的相对路径描述符。

```
compile project(':project-foo')
```

文件依赖

依赖存在于本地文件系统中，举个栗子，如oracle的OJDBC驱动，中央仓库中没有又没有自建私服此时需要放到项目lib下进行手工加载那么便可采用此种方式，可以通过 `FileCollection` 接口及其子接口提供的方法加载这些依赖(支持文件通配符)

```
dependencies {
    // 指定多个依赖
    compile files('hibernate.jar', 'libs/spring.jar')

    // 读取lib文件夹下的全部文件作为项目依赖
    compile fileTree('libs')

    // 根据指定基准目录\包含\排除条件加载依赖
    compile fileTree(dir:'libs',include:'spring*.jar',exclude:'hibernate*.jar')
}
```

内置依赖

跟随Gradle发行包或者基于Gradle API的一些依赖，通常在插件开发时使用，当前提供了如下三种

```
dependencies {
    // 加载Gradle自带的groovy作为依赖
    compile localGroovy()

    // 使用Gradle API作为依赖
    compile gradleApi()

    //使用 Gradle test-kit API 作为依赖
    testCompile gradleTestKit()
}
```

子模块依赖

简单来说就是声明 依赖的依赖 或者 依赖的传递依赖，一般情况下如果依赖的库并未用构建工具构建（尤其是一些上古时代的老库），那么Gradle是无法透过源文件去查找该库的传递性依赖的，通常而言，一个模块采用XML(POM文件)来描述库的元数据和它的传递性依赖。Gradle可以借由此方式提供相同的能力，当然这种方式也会可以改写原有的传递性依赖。这里让 `druid` 连接池依赖了 `ptj.tiger` 的一个库。

```
dependencies {  
    // 让ptj.tiger作为druid的传递性依赖  
    compile module("com.alibaba:druid:1.0.26") {  
        dependency("cn.pkaq:ptj.tiger:+")  
    }  
  
    runtime module("org.codehaus.groovy:groovy:2.4.7") {  
        // 停用groovy依赖的commons-cli库的依赖传递  
        dependency("commons-cli:commons-cli:1.0") {  
            transitive = false  
        }  
        // 让groovy依赖的ant模块的依赖ant-launcher停用传递性依赖并依赖ant-junit.....  
        module(group: 'org.apache.ant', name: 'ant', version: '1.9.6') {  
            dependencies "org.apache.ant:ant-launcher:1.9.6@jar",  
                         "org.apache.ant:ant-junit:1.9.6"  
        }  
    }  
}
```

更新依赖

在执行build、compile等任务时会解析项目配置的依赖并按照配置的仓库去搜寻下载这些依赖。默认情况下，Gradle会依照Gradle缓存->你配置的仓库的顺序依次搜寻这些依赖，并且一旦找到就会停止搜索。如果想要忽略本地缓存每次都进行远程检索可以通过在执行命令时添加 `--refresh-dependencies` 参数来强制刷新依赖。

```
gradle build --refresh-dependencies
```

当远程仓库上传了相同版本依赖时，有时需要为缓存指定一个时效去检查远程仓库的依赖笨版本，Gradle提供了 `cacheChangingModulesFor(int, java.util.concurrent.TimeUnit)` , `cacheDynamicVersionsFor(int, java.util.concurrent.TimeUnit)` 两个方法来设置缓存的时效

```
configurations.all {
    //每隔24小时检查远程依赖是否存在更新
    resolutionStrategy.cacheChangingModulesFor 24, 'hours'
    //每隔10分钟..
    //resolutionStrategy.cacheChangingModulesFor 10, 'minutes'
    // 采用动态版本声明的依赖缓存10分钟
    resolutionStrategy.cacheDynamicVersionsFor 10*60, 'seconds'
}

dependencies {
    // 添加changing: true
    compile group: "group", name: "module", version: "1.1-SNAPSHOT", changing: true
    //简写方式
    //compile('group:module:1.1-SNAPSHOT') { changing = true }
}
```

缓存管理

缓存位置管理

Gradle在按照配置的仓库去搜寻下载依赖时，下载的依赖默认会缓存到 `USER_HOME/.gradle/caches` 目录下，当然也可以手工修改这个位置。具体可以参考如下三种方式：

- 通过添加系统变量 `GRADLE_USER_HOME`
- 设置虚拟机参数 `org.gradle.user.home` 属性
- 通过命令行 `-g` 或者 `--gradle-user-home` 参数设置

离线模式（总是采用缓存内容）

Gradle提供了一种离线模式，可以让你构建时总是采用缓存的内容而无需去联网检查，如果你并未采用动态版本特性且可以确保项目中依赖的版本都已经缓存到了本地，这无疑是提高构建速度的一个好选择。开启离线模式只需要在执行命令时候添加 `--offline` 参数即可。当然，采用这种模式的也是有代价的，如果缓存中搜寻不到所需依赖会导致构建失败。

```
gradle build --offline
```


借助 `maven-publish` 插件可以轻松地将 `jar` 包发布到仓库中。这个过程没啥幺蛾子直接上代码吧。了解更多配置可以查看 [Maven plugin插件章节](#)

```
apply plugin: 'maven-publish'
apply plugin: 'java'

// 打包源文件
task sourceJar(type: Jar) {
    from sourceSets.main.allSource
    classifier = 'sources'
}

task javadocJar(type: Jar, dependsOn: javadoc) {
    classifier = 'javadoc'
    from javadoc.destinationDir
}

publishing {
    // 目标仓库
    repositories {
        maven {
            url "xxxx"
        }
    }

    publications {
        mavenJava(MavenPublication) {
            // 设置gav属性
            groupId 'org.pkaq'
            artifactId 'tiger'
            version '1.1'

            from components.java
            artifact sourceJar

            // 设置pom相关信息
            pom.withXml {
                Node root = asNode()
                root.appendNode('description', 'bazinga!')
            }
        }
    }
}

//生成一个元的pom文件
model {
    tasks.generatePomFileForMavenJavaPublication {
        destination = file("${buildDir}/generated-pom.xml")
    }
}
```


本小节示例脚本

```
apply plugin: "java"

ext {
    bootVersion = "1.4.2.RELEASE"
    tomcat_embed = "8.5.4"
}

repositories {
    maven { url "https://repo.spring.io/libs-release" }
    jcenter()
}

dependencies {
    compile "org.springframework.boot:spring-boot-starter-web:${bootVersion}",
            "org.apache.tomcat.embed:tomcat-embed-jasper:${tomcat_embed}"
}
```

检查依赖

在引用的依赖或传递性依赖存在版本冲突时，**Gradle**采用的策略是优先选取最新的依赖版本解决版本冲突问题。解决此类问题我们可以通过上一章节中介绍的各种依赖管理方法进行排除、强制指定一个版本或者干脆禁用依赖传递特性解决。但如何知道哪些依赖传递了哪些子依赖，哪些传递的依赖又被**Gradle**进行了 隐性升级 呢。采用下面的命令可以查看各个范围的依赖树。

```
gradle dependencies > dep.log
```

输出结果:

dependencies

Root project

archives - Configuration for archive artifacts.

