**Groovy简介与使用**

展开

* [简介](https://blog.csdn.net/jiangqian6481/article/details/83717442#%E7%AE%80%E4%BB%8B)
  + [使用](https://blog.csdn.net/jiangqian6481/article/details/83717442#%E4%BD%BF%E7%94%A8)
  + [应用](https://blog.csdn.net/jiangqian6481/article/details/83717442#%E5%BA%94%E7%94%A8)
* [Groovy语法特性(相比于Java)](https://blog.csdn.net/jiangqian6481/article/details/83717442#groovy%E8%AF%AD%E6%B3%95%E7%89%B9%E6%80%A7%E7%9B%B8%E6%AF%94%E4%BA%8Ejava)
* [Groovy与Java项目集成使用](https://blog.csdn.net/jiangqian6481/article/details/83717442#groovy%E4%B8%8Ejava%E9%A1%B9%E7%9B%AE%E9%9B%86%E6%88%90%E4%BD%BF%E7%94%A8)
  + [GroovyShell](https://blog.csdn.net/jiangqian6481/article/details/83717442#groovyshell)
  + [GroovyClassLoader](https://blog.csdn.net/jiangqian6481/article/details/83717442#groovyclassloader)
  + [GroovyScriptEngine](https://blog.csdn.net/jiangqian6481/article/details/83717442#groovyscriptengine)
  + [JSR-223](https://blog.csdn.net/jiangqian6481/article/details/83717442#jsr-223)
* [Groovy实现相关原理](https://blog.csdn.net/jiangqian6481/article/details/83717442#groovy%E5%AE%9E%E7%8E%B0%E7%9B%B8%E5%85%B3%E5%8E%9F%E7%90%86)
  + [Groovy代码文件与class文件的对应关系](https://blog.csdn.net/jiangqian6481/article/details/83717442#groovy%E4%BB%A3%E7%A0%81%E6%96%87%E4%BB%B6%E4%B8%8Eclass%E6%96%87%E4%BB%B6%E7%9A%84%E5%AF%B9%E5%BA%94%E5%85%B3%E7%B3%BB)
    - [对于没有任何类定义](https://blog.csdn.net/jiangqian6481/article/details/83717442#%E5%AF%B9%E4%BA%8E%E6%B2%A1%E6%9C%89%E4%BB%BB%E4%BD%95%E7%B1%BB%E5%AE%9A%E4%B9%89)
    - [对于仅有一个类](https://blog.csdn.net/jiangqian6481/article/details/83717442#%E5%AF%B9%E4%BA%8E%E4%BB%85%E6%9C%89%E4%B8%80%E4%B8%AA%E7%B1%BB)
    - [对于多个类](https://blog.csdn.net/jiangqian6481/article/details/83717442#%E5%AF%B9%E4%BA%8E%E5%A4%9A%E4%B8%AA%E7%B1%BB)
    - [对于有定义类的脚本](https://blog.csdn.net/jiangqian6481/article/details/83717442#%E5%AF%B9%E4%BA%8E%E6%9C%89%E5%AE%9A%E4%B9%89%E7%B1%BB%E7%9A%84%E8%84%9A%E6%9C%AC)
* [Spring对Groovy以及动态语言的支持](https://blog.csdn.net/jiangqian6481/article/details/83717442#spring%E5%AF%B9groovy%E4%BB%A5%E5%8F%8A%E5%8A%A8%E6%80%81%E8%AF%AD%E8%A8%80%E7%9A%84%E6%94%AF%E6%8C%81)
* [Groovy运行沙盒](https://blog.csdn.net/jiangqian6481/article/details/83717442#groovy%E8%BF%90%E8%A1%8C%E6%B2%99%E7%9B%92)

*本文首发于简书*[*https://www.jianshu.com/p/2c6b95097b2c*](https://www.jianshu.com/p/2c6b95097b2c)

**简介**

[Groovy](http://www.groovy-lang.org/)是构建在JVM上的一个轻量级却强大的动态语言, 它结合了Python、Ruby和Smalltalk的许多强大的特性.

Groovy就是用Java写的 , Groovy语法与Java语法类似, Groovy 代码能够与 Java 代码很好地结合，也能用于扩展现有代码, 相对于Java, 它在编写代码的灵活性上有非常明显的提升，Groovy 可以使用其他 Java 语言编写的库.

