

Gradle是什么？

Gradle是一个基于Groovy的DSL的项目自动化构建工具，Gradle主要是面对Java应用，了解gradle之前，先看一看其他常用的构建工具。

其他构建工具

Ant

Ant是一款基于xml的一个构建工具，简单展示一下Ant的语法如下：

build.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<project name="HelloWorld" default="run" basedir=". ">
  <property name="src" value="src"/>
  <property name="dest" value="classes"/>
  <property name="hello_jar" value="hello1.jar"/>
  <target name="init">
    <mkdir dir="${dest}"/>
  </target>
  <target name="compile" depends="init">
    <javac srcdir="${src}" destdir="${dest}"/>
  </target>
  <target name="build" depends="compile">
    <jar jarfile="${hello_jar}" basedir="${dest}"/>
  </target>
  <target name="run" depends="build">
    <java classname="test.ant.HelloWorld" classpath="${hello_jar}"/>
  </target>
  <target name="clean">
    <delete dir="${dest}" />
    <delete file="${hello_jar}" />
  </target>
  <target name="rerun" depends="clean,run">
    <ant target="clean" />
    <ant target="run" />
  </target>
</project>
```

上面是一个HelloWorld.java的Ant构建文件，将源码放在src目录下，build.xml放工程目录下，命令行执行ant即可执行默认的run这个target。自动构建并执行。

优缺点

ant作为第一款java的现代化自动化构建工具，将开发人员从javac, copy, jar等指令中解脱出来，并且配置灵活，但是ant比较适合简单的java项目，当项目增大时，xml同样会变大到很难管理。并且ant没有依赖管理功能的，对网络上开放的各种库不能自动下载并依赖构建。

Ant with Ivy

Ivy是跟踪管理项目依赖的工具，具有很好的灵活性和可配置性，其由于可以和Ant进行配合集成而被广泛使用。

简单展示一下Ant的语法如下：

ivy.xml

```
<ivy-module version="2.0">
  <info organisation="org.apache" module="hello-ivy"/>
  <dependencies>
    <dependency org="commons-lang" name="commons-lang" rev="2.0"/>
    <dependency org="commons-cli" name="commons-cli" rev="1.0"/>
  </dependencies>
</ivy-module>
```

build.xml

```
<project xmlns:ivy="antlib:org.apache.ivy.ant" name="hello-ivy" default="run">

  ...

  <!-- =====
        target: resolve
        ===== -->
  <target name="resolve" description="--> retrieve dependencies with ivy">
    <ivy:retrieve />
  </target>
</project>
```

其相比原Ant的build.xml文件，增加了一个ivy.xml用来配置依赖，build.xml文件中增加ivy插件的支持即可。

优缺点

相比ant，ivy插件的增加解决了依赖管理的问题，并且保持了ant对构建过程控制上的灵活的优点，但是依旧存在大型项目xml文件过大难以管理的问题。

Maven

Maven也是基于xml的一个构建工具，基于项目对象模型（POM）的概念，不过其文件结构有所差异，先看一下maven管理的xml文件的简单示例：

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.yiibai.project-group</groupId>
  <artifactId>project</artifactId>
  <version>1.0</version>
  <dependencies>
    <dependency>
      <groupId>org.mybatis</groupId>
      <artifactId>mybatis</artifactId>
      <version>3.2.5</version>
    </dependency>
  </dependencies>
</project>
```

在执行任务或目标时，Maven 会使用当前目录中的 POM(Project Object Model)。它读取POM得到所需要的配置信息，然后执行目标。

Ant需要我们将执行task所需的全部命令都一一列出，而Maven依靠约定提供现成的可调用的目标（goal），并且Maven具备了从网络自动下载依赖的能力。

优缺点

Maven不仅是构建工具，其主要特点在于依赖管理，其提供了中央仓库，提供了自动下载依赖的能力，并且根据约定好的关键字和项目结构简化了项目配置流程，能够快速搭建项目。

同样的缺点一样很明显，其不容易写出定制化比较强的项目，并且同样存在xml文件较大的问题。

Gradle

Gradle是一个基于Groovy的DSL的项目自动化构建工具，Gradle主要是面对Java应用，目前也支持Groovy，Kotlin，Scala工程的构建。

特点：

- 一个像 Ant 一样的非常灵活的通用构建工具
- 一种可切换的, 像 maven 一样的基于合约构建的框架
- 支持强大的多工程构建
- 支持强大的依赖管理(基于 Apache Ivy)
- 支持已有的 maven 和 ivy 仓库
- 支持传递性依赖管理, 而不需要远程仓库或者 pom.xml 或者 ivy 配置文件
- 优先支持 Ant 式的任务和构建
- 基于 groovy 的构建脚本
- 有丰富的领域模型来描述你的构建

通俗来说：**gradle结合ant和maven两者的优点，Gradle帮助我们在构建流程中，控制其构建流程，管理项目中的差异，配置不同的构建依赖，动态更改构建参数，自动打包和部署。**

简单的Gradle示例：

build.gradle

```
apply plugin: 'java'
//eclipse插件可以帮助我们自动生成eclipse项目文件(gradle eclipse)
apply plugin: 'eclipse'
//配置MANIFEST.MF
sourceCompatibility = 1.7
version = '1.0'
jar {
    manifest {
        attributes 'Implementation-Title': 'Gradle Quickstart',
                   'Implementation-Version': version
    }
}

repositories {
    mavenCentral()
}
//依赖
dependencies {
    compile group: 'commons-collections', name: 'commons-collections', version:
    '3.2.2'
    testCompile group: 'junit', name: 'junit', version: '4.+'
}
//测试阶段加入一个系统属性
test {
    systemProperties 'property': 'value'
}
//发布 JAR 文件
uploadArchives {
    repositories {
        flatDir {
            dirs 'repos'
        }
    }
}
```

上面的build.gradle文件展示了如何管理一个简单的java项目。执行gradle build即可执行构建，构建的过程输出如下：

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

BUILD SUCCESSFUL in 0s
6 actionable tasks: 6 executed
```

同时还有 clean, assemble, check等gradle指令，相关原理及使用我们在后面会介绍到。

Gradle基本语法及用法

Gradle 的核心是基于 Groovy 的 领域特定语言 (DSL)

Groovy基本语法

注释

同Java，支持单行（使用 `//`）、多行（`/* */`）和文档注释（使用 `/** */`）。

变量定义

```
def name = "Groovy"
String name = "Java"
```

引号标识符（Quoted identifiers）

Groovy在点表达式（dotted expression）后面可以使用引号标识符，比如 `persion.name` 可以表示为 `persion.'name'` 或 `persion."name"`

```
def map = [:]
map."show-add-when-open" = "ALLOWED"
map.'with-dash-signs-and-single-quotes' = "ALLOWED"
```

字符串

```

def name = 'Groovy' //java.lang.String 不支持插值
def strippedFirstNewline = '''line one
line two
line three
''' //java.lang.String, 支持多行
def normalStr = "Groovy" //java.lang.String
def interpolatedStr = "my name is ${normalStr}" //groovy.lang.GString-字符串插值

```

字符串插值

```

def person = [name: 'Groovy', age: 28]
println "$person.name is $person.age years old"

def pi = 3.14
println "Pi = ${pi.toString()}"

```

循环

```

//while循环
while(count<5) {
    println(count);
    count++;
}
//for循环
for(int i = 0;i<5;i++) {
    println(i);
}
//for-in 循环
int[] array = [0,1,2,3];
for(int i in array) {
    println(i);
}

```

集合

List

Groovy的列表使用的是 `java.util.ArrayList`，用中括号 `[]` 括住，使用逗号分隔：