No dependencies

compile - Dependencies for source set 'main'.

```
+--- org.springframework.boot:spring-boot-starter-web:1.4.2.RELEASE
|    +--- org.springframework.boot:spring-boot-starter:1.4.2.RELEASE
|         +--- org.springframework.boot:spring-boot:1.4.2.RELEASE
|              +--- org.springframework:spring-core:4.3.4.RELEASE
|                   \--- commons-logging:commons-logging:1.2
|                   \--- org.springframework:spring-context:4.3.4.RELEASE
|                        +--- org.springframework:spring-aop:4.3.4.RELEASE
|                             +--- org.springframework:spring-beans:4.3.4.RELEASE
|                                 | \--- org.springframework:spring-core:4.3.4.RELEASE (*)
|                                 | \--- org.springframework:spring-core:4.3.4.RELEASE (*)
|                                 +--- org.springframework:spring-beans:4.3.4.RELEASE (*)
|                                 +--- org.springframework:spring-core:4.3.4.RELEASE (*)
|                                 \--- org.springframework:spring-expression:4.3.4.RELEASE
....
....
省略的
....
....
\--- org.apache.tomcat.embed:tomcat-embed-jasper:8.5.4
     +--- org.apache.tomcat.embed:tomcat-embed-core:8.5.4 -> 8.5.6
     +--- org.apache.tomcat.embed:tomcat-embed-el:8.5.4 -> 8.5.6
     \--- org.eclipse.jdt.core.compiler:ecj:4.5.1
```

后面 `dep.log` 文件名可以随意，然而，你一定在想为什么有些带了 `(*)` 有的带了 `->` 有的什么都没有呢，这是什么鬼。前面已经说过，当发生版本冲突时 **Gradle** 会采用最新版本解决。仔细观察带了 `(*)` 的依赖你会发现这些依赖被不同的库重复依赖了若干次，这里 `(*)` 的意思即是表示该依赖被忽略掉了。而 `->` 则表示其它的定级依赖的传递依赖中存在更高版本的依赖，该版本将会使用 `->` 后面的版本来替代。

反向查找

如果你想知道某个依赖到底被哪个库引用过，可以采用下面的命令进行反向查找

```
gradle dependencyInsight --dependency tomcat-embed-core > reverse.log
```

```

:dependencyInsight
org.apache.tomcat.embed:tomcat-embed-core:8.5.6 (conflict resolution)
+--- org.apache.tomcat.embed:tomcat-embed-websocket:8.5.6
|    \--- org.springframework.boot:spring-boot-starter-tomcat:1.4.2.RELEASE
|         \--- org.springframework.boot:spring-boot-starter-web:1.4.2.RELEASE
|              \--- compile
\--- org.springframework.boot:spring-boot-starter-tomcat:1.4.2.RELEASE (*)

org.apache.tomcat.embed:tomcat-embed-core:8.5.4 -> 8.5.6
\--- org.apache.tomcat.embed:tomcat-embed-jasper:8.5.4
      \--- compile

(*) - dependencies omitted (listed previously)

BUILD SUCCESSFUL

Total time: 6.936 secs

```

上面的报告中可以看到 8.5.6 这个版本后面标注了 (conflict resolution) 说明了该版本是用于解决冲突选用的版本。

冲突即停

Gradle默认采用自动升级版本的方式解决依赖冲突，有时这种 隐式升级 可能会带来一些不必要的麻烦，此时我们可以通过更改这种默认行为来让Gradle发现版本冲突时立即停止构建并抛出错误信息。更改脚本：

```

configurations.all {
    resolutionStrategy {
        failOnVersionConflict()
    }
}

```

执行 `gradle build` 的输出结果：

```

:compileJava FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Could not resolve all dependencies for configuration ':compileClasspath'.
> A conflict was found between the following modules:
    - org.apache.tomcat.embed:tomcat-embed-core:8.5.4
    - org.apache.tomcat.embed:tomcat-embed-core:8.5.6

```

可以看到在执行`gradle build`时由于`tomcat-embed-core`存在版本冲突导致了构建失败，并提示了冲突的两个版本。

依赖报告

Gradle官方提供了一个名叫 `project-report` 插件可以让依赖查看更加简单方便，要用此插件只需要在脚本中添加 `apply plugin: 'project-report'` 即可。该插件提供的任务可以参考下面的表格，所有输出结果将被放在 `build/report` 下。

任务名称	描述
<code>dependencyReport</code>	将项目依赖情况输出到txt文件中 功能同 <code>gradle dependencies</code> <code>> build/dependencies.txt</code>
<code>htmlDependencyReport</code>	生成HTML版本的依赖情况
<code>propertyReport</code>	生成项目属性报告
<code>taskReport</code>	生成项目任务报告
<code>projectReport</code>	生成项目报告,包括前四个

title: 多模块项目的构建

随着项目规模的扩展和功能的增加，任何一个简单的项目基本都会进行模块的拆分独立，然后通过项目之间的依赖进行重新整合。

项目拆分带来的好处也显而易见。拆分可以是项目结构更加清晰，职责划分更加明确，相关模块的负责人只需专注于自己负责的子模块而无需对整个项目进行一次构建。

如果不采用构建工具或许最原始的方式无非于将一些相对独立的和公共的模块代码拆分成独立项目，通过手工打jar包替换的方式进行多项目之间的依赖管理，这无疑是低效、繁琐且没有价值的工作。

Gradle对多模块项目有着优秀的支持，现在借助于Gradle可以把你从一些低价值的劳动中解放出来。你可以肆意的采用水平或者分层的方式组织你的项目模块，肆意的进行模块之间的依赖，甚至是模块之间深层次的依赖。举个栗子，如果有个web工程依赖于一个 util 工程和一个 common-service 工程，那么当 util 和 common-service 进行了任何修改，在web工程进行打包的时候会自动编译打包依赖的 util 和 common-service 工程，如果你采用了合理的 依赖声明 那么在发布war的时候这两个工程的会被自动打成 jar 包打进最终的 war 中。这一章节你将会了解到如下内容

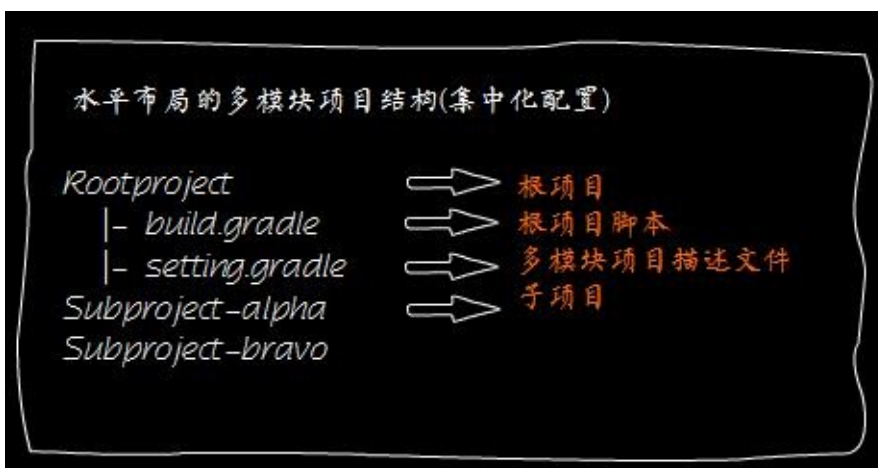
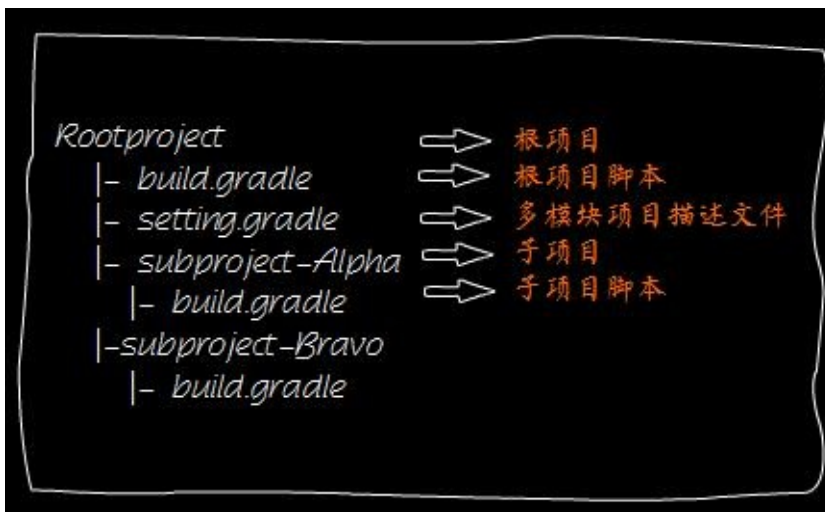
- 多模块项目结构
- 多模块项目脚本
- 构建分层布局的多模块项目
- 构建集中化配置的多模块项目