**使用**

[下载SDK](http://www.groovy-lang.org/download.html)

* Groovy Console
* 安装IDEA groovy插件

**应用**

ElasticSearch, Jenkins 都支持执行Groovy脚本  
项目构建工具Gradle就是Groovy实现的

**Groovy语法特性(相比于Java)**

1. 不需要分号
2. return关键字可省略, 方法的最后一句表达式可作为返回值返回 (视具体情况使用, 避免降低可读性)
3. 类的默认作用域是public, 不需要getter/setter方法
4. def关键字定义的变量类型都是Object, 任何变量, 方法都能用def定义/声明 , 在 Groovy 中 “一切都是对象 "
5. 导航操作符 ( ?. )可帮助实现对象引用不为空时方法才会被调用
6. // java
7. if (object != null) {
8. object.getFieldA();
9. }
10. // groovy
11. object?.getFieldA()
12. 命令链, Groovy 可以使你省略顶级语句方法调用中参数外面的括号。“命令链”功能则将这种特性继续扩展，它可以将不需要括号的方法调用串接成链，既不需要参数周围的括号，链接的调用之间也不需要点号
13. def methodA(String name) {
14. println("A: " + name)
15. return this
16. }
17. def methodB(String name) {
18. println("B: " + name)
19. return this
20. }
21. def methodC() {
22. println("C")
23. return this
24. }
25. def methodD(String name) {
26. println("D: " + name)
27. return this
28. }
29. methodA("xiaoming")
30. methodB("zhangsan")
31. methodC()
32. methodD("lisi")
33. // 不带参数的链中需要用括号
34. methodA "xiaoming" methodB "zhangsan" methodC() methodD "lisi"

闭包. 闭包是一个短的匿名代码块。每个闭包会被编译成继承groovy.lang.Closure类的类，这个类有一个叫call方法，通过该方法可以传递参数并调用这个闭包.

1. def hello = {println "Hello World"}
2. hello.call()
3. // 包含形式参数
4. def hi = {
5. person1, person2 -> println "hi " + person1 + ", "+ person2
6. }
7. hi.call("xiaoming", "xiaoli")
8. // 隐式单个参数, 'it'是Groovy中的关键字
9. def hh = {
10. println("haha, " + it)
11. }
12. hh.call("zhangsan")
13. with语法, (闭包实现)
14. // Java
15. public class JavaDeamo {
16. public static void main(String[] args) {
17. Calendar calendar = Calendar.getInstance();
18. calendar.set(Calendar.MONTH, Calendar.DECEMBER);
19. calendar.set(Calendar.DATE, 4);
20. calendar.set(Calendar.YEAR, 2018);
21. Date time = calendar.getTime();
22. System.out.println(time);
23. }
24. }
25. // Groovy
26. Calendar calendar = Calendar.getInstance()
27. calendar.with {
28. // it 指 calendar 这个引用
29. it.set(Calendar.MONTH, Calendar.DECEMBER)
30. // 可以省略it, 使用命令链
31. set Calendar.DATE, 4
32. set Calendar.YEAR, 2018
33. // calendar.getTime()
34. println(getTime())
35. // 省略get, 对于get开头的方法名并且
36. println(time)
37. }

数据结构的原生语法, 写法更便捷

1. def list = [11, 12, 13, 14] // 列表, 默认是ArrayList
2. def list = ['Angular', 'Groovy', 'Java'] as List // 字符串列表
3. // 同list.add(8)
4. list << 8
5. [1, 2, [3, 4], 5] // 嵌套列表
6. ['Groovy', 21, 2.11] // 异构的对象引用列表
7. [] // 一个空列表
8. def set = ["22", "11", "22"] as Set // LinkedHashSet, as运算符转换类型
9. def map = ['TopicName': 'Lists', 'TopicName': 'Maps'] // map, LinkedHashMap
10. [:] // 空map
11. // 循环
12. map.each {
13. print it.key
14. }

Groovy Truth

所有类型都能转成布尔值，比如null, void 对象, 等同于 0 或空的值，都会解析为false，其他则相当于true

groovy支持DSL（Domain Specific Languages领域特定语言), DSL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

借助命令链编写DSL

// groovy代码

show = { println it }

square\_root = { Math.sqrt(it) }

def please(action) {

[the: { what ->

[of: { n -> action(what(n)) }]

}]

}

// DSL 语言: please show the square\_root of 100 (请显示100的平方根)

// 调用, 等同于：please(show).the(square\_root).of(100)

please show the square\_root of 100

// ==> 10.0



1. Java 的 == 实际相当于 Groovy 的 is() 方法，而 Groovy 的 == 则是一个更巧妙的 equals()。 在Groovy中要想比较对象的引用，不能用 ==，而应该用 a.is(b)
   * <http://www.groovy-lang.org/syntax.html>
   * Differences with Java: <http://www.groovy-lang.org/differences.html>

