集成Groovy到Java应用中

2024-01-31

# Groovy集成机制

The Groovy language proposes several ways to integrate itself into applications (Java or even Groovy) at runtime, from the most basic, simple code execution to the most complete, integrating caching and compiler customization.

All the examples written in this section are using Groovy, but the same integration mechanisms can be used from Java.

## Eval

The groovy.util.Eval class is the simplest way to execute Groovy dynamically at runtime. This can be done by calling the me method:

import groovy.util.Eval  
  
assert Eval.me('33\*3') == 99  
assert Eval.me('"foo".toUpperCase()') == 'FOO'

Eval supports multiple variants that accept parameters for simple evaluation:

assert Eval.x(4, '2\*x') == 8   
assert Eval.me('k', 4, '2\*k') == 8   
assert Eval.xy(4, 5, 'x\*y') == 20   
assert Eval.xyz(4, 5, 6, 'x\*y+z') == 26

* Simple evaluation with one bound parameter named x
* Same evaluation, with a custom bound parameter named k
* Simple evaluation with two bound parameters named x and y
* Simple evaluation with three bound parameters named x, y and z

The Eval class makes it very easy to evaluate simple scripts, but doesn’t scale: there is no caching of the script, and it isn’t meant to evaluate more than one-liners.

## GroovyShell

### Multiple sources

The groovy.lang.GroovyShell class is the preferred way to evaluate scripts with the ability to cache the resulting script instance. Although the Eval class returns the result of the execution of the compiled script, the GroovyShell class offers more options.

def shell = new GroovyShell()   
def result = shell.evaluate '3\*5'   
def result2 = shell.evaluate(new StringReader('3\*5'))   
assert result == result2  
def script = shell.parse '3\*5'   
assert script instanceof groovy.lang.Script  
assert script.run() == 15

* create a new GroovyShell instance
* can be used as Eval with direct execution of the code
* can read from multiple sources (String, Reader, File, InputStream)
* can defer execution of the script. parse returns a Script instance
* Script defines a run method

### Sharing data between a script and the application

It is possible to share data between the application and the script using a groovy.lang.Binding:

def sharedData = new Binding()   
def shell = new GroovyShell(sharedData)   
def now = new Date()  
sharedData.setProperty('text', 'I am shared data!')   
sharedData.setProperty('date', now)   
  
String result = shell.evaluate('"At $date, $text"')   
  
assert result == "At $now, I am shared data!"

* create a new Binding that will contain shared data
* create a GroovyShell using this shared data
* add a string to the binding
* add a date to the binding (you are not limited to simple types)
* evaluate the script

Note that it is also possible to write from the script into the binding:

def sharedData = new Binding()   
def shell = new GroovyShell(sharedData)   
  
shell.evaluate('foo=123')   
  
assert sharedData.getProperty('foo') == 123

* create a new Binding instance
* create a new GroovyShell using that shared data
* use an **undeclared** variable to store the result into the binding
* read the result from the caller

It is important to understand that you need to use an undeclared variable if you want to write into the binding. Using def or an explicit type like in the example below would fail because you would then create a *local variable*:

def sharedData = new Binding()  
def shell = new GroovyShell(sharedData)  
  
shell.evaluate('int foo=123')  
  
try {  
 assert sharedData.getProperty('foo')  
} catch (MissingPropertyException e) {  
 println "foo is defined as a local variable"  
}

You must be very careful when using shared data in a multithreaded environment. The Binding instance that you pass to GroovyShell is **not** thread safe, and shared by all scripts.

It is possible to work around the shared instance of Binding by leveraging the Script instance which is returned by parse:

def shell = new GroovyShell()  
  
def b1 = new Binding(x:3)   
def b2 = new Binding(x:4)   
def script = shell.parse('x = 2\*x')  
script.binding = b1  
script.run()  
script.binding = b2  
script.run()  
assert b1.getProperty('x') == 6  
assert b2.getProperty('x') == 8  
assert b1 != b2

* will store the x variable inside b1
* will store the x variable inside b2

However, you must be aware that you are still sharing the **same instance** of a script. So this technique cannot be used if you have two threads working on the same script. In that case, you must make sure of creating two distinct script instances:

def shell = new GroovyShell()  
  
def b1 = new Binding(x:3)  
def b2 = new Binding(x:4)  
def script1 = shell.parse('x = 2\*x')   
def script2 = shell.parse('x = 2\*x')   
assert script1 != script2  
script1.binding = b1   
script2.binding = b2   
def t1 = Thread.start { script1.run() }   
def t2 = Thread.start { script2.run() }   
[t1,t2]\*.join()   
assert b1.getProperty('x') == 6  
assert b2.getProperty('x') == 8  
assert b1 != b2

* create an instance of script for thread 1
* create an instance of script for thread 2
* assign first binding to script 1
* assign second binding to script 2
* start first script in a separate thread
* start second script in a separate thread
* wait for completion

In case you need thread safety like here, it is more advisable to use the [GroovyClassLoader](#groovyclassloader) directly instead.

### Custom script class

We have seen that the parse method returns an instance of groovy.lang.Script, but it is possible to use a custom class, given that it extends Script itself. It can be used to provide additional behavior to the script like in the example below:

abstract class MyScript extends Script {  
 String name  
  
 String greet() {  
 "Hello, $name!"  
 }  
}

The custom class defines a property called name and a new method called greet. This class can be used as the script base class by using a custom configuration:

import org.codehaus.groovy.control.CompilerConfiguration  
  
def config = new CompilerConfiguration()   
config.scriptBaseClass = 'MyScript'   
  
def shell = new GroovyShell(this.class.classLoader, new Binding(), config)   
def script = shell.parse('greet()')   
assert script instanceof MyScript  
script.setName('Michel')  
assert script.run() == 'Hello, Michel!'