title: 多模块项目结构

在进行介绍之前，这里有必要先解释一下项目（project）和模块（module）的概念。Gradle中默认是不存在模块概念的，这里指的多模块项目构建对应到Gradle中即是多项目构建，为了便于理解，所以本文中一律按多模块进行描述。

关于多项目和多模块的分歧无法溯源，或许是由于 eclipse 和 idea 不同定义而起吧。在 eclipse 中不存在模块概念，走的是 workspace - working set - project 路线，而在 idea 中，则是按 project - module 进行组织项目。自从转为 idea 党之后，私以为 idea 的路子更科学，所以这里按 idea 的套路进行描述。

下面展示了分层布局和水平布局的两种多模块项目结构



为了更加直观，这里项目结构采用了分层布局的方式。这也是Gradle默认的布局方式。这里根项目是必不可少的，尽管在分层布局中它看上去更像是一个顶级的 working set。其中根项目下的 build.gradle 文件可以对一些公共属性、插件、依赖等进行统一配置， setting.gradle 则是描述项目关系的文件。

或许你已经注意到，在水平布局的示例图中子项目下没有 `build.gradle` 脚本文件,这是由于在构建多模块项目时 **Gradle** 允许你自由的选用集中化配置和分开式配置。在水平布局的结构图中，所画的即是采用集中化配置的结构。

title: 多模块项目脚本结构-settings文件

我们了解到一个典型的多模块项目需要有一个根模块项目以及模块描述文件(默认为 `settings.gradle`)，下面我们将通过一个 [多项目示例](#) 来对多模块项目的结构以及脚本进行更深入的了解。

如果要想让 `Gradle` 支持多构建，只需为你的项目确定一个根模块项目 并且在根模块项目下添加 `settings.gradle` 文件用以描述模块（项目）关系

```
include 'base','main'
```

这里将模块（项目）名称路径的字符串以数组的形式传递给了 `include` 函数，`Gradle` 会以相对于当前目录按照 `include` 给定的模块（项目）路径查找对应的子模块，如果要声明的模块（项目）有多个层次可以用 `:` 进行描述，假设 `main` 下面又分了 `Alpha` , `Bravo` 那么声明方式则按如下方式书写

```
include 'base','main:Alpha','main:Bravo'
```

这里的 `settings` 文件实际上是对应的 `Settings` 接口，`Settings` 接口中提供的函数在本脚本中都是可用的，具体细节请查阅 [>Settings接口DSL<](#) 文档进行了解。

在构建的初始化阶段，`Gradle` 会读取这个文件，并创建一个 `Settings` 类型的实例。`Gradle` 会依据此文件进行多模块项目构建，默认情况下 `Gradle` 会从同级的 `master` 目录下寻找此文件，如果未找到则会去父级目录寻找。如果搜寻不到 `settings` 文件，那么 `Gradle` 会把模块当做单独构建的项目去执行单独构建。这里 `Gradle` 提供了一个命令行参数 `-u` 或 `--no-search-upward` 来控制 `Gradle` 不去父目录搜寻 `settings` 文件。

默认情况下，`Gradle` 会采用 `settings.gradle` 作为文件名去查找，但如果处于某种不可描述的原因要采用其它名称的话也是可以的，调用命令时可以通过 `-c` 或者 `--settings-file` 参数来指定 `settings` 文件的位置以及名称。

tip: 通常情况下，不存在孤立的子模块，所以构建的执行顺序往往是由依赖顺序所决定的。但如果无法通过依赖顺序决定，那么 `Gradle` 会简单的按照首字母顺序决定构建顺序。

title: 多模块项目脚本结构-build文件

在根项目下，除了 settings.gradle 文件之外，通常还需要提供一个 build.gradle 文件，该文件用以定义子模块行为以及描述项目的一些公共插件、属性、依赖等。

下面的示例中，定义了所有模块共享的group、版本号，所有子模块共享的插件，以及针对 main 项目的定制化配置。

```
// 所有模块都采用统一的版本号以及groupName
allprojects {
    group = 'org.pkaq.gradle.multi'
    version = "0.1.0"
}

// 为所有子模块都应用java插件
subprojects {
    apply plugin: 'java'
}

//为main模块定义特定行为,采用war插件并且依赖base模块
project(':main'){

    apply plugin: 'war'

    dependencies {
        compile project(':base')
    }

}
```