**Groovy与Java项目集成使用**

项目中引入groovy依赖

<dependency>

<groupId>org.codehaus.groovy</groupId>

<artifactId>groovy-all</artifactId>

<version>x.y.z</version>

</dependency>



常见的集成机制:

GroovyShell

GroovyClassLoader

GroovyScriptEngine

JSR 223 javax.script API

* <http://www.groovy-lang.org/integrating.html>

**GroovyShell**

GroovyShell允许在Java类中（甚至Groovy类）求任意Groovy表达式的值。您可使用Binding对象输入参数给表达式，并最终通过GroovyShell返回Groovy表达式的计算结果

解析为脚本(groovy.lang.Script)运行

GroovyShell groovyShell = new GroovyShell();

groovyShell.evaluate("println \"hello world\"");



**GroovyClassLoader**

用 Groovy 的 GroovyClassLoader ，动态地加载一个脚本并执行它的行为。GroovyClassLoader是一个定制的类装载器，负责解释加载Java类中用到的Groovy类。

GroovyClassLoader loader = new GroovyClassLoader();

Class groovyClass = loader.parseClass(new File(groovyFileName)); // 也可以解析字符串

GroovyObject groovyObject = (GroovyObject) groovyClass.newInstance();

groovyObject.invokeMethod("run", "helloworld");



**GroovyScriptEngine**

groovy.util.GroovyScriptEngine 类为 GroovyClassLoader 其上再增添一个能够处理脚本依赖及重新加载的功能层, GroovyScriptEngine可以从指定的位置（文件系统，URL，数据库，等等）加载Groovy脚本

你可以使用一个CLASSPATH集合（url或者路径名称）初始化GroovyScriptEngine，之后便可以让它根据要求去执行这些路径中的Groovy脚本了.GroovyScriptEngine同样可以跟踪相互依赖的脚本，如果其中一个被依赖的脚本发生变更，则整个脚本树都会被重新编译和加载。

GroovyScriptEngine groovyScriptEngine = new GroovyScriptEngine(file.getAbsolutePath());

groovyScriptEngine.run("hello.groovy", new Binding())



**JSR-223**

JSR-223 是 Java 中标准的脚本框架调用 API。从 Java 6 开始引入进来，主要目用来提供一种常用框架，以便从 Java 中调用多种语言

ScriptEngine groovyEngine = new ScriptEngineManager().getEngineByName("groovy");

// 编译成类

groovyEngine.compile(script)

// 直接执行

groovyEngine.eval(script)



**Groovy实现相关原理**

groovy负责词法、语法解析groovy文件，然后用ASM生成普通的java字节码文件，供jvm使用。

**Groovy代码文件与class文件的对应关系**

作为基于JVM的语言，Groovy可以非常容易的和Java进行互操作，但也需要编译成class文件后才能运行，所以了解Groovy代码文件和class文件的对应关系，有助于更好地理解Groovy的运行方式和结构。

**对于没有任何类定义**

如果Groovy脚本文件里只有执行代码，没有定义任何类（class），则编译器会生成一个Script的子类，类名和脚本文件的文件名一样，而脚本的代码会被包含在一个名为run的方法中，同时还会生成一个main方法，作为整个脚本的入口。

**对于仅有一个类**

如果Groovy脚本文件里仅含有一个类，而这个类的名字又和脚本文件的名字一致，这种情况下就和Java是一样的，即生成与所定义的类一致的class文件, Groovy类都会实现groovy.lang.GroovyObject接口。

**对于多个类**

如果Groovy脚本文件含有一个或多个类，groovy编译器会很乐意地为每个类生成一个对应的class文件。如果想直接执行这个脚本，则脚本里的第一个类必须有一个static的main方法。

**对于有定义类的脚本**

如果Groovy脚本文件有执行代码, 并且有定义类, 那么所定义的类会生成对应的class文件, 同时, 脚本本身也会被编译成一个Script的子类，类名和脚本文件的文件名一样

**Spring对Groovy以及动态语言的支持**

Spring 从2.0开始支持将动态语言集成到基于 Spring 的应用程序中。Spring 开箱即用地支持 Groovy、JRuby 和 BeanShell。以 Groovy、JRuby 或任何受支持的语言编写的应用程序部分可以无缝地集成到 Spring 应用程序中。应用程序其他部分的代码不需要知道或关心单个 Spring bean 的实现语言。

