破解Gradle(五) Task完全掌握

罗恩不带土 LV.3
2022年01月11日 23:26 · 阅读 387

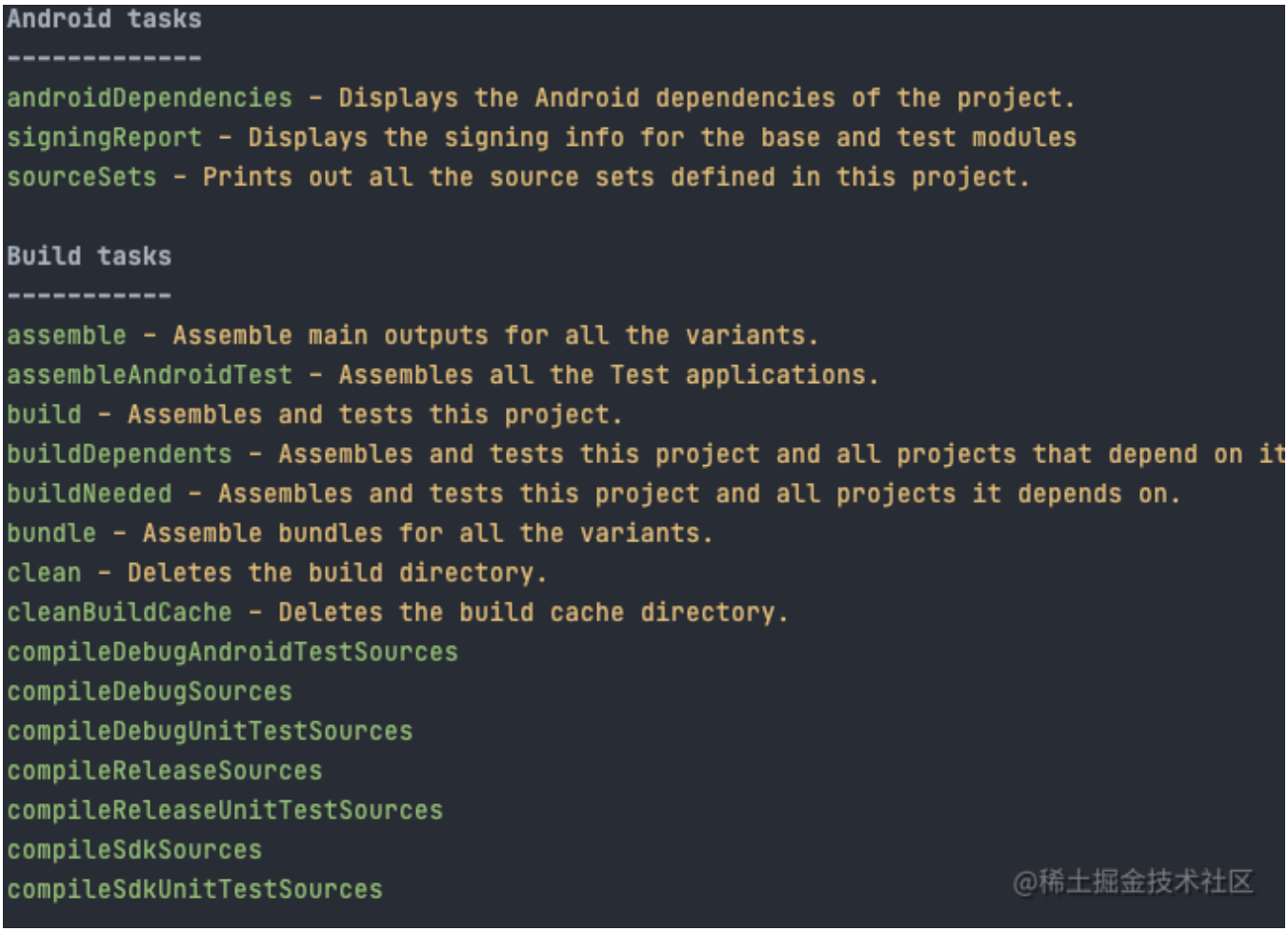
关注

一个 Task 是 Gradle 里项目构建的原子执行单元，Gradle 通过将一个个Task串联起来完成具体的构建任务，每个 Task 都属于一个 Project。