如果您阅读了上一小节， settings 文件是对 Settings 接口的脚本化编程实现，那么此处同理，借由 Project API，可对模块的行为进行定制。

方法名	描述
allprojects	配置当前模块以及所有子模块行为
subprojects	配置所有子模块行为
project	配置指定子模块行为

可以查阅 [> Project DSL <](#) 了解 Project 接口的更多操作。

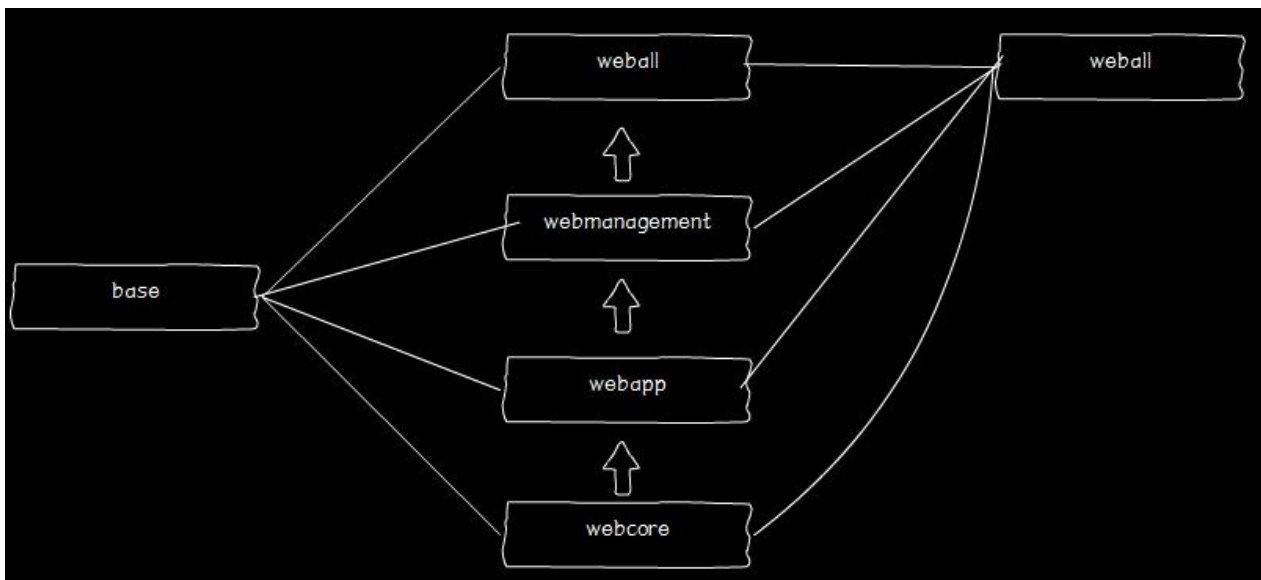
此时可以通过执行 gradle build 来进行构建，这会按照依赖的顺序构建所有子模块，如果要单独构建某个子模块那么可以参照 gradle :main:build 的方式进行单独构建，正如你所见用：分隔项目和 task 即可。

title: 多模块项目的构建-配置分离

当项目足够复杂的时候，采用集中化配置显然不是一个好计谋，尤其是如果你有几十个模块的时候，当你打开一个几百行的脚本无疑会感到一种不由自主的恶心在背后拍拍你的肩膀说："兄弟 吐吧".... 此时，将不同项目的定制行为分离到各自的脚本里无疑会显著的减轻你的饱腹感。这不仅可以是项目结构看起来更加清晰，脚本更加易读，分离化的配置还可以让你更好的关注子模块的特定行为。 要分离模块配置只需要很简单的两步：

1. 在子模块建立 `build.gradle` 文件
2. 将原来的 `project(':xxx'){}` 内的脚本移动到上面建立的文件中去

下面的图展示了一个稍微复杂一点的 分层布局的 、 多模块的 、 分离配置的 的多模块项目示例，其中所有的二级模块都依赖 `base` ，最终的 `weball` 又依赖于所有的二级模块。具体代码可以查看 [> 就是这里 <](#)



title: 多模块项目的构建-水平布局和分层布局

如果你阅读了前面的内容，那么你已经掌握了分层布局。没错，Gradle 默认即是采用的分层布局。只需要正确的配置 `settings` 文件即可。某些情况下，若是出于一些不可描述的原因需要进行水平布局，那么 Gradle 也提供了良好的支持。如果你之前曾经翻阅过 `Settings` 接口的手册，那么或许你已经发现 `Settings` 接口提供了一个 `includeFlat` 方法，借助此方法即可实现水平布局的多模块项目。

```
includeFlat 'base'
```

由于此处 `main` 即是根项目，所以无需再包含 `main`

完整示例 -> [> 水平布局的多模块项目示例 <](#)

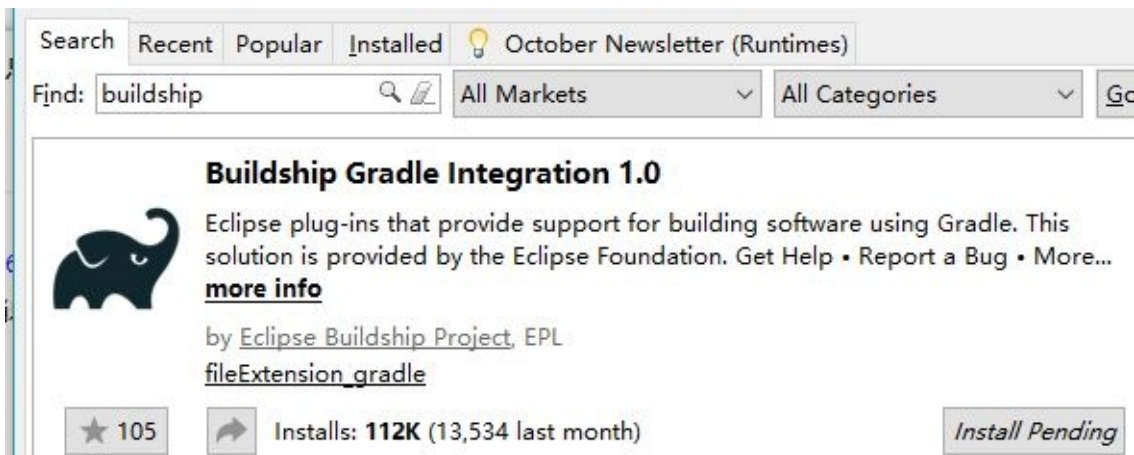
心法在于两点:

1.根项目与被依赖的项目(在水平意义上的子项目,视觉上根项目的兄弟项目)保持平级。 2.配置根项目的 `setting.gradle`，采用 `includeFlat` 来描述子项目路径(由于采用的是水平布局,默认根路径就是当前根项目的上级路径,所以无需用 `../` 上跳)

Eclipse中Gradle插件的使用

安装

无论你使用的是 eclipse/sts/myeclipse 还是任何eclipse的变种,可以通过点击 `Help->eclipse marketplace` 去插件市场搜索buildship然后点击 `install` 进行安装,此过程完全傻瓜操作,不赘述.



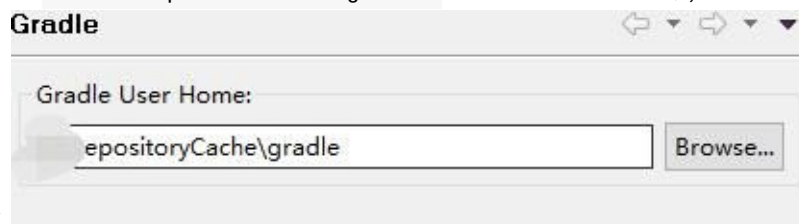
如果你是一名传统程序猿习惯通过site安装,可以参考下面的表格通过点击 `Help->Install New Software` 进行手动安装

Eclipse Version	Type	Update Site
Neon (4.6)	release	http://download.eclipse.org/buildship/updates/e46/releases/1.0
	milestone	http://download.eclipse.org/buildship/updates/e46/milestones/1.0
	snapshot	http://download.eclipse.org/buildship/updates/e46/snapshots/1.0
Mars (4.5)	release	http://download.eclipse.org/buildship/updates/e45/releases/1.0
	milestone	http://download.eclipse.org/buildship/updates/e45/milestones/1.0
	snapshot	http://download.eclipse.org/buildship/updates/e45/snapshots/1.0

... 更早版本可以参考上面两条自己修改地址 具体方式只需要把你所需的eclipse版本中间的小数点去掉即可 修改e45/e46参数.

