Gradle基础 构建生命周期和Hook技术

Joe_H LV.3
2018年05月18日 20:21 · 阅读 8371

关注

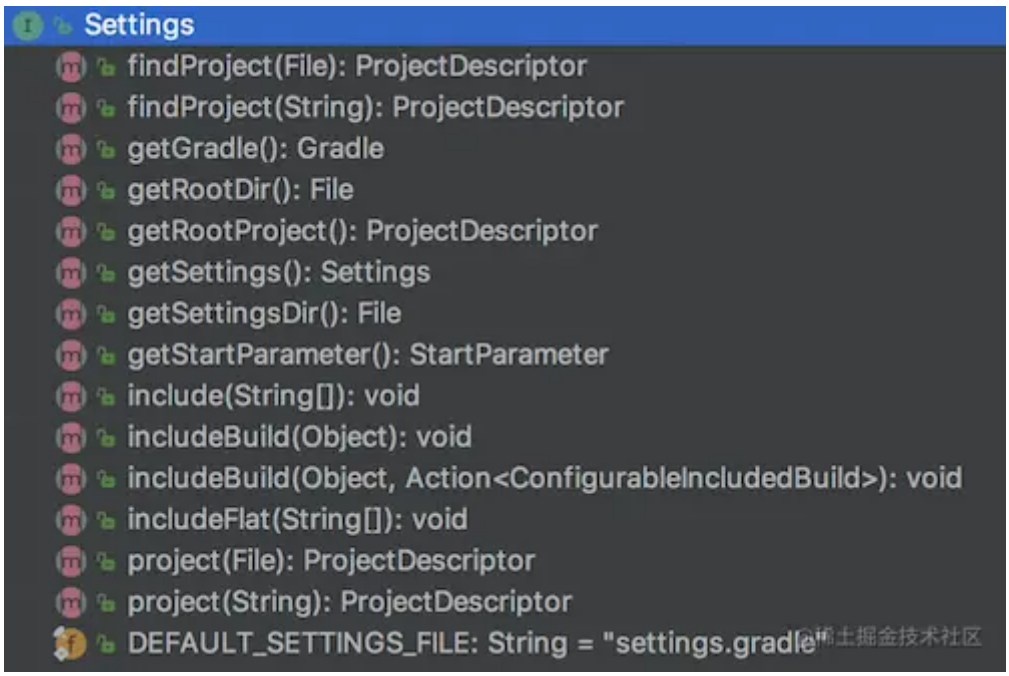
对于初学者来说，面对各种各样的Gradle构建脚本，想要梳理它的构建流程，往往不知道从何入手。Gradle的构建过程有着固定的生命周期，理解Gradle的生命周期和Hook点，有助于帮你梳理、扩展项目的构建流程。

构建的生命周期

任何Gradle的构建过程都分为三部分：初始化阶段、配置阶段和执行阶段。

初始化阶段

初始化阶段的任务是创建项目的层次结构，并且为每一个项目创建一个 `Project` 实例。 与初始化阶段相关的脚本文件是 `settings.gradle` （包括 `<USER_HOME>/.gradle/init.d` 目录下的所有.gradle脚本文件，这些文件作用于本机的所有构建过程）。一个 `settings.gradle` 脚本对应一个 `Settings` 对象，我们最常用来声明项目的层次结构的 `include` 就是 `Settings` 类下的一个方法，在Gradle初始化的时候会构造一个 `Settings` 实例对象，它包含了下图中的方法，这些方法都可以直接在 `settings.gradle` 中直接访问。



比如可以通过如下代码向Gradle的构建过程添加监听：

```javascript
gradle.addBuildListener(new BuildListener() {
  void buildStarted(Gradle var1) {
    println '开始构建'
  }
  void settingsEvaluated(Settings var1) {
    println 'settings评估完成（settins.gradle中代码执行完毕）'
    // var1.gradle.rootProject 这里访问Project对象时会报错，还未完成Project的初始化
  }
  void projectsLoaded(Gradle var1) {
    println '项目结构加载完成（初始化阶段结束）'
    println '初始化结束，可访问根项目：' + var1.gradle.rootProject
  }
  void projectsEvaluated(Gradle var1) {
    println '所有项目评估完成（配置阶段结束）'
  }
  void buildFinished(BuildResult var1) {
    println '构建结束 '
  }
})
```

执行 `gradle build`，打印结果如下：

```lua
settings评估完成（settins.gradle中代码执行完毕）
项目结构加载完成（初始化阶段结束）
初始化结束，可访问根项目：root project 'GradleTest'
所有项目评估完成（配置阶段结束）
:buildEnvironment

------------------------------------------------------------
Root project
------------------------------------------------------------

classpath
No dependencies

BUILD SUCCESSFUL

Total time: 0.959 secs
构建结束
```