Task API文档

开始前，我们执行 `./gradlew tasks` 来查看项目中有多少Task。

可以看到如下输出：



上图就是当前工程中的每条task都已罗列出，并且有黄色的输出表示当前 `task` 的描述。

一、Task定义及配置

常见的有两种定义方式：

groovy 复制代码

```groovy
//直接通过task函数创建
task helloTask{
  println "i am hellloTask"
}

//通过TaskContationer去创建Task
this.tasks.creat(name: "helloTask2"){
  println "i am helloTasks2"
}
```

> TaskContianer 是用来管理所有的 Task 实例集合的，可以通过 Project.getTasks() 来获取 TaskContainer 实例。

不管哪种创建Task的同时，我们还可以配置它的一些列属性，如：

groovy 复制代码

```
//定义分组 添加注释
task helloTask(group: 'test', description: 'task study'){
  println 'i am hellpTask'
}

this.tasks.create(name: 'hellpTask2'){
  setGroup('imooc')
  setDescription('task study')
  println 'i am hellpTask2'
}
```

添加了group后，我们打开侧边Tasks目录下就会多出个test的分组，更方便查找。

目前 官方所支持的属性 可以总结为如下表格：

选型	描述	默认值
"name"	task 名字	无，必须指定
"type"	需要创建的 task Class	DefaultTask
"action"	当 task 执行的时候，需要执行的闭包 closure 或 行为 Action	null
"overwrite"	替换一个已存在的 task	false
"dependsOn"	该 task 所依赖的 task 集合	[]
"group"	该 task 所属组	null
"description"	task 的描述信息	null
"constructorArgs"	传递到 task Class 构造器中的参数	null

在开发中，如何引用另一个task的属性？如下所示：

groovy 复制代码

```
//使用 defaultTasks 关键字 来将一些任务标识为默认的执行任务
defaultTasks "Gradle_First", "Gradle_Last"

task Gradle_First() {
```

```
  //使用 ext 给 task 自定义需要的属性
    ext.good = true
}

task Gradle_Last() {
    doFirst {
        println Gradle_First.goodg
    }
    doLast {
      //使用 "$" 来引用另一个 task 的属性
        println "I am not $Gradle_First.name"
    }
}
```

二、 Task执行详解

在gradle脚本的配置阶段都会执行，也就是说不管执行脚本里的哪个任务，所有的task里的配置代码都会被执行，但是我们期望的是调用一个方法的时候，这个方法里的代码才会执行。这里就要涉及到Task Action的概念了。

Task通常用 `doFirs` t和 `dolast` 两个方式用于在执行期间进行操作：

groovy 复制代码

```
task helloTask(group: 'test', description: 'task study'){
  println 'i am hellpTask'
  //表示 task 执行最开始的时候被调用的 Action。
  doFirst{
    println 'the task group is: ' + group
  }

  //表示 task 将执行完的时候被调用的 Action。
  doLast{
    println 'the task doLast'
  }
}
helloTask.doFirst{
   println 'the task description is: ' + description
}
```

当我们执行 `./gradlew helloTask` 可以看到输出结果：

groovy 复制代码

```
the task description is: task study
the task group is: test
the task doLast
```

> 从这里可以知道了单独调用方法会先执行，然后配置块中的再执行。

接下来，就用一个实战更加了解 `doFirst` 和 `doLast` 的用法，来实现计算build执行期间的耗时：

groovy 复制代码

```
//计算build执行时长
def startBuildTime, endBuildTime

 //保证要找的task已经配置完毕
this.afterEvaluate { Project project ->
   // 执行build任务时，第一个被执行的Task
        def preBuildTask = project.tasks.getByName('preBuild')
        preBuildTask.doFirst {
          //获取第一个 task 开始执行时刻的时间戳
            startBuildTime = System.currentTimeMillis()
        }

        //找到当前 project 下最后一个执行的 task，即 build task
        def buildTask = project.tasks .getByName('build')
        buildTask.doLast {
            //获取最后一个 task 执行完成前一瞬间的时间戳
            endBuildTime = System.currentTimeMillis()
            println "Current project execute time is ${endBuildTime - startBuildTime}"
        }
}
```