```

def letters = ['a', 'b', 'c', 'd'] //java.util.ArrayList
def linkedList = [2, 3, 4] as LinkedList // 使用as操作符 显式声明为
java.util.LinkedList
LinkedList otherLinked = [3, 4, 5] // 显式指明类型

letters[2] = 'C' // 直接修改
assert letters[1, 3] == ['b', 'd'] // 提取指定元素
assert letters[2..4] == ['C', 'd', 'e'] // 支持范围 (ranges) 操作

```

Map

```
def colors = [red: '#FF0000', green: '#00FF00', blue:
'#0000FF']//java.util.LinkedHashMap
//使用
assert colors['red'] == '#FF0000' // 使用中括号访问
assert colors.green == '#00FF00' // 使用点表达式访问
//赋值
colors['pink'] = '#FF00FF'
colors.yellow = '#FFFF00'
```

类与对象

定义和使用上都和java没有太大区别，只有以下几点的是Groovy特有的：

- `public` 修饰的字段会被自动转换成属性变量，这样可以避免很多冗余的get和set方法。（属性变量默认自带setter，getter方法）
- 如果属性或方法没有访问权限修饰符，那么默认是public，而java中是protected。
- 类名不需要和文件名相同。
- 一个文件中可以定义多个一级类。如没有定义类，则这个groovy文件被认为是脚本文件。
- 构造方法更多样

```
class PersonConstructor {
    String name
    Integer age

    PersonConstructor(name, age) {
        this.name = name
        this.age = age
    }
}

def person1 = new PersonConstructor('Marie', 1) //普通构造方式
def person2 = ['Marie', 2] as PersonConstructor //as 方式
PersonConstructor person3 = ['Marie', 3]

//命名参数构造方法不需要用户定义，当一个类没有构造方法的时候，其默认有一个命名参数构造方法。
class PersonConstructor {
    String name
    Integer age
}

def person4 = new PersonConstructor()
def person5 = new PersonConstructor(name: 'Marie')
def person6 = new PersonConstructor(age: 1)
def person7 = new PersonConstructor(name: 'Marie', age: 2)
```

- 方法定义

//方法定义使用关键字def或者返回值即可。groovy中的方法都有返回值，如果没有写return语句，groovy会计算方法中的最后一行语句并将其结果返回。

```
def someMethod() { 'method called' }
String anotherMethod() { 'another method called' }
def thirdMethod(param1) { "$param1 passed" }
static String fourthMethod(String param1) { "$param1 passed" }
```

闭包

闭包（Closure）是一段匿名代码块，其可以作为参数传递给一个方法，先展示一下最简单的闭包：

```
def clos = {println "Hello World"};
clos.call();//结果打印Hello World
```

闭包的形参

```
def clos = {param->println "Hello ${param}"};
clos.call("World");//结果打印Hello World

def clos = {println "Hello ${it}"};//it是Groovy的一个关键字，代表隐式单个参数
clos.call("World");//结果打印Hello World

def str1 = "Hello";
def clos = {param -> println "${str1} ${param}"}
str1 = "Welcome"
clos.call("World");//结果打印Welcome World
```

方法中使用闭包

```
def static display(clo) {
    clo.call("Inner");
}

display({param->println "Hello ${param}"});
```

集合中的闭包

```
//List
def lst = [11, 12, 13, 14];
lst.each {println it};//it 是默认的param
//Map
def mp = ["TopicName" : "Maps", "TopicDescription" : "Methods in Maps"]
mp.each {println "${it.key} maps to: ${it.value}"}
```

闭包中的this, owner和delegate


```

class Test {
    def num = 1
    def add1 = {
        this.num++
    }
    def add2 = {
        owner.num++
    }
    def add3 = {
        delegate.num++
    }
}

```

在默认情况下，这三者都是指的定义闭包的封装类，也就是Test类。不同的是，delegate的指向是可以改变的。

“<<”操作符

在Java中，<< 符号是左移运算符，而在Groovy中，除了作为左移运算符，还可以有其他的功能：

```

//List和Set的添加
def list = ['a','b','c']
list<<'d' //即java中的add
//StringBuffer的append
def sb = new StringBuffer()
sb<<'Hello'
//对流对象的write
def wtr= new OutputStreamWriter(new FileOutputStream('TheOutput.txt'))
wtr<< 'abc'

```

元对象编程

元对象编程或MOP(Meta Object Protocol)可以用于动态调用方法，并且可以即时创建类和方法。

Groovy实现MOP常见有2种方法：

1. 重写"invokeMethod"和"set/getProperty"方法，和重写"methodMissing"和"propertyMissing"。
2. 通过MetaClass来实现的MOP机制

重写

GroovyObject对象的几个方法如下

```

public interface GroovyInterceptable extends GroovyObject {
}
public interface GroovyObject {
    Object invokeMethod(String var1, Object var2);

    Object getProperty(String var1);

    void setProperty(String var1, Object var2);

    MetaClass getMetaClass();

    void setMetaClass(MetaClass var1);
}

```

举个例子，现在假设有个空的类 Student,然后给student不存在的属性name赋值，就会抛出 MissingPropertyException,调用不存在的方法的话，则抛出 MissingMethodException异常。

```

static class Student {
}

public static metaTest(){
    def myStudent = new Student()
    myStudent.name = "Zheng"
    myStudent.testMethod()
}

```

如何避免这种情况呢，就是重写上面的"invokeMethod"和"set/getProperty"方法

```

static class Student implements GroovyObject {
    protected dynamicProps = [:]

    void setProperty(String pName, val) {
        dynamicProps[pName] = val
    }

    def getProperty(String pName) {
        dynamicProps[pName]
    }

    def invokeMethod(String name, Object args) {
        return "called invokeMethod $name $args"
    }
}

```

这样再次调用student的方法，使用其属性就没有问题了。

另外，与"set/getProperty"和invokeMethod类似，通过重写propertyMissing和methodMissing也可以在找不到对应方法的时候，执行到methodMissing。这里不再赘述。