配置阶段

配置阶段的任务是执行各项目下的 `build.gradle` 脚本，完成Project的配置，并且构造 `Task` 任务依赖关系图以便在执行阶段按照依赖关系执行 `Task` 。 该阶段也是我们最常接触到的构建阶段，比如应用外部构建插件 `apply plugin: 'com.android.application'` ，配置插件的属性 `android{ compileSdkVersion 25 ...}` 等。每个 `build.gralde` 脚本文件对应一个 `Project` 对象，在初始化阶段创建， `Project` 的接口文档。 配置阶段执行的代码包括 `build.gralde` 中的各种语句、闭包以及 `Task` 中的配置段语句，在根目录的 `build.gradle` 中添加如下代码：

arduino 复制代码

```
println 'build.gradle的配置阶段'

// 调用Project的dependencies(Closure c)声明项目依赖
dependencies {
    // 闭包中执行的代码
    println 'dependencies中执行的代码'
}

// 创建一个Task
task test() {
  println 'Task中的配置代码'
  // 定义一个闭包
  def a = {
    println 'Task中的配置代码2'
  }
  // 执行闭包
  a()
  doFirst {
    println '这段代码配置阶段不执行'
  }
}

println '我是顺序执行的'
```

调用 `gradle build` ，得到如下结果：

markdown 复制代码

```
build.gradle的配置阶段
dependencies中执行的代码
Task中的配置代码
Task中的配置代码2
我是顺序执行的
:buildEnvironment

------------------------------------------------------------
Root project
------------------------------------------------------------

classpath
```

```
No dependencies

BUILD SUCCESSFUL

Total time: 1.144 secs
```

一定要注意，配置阶段不仅执行 `build.gradle` 中的语句，还包括了 `Task` 中的配置语句。从上面执行结果中可以看到，在执行了dependencies的闭包后，直接执行的是任务test中的配置段代码（ `Task` 中除了Action外的代码段都在配置阶段执行）。 另外一点，无论执行Gradle的任何命令，**初始化阶段和配置阶段的代码都会被执行**。 同样是上面那段Gradle脚本，我们执行帮助任务 `gradle help` ，任然会打印出上面的执行结果。我们在排查构建速度问题的时候可以留意，是否部分代码可以写成任务Task，从而减少配置阶段消耗的时间。

执行阶段

在配置阶段结束后，Gradle会根据任务Task的依赖关系创建一个有向无环图，可以通过 `Gradle` 对象的 `getTaskGraph` 方法访问，对应的类为TaskExecutionGraph，然后通过调用 `gradle <任务名>` 执行对应任务。

下面我们展示如何调用子项目中的任务。

1. 在根目录下创建目录subproject，并添加文件build.gradle
2. 在settings.gradle中添加 `include ':subproject'`
3. 在subproject的build.gradle中添加如下代码

arduino 复制代码

```
task grandpa {
  doFirst {
    println 'task grandpa: doFirst 先于 doLast 执行'
  }
  doLast {
    println 'task grandpa: doLast'
  }
}

task father(dependsOn: grandpa) {
  doLast {
    println 'task father: doLast'
  }
}

task mother << {
  println 'task mother 先于 task father 执行'
```

```
}

task child(dependsOn: [father, mother]){
  doLast {
    println 'task child 最后执行'
  }
}

task nobody {
  doLast {
    println '我不执行'
  }
}
// 指定任务father必须在任务mother之后执行
father.mustRunAfter mother
```

它们的依赖关系如下：

ruby 复制代码

```
:subproject:child
+--- :subproject:father
|    \--- :subproject:grandpa
\--- :subproject:mother
```