动态语言支持将 Spring 从一个以 Java 为中心的应用程序框架改变成一个以 *JVM* 为中心的应用程序框架

Spring 通过 ScriptFactory 和 ScriptSource 接口支持动态语言集成。ScriptFactory 接口定义用于创建和配置脚本 Spring bean 的机制。理论上，所有在 JVM 上运行语言都受支持，因此可以选择特定的语言来创建自己的实现。ScriptSource 定义 Spring 如何访问实际的脚本源代码；例如，通过文件系统, URL, 数据库。

在使用基于 Groovy 的 bean 时，则有几种选择：

* 将 Groovy 类编译成普通的 Java 类文件
* 在一个 .groovy 文件中定义 Groovy 类或脚本
* 在 Spring 配置文件中以内联方式编写 Groovy 脚本

1. 配置编译的 Groovy 类, 和Java一样的用法, 定义groovy class, 使用<bean/>创建bean

class Test {

def printDate() {

println(new Date());

}

}



<bean id="test" class="com.qj.study.groovytest.spring.Test" />

* 1

ClassPathXmlApplicationContext context = newClassPathXmlApplicationContext("applicationContext.xml");

Test bean = (Test) context.getBean("test");

bean.printDate();



1. 配置来自 Groovy 脚本的 bean
   * <bean/>
   * <lang:groovy>

* <bean/>示例:

<bean id="demo" class="org.springframework.scripting.groovy.GroovyScriptFactory">

<constructor-arg value="classpath:script/ScriptBean.groovy"/>

</bean>

<bean class="org.springframework.scripting.support.ScriptFactoryPostProcessor"/>

* <lang:groovy/>示例:

<lang:groovy id="demo" script-source="classpath:script/ScriptBean.groovy">

</lang:groovy>

<bean class="org.springframework.scripting.support.ScriptFactoryPostProcessor"/>



实现过程:

Groovy 语言集成通过 ScriptFactory 的 GroovyScriptFactory 实现得到支持

当 Spring 装载应用程序上下文时，它首先创建工厂 bean（这里是GroovyScriptFactory 类型的bean）。然后，执行 ScriptFactoryPostProcessor bean中的postProcessBeforeInstantiation方法，用实际的脚本对象替换所有的工厂 bean。

ScriptFactoryPostProcessor:

public Object postProcessBeforeInstantiation(Class<?> beanClass, String beanName) {

// 只处理ScriptFactory类型的bean

if (!ScriptFactory.class.isAssignableFrom(beanClass)) {

return null;

}

// ...

// 加载并解析groovy代码, 在scriptBeanFactory中注册BeanDefinition

prepareScriptBeans(bd, scriptFactoryBeanName, scriptedObjectBeanName);

// ...

}

// prepareScriptBeans调用createScriptedObjectBeanDefinition

protected BeanDefinition createScriptedObjectBeanDefinition(BeanDefinition bd, String scriptFactoryBeanName,

ScriptSource scriptSource, @Nullable Class<?>[] interfaces) {

GenericBeanDefinition objectBd = new GenericBeanDefinition(bd);

objectBd.setFactoryBeanName(scriptFactoryBeanName);

// 指定工厂方法, ScriptFactory.getScriptedObject, 创建脚本的Java对象

objectBd.setFactoryMethodName("getScriptedObject");

objectBd.getConstructorArgumentValues().clear();

objectBd.getConstructorArgumentValues().addIndexedArgumentValue(0, scriptSource);

objectBd.getConstructorArgumentValues().addIndexedArgumentValue(1, interfaces);

return objectBd;

}



创建bean的时候, SimpleInstantiationStrategy.instantiate

// 调用工厂方法创建beanInstance

Object result = factoryMethod.invoke(factoryBean, args);

if (result == null) {

result = new NullBean();

}



GroovyScriptFactory.getScriptedObject

// 通过groovyClassLoader 加载并解析类

this.scriptClass = getGroovyClassLoader().parseClass(

scriptSource.getScriptAsString(), scriptSource.suggestedClassName());

if (Script.class.isAssignableFrom(this.scriptClass)) {

// 如果是groovy 脚本, 那么运行脚本, 将结果的类作为Bean的类型

Object result = executeScript(scriptSource, this.scriptClass);

this.scriptResultClass = (result != null ? result.getClass() : null);

return result;

}

else {

// 不是脚本, 直接返回类

this.scriptResultClass = this.scriptClass;