```
def methodMissing(String name, args) {  
    return "unknown method $name(${args.join(',')})"  
}
```

MetaClass来实现的MOP机制

比如我们在使用String类是，想向其中添加一个新方法，让其能够打印一下Hello：

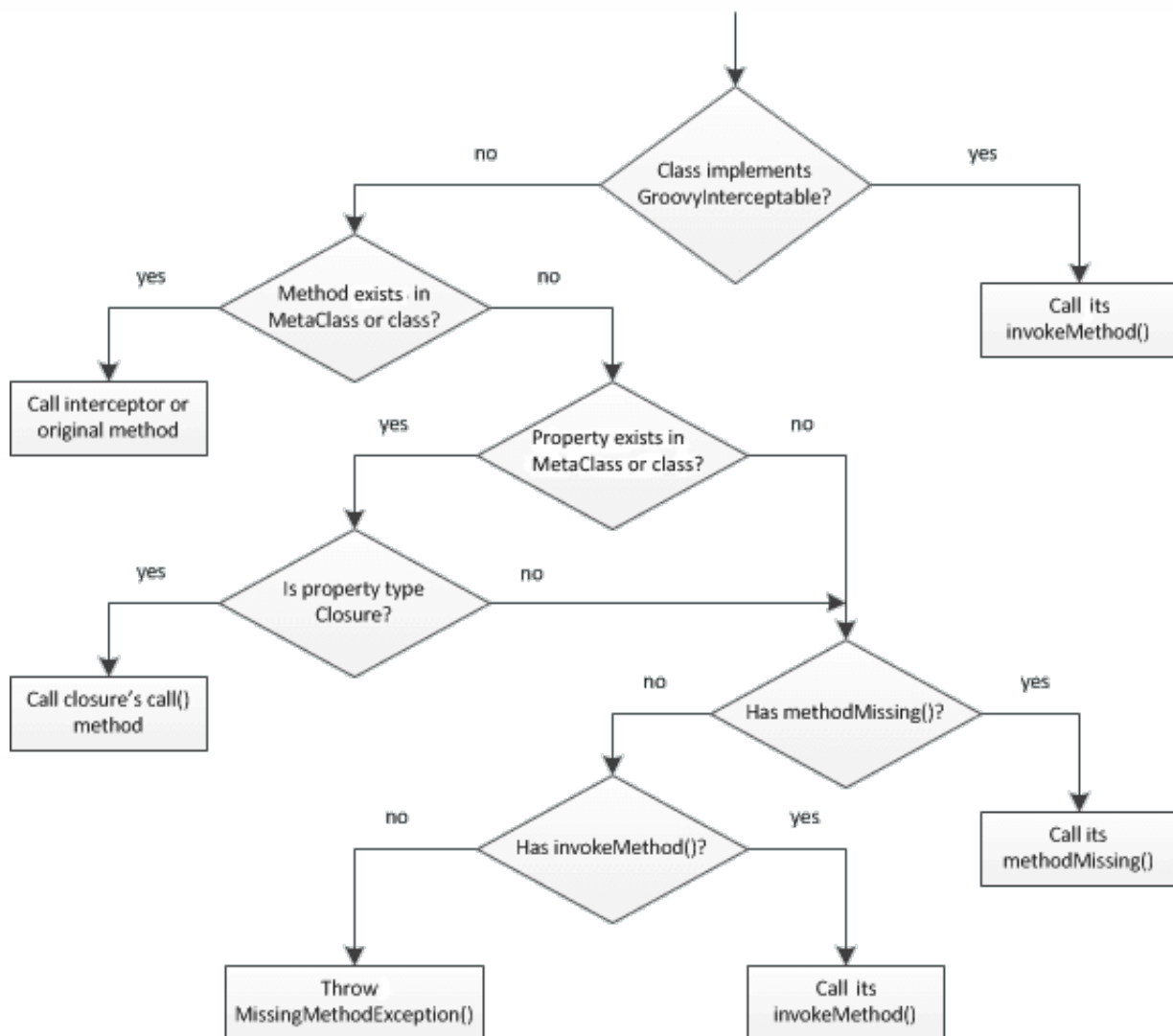
```
public static stringMeta(){  
    String.metaClass.hello = {->  
        println "Hello ${delegate}"  
    }  
  
    String me = "ChenZheng"  
    me.hello()//Hello ChenZheng  
}
```

这是如何做到的？

ExpandoMetaClass是MetaClass接口的一个实现类，可以通过GroovyObjectSupport的metaClass属性直接引入

未包含任何参数的 `hello()` 闭包被添加到 `String` 的 `ExpandoMetaClass` (EMC) 中。每个类都包含在一个 EMC 中，EMC 将拦截对它的方法调用。这意味着即使 `String` 为 `final`，仍然可以将方法添加到其 EMC 中。因此，现在看上去仿佛 `String` 有一个 `hello()` 方法。

Groovy拦截机制：



Groovy中的DSL

DSL或域特定语言旨在简化以Groovy编写的代码，使得它对于普通用户变得容易理解。以下示例显示了具有域特定语言的确切含义。Groovy是一种通用语言，但是使用Groovy可以很容易的写出一个新的DSL，用于特定领域的用途，这是由于Groovy本身的语法特性决定的---比如省略方法调用的（）和；，不需要class定义直接执行脚本等。

比如如果我们想要写一个新的DSL，用来解析一下下面这段很像build.gradle的文件：

```
//很像build.gradle的样子
person {
    name {
        firstname 'DSL'
        lastname 'Groovy'
    }
    action{
        eat "apple"
        drive "mini cooper"
    }
}
```

通过DSL，我们希望能够将这个脚本解析为对应的对象，并在打印姓名和行为。

定义对象：

```
class Name {
    String firstname
    String lastname
    String toString() {
        "$firstname.$lastname"
    }
}
class Action {
    //略
}
class Person {
    Name name
    Action action
    Person(name, action) {
        this.name = name
        this.action = action
    }
    String toString() {
        "My name is $name , I $action"
    }
}
```

```

def scriptClosure = { Closure personClosure ->
    def person = new Person(new Name(), new Action())
    personClosure.delegate = new PersonDelegate(person)
    personClosure.resolveStrategy = Closure.DELEGATE_FIRST
    personClosure()
    return person
}

def createMetaClass(Class clazz, Closure closure) {
    // 为传入的Class对象创建一个ExpandoMetaClass实例, 但不将该ExpandoMetaClass实例注册
    // 到MetaClassRegistry对象中
    def emc = new ExpandoMetaClass(clazz, false)
    //该closure用来初始化ExpandoMetaClass对象
    closure(emc)
    // 完成初始化过程
    emc.initialize()
    return emc
}

def executeScript(dslScriptCode, rootName, closure) {
    Script dslScript = new GroovyShell().parse(dslScriptCode) // 读取并解析DSL代
    // 码, 返回一个Script对象
    dslScript.metaClass = createMetaClass(dslScript.class) { emc ->
        //动态新增一个名为"$rootName"的方法, 注意"$rootName"的值决定于运行时, 比如本例中
        //的值为person
        emc."$rootName" = closure
    }
    return dslScript.run() // 执行DSL代码
}

//最后只需要执行
def person = executeScript(dslScriptCode2, 'person', scriptClosure)//dslScriptCode2
//便是上面的DSL脚本
println person
//即可获取解析后的对象并打印
//My name is DSL.Groovy , I eat apple and drive mini cooper

```

delegate类

```

class PersonDelegate {
    def person

    PersonDelegate(person) {
        this.person = person
    }

    def methodMissing(String name, Object args) {
        if ('name' == name && args[0] instanceof Closure) {

```

```

        def nameClosure = args[0]
        /*
            给nameClosure的delegate赋值，nameClosure就是name旁边的那个closure即一
            对大括号{}以及大括号中的内容
        */
        nameClosure.delegate = new NameDelegate(person)
        nameClosure.resolveStrategy = Closure.DELEGATE_FIRST // 指明closure中变
        量和方法的解析策略，本例选择DELEGATE_FIRST
        nameClosure()
    }
    if ('action' == name && args[0] instanceof Closure) {
        def actionClosure = args[0]
        actionClosure.delegate = new ActionDelegate(person)
        actionClosure.resolveStrategy = Closure.DELEGATE_FIRST
        actionClosure()
    }
}

def propertyMissing(String name) {}
}

class NameDelegate {
    def person

    NameDelegate(person) {
        this.person = person
    }
    /*下面这些getter和setter是为了实现下面这种赋值而写的：
    person {
        name {
            firstname = 'Groovy'
            lastname = 'DSL'
        }
    }
    */

    def getFirstname() {
        return person.name.firstname
    }

    def setFirstname(String firstname) {
        person.name.firstname = firstname
    }

    def getLastName() {
        return person.name.lastname
    }
}

```

```

def setLastname(String lastname) {
    person.name.lastname = lastname
}

def methodMissing(String name, Object args) {
    if ('firstname' == name) {
        person.name.firstname = args[0]
    } else if ('lastname' == name) {
        person.name.lastname = args[0]
    }
}

def propertyMissing(String name) {}
}

```

Gradle语法

通过上面章节的DSL的演示，再回来看build.gradle的写法就会更容易理解

```

buildscript {
    repositories {
        jcenter()
        mavenLocal()
        maven{
            url "file:///D:/repo"
        }
    }

    dependencies {
        classpath 'com.android.tools.build:gradle:1.2.3'
    }
}

```

和上面的DSL解析类似，Gradle也是提供了这么一套专门用来项目构建的DSL解析框架，并且支持Plugin框架，允许第三方plugin起作用。

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)**。

Project和Task

每一个构建都是由一个或多个 projects 构成的，比如一个用于将JAVA代码编译生成JAR并部署的 Project，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, 或者生成某个目录的压缩文件，Project为Task提供执行上下文。

下面展示一个最简单的task的例子：

build.gradle