执行 `gradle :subproject:child` ，得到如下打印结果：

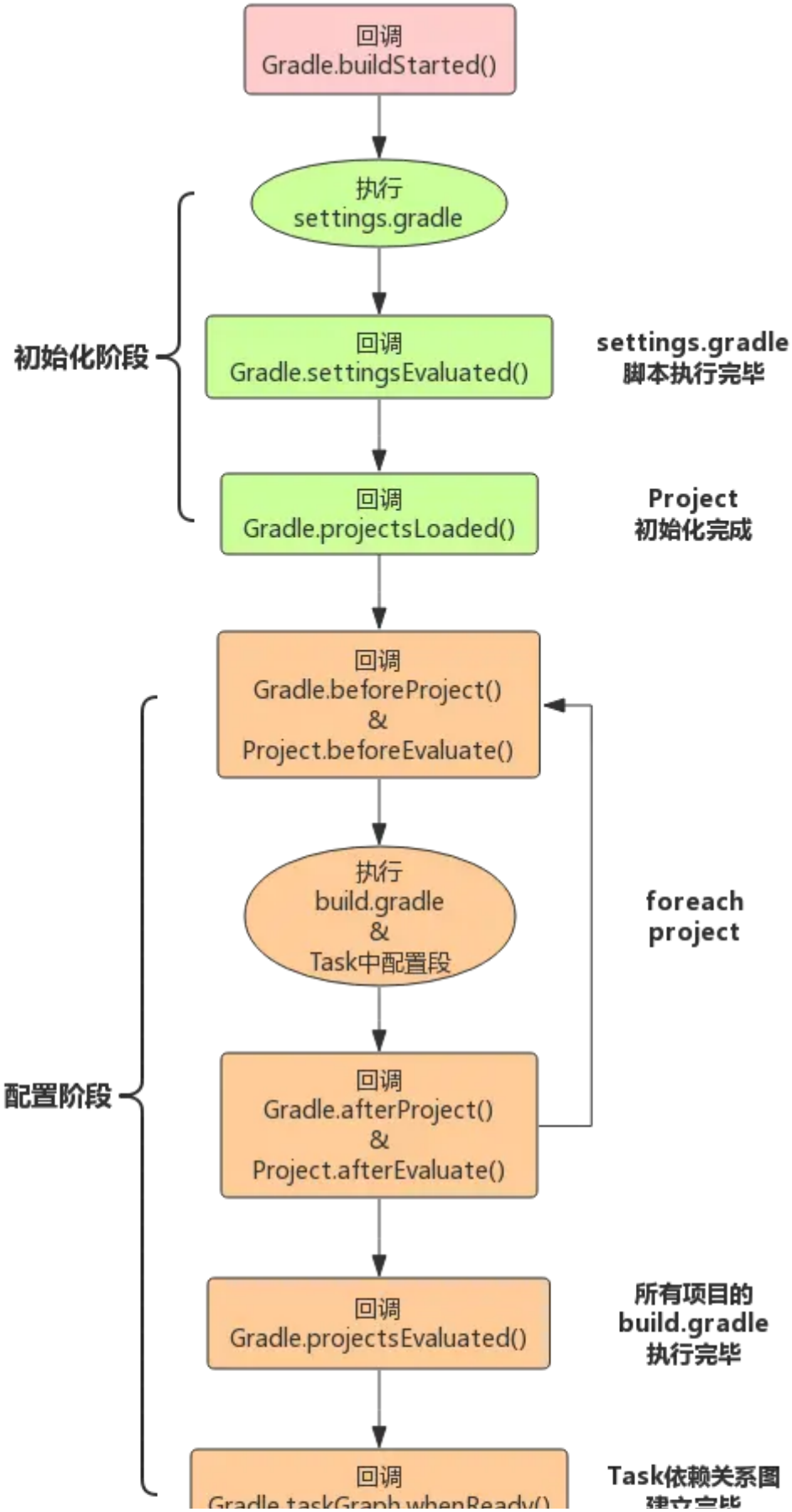
ruby 复制代码

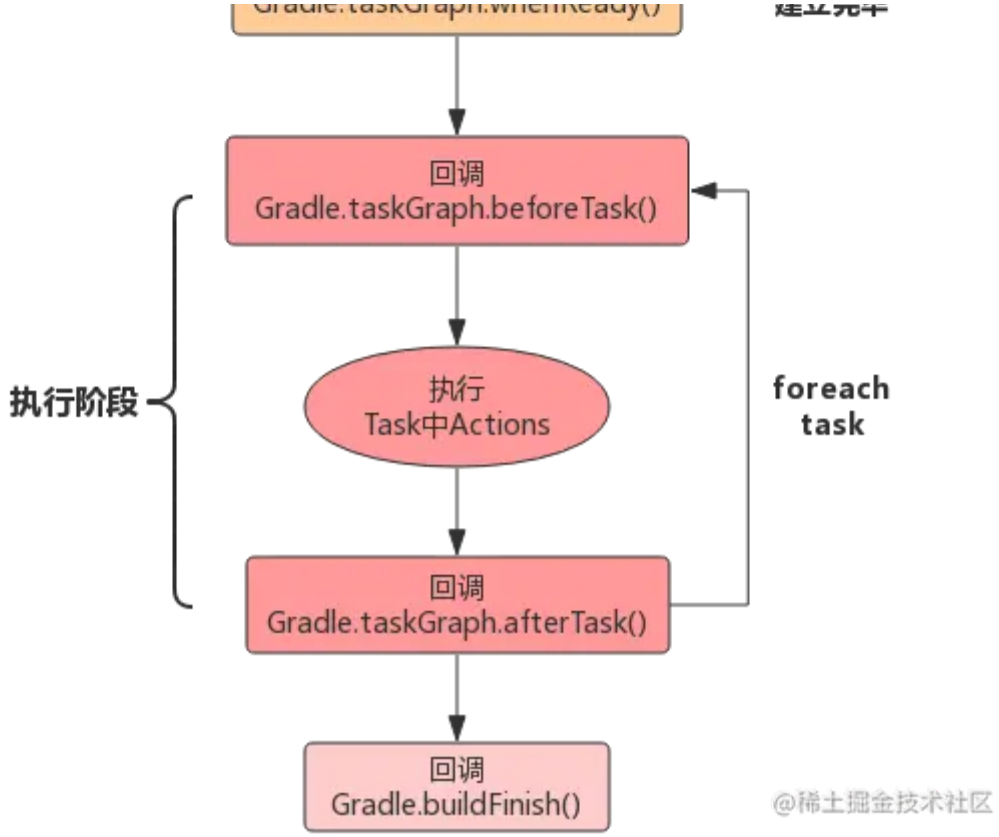
```
:subproject:mother
task mother 先于 task father 执行
:subproject:grandpa
task grandpa: doFirst 先于 doLast 执行
task grandpa: doLast
:subproject:father
task father: doLast
:subproject:child
task child 最后执行

BUILD SUCCESSFUL

Total time: 1.005 secs
```

因为在配置阶段，我们声明了任务mother的优先级高于任务father，所以mother先于father执行，而任务father依赖于任务grandpa，所以grandpa先于father执行。任务nobody不存在于child的依赖关系中，所以不执行。

Hook点

Gradle提供了非常多的钩子供开发人员修改构建过程中的行为，为了方便说明，先看下面这张图。





Gradle在构建的各个阶段都提供了很多回调，我们在添加对应监听时要注意，**监听器一定要在回调的生命周期之前添加**，比如我们在根项目的build.gradle中添加下面的代码就是错误的：

arduino 复制代码

```
gradle.settingsEvaluated { setting ->
  // do something with setting
}

gradle.projectsLoaded {
  gradle.rootProject.afterEvaluate {
    println 'rootProject evaluated'
  }
}
```