三、 Task的依赖和执行顺序

在task执行顺序方式，有三种方式可以规定task的执行顺序：

3.1 dependsOn强依赖方式

通过下面例子来了解Task如何进行依赖：

groovy 复制代码

```
task taskX{
    doLast{
        println 'taskX'
    }
}
```

```
task taskY{
    doLast{
        println 'taskY'
    }
}

//通过定义dependsOn依赖了taskX 和tasky
task taskZ(dependsOn: [taskX, taskY]){
  doLast{
    print 'taskZ'
  }
}
//另一种方式
taskZ.dependsOn(taskX, taskY)
```

再执行后就可以看到先执行了taskX和taskY，在执行taskZ。

groovy 复制代码

```
> Task :app:taskX
taskX

> Task :app:taskY
taskY

> Task :app:taskZ
taskZ
```

如果定义一个task不知道依赖具体的task的时候该怎么办？这时就要我们动态的进行依赖。

groovy 复制代码

```
task lib1 {
    doLast{
        println 'lib1'
    }
}

task taskZ(dependsOn: [taskX, taskY]){
    //依赖所有以lib开头的任务
    dependsOn this.tasks.findAll{task ->
        return task.name.startsWith('lib')
    }
    doLast{
        println 'taskY'
    }
}
```

下面还是用一个例子来介绍下task依赖在实战中的使用，来**解析一个xml文件并进行输出**：

groovy 复制代码

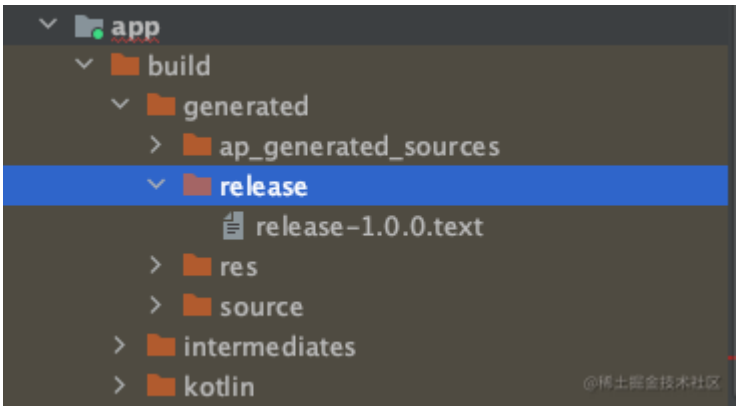
```groovy
//解析xml文件
task handleReleaseFile{
    def srcFile = file('releases.xml')
    def destDir = new File(this.buildDir, 'generated/release/')
    doLast{
        println '开始解析对应的xml文件'
        destDir.mkdir()
        def releases = new XmlParser().parse(srcFile)
        releases.release.each{ releaseNode ->
            //解析每个reslease结点的内容
            def name = releaseNode.versionName.text()
            def versionCode = releaseNode.versiointCode.text()
            def versionInfo = releaseNode.versionInfo.text()
            //创建文件并写入结点数据
            def destFile = new File(destDir, "release-${name}.text")
            destFile.withWriter{ writer ->
                writer.write("${name} -> ${versionCode} -> ${versionInfo}")
            }
        }
    }
}

task handleReleaseFileTest(dependsOn: handleReleaseFile){
    //得到该目录
    def dir = fileTree(this.buildDir.path + 'generated/release/')
    doLast {
        dir.each {
            println 'the file name is:' + it
        }
        println '输出完成....'
    }
}
```

然后执行 `./gradlew handleReleaseFileTest` 命令，可以得到如下输出：

groovy 复制代码

```groovy
> Task :app:handleReleaseFile
开始解析对应的xml文件

> Task :app:handleReleaseFileTest
输出完成....
```

在我们执行测试任务的时候，也会执行相对应的依赖任务。之后在自定义的输出目录下面就会有相应的文件生成：



上面提到的 `release.xml` 文件如下：

xml 复制代码

```xml
<release>
  <release>
    <versionCode>100</versionCode>
    <versionName>1.0.0</versionName>
    <versionInfo>App的第一个版本， 上线了一些基础功能</versionInfo>
  </release>
</release>
```