* create a CompilerConfiguration instance
* instruct it to use MyScript as the base class for scripts
* then use the compiler configuration when you create the shell
* the script now has access to the new method greet

You are not limited to the sole *scriptBaseClass* configuration. You can use any of the compiler configuration tweaks, including the [compilation customizers](core-domain-specific-languages.xml#compilation-customizers).

## GroovyClassLoader

In the [previous section](#integ-groovyshell), we have shown that GroovyShell was an easy tool to execute scripts, but it makes it complicated to compile anything but scripts. Internally, it makes use of the groovy.lang.GroovyClassLoader, which is at the heart of the compilation and loading of classes at runtime.

By leveraging the GroovyClassLoader instead of GroovyShell, you will be able to load classes, instead of instances of scripts:

import groovy.lang.GroovyClassLoader  
  
def gcl = new GroovyClassLoader()   
def clazz = gcl.parseClass('class Foo { void doIt() { println "ok" } }')   
assert clazz.name == 'Foo'   
def o = clazz.newInstance()   
o.doIt()

* create a new GroovyClassLoader
* parseClass will return an instance of Class
* you can check that the class which is returns is really the one defined in the script
* and you can create a new instance of the class, which is not a script
* then call any method on it

A GroovyClassLoader keeps a reference of all the classes it created, so it is easy to create a memory leak. In particular, if you execute the same script twice, if it is a String, then you obtain two distinct classes!

import groovy.lang.GroovyClassLoader  
  
def gcl = new GroovyClassLoader()  
def clazz1 = gcl.parseClass('class Foo { }')   
def clazz2 = gcl.parseClass('class Foo { }')   
assert clazz1.name == 'Foo'   
assert clazz2.name == 'Foo'  
assert clazz1 != clazz2

* dynamically create a class named "Foo"
* create an identical looking class, using a separate parseClass call
* make sure both classes have the same name
* but they are actually different!

The reason is that a GroovyClassLoader doesn’t keep track of the source text. If you want to have the same instance, then the source **must** be a file, like in this example:

def gcl = new GroovyClassLoader()  
def clazz1 = gcl.parseClass(file)   
def clazz2 = gcl.parseClass(new File(file.absolutePath))   
assert clazz1.name == 'Foo'   
assert clazz2.name == 'Foo'  
assert clazz1 == clazz2

* parse a class from a File
* parse a class from a distinct file instance, but pointing to the same physical file
* make sure our classes have the same name
* but now, they are the same instance

Using a File as input, the GroovyClassLoader is capable of **caching** the generated class file, which avoids creating multiple classes at runtime for the same source.

## GroovyScriptEngine

The groovy.util.GroovyScriptEngine class provides a flexible foundation for applications which rely on script reloading and script dependencies. While GroovyShell focuses on standalone Scripts and GroovyClassLoader handles dynamic compilation and loading of any Groovy class, the GroovyScriptEngine will add a layer on top of GroovyClassLoader to handle both script dependencies and reloading.

To illustrate this, we will create a script engine and execute code in an infinite loop. First of all, you need to create a directory with the following script inside:

**ReloadingTest.groovy**

class Greeter {  
 String sayHello() {  
 def greet = "Hello, world!"  
 greet  
 }  
}  
  
new Greeter()

then you can execute this code using a GroovyScriptEngine:

def binding = new Binding()  
def engine = new GroovyScriptEngine([tmpDir.toURI().toURL()] as URL[])   
  
while (true) {  
 def greeter = engine.run('ReloadingTest.groovy', binding)   
 println greeter.sayHello()   
 Thread.sleep(1000)  
}

* create a script engine which will look for sources into our source directory
* execute the script, which will return an instance of Greeter
* print the greeting message

At this point, you should see a message printed every second:

Hello, world!  
Hello, world!  
...

**Without** interrupting the script execution, now replace the contents of the ReloadingTest file with:

**ReloadingTest.groovy**

class Greeter {  
 String sayHello() {  
 def greet = "Hello, Groovy!"  
 greet  
 }  
}  
  
new Greeter()

And the message should change to:

Hello, world!  
...  
Hello, Groovy!  
Hello, Groovy!  
...

But it is also possible to have a dependency on another script. To illustrate this, create the following file into the same directory, without interrupting the executing script:

**Dependency.groovy**

class Dependency {  
 String message = 'Hello, dependency 1'  
}

and update the ReloadingTest script like this:

**ReloadingTest.groovy**

import Dependency  
  
class Greeter {  
 String sayHello() {  
 def greet = new Dependency().message  
 greet  
 }  
}  
  
new Greeter()

And this time, the message should change to:

Hello, Groovy!  
...  
Hello, dependency 1!  
Hello, dependency 1!  
...

And as a last test, you can update the Dependency.groovy file without touching the ReloadingTest file:

**Dependency.groovy**

class Dependency {  
 String message = 'Hello, dependency 2'  
}

And you should observe that the dependent file was reloaded:

Hello, dependency 1!  
...  
Hello, dependency 2!  
Hello, dependency 2!

## CompilationUnit

Ultimately, it is possible to perform more operations during compilation by relying directly on the org.codehaus.groovy.control.CompilationUnit class. This class is responsible for determining the various steps of compilation and would let you introduce new steps or even stop compilation at various phases. This is for example how stub generation is done, for the joint compiler.

However, overriding CompilationUnit is not recommended and should only be done if no other standard solution works.

Unresolved directive in guide-integrating.adoc - include::../../../subprojects/groovy-jsr223/src/spec/doc/\_integrating-jsr223.adoc[leveloffset=+1]