```
task hello {
    doLast {
        println 'HelloWorld'
    }
}
```

通过gradle hello执行，输出为：

```
Zheng's-MacBook-Pro-2:HelloWorld zhengchen$ gradle hello

Starting a Gradle Daemon (subsequent builds will be faster)

<---

> Task :hello

HelloWorld

BUILD SUCCESSFUL in 3s
1 actionable task: 1 executed
```

任务依赖

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

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

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

Note:<<运算符通过Task leftShift(Closure action) 重载为doLast了。

intro依赖于hello，所以当执行intro时，会先执行hello，然后才会执行intro。

任务排序

目前, 有 2 种可用的排序规则: "must run after" 和 "should run after".

当你使用 "must run after" 时即意味着 taskB 必须总是在 taskA 之后运行, 无论 taskA 和 taskB 是否将要运行:

```
taskB.mustRunAfter(taskA)
```

"should run after" 规则其实和 "must run after" 很像, 只是没有那么的严格, 在 2 种情况下它会被忽略:

1. 使用规则来阐述一个执行的循环.
2. 当并行执行并且一个任务的所有依赖除了“should run after”任务其余都满足了, 那么这个任务无论它的“should run after”依赖是否执行, 它都可以执行. (编者: 翻译待商榷, 提供具体例子)

总之, 当要求不是那么严格时, “should run after”是非常有用的.

```
task taskX << {  
    println 'taskX'  
}  
task taskY << {  
    println 'taskY'  
}  
taskY.mustRunAfter taskX
```

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

任务类型

示例:

```
task copyDocs(type: Copy) {  
    from 'src/main/doc'  
    into 'build/target/doc'  
    includeEmptyDirs = false  
}
```

指定该task的类型为Copy, 其中from, into都是Copy的方法, includeEmptyDirs是Copy的属性, 通过指定Type可以一定程度上简化书写的任务代码。

Task类型及文档:

- [AntlrTask](#)
- [BuildEnvironmentReportTask](#)
- [Checkstyle](#)
- [CodeNarc](#)
- [CompareGradleBuilds](#)
- [Copy](#)
- [CreateStartScripts](#)
- [Delete](#)
- [Ear](#)
- [Exec](#)
- [FindBugs](#)
- [GenerateIvyDescriptor](#)
- [GenerateMavenPom](#)
- [GenerateBuildDashboard](#)

- [GradleBuild](#)
- [GroovyCompile](#)
- [Groovydoc](#)
- [HtmlDependencyReportTask](#)
- [JacocoReport](#)
- [JacocoMerge](#)
- [JacocoCoverageVerification](#)
- [Jar](#)
- [JavaCompile](#)
- [Javadoc](#)
- [JavaExec](#)
- [JDepend](#)
- [Pmd](#)
- [PublishToIvyRepository](#)
- [PublishToMavenRepository](#)
- [ScalaCompile](#)
- [ScalaDoc](#)
- [InitBuild](#)
- [Sign](#)
- [Sync](#)
- [Tar](#)
- [Test](#)
- [TestReport](#)
- [Upload](#)
- [War](#)
- [Wrapper](#)
- [WriteProperties](#)
- [Zip](#)

任务规则

```
tasks.addRule("Pattern: ping<ID>") { String taskName ->
    if (taskName.startsWith("ping")) {
        task(taskName) << { //动态任务
            println "Pinging: " + (taskName - 'ping')
        }
    }
}
```

```
> gradle -q pingServer1
Pinging: Server1
```

默认任务

Gradle 允许在脚本中定义一个或多个默认任务.

```
defaultTasks 'clean', 'run'

task clean << {
    println 'Default Cleaning!'
}

task run << {
    println 'Default Running!'
}

task other << {
    println "I'm not a default task!"
}
```

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

等价于 **gradle -q clean run**.

依赖管理

示例:

```
repositories {
    mavenCentral()
}

dependencies {
    compile 'com.android.support:recyclerview-v7:25.2.0'
    compile 'com.android.support:cardview-v7:25.2.0'
    testCompile 'junit:junit:4.12'
    testCompile 'com.google.truth:truth:0.29'
}
```

在Java插件中, Gradle可以使用Maven和Ivy的Repository, 同时它还可以使用本地文件系统作为Repository。不同的依赖可能用于不同的情景下, 比如编译时依赖这一组, 而运行时又依赖另一组, 所以在 Gradle 里, 依赖可以组合成*configurations(配置)*, 比如compile 就是一个configuration (其实就是依赖组的意思)。Java插件定义了以下几种以来配置。当然我们也可以自定义一个新的配置。

名称	扩展	被使用时运行的任务	含义
compile	-	compileJava	编译时的依赖
runtime	compile	-	运行时的依赖
testCompile	compile	compileTestJava	编译测试所需的额外依赖
testRuntime	runtime	test	仅供运行测试的额外依赖
archives	-	uploadArchives	项目产生的信息单元（如:jar包）
default	runtime	-	使用其他项目的默认依赖项，包括该项目产生的信息单元以及依赖

如果我们想定义一组新的依赖配置，可以通过以下方式：

```

configurations {
    pmd
}
dependencies {
    pmd group: 'pmd', name: 'pmd', version: '4.2.5'
}

task check << {
    ant.taskdef(name: 'pmd', classname: 'net.sourceforge.pmd.ant.PMDTask',
    classpath: configurations.pmd.asPath)
}

```

这是Gradle官网的例子：在Gradle中调用Ant，首先我们通过Configuration声明一组依赖，然后在Ant定义中将该Configuration所表示的classpath传给Ant：

Gradle属性（Property）

默认属性

Gradle在默认情况下已经为Project定义了很多Property，其中比较常用的有：

<https://docs.gradle.org/current/dsl/org.gradle.api.Project.html>

属性	含义
project	Project本身
name	Project的名字
path	Project的绝对路径
description	Project的描述信息
buildDir	Project构建结果存放目录
version	Project的版本号

```
task showProjectProperties << {  
    println project.name  
    println project.buildDir  
}
```

结果

```
Property  
/Users/zhengchen/Documents/笔记/Gradle使用/Demo/Gradle/Property/build
```

build.gradle文件中定义

```
ext.property1 = "this is property1"  
ext {  
    property2 = "this is property2"  
}  
task showProperties << {  
    println property1  
    println property2  
}
```

结果

```
> Task :showProperties  
this is property1  
this is property2
```

命令行参数传递属性

```
task showCommandLieProperties << {  
    println property3  
}
```

```
gradle showCommandLieProperties
FAILURE: Build failed with an exception.
> Could not get unknown property 'property3' for task ':showCommandLieProperties' of
type org.gradle.api.DefaultTask.

gradle -Pproperty3="Hello" showCommandLieProperties
> Task :showCommandLieProperties
Hello
```

gradle.properties

在项目目录放置一个 gradle.properties 文件，这些属性可以在build.gradle中获取到

gradle.properties

```
gradlePropertiesProp=gradlePropertiesValue
sysProp=shouldBeOverWrittenBySysProp
envProjectProp=shouldBeOverWrittenByEnvProp
systemProp.system=systemValue
```

build.gradle

```
task printProps << {
    println commandLineProjectProp
    println gradlePropertiesProp
    println systemProjectProp
    println envProjectProp
}
```

文件操作

构建时经常要操作文件，所以Gradle增加了一些API专门处理文件。

定位文件：

`Project.file()`

```
// 使用一个相对路径
File configFile = file('src/config.xml')
// 使用一个绝对路径
configFile = file(configFile.absolutePath)
// 使用一个项目路径的文件对象
configFile = file(new File('src/config.xml'))`
```

文件集合

```

FileCollection collection = files('src/file1.txt',
                                new File('src/file2.txt'),
                                ['src/file3.txt', 'src/file4.txt'])
                                // 对文件集合进行迭代

collection.each {File file ->
    println file.name
}

```

文件树

文件树就是一个按照层次结构分布的文件集合,例如,一个文件树可以代表一个目录树结构或者一个 ZIP 压缩文件的内容.它被抽象为 `FileTree` 结构,`FileTree` 继承自 `FileCollection`,所以你可以像处理文件集合一样处理文件树, Gradle 有些对象实现了 `FileTree` 接口,例如 [源集合](#). 使用

`Project.fileTree()` 方法可以得到 `FileTree` 的实例,它会创建一个基于基准目录的对象,然后视需要使用一些 `Ant-style` 的包含和去除规则.