当构建走到build.gradle时说明初始化过程已经结束了，所以上面的回调都不会执行，把上述代码移动到settings.gradle中就正确了。

下面通过一些例子来解释如何Hook Gradle的构建过程。

- **为所有子项目添加公共代码**

在根项目的build.gradle中添加如下代码：

arduino 复制代码

```
gradle.beforeProject { project ->
  println 'apply plugin java for ' + project
  project.apply plugin: 'java'
}
```

这段代码的作用是为所有子项目应用Java插件，因为代码是在根项目的配置阶段执行的，所以并不会应用到根项目中。 这里说明一下Gradle的 `beforeProject` 方法和Project的 `beforeEvaluate` 的执行时机是一样的，只是 `beforeProject` 应用于所有项目，而 `beforeEvaluate` 只应用于调用的Project，上面的代码等价于：

arduino 复制代码

```
allprojects {
  beforeEvaluate { project ->
    println 'apply plugin java for ' + project
    project.apply plugin: 'java'
  }
}
```

`after***` 也是同理的，但 `afterProject` 还有一点不一样，无论Project的配置过程是否出错， `afterProject` 都会收到回调。

- **为指定Task动态添加Action**

arduino 复制代码

```
gradle.taskGraph.beforeTask { task ->
  task << {
    println '动态添加的Action'
  }
}

task Test {
  doLast {
    println '原始Action'
  }
}
```