}



protected Object executeScript(ScriptSource scriptSource, Class<?> scriptClass) throws ScriptCompilationException {

try {

GroovyObject goo = (GroovyObject) ReflectionUtils.accessibleConstructor(scriptClass).newInstance();

// GroovyObjectCustomizer 是一个回调，Spring 在创建一个 Groovy bean 之后会调用它。可以对一个 Groovy bean 应用附加的逻辑，或者执行元编程

if (this.groovyObjectCustomizer != null) {

this.groovyObjectCustomizer.customize(goo);

}

if (goo instanceof Script) {

// A Groovy script, probably creating an instance: let's execute it.

return ((Script) goo).run();

}

else {

// An instance of the scripted class: let's return it as-is.

return goo;

}

}

catch (NoSuchMethodException ex) {

// ...

}



最终在ScriptFactoryPostProcessor中, scriptBeanFactory保存了所有通过脚本创建的bean, scriptSourceCache缓存了所有的脚本信息

final DefaultListableBeanFactory scriptBeanFactory = new DefaultListableBeanFactory();

/\*\* Map from bean name String to ScriptSource object \*/

private final Map<String, ScriptSource> scriptSourceCache = new HashMap<String, ScriptSource>();

* refresh参数

<lang:groovy id="refresh" refresh-check-delay="1000"

script-source="classpath:script/RefreshBean.groovy">

</lang:groovy>



创建的是JdkDynamicAopProxy代理对象, 在每一次调用这个代理对象的方法的时候, 都回去校验被代理对象是否需要刷新, 通过比对脚本文件的最后更新时间和设定的更新时间间隔, 如果需要刷新则重新加载这个groovy文件, 并编译, 然后创建一个新的bean并注册进行替换

3.内联方式配置

inline script标签, 从配置中读取源代码

<lang:groovy id="inline">

<lang:inline-script>

<![CDATA[

class InlineClass {

// xxxxx ...

}

]]>

</lang:inline-script>

</lang:groovy>

* 9

综上, 扩展一下, 脱离xml配置, 可以从数据库中定时加载groovy代码, 构建/更新/删除BeanDefinition

* <https://docs.spring.io/spring/docs/4.3.8.RELEASE/spring-framework-reference/html/dynamic-language.html#dynamic-language-beans-ctor-injection>

**Groovy运行沙盒**

沙盒原理也叫沙箱，英文sandbox。在计算机领域指一种虚拟技术，且多用于计算机安全技术。安全软件可以先让它在沙盒中运行，如果含有恶意行为，则禁止程序的进一步运行，而这不会对系统造成任何危害。

举个例子:

docker容器可以理解为在沙盒中运行的进程。这个沙盒包含了该进程运行所必须的资源。不同的容器之间相互隔离。CGroup实现资源控制, Namespace实现访问隔离, rootfs实现文件系统隔离。

对于嵌入Groovy的Java系统, 如果暴露接口, 可能存在的隐患有

* 通过Java的Runtime.getRuntime().exec()方法执行shell, 操作服务器…
* 执行System.exit(0)
* dump 内存中的Class, 修改内存中的缓存数据

[ElasticSearch Groovy 脚本 远程代码执行漏洞](https://www.aliyun.com/jiaocheng/296620.html)

Groovy提供了编译自定义器(Compilation customizers), 无论你使用 groovyc 还是采用 GroovyShell 来编译类，要想执行脚本，实际上都会使用到**编译器配置**（*compiler configuration*）信息。这种配置信息保存了源编码或类路径这样的信息，而且还用于执行更多的操作，比如默认添加导入，显式使用 AST(语法树) 转换，或者禁止全局 AST 转换, 编译自定义器的目标在于使这些常见任务易于实现。CompilerConfiguration 类就是切入点。

groovy sandbox的实现 -> <https://github.com/jenkinsci/groovy-sandbox>

实现过程:

groovy-sandbox实现了一个SandboxTransformer, 扩展自CompilationCustomizer, 在Groovy代码编译时进行转换. 脚本转换后, 让脚本执行的每一步都会被拦截, 调用Checker进行检查

可拦截所有内容，包括

* 方法调用（实例方法和静态方法）
* 对象分配（即除了“this（…）”和“super（…）”之外的构造函数调用
* 属性访问（例如，z = foo.bar，z = foo。“bar”）和赋值（例如，foo.bar = z，foo。“bar”= z）
* 数组访问和赋值

当然, 执行性能也会受到一些的影响

示例: Jenkins Pipline支持在Groovy沙盒中执行Groovy脚本  