```

// 使用路径创建一个树
tree = fileTree('src').include('**/*.java')
// 遍历文件树
tree.each {File file ->
    println file
}
// 访问文件数的元素
tree.visit {element ->
    println "$element.relativePath => $element.file"
}

```

复制文件


```

task anotherCopyTask(type: Copy) {
    // 复制 src/main/webapp 目录下的所有文件
    from 'src/main/webapp'
    // 复制一个任务输出的文件
    from copyTask
    // 显式使用任务的 outputs 属性复制任务的输出文件
    from copyTaskWithPatterns.outputs
    // 复制一个 ZIP 压缩文件的内容
    from zipTree('src/main/assets.zip')
    // 最后指定目标目录
    into { getDestDir() }
    //过滤规则
    include '**/*.html'
    include '**/*.jsp'
    exclude { details -> details.file.name.endsWith('.html') &&
        details.file.text.contains('staging') }
    // 使用一个闭合映射文件名
    rename { String fileName ->
        fileName.replace('-staging-', '')
    }
    // 使用正则表达式映射文件名
    rename '(.+)-staging-(.+)', '$1$2'
    rename(/(.+)-staging-(+)/, '$1$2')
}

```

创建归档文件

```

task zip(type: Zip) {
    from 'src/dist'
    into('libs') {
        from configurations.runtime
    }
    baseName = 'customName'//属性
}

```

属性名

属性名	类型	默认值	描述
archiveName	String	baseName-appendix-version-classifier.extension,如果其中任何一个都是空的,则不添加名称	归档文件的基本文件名
archivePath	File	destinationDir/archiveName	生成归档文件的绝对路径。
destinationDir	File	取决于文档类型, JAR 和 WAR 使用 project.buildDir/distributions. project.buildDir/libraries.ZIP 和 TAR	归档文件的目录
baseName	String	project.name	归档文件名的基础部分。
appendix	String	null	归档文件名的附加部分。
version	String	project.version	归档文件名的版本部分。
classifier	String	null	归档文件名的分类部分
extension	String	取决于文档类型和压缩类型: zip, jar, war, tar, tgz 或者 tbz2.	归档文件的扩展名

基于Gradle构建Java程序

参考：

gradle应用场景最多的地方就是Java，Java工程构建是需要用到Java Plugin，Java Plugin就是在Gradle的Project中加入了很多的Task和Property，并且在项目中引入了构建生命周期的概念。

先举一个Java工程的简单例子

```

apply plugin: 'java'
repositories {
    mavenCentral()
}
dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.+'
}
uploadArchives {
    repositories {
        flatDir {
            dirs 'repos'
        }
    }
}
}

```

文件结构：

```

├─ build.gradle
├─ src
│   ├── main
│   │   ├── java
│   │   │   └── org
│   │   │       └── gradle
│   │   │           └── Person.java
│   │   └── resources
│   │       └── org
│   │           └── gradle
│   │               └── resource.xml
│   └── test
│       ├── java
│       │   └── org
│       │       └── gradle
│       │           └── PersonTest.java
│       └── resources
│           └── org
│               └── gradle
│                   └── test-resource.xml

```

实际上这个目录结构是Java插件的默认工程目录，所以我们不需要单独配置源码和资源文件的位置。

执行

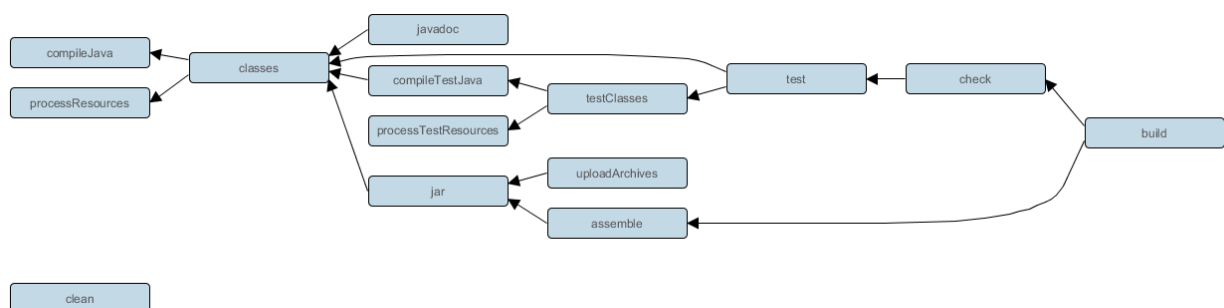
```
gradle build
```

就会分别执行

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

BUILD SUCCESSFUL

这就需要介绍一下前面提到的生命周期的概念



上图为各个任务的依赖关系，可以看出来build任务的执行顺序和上图是一致的。

Source sets

Java插件引入了Source Set的概念，source set的意思就是一组源文件（包括java源码和资源文件）。并且包含编译的classpath和运行时的classpath。比如当我们想要用一组源文件来做集成测试，另一组用来做api。这时就需要去配置sourceset了。

java插件本身定义了两个标准source set—main和test。main source包含实际用来编译成JAR文件的源码，test source则包含你的测试代码，通过JUnit或者TestNG编译并执行测试。

比如当你的工程目录没有按照Java插件的默认方式书写，可以通过配置SourceSets来指定代码和资源目录。

目录结构：

```

├─ build.gradle
├─ src
│   ├── java
│   │   ├── org
│   │   │   └── gradle
│   │   │       └── Person.java
│   └── resources
│       ├── org
│       │   └── gradle
│       │       └── resource.xml

```

```

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

```

除了指定java, resources外, sourceSets还有很多其他属性, 如compileClasspath, runtimeClasspath用来指定编译期和运行期的classpath。

同时我们也可以定义新的sourceSets, 添加新的sourceSets同时, Java 插件会增加对应的依赖配置 (确切的说是四个: *sourceSetCompile*, *sourceSetCompileOnly*, *sourceSetCompileClasspath*, *sourceSetRuntime*), 你可以为这个sourceSets定义单独的依赖:

```

sourceSets {
    intTest
}
dependencies {
    intTestCompile 'junit:junit:4.12'
    intTestRuntime 'org.ow2.asm:asm-all:4.0'
}

```

多个Projects

下面是一个典型的多项目结构

```

multiproject/
├─ api/
├─ services/webService/
├─ shared/
├─ services/shared/

```

这个工程包含了4个项目(project)，项目中间存在着依赖关系，其中api项目产生一个jar文件用来提供给用户访问webservice，webservice项目则是一个能够返回XML的webapp。shared项目包含了api和webservice共用的一些代码。services/shared则依赖shared项目。

定义一个多项目的构建，需要有一个设置文件**settings.gradle**，其中指定了该工程包含哪些项目。

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

在项目根目录下，还有一个build.gradle文件，其中包含了应用于所有子项目的配置。

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

    repositories {
        mavenCentral()
    }

    dependencies {
        testCompile 'junit:junit:4.12'
    }

    version = '1.0'

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

多项目工程中，经常要用到的是依赖另一个项目，这种项目之间的依赖通过

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

的方式进行。

Gradle在Android工程中的使用

Gradle相关文件

先看一下Android项目中常见的和gradle相关的目录和文件。