在任务Test执行前，动态添加了一个doLast动作。

- **获取构建各阶段耗时情况**

arduino 复制代码

```
long beginOfSetting = System.currentTimeMillis()
```

```
gradle.projectsLoaded {
  println '初始化阶段，耗时：' + (System.currentTimeMillis() - beginOfSetting) + 'ms'
}

def beginOfConfig
def configHasBegin = false
def beginOfProjectConfig = new HashMap()
gradle.beforeProject { project ->
  if (!configHasBegin) {
    configHasBegin = true
    beginOfConfig = System.currentTimeMillis()
  }
  beginOfProjectConfig.put(project, System.currentTimeMillis())
}
gradle.afterProject { project ->
  def begin = beginOfProjectConfig.get(project)
  println '配置阶段，' + project + '耗时：' + (System.currentTimeMillis() - begin) + 'ms'
}
def beginOfProjectExcute
gradle.taskGraph.whenReady {
  println '配置阶段，总共耗时：' + (System.currentTimeMillis() - beginOfConfig) + 'ms'
  beginOfProjectExcute = System.currentTimeMillis()
}
gradle.taskGraph.beforeTask { task ->
  task.doFirst {
    task.ext.beginOfTask = System.currentTimeMillis()
  }
  task.doLast {
    println '执行阶段，' + task + '耗时：' + (System.currentTimeMillis() - task.beginOfTask) +
  }
}
gradle.buildFinished {
  println '执行阶段，耗时：' + (System.currentTimeMillis() - beginOfProjectExcute) + 'ms'
}
```

将上述代码段添加到settings.gradle脚本文件的开头，再执行任意构建任务，你就可以看到各阶段、各任务的耗时情况。

- **动态改变Task依赖关系**

有时我们需要在一个已有的构建系统中插入我们自己的构建任务，比如在执行Java构建后我们想要删除构建过程中产生的临时文件，那么我们就可以自定义一个名叫cleanTemp的任务，让其依赖于build任务，然后调用cleanTemp任务即可。 但是这种方式适用范围太小，比如在使用IDE执行构建时，IDE默认就是调用build任务，我们没法修改IDE的行为，所以我们需要将自定义的任务插入到原有的任务关系中。

1. **寻找插入点** 如果你对一个构建的任务依赖关系不熟悉的话，可以使用一个插件来查看，在根项目的build.gradle中添加如下代码:

arduino 复制代码

```
buildscript {
  repositories {
    maven {
      url "https://plugins.gradle.org/m2/"
    }
  }
  dependencies {
    classpath "gradle.plugin.com.dorongold.plugins:task-tree:1.2.2"
  }
}
apply plugin: "com.dorongold.task-tree"
```

然后执行 `gradle <任务名> taskTree --no-repeat` ，即可看到指定Task的依赖关系，比如在Java构建中查看build任务的依赖关系：

lua 复制代码

```
:build
+--- :assemble
|    \--- :jar
|         \--- :classes
|              +--- :compileJava
|              \--- :processResources
\--- :check
     \--- :test
          +--- :classes *
          \--- :testClasses
               +--- :compileTestJava
               |    \--- :classes *
               \--- :processTestResources
```

我们看到build主要执行了assemble包装任务和check测试任务，那么我们可以将我们自定义的cleanTemp插入到build和assemble之间。 2. **动态插入自定义任务** 我们先定义一个自定的任务cleanTemp，让其依赖于assemble。

arduino 复制代码

```
task cleanTemp(dependsOn: assemble) {
  doLast {
    println '清除所有临时文件'
  }
}
```

接着，我们将cleanTemp添加到build的依赖项中。

```erlang
afterEvaluate {
  build.dependsOn cleanTemp
}
```

注意，**dependsOn方法只是添加一个依赖项，并不清除之前的依赖项**，所以现在的依赖关系如下：

```lua
:build
+--- :assemble
|    \--- :jar
|         \--- :classes
|              +--- :compileJava
|              \--- :processResources
+--- :check
|    \--- :test
|         +--- :classes
|         |    +--- :compileJava
|         |    \--- :processResources
|         \--- :testClasses
|              +--- :compileTestJava
|              |    \--- :classes
|              |         +--- :compileJava
|              |         \--- :processResources
|              \--- :processTestResources
\--- :cleanTemp
     \--- :assemble
          \--- :jar
               \--- :classes
                    +--- :compileJava
                    \--- :processResources
```

可以看到，cleanTemp依赖于assemble，同时build任务多了一个依赖，而build和assemble原有的依赖关系并没有改变，执行 `gradle build` 后任务调用结果如下：

```vbnet
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:jar UP-TO-DATE
:assemble UP-TO-DATE
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
```

```
:test UP-TO-DATE
:check UP-TO-DATE
:cleanTemp
清除所有临时文件
:build

BUILD SUCCESSFUL
```

结语

理解Gradle构建的生命周期是学习Gradle构建系统的基础，对于梳理构建系统执行流程以及编写自己的构建流程都是非常有帮助的，希望这篇文章能够帮助到迷茫的初学者。

Gradle插件项目

AndroidWebP：在Android构建过程中将图片自动转换至WebP格式的Gradle插件。

分类： Android 标签： Android Java gradle WebP