配置

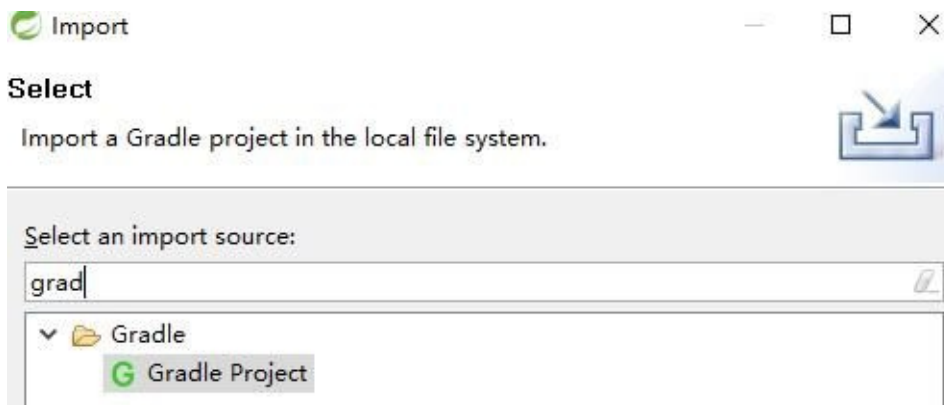
安装完毕后,点击 Window->preferences->gradle 进行一些配置操作,其实就一个配置 就是配