```
.
├── build.gradle
├── gradle
├── gradle.properties
└── gradlew
```

```
|— gradlew.bat
|— local.properties
|— settings.gradle
|   |— build.gradle
|— app
```

其中根目录的build.gradle 和 setting.gradle前面已经说过了。子项目中的build.gradle则是配置该子项目的。

gradle.properties前面介绍属性的时候也说过了，就是可以配置gradle的外部属性，可以在build.gradle中使用其配置的属性。

gradlew和gradlew.bat则是分别为 UN*X和Windows环境下的Gradle启动脚本，其是和gradle/wrapper一块使用的。

在第一次执行./gradlew 时，其会根据wrapper中的设置，下载对应版本的Gradle(本地已下载的话就直接使用)，这样用户就可以不用自己下载配置gradle啦，gradle/wrapper/gradle-wrapper.properties就是配置gradle使用版本，保存路径等。

local.properties则是保存的用户本地的一些配资比如sdk目录，其一般不要添加到版本控制中去。

在电脑中，先要知道Gradle 的几个存储路径，其中：

Gradle目录：/Users/zhengchen/.gradle/wrapper/dists/gradle-4.1-all

Android的gradle插件：/Users/zhengchen/.gradle/caches/modules-2/files-2.1/com.android.tools.build

这里需要明确在我们使用过程中容易混淆的概念，其中gradle-4.1-all是gradle的版本，其在Android工程中多配置在gradle/wrapper/gradle-wrapper.properties中。而配置在根目录build.gradle中的

```
dependencies {
    classpath 'com.android.tools.build:gradle:2.3.2'
}
```

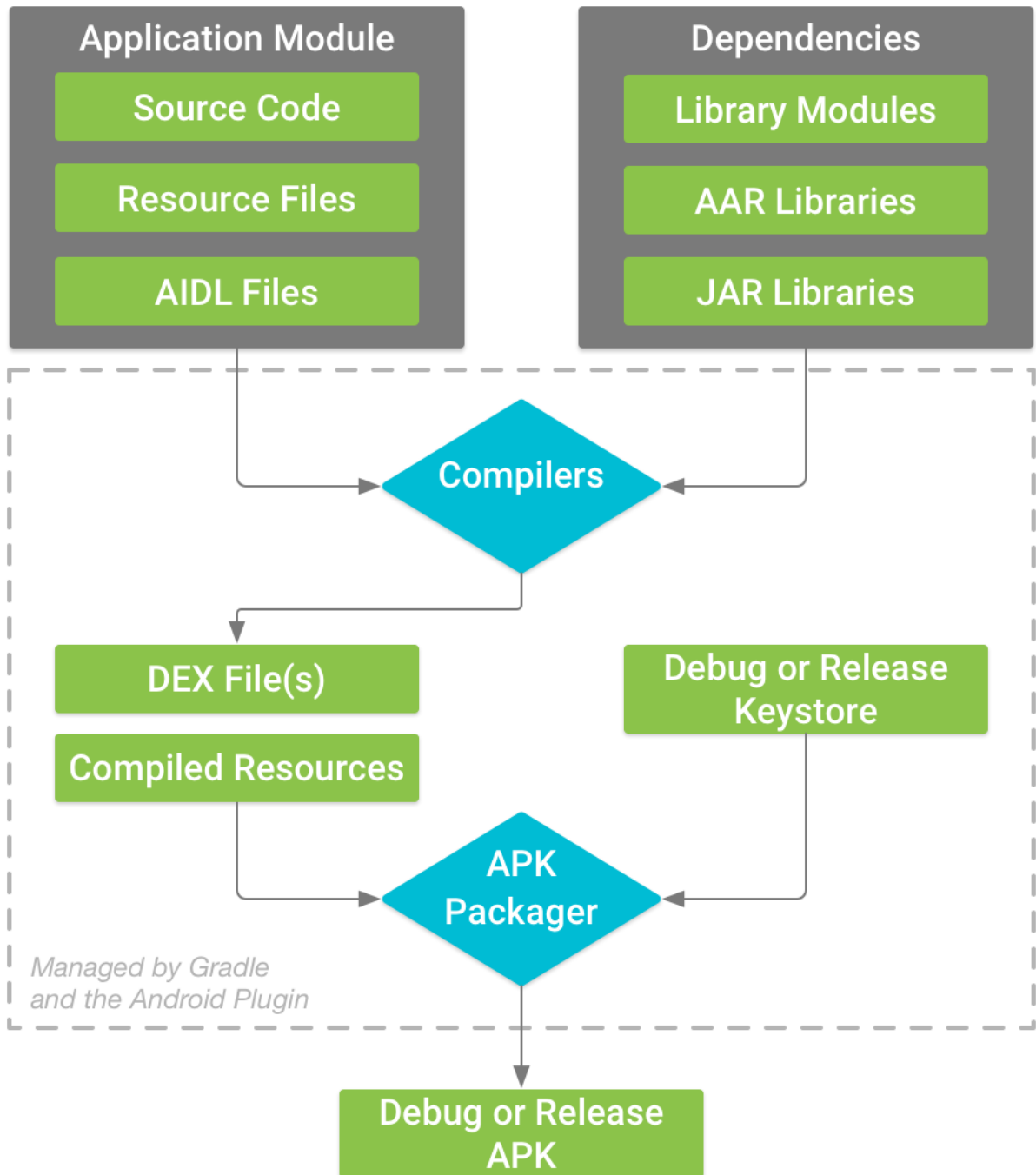
则是Android插件的版本，一般来说Android插件版本的升高，可以会带动要求gradle版本升高。

Android的gradle插件包含：

插件名	含义
com.android.application	android应用
com.android.library	Android库module
com.android.test	单元测试module
android-reporting	多项目中在根build.gradle中生成单个report

我们常用的就是application插件和library插件，分别用来构建Android应用和Android库工程。

构建流程



自定义构建变体

构建变体Build Variant包含很多个构建配置，是构建类型与产品风味的交叉产物。Gradle 和 Android 插件可以自定义构建过程中的这些配置，通过自定义配置可以使用最少的代码修改，生成不同类型的 APK。

构建类型

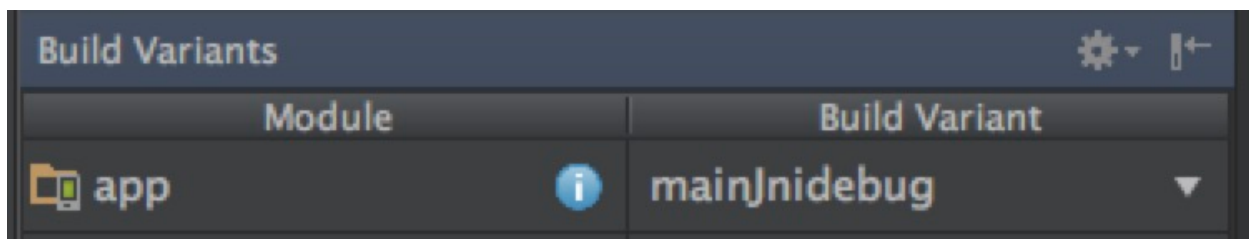
也就是buildTypes，release和debug是两个默认的构建类型，除此之外你还可以创建自己的构建类型，并可以在构建类型中配置诸如Debuggable PseudoLocalesEnabled TestCoverageEnabled JniDebuggable RenderscriptDebuggable RenderscriptOptimLevel MinifyEnabled SigningConfig 等等配置（可以查看DefaultBuildType及父类BaseConfigImpl源码）。

示例：