3.2 通过Task输入输出指定

也可以通过Task来指定输入输出，那我们就使用这种方式来完成一个自动维护版本发布文档的gradle脚本。

groovy 复制代码

```groovy
ext{
    versionName = '1.0.0'
    versionCode = '100'
    versionInfo = 'App的第一个版本， 上线了一些基础功能'
    destFile = file('release.xml')
    if (destFile != null && !destFile.exists()){
        destFile.createNewFile()
    }
}

task writeTask{
    //为task指定输入
    inputs.property('versionCode', this.versionCode)
    inputs.property('versionName', this.versionName)
    inputs.property('versionInfo', this.versionInfo)
    //为Task指定输出
    outputs.file this.destFile

    doLast {
```

```
        //将输入的内容写进去
        def data = inputs.getProperties()
        File file = outputs.getFiles().getSingleFile()
        //将map转化为实体对象
      def versionMsg = new VersionMsg(data)

      //将实体对象写入到xml中
                .......
   }

  task readTask{
    inputs.file destFile
    doLast {
        //读取输入文件的内容并显示
        def file = inputs.files.singleFile
        println file.text
    }
}

  //最后对它进行测试
  task taskTest{
    dependsOn readTask, writeTask
    doLast {
        println '输入输出任务结束'
    }
}
```

在这里我们定义了readTask和writeTask，一个负责读取另一个则负责写入。在writeTask任务中指定了输出文件destFile，并把版本信息写入进去。最后在TaskTest任务重使用的dependOn将两个task关联起来，这样就完成了这个例子--releaseinfo.gradle

可是每一次执行都要我们手动执行，有没有可以直接在构建中就帮我们执行了这个任务?

groovy 复制代码

```
//挂载到build构建中去
project.afterEvaluate {project ->
    def buildTask = project.tasks.getByName('build')
    if (buildTask == null){
        throw GradleException('the build task is not found')
    }

    //buildTask.finalizedBy "writeTask"
    buildTask.doLast{
        writeTask.finalizedBy()
    }
}
```

3.3 通过API指定执行顺序

我们还可以在闭包中通过 `mustRunAfter` 方法指定task的依赖顺序，它必须结合 `dependesOn` 强依赖进行配套使用。

groovy 复制代码

```
task taskX {
  //如果同时执行taskX、testY这2个task，会确保testY执行完之后才执行taskX这个task，用这个来保证执行顺序
    mustRunAfter "taskY"

    doFirst {
        println "this is taskX"
    }
}

task taskY {
    // 使用 mustRunAfter 指定依赖的（一至多个）前置 task
    // 也可以使用 shouldRunAfter 的方式，但是是非强制的依赖
                // shouldRunAfter taskA
    doFirst {
        println "this is taskY"
    }
}

task taskZ(dependsOn: [taskX, taskY]) {
    mustRunAfter "taskY"
    doFirst {
        println "this is taskZ"
    }
}
```

四、Task类型

Task已经帮我们定义了很多Task Type供我们使用，平时用的时候多查看下官方文档Gradle DSL API文档中的Task types那一栏下。

Gradle本身还提供了一些已有的Task供我们使用，比如Copy、Delete等。因此我们定义Task的时候是可以继承已有的Task，如下实例代码：

groovy 复制代码

```
// 将 doc 复制到 build/target 目录下
task copyDocs(type: Copy) {
    from 'src/main/doc'
    into 'build/target/doc'
}
```

```
// 删除根目录下的 build 文件
task clean(type: Delete) {
    delete rootProject.buildDir
}
```

另外我们也可以自定义task来引用:

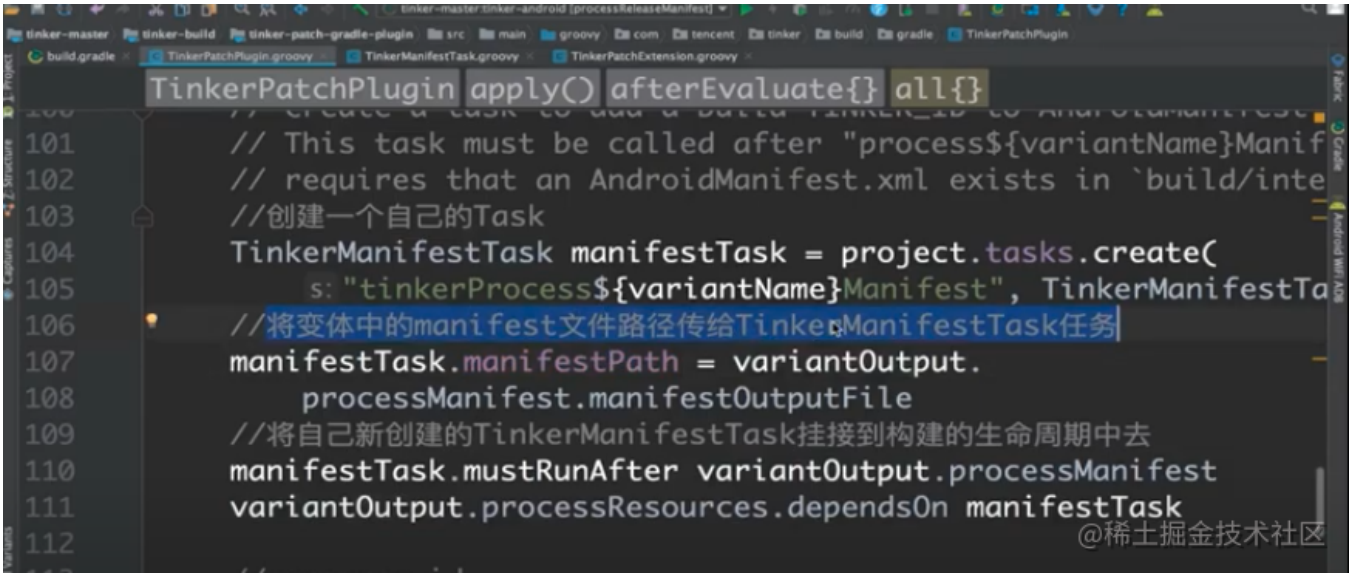
groovy 复制代码

```
class SayHelloTask extends DefaultTask {

    String msg = "default name";
    int age = 20

  //@TaskAction 表示该Task要执行的动作,即在调用该Task时，sayhello()方法将被执行
    @TaskAction
    void sayHello() {
        println "Hello $msg ! Age is ${age}"
    }
}

// hello使用了默认的message值
task hello(type:SayHelloTask)

// 重新设置了message的值
task hello1(type:SayHelloTask){
    message ="I am an android developer"
}
```

五、挂接自定Task到构建过程

如果想把自定义Task挂接到生命周期中间部分，而不是开头或者结束位置，又是如何实现?

学习这个最好是查看**Tinker源码**，在里面可以找到定义了如何挂接自定义Task到构建过程中:



可以转化为构建图理解起来更清晰，这样自定义的task就放在了 `processReleaseManifest` 之后 `processReleaseResources` 之前了。

六、TaskContainer接口解析

TaskContianer 是用来管理所有的 Task 实例集合的，可以通过 Project.getTasks() 来获取 TaskContainer 实例。

groovy 复制代码

```
//查找task
findByPath(path: String): Task
getByPath(path: String): Task
getByName(name: String): Task
withType(type: Class): TaskCollection
```

```
matching(condition: Closure): TaskCollection

//创建task
create(name: String): Task
create(name: String, configure: Closure): Task
create(name: String, type: Class): Task
create(options: Map<String, ?>): Task
create(options: Map<String, ?>, configure: Closure): Task

//当task被加入到TaskContainer时的监听
whenTaskAdded(action: Closure)
```

可以看下具体的使用例子：

groovy 复制代码

```
//当有task创建时
getTasks().whenTaskAdded { Task task ->
    println "The task ${task.getName()} is added to the TaskContainer"
}

//采用create(name: String)创建
getTasks().create("task1")

//采用create(options: Map<String, ?>)创建
getTasks().create([name: "task2", group: "MyGroup", description: "这是task2描述", dependsOn:

//采用create(options: Map<String, ?>, configure: Closure)创建
getTasks().create("task3", {
    group "MyGroup"
    setDependsOn(["task1", "task2"])
    setDescription "这是task3描述"
})

//通过名字查找指定的task
def task3 = getTasks().findByName("task3")
println "findByName() return task is " + task3

def taskList = getTasks().withType(DefaultTask)
def count = 0
//遍历所有的task，打印出其名字
taskList.all { Task t ->
    println "${count++} task name is ${t.name}"
}
```

今天就到这里了，接下去的Gradle开发都离不开 `task` 和 `project` 这两块核心内容，还是要好好掌握住的。

参考

深入理解Android之Gradle

深度探索 Gradle 自动化构建技术（三、Gradle 核心解密） Project文档

Android Gradle学习(三)：Task进阶学习

Task文档

TaskContainer API

分类： Android 标签： gradle