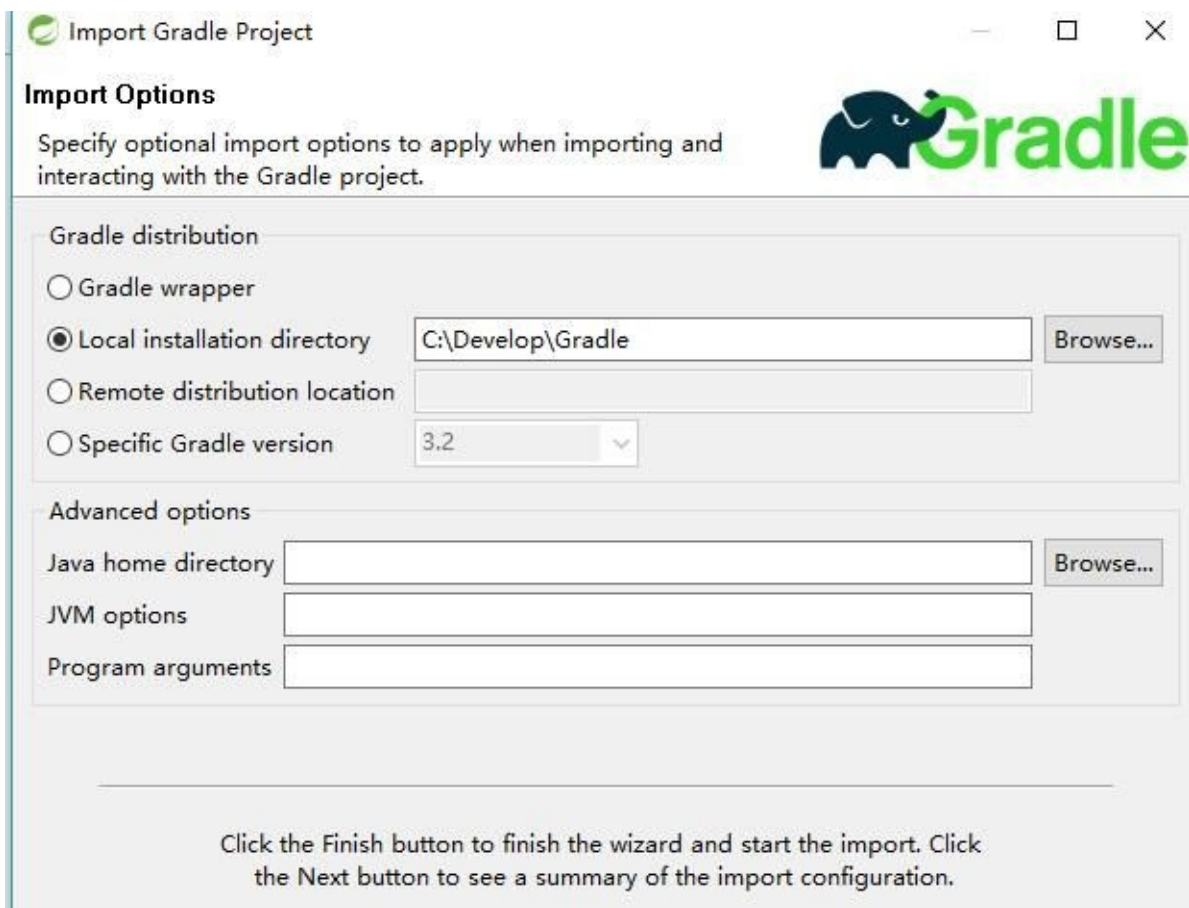


置下本地缓存目录

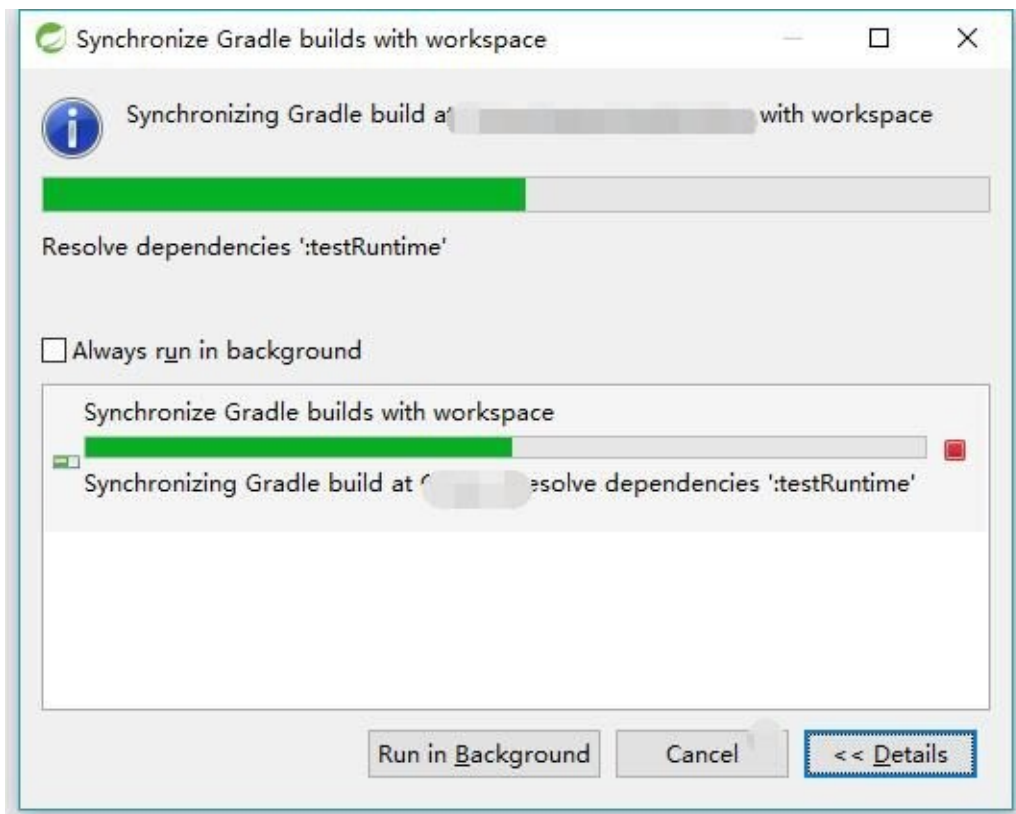
导入

选择Gradle项目进行导入



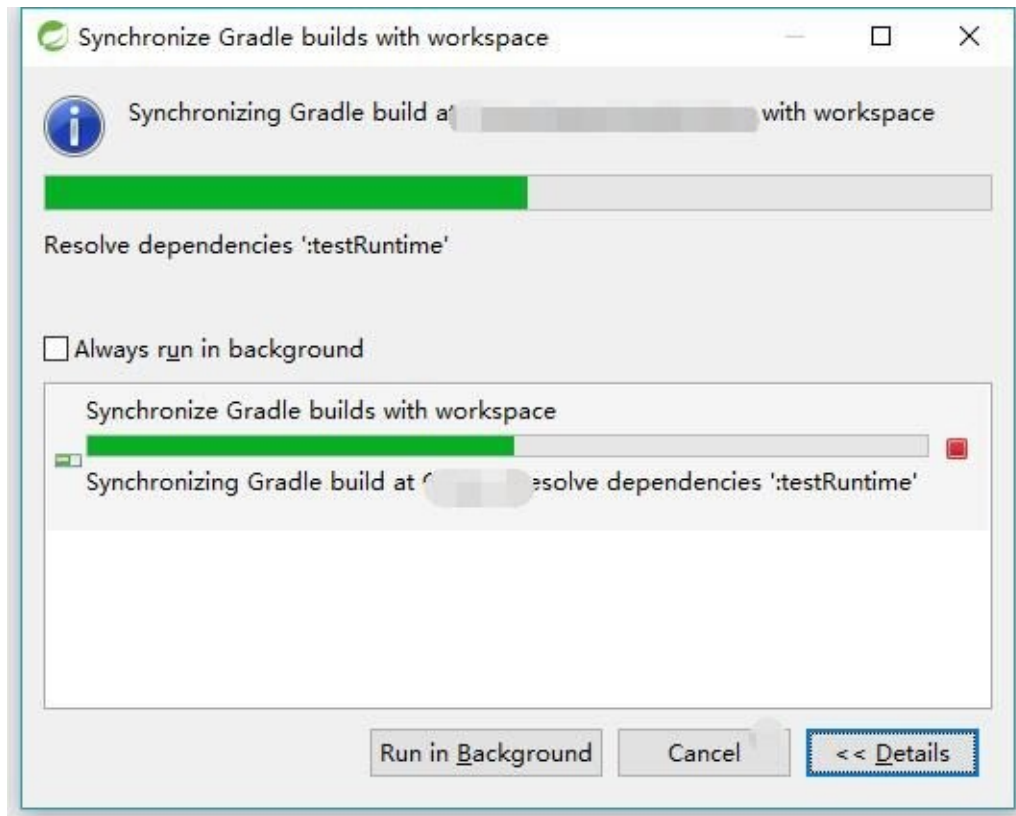


经过一段时间的等待（验证脚本、下载依赖）后，导入即可完成。



创建

经过一段时间的等待（验证脚本、下载依赖）后，导入即可完成。

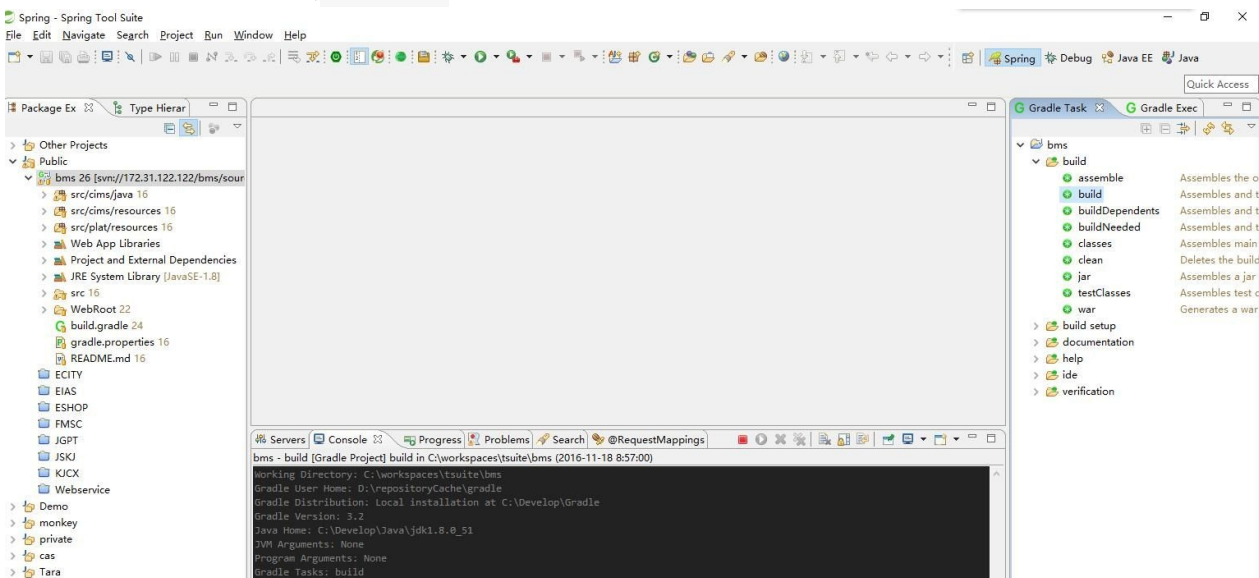


创建

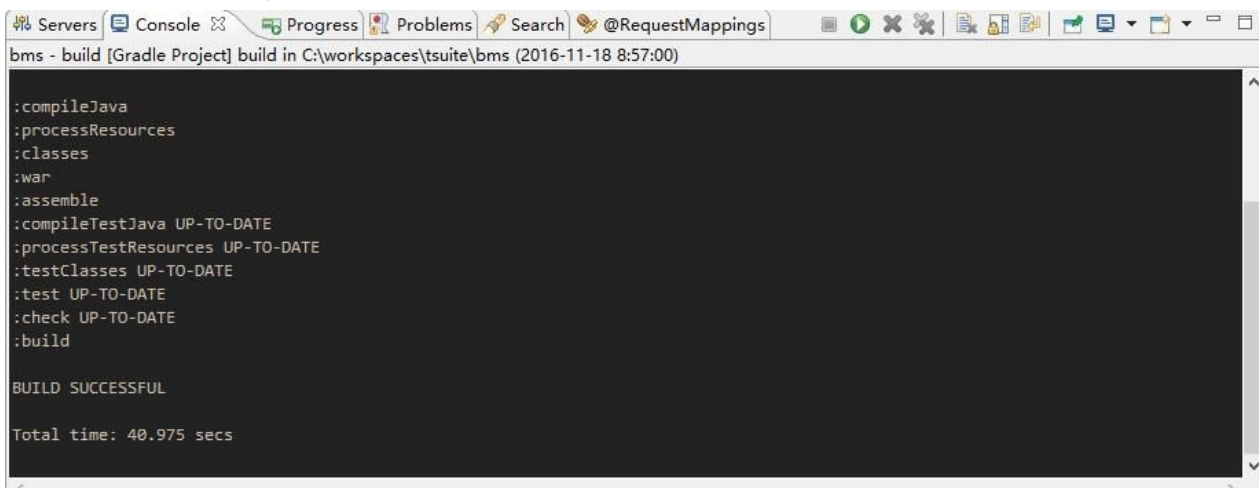
按照常规方式创建项目,创建时选择gradle project即可,创建向导与导入向导如出一辙,不再赘述