```

buildTypes {
    release {
        buildConfigField "boolean", "SERVER_TEST", SERVER_ENVIRONMENT_TEST
        minifyEnabled true
        shrinkResources true
        proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-
rules.pro'
    }
    debug {
        buildConfigField "boolean", "SERVER_TEST", SERVER_ENVIRONMENT_TEST
    }
    jnidebug {
        // This copies the debuggable attribute and debug signing configurations.
        initWith debug
        applicationIdSuffix ".jnidebug"
        jniDebuggable true
    }
}

```



新增加的这个构建配置jnidebug，代表其debuggable属性和签名等都和debug类型一致，applicationID增加“.jnidebug”后缀，并且允许开启jnidebug。构建出来的apk为app-main-jnidebug.apk。可以再manifest中看到package="com.sogou.gamecenter.jnidebug"。

想要知道所有的build type的属性和方法可以参考官方文档：

<http://google.github.io/android-gradle-dsl/current/com.android.build.gradle.internal.dsl.BuildType.html>

产品风味

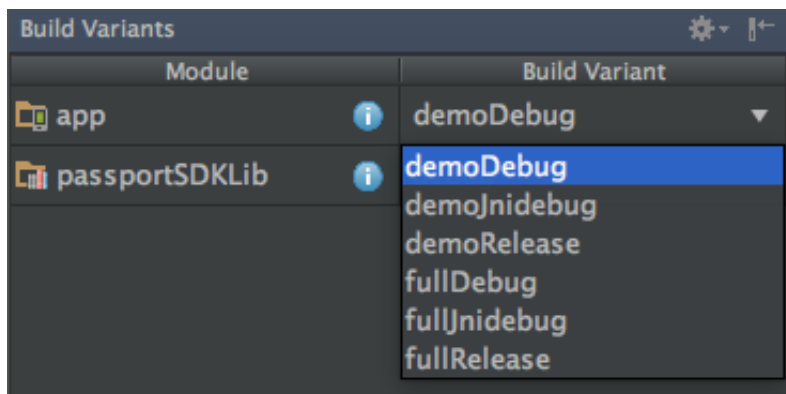
也就是productFlavors（也不知道为啥起这个名字），产品风味代表您可以发布给用户的不同应用版本，例如免费和付费的应用版本(或者国内版和国际版—采用不同的登录方式，不同的产品逻辑)。您可以将产品风味自定义为使用不同的代码和资源，同时对所有应用版本共有的部分加以共享和重复利用。

示例：

```

buildTypes {...}
productFlavors {
    demo {
        applicationIdSuffix ".demo"
        versionNameSuffix "-demo"
    }
    full {
        applicationIdSuffix ".full"
        versionNameSuffix "-full"
    }
}
}

```



除此之外还能配置什么呢？

1，配合sourceSets，可以配置不同flavors下的源码目录和资源目录（共同源码在main里配置），这样就可以做到不同应用版本的配置。

2，ProductFlavor继承DefaultProductFlavor，DefaultProductFlavor继承BaseConfigImpl。所以除了BaseConfigImpl中的ApplicationIdSuffix，BuildConfigFields等配置，还可以单独配置MinSdkVersion，TargetSdkVersion，VersionCode，VersionName等

依赖项

构建系统管理来自您的本地文件系统以及来自远程存储区的项目依赖项。范例：

```

dependencies {
    // Dependency on the "mylibrary" module from this project
    compile project(":mylibrary")
    // Remote binary dependency
    compile 'com.android.support:appcompat-v7:26.1.0'
    // Local binary dependency
    compile fileTree(dir: 'libs', include: ['*.jar'])
}

```

模块依赖项

`compile project(':mylibrary')` 行声明了一个名为“mylibrary”的[本地 Android 库模块](#)作为依赖项，并要求构建系统在构建应用时编译并包含该本地模块。

远程二进制依赖项

`compile 'com.android.support:appcompat-v7:26.1.0'` 行会通过指定其 JCenter 坐标，针对 [Android 支持库](#) 的 26.1.0 版本声明一个依赖项。默认情况下，Android Studio 会将项目配置为使用顶级构建文件中的 JCenter 存储区。当您项目与构建配置文件同步时，Gradle 会自动从 JCenter 中抽取依赖项。或者，您也可以通过[使用 SDK 管理器](#)下载和安装特定的依赖项。

本地二进制依赖项

`compile fileTree(dir: 'libs', include: ['*.jar'])` 行告诉构建系统在编译类路径和最终的应用软件包中包含 `app/libs/` 目录内的任何 JAR 文件。如果您有模块需要本地二进制依赖项，请将这些依赖项的 JAR 文件复制到项目内部的 `<moduleName>/libs` 中。

配置依赖项

除了 `compile` 关键字外，还有几个其他关键字用来配置依赖项，其决定了何时使用该依赖项。

- `compile`

指定编译时依赖项。Gradle 将此配置的依赖项添加到类路径和应用的 APK。这是默认配置。

- `apk`

指定 Gradle 需要将其与应用的 APK 一起打包的仅运行时依赖项。您可以将此配置与 JAR 二进制依赖项一起使用，而不能与其他库模块依赖项或 AAR 二进制依赖项一起使用。

- `provided`

指定 Gradle 不与应用的 APK 一起打包的编译时依赖项。如果运行时无需此依赖项，这将有助于缩减 APK 的大小（比如一些编译时注解的依赖，其作用是生成编译时的代码，并不需要将该依赖打包到 apk）。您可以将此配置与 JAR 二进制依赖项一起使用，而不能与其他库模块依赖项或 AAR 二进制依赖项一起使用。

此外，您可以通过将构建变体或测试源集的名称应用于配置关键字，为特定的构建变体或[测试源集](#)配置依赖项，如下例所示。

```
dependencies {
    fullReleaseCompile project(path: ':library', configuration: 'release')
    fullDebugCompile project(path: ':library', configuration: 'debug')
    // Adds a compile time dependency for local tests.
    testCompile 'junit:junit:4.12'
    // Adds a compile time dependency for the test APK.
    androidTestCompile 'com.android.support.test.espresso:espresso-core:2.2.2'
}
```

签名

构建系统让您能够在构建配置中指定签名设置，并可在构建过程中自动签名您的 APK。构建系统通过使用已知凭据的默认密钥和证书签名调试版本，以避免在构建时提示密码。

签名设置示例：

```

defaultConfig {...}
signingConfigs {
    release {
        storeFile file("myreleasekey.keystore")
        storePassword "password"
        keyAlias "MyReleaseKey"
        keyPassword "password"
    }
}
buildTypes {
    release {
        ...
        signingConfig signingConfigs.release
    }
}
}

```

签名的密钥放在工程中不是很安全，可以通过系统环境变量获取或者命令行获取。

```

storePassword System.getenv("KSTOREPWD")
keyPassword System.getenv("KEYPWD")

storePassword System.console().readLine("\nKeystore password: ")
keyPassword System.console().readLine("\nKey password: ")

```

常用小技巧

渠道修改

多渠道打包实际上并没有一个定式，只要app运行时能够通过某种方式获取到当前app的渠道即可。比如可以通过不同渠道的apk打包时修改为不同的applicationId，或者在AndroidManifest中修改meta-data元数据，在运行时再获取元数据的值，也可以通过资源文件的差异来区分，还有一些快速打渠道包的方案比如美团的通过在META-INF中添加渠道信息并在运行时获取的方式。

在这里使用举个通过gradle配置不同的product flavor，来修改AndroidManifest文件中的元数据，或者applicationId的方式来打渠道包。

Demo: multichannel

```

productFlavors {
    //默认, 继承自defaultConfig
    defaultFlavors {
        manifestPlaceholders = [CHANNEL: "default"]
    }
    //applicationId增加sogou后缀
    sogou {
        applicationIdSuffix "sogou"
        manifestPlaceholders = [CHANNEL: "sogou"]
    }
    //修改applicationId
    baidu {
        applicationId "com.example.zhengchen.channels.baidu"
        manifestPlaceholders = [CHANNEL: "baidu"]
    }
    //通过修改AndroidManifest.xml的meta-data来修改channel
    tencent {
        manifestPlaceholders = [CHANNEL: "tencent"]
    }
    c360 {
        manifestPlaceholders = [CHANNEL: "c360"]
    }
}

```

生成的APK改名

这个可以接上一个渠道修改，通过gradle build，自动生成多个渠道的包

```

applicationVariants.all { variant ->
    variant.outputs.each { output ->
        def file = output.outputFile
        output.outputFile = new File(file.parent,
            "${applicationId}-${variant.flavorName}-${variant.versionName}-
            ${System.currentTimeMillis()}.apk".toLowerCase())
        }
    }
}

```

生成的APK为：

```

.
├─ com.example.zhengchen.channels-c360-1.0-1507801404.apk
├─ com.example.zhengchen.channels-defaultflavors-1.0-1507801404.apk
├─ com.example.zhengchen.channels-tencent-1.0-1507801404.apk
├─ com.example.zhengchen.channels.baidu-baidu-1.0-1507801404.apk
└─ com.example.zhengchen.channels.sogou-sogou-1.0-1507801404.apk

```

除了修改文件名，还有很多属性都可以在这里统一修改，具体可参考：

<http://tools.android.com/tech-docs/new-build-system/user-guide#TOC-Shrinking-Resources>

更多的构建技巧还可以参考：<https://developer.android.com/studio/build/gradle-tips.html>

国际版和国内版

product flavor除了可以区分不同渠道，还可以拥有自己的源代码和资源文件。比如项目中有两个 product flavors，分别叫做internal和international。而国际版和国内版的大部分的源码逻辑和资源文件是一致的。但是有可能有部分的代码逻辑区别（比如国际版不允许在代码中出现获取手机imsi代码）。也可能存在不同的图片和icon资源。也可能是国际版和国内版使用不同的library（比如不同的地图sdk）

这个时候通过gradle配置不同的product flavor和dependencies。就可以满足上面的需求。

Demo: international

自动测试

我们都知道测试对于app稳定的重要性，对于android项目来说，创建项目时，就自动添加了test和androidTest两个sourceSets。同时你也可以在dependencies中配置测试任务的独立依赖。

test目录的代码是编写的本地单元测试，其不依赖android框架或者依赖一个模拟的android框架。

androidTest则是在真实硬件或者模拟器上测试，通过访问 `Instrumentation` API，利用测试代码控制被测的app，模拟用户的真实交互。

由于这里主要讲解gradle的使用，就不深入测试部分的内容了。有兴趣的同学可以分别参考<https://github.com/googlesamples/android-testing/tree/master/unit/BasicSample>

和<https://github.com/googlesamples/android-testing/tree/master/unit/BasicUnitAndroidTest>了解这两种测试。

自定义Gradle插件

插件基础

不管是java插件还是android插件，都是在gradle框架下进行的定制开发，其实我们也可以自定义插件，在使用时应用该插件后，通过插件自定义的配置，影响构建过程，输出想要的结果。

由于Gradle主要的两个概念就是Project和Task，在插件中我们通过向project中加入新的task，定义configurations和property等方式实现自定义的流程。

自定义插件其实就是一个groovy的module，插件继承implements Plugin，并且可以通过创建extensions来创建新的代码块（例如android{}，sourceSets{}都是一个extension）。

插件通过可以建立不同的task，并可以将这些task通过任务依赖或者任务排序插入到原有的构建流程中。这样在使用插件的时候其实是有两种方式的，一种方式是直接通过gradle taskName，或者可视化的gradle任务执行该task。还有一种是正常的构建流程即可，插件中的task会根据依赖关系自动执行。

自定义插件的书写可以在build.gradle写，也可以在当前工程写，不过最好是在一个单独的项目中写，然后通过maven库等分享给其他同事或用户使用。

一个独立工程的Gradle插件，就是一个Groovy工程，其产出就是一个包含插件class的jar文件，一个简单的插件工程的gradle文件类似如下：

```
apply plugin: 'groovy'

dependencies {
    compile gradleApi()
    compile localGroovy()
}
```

同时还需要在src/main/resources/META-INF/gradle-plugins目录下，提供一个properties文件，以便于Gradle能够找到这个插件。比如下文demo中的

sdk.plugin.sdkpublishplugin.properties：

```
implementation-class=sdk.plugin.SdkPublishPlugin
```

文件名和plugin的id对应（也就是使用的时候apply plugin: 'sdk.plugin.sdkpublishplugin'）。而文件内的实现类指定的是plugin实现的类名。

参考：https://docs.gradle.org/4.1/userguide/custom_plugins.html

下面通过一个实际的插件工程来演示如何书写并使用gradle插件的。

插件实战

背景：对于一款sdk项目，我们要提供给接入方sdk来接入。我们的输出就不再是一个app，而可能是：

- 1，一个library工程（不包含源码，可能是jar，也可能是aar）
- 2，一个demo工程
- 3，doc目录，内部是sdk的javadoc
- 4，接入文档
- 5，再复杂一点，可能需要输出包含模块A（比如支付模块）和不包含模块A的两个版本（甚至多个版本交叉组合）。

如果这些过程每次都通过程序员手动编译，修改，复制粘贴，多次编译等操作，一来容易出错，二来耗费大量精力。

所以这个实战就是专门写一款为**自己的sdk项目自动化打包的Android插件**。

Demo：[plugin](#)

设计思路

设计一款插件，一般不是针对某一项特殊的工程，那就失去了插件的意义，直接在gradle中写task就可以了。为了具有通用的效果，对一般的SDK项目，都会有一个library模块，一个demo模块用来展示。所以我们的思路就是，该插件应用在library模块中，并且可以配置demo模块的名称，结果输出目录，是否生成javadoc等。

最终插件使用时类似如下：

```
apply plugin: 'sdk.plugin.sdkpublishplugin'
sdkPlugin{
    //sdk的demo工程名称
    demoApp = "demo"
    //接入文档的目录
    docDir = "doc"
    //是否自动生成javaDoc
    generateJavaDoc = true
    //sdk中使用到了哪些lib
    sdklibs = ["libammsdk-2.0.jar","open_sdk_r5756_lite.jar"]
}
```

其中：

sdkPlugin	一个自定义的Extension
demoApp	demo工程名称，最终会被拷贝到输出目录
docDir	sdk文档的目录，同样会被拷贝到输出目录
generateJavaDoc	是否生成JavaDoc，是的话，会在输出目录生成Javadoc文档网页
sdklibs	配置sdk 使用到的lib库

插件工作流程：

- 1，为工程创建一个名为sdkPlugin的extension
- 2，根据sdkPlugin中几项的配置分别创建任务：拷贝任务，JavaDoc生成任务，settings.gradle生成任务。lib工程生成任务。

最终其输出的是一个包含doc，javaDoc，lib工程，demo工程的可直接导入到Android Studio中的工程。

常用Android插件

参考：

- 1, <https://docs.gradle.org/4.1/userguide/userguide.html>
- 2, 用户指南中文版: <https://github.com/DONGChuan/GradleUserGuide>
- 3, <http://ant.apache.org/ivy/history/latest-milestone/tutorial/start.html>
- 4, <http://www.groovy-lang.org/documentation.html>
- 5, <https://chaosleong.gitbooks.io/gradle-for-android/content/>
- 6, <https://developer.android.com/studio/build/index.html>
- 7, <http://www.blogjava.net/BlueSUN/archive/2007/07/15/130318.html>

8,<http://www.blogjava.net/BlueSUN/archive/2008/05/17/201026.html>

9,<http://www.cnblogs.com/davenkin/p/gradle-learning-1.html>

10,<http://tools.android.com/tech-docs/new-build-system/user-guide>