转入

导入成功后找到gradle窗格,可见 Gradle task 和 Gradle exec 两个页签,其中 task 页签列出了所有的可执行的任务,exec页签 则会在执行任务后展示执行日志




当然控制台也会打印执行的详细信息



补充

插件自带的编辑器是没有语法着色的,可以安装下面这个,可以对关键字进行高亮展示.当然,语法提示的话,肯定是没有的



Minimalist Gradle Editor 1.0.1

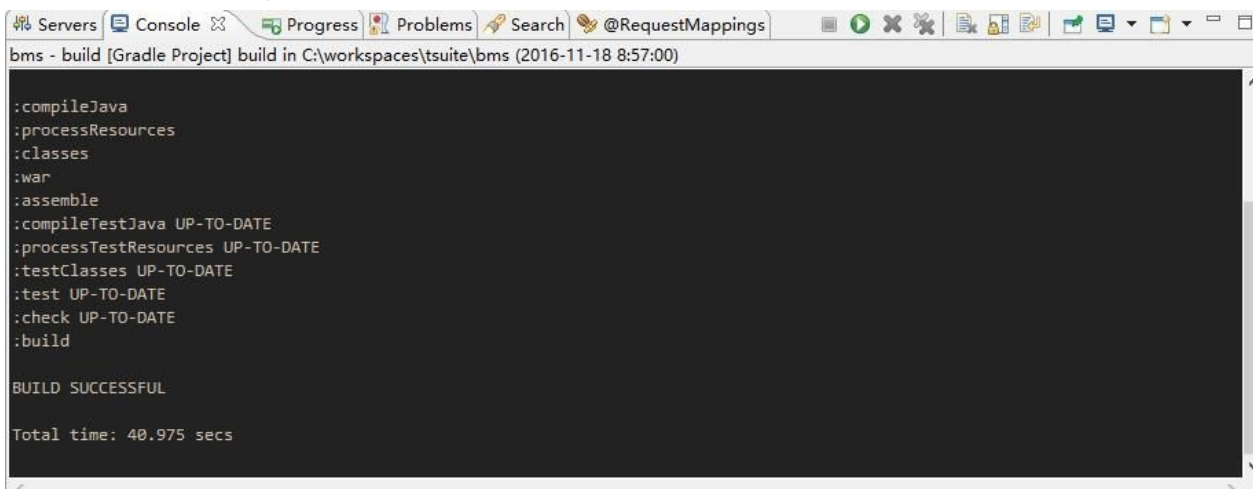
<https://github.com/Nodeclipse/GradleEditor> Minimalist Gradle Editor for build.gradle files with highlight for keywords, strings and matching brackets and android... [more info](#)

by [Nodeclipse/Enide](#), GPL

[gradle editor highlight build android fileExtension_gradle](#)

★ 18
Installs: 23.4K (1,550 last month)
Install

当然控制台也会打印执行的详细信息



The screenshot shows the Eclipse IDE's console window. The title bar indicates the project is 'bms' and the build is for '[Gradle Project] build in C:\workspaces\tsuite\bms (2016-11-18 8:57:00)'. The console output lists the following tasks: :compileJava, :processResources, :classes, :war, :assemble, :compileTestJava UP-TO-DATE, :processTestResources UP-TO-DATE, :testClasses UP-TO-DATE, :test UP-TO-DATE, :check UP-TO-DATE, and :build. The build concludes with 'BUILD SUCCESSFUL' and a total time of 40.975 seconds.

```
bms - build [Gradle Project] build in C:\workspaces\tsuite\bms (2016-11-18 8:57:00)

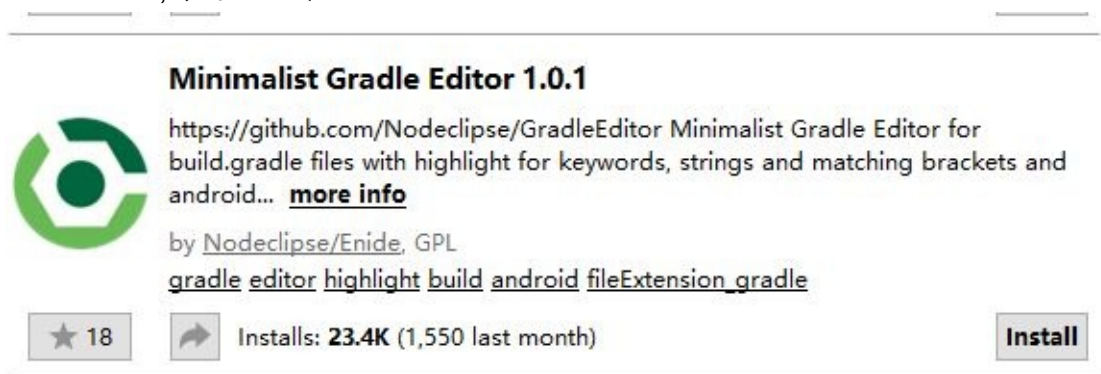
:compileJava
:processResources
:classes
:war
:assemble
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:test UP-TO-DATE
:check UP-TO-DATE
:build

BUILD SUCCESSFUL

Total time: 40.975 secs
```

补充

插件自带的编辑器是没有语法着色的,可以安装下面这个,可以对关键字进行高亮展示.当然,语法提示的话,肯定是没有的



This block displays a plugin card for 'Minimalist Gradle Editor 1.0.1'. It features a green circular logo on the left. The text provides the GitHub repository link (<https://github.com/Nodeclipse/GradleEditor>), a brief description of its functionality (highlighting keywords, strings, and brackets in build.gradle files), and the license (GPL). Below the description, it shows a star count of 18 and installation statistics of 23.4K (1,550 last month). An 'Install' button is located at the bottom right.

Minimalist Gradle Editor 1.0.1

<https://github.com/Nodeclipse/GradleEditor> Minimalist Gradle Editor for build.gradle files with highlight for keywords, strings and matching brackets and android... [more info](#)

by [Nodeclipse/Enide](#), GPL

[gradle editor highlight build android fileExtension_gradle](#)

★ 18 Installs: **23.4K** (1,550 last month) **Install**

IDEA中Gradle插件的使用

Idea本身已经集成了Gradle插件,你可以在File | Settings | Build, Execution, Deployment | Build Tools | Gradle 下找到 相关配置

配置

本文基于IDEA 2016.2

配置项	说明
Linked Gradle projects	所有Gradle添加了Gradle支持的项目列表

项目相关配置	说明
Use auto-import	是否开启自动导入,若开启修改gradle脚本文件后会自动检测变化并对项目进行刷新
Create directories for empty content roots automatically	导入或者创建gradle项目时,是否自动创建 标准目录结构
Create separate module per source set	让每个模块单独采用Gradle约定的source set结构去执行构建
Use default gradle wrapper (recommended)	使用Gradle Wrapper(如果一定要翻译的话,可以称之为Gradle 包装器),这可以使得项目组成员不必预先安装好gradle即可执行gradle脚本,同时也便于统一项目所使用的gradle版本,当然虽说是不必预装其实是它会自己去官网帮你下载一个,然而gradle安装包体积不小同时又有墙的过滤,所以开启此项最好事先备好梯子.
Use gradle wrapper task configuration	自定义Gradle Wrapper配置,如可以更改发行包下载地址为你的内网地址便不存在上一条中的翻墙\下载失败\速度慢的问题了,示例代码 <pre>task wrapper(type: Wrapper) { distributionUrl = "http://mycompanyserver/gradle-2.10-bin.zip" jarFile = "/mylocation/gradle/wrapper/gradle-wrapper.jar" scriptFile = "/mylocation/gradle/gradlew" }</pre>
Use local gradle distribution	采用本地安装的Gradle执行脚本
Gradle home	选择你的Gradle安装目录即可,无需选择到bin
Gradle JVM	选择构建Gradle项目使用的JVM,默认是项目采用的JDK

全局相关配置	说明
Offline work	离线模式,开启离线模式后,Gradle将不会联网查找依赖,而是仅从本地缓存中查找,所以要慎重开启此选项
Service directory path	修改Gradle的默认缓存目录,该项也可以通过添加GRADLE_USER_HOME环境变量进行设置,默认值是«USER_HOME»/.gradle文件夹,更多设置方式参考 如何修改Gradle的缓存目录
Gradle VM options -	设置jvm参数,可以采用空格分隔不同的参数设置 比如 "-client -ea -Xmx1024m" 也可以通过Gradle的-D参数对相关参数进行设置

应用

看到你的IDEA右侧竖向的标签页,找到gradle点开可以在这里看到你的项目以及相应任务,双击或者右键可以执行相应任务. 如果未开启auto import选项,可以点击菜单上方的蓝色圈圈 进行手动同步

为已有项目添加Gradle支持

可以通过重新import 或者 关闭项目 重新打开 会自动弹出引导窗口 进行相关配置 这里需要保证你的gradle脚本没有问题,否则及时完成引导设置右侧也无法看到gradle页签的.

项目文件

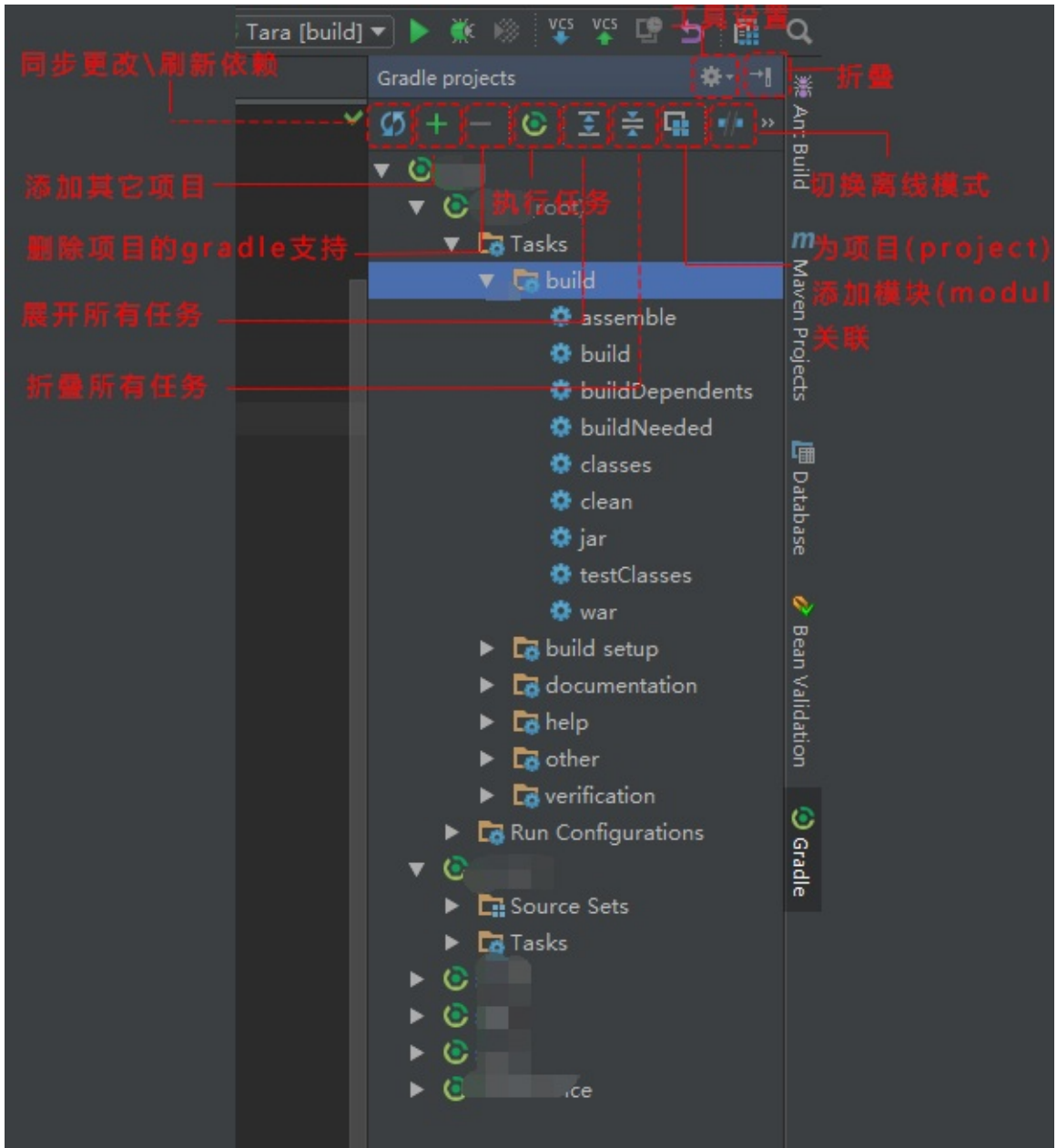
- .gradle gradle项目产生文件 (自动编译工具产生的文件)
- build 自动构建时生成文件的地方
- gradle 自动完成gradle环境支持文件夹
- build.gradle gradle 项目自动编译的配置文件
- gradle.properties gradle 运行环境配置文件
- gradlew 自动完成 gradle 环境的linux mac 脚本, 配合gradle 文件夹使用 代替gradle 命令实现自动完成gradle环境搭建,配合gradle文件夹的内容
- gradlew.bat 自动完成 gradle 环境的windows 脚本, 配合gradle 文件夹使用

上面两个文件会做如下几件事情 1.解析 gradle/wrapper/gradle-wrapper.properties 文件, 获取项目需要的 gradle 版本下载地址 2.判断本地用户目录下的 ./gradle 目录下是否存在该版本, 不存在该版本, 走第3点, 存在走第4点 3.下载 gradle-wrapper.properties 指定版本, 并解压到用户目录的下 ./gradle 文件下 4.利用 ./gradle 目录下对应的版本的 gradle 进行相应自动编译操作

- setting.gradle gradle 项目的子项目包含文件

Gradle工具窗口

可以通过View -> Tool windows -> Gradle 打开,也可以从右侧纵向标签页点击打开.



不是我画的,出处不明,仅供参考

珍爱生命 从不做伸手党开始

