ADAMS

Advanced **D**ata mining **A**nd **M**achine learning **S**ystem

Module: adams-groovy

Peter Reutemann

January 23, 2012

Contents

Chapter 1

Introduction

Developing a new actor for ADAMS is not hard. But it can take too long, if
you simply want to test something: writing the code, compiling, packaging and,
finally, executing it. Dynamic languages that run in the Java Virtual Machine
(JVM), like Groovy ([2]), come in handy. Here, you simply have to write the
code and then execute it. There is no need to compile or package anything.

Groovy's syntax is similar to the Java one. Here is an example, taking a
Java class:

```
import java.util.Vector;
public class FunkyVector extends Vector {
  public String toString() {
    String result = "Funky output: ";
    result += super.toString();
    return result;
  }
}
```

And turning it into a Groovy script:

```
import java.util.Vector
class FunkyVector extends Vector {
  public String toString() {
    def result = "Funky output: "
    result += super.toString()
    return result
  }
}
```

Apart from the missing semi-colons and the missing data type when defining a
local variable, the code looks pretty much the same. But Groovy also comes
with other features, like Closures [1] or lazy transformation [2]. For more details,
you might want to check out the Groovy documentation [3].

---

[1] http://groovy.codehaus.org/Closures
[2] http://groovy.codehaus.org/Lazy+transformation
[3] http://groovy.codehaus.org/User+Guide

Chapter 2

Writing actors

Writing a Groovy actor works just like writing a regular ADAMS actor in Java.

2.1 Superclass and wrapper

First, you create an empty text file for you new actor. Second, you choose what superclass you want to derive it from:

- **Standalone** – `adams.flow.standalone.AbstractScript`
- **Source** – `adams.flow.source.AbstractScript`
- **Transformer** – `adams.flow.transformer.AbstractScript`
- **Sink** – `adams.flow.sink.AbstractScript`

This also determines, which ADAMS wrapper actor you need to use for executing your external script:

- **Standalone** – `adams.flow.standalone.Groovy`
- **Source** – `adams.flow.source.Groovy`
- **Transformer** – `adams.flow.transformer.Groovy`
- **Sink** – `adams.flow.sink.Groovy`

You simply use your script file as the `scriptFile` property and now you only have to write the actual code.

2.2   Implementation

As for writing your code, you merely have to implement all the abstract methods
from your `AbstractScript` superclass. The following code shows a minimalistic
transformer, which accepts and generates `Integer` objects. In the `doExecute()`
method, as it stands, it does not do anything with the incoming data, it merely
forwards a new `Token` with the data it received.

```
import adams.flow.core.Token
import adams.flow.transformer.AbstractScript

class SimpleTransformer extends AbstractScript {
  public String globalInfo() {
    return "My simple transformer"
  }
  public Class[] accepts() {
    return [Integer.class] as Object[]
  }
  public Class[] generates() {
    return [Integer.class] as Object[]
  }
  protected String doExecute() {
    m_OutputToken = new Token(m_InputToken.getPayload())
    return null
  }
}
```

A slightly more complex version computes the square of the incoming integer:

```
  ...
  protected String doExecute() {
    Integer input = (Integer) m_InputToken.getPayload()
    m_OutputToken = new Token(input * input)
    return null
  }
  ...
```

2.3 Parameters

Of course, most of the actors that you will write, will require some form of parametrization. Instead of defining options in the script itself, the ADAMS wrapper actor takes on the role of providing parameters. Each of the `Groovy` wrapper actors has a property called **scriptOptions** which takes a blank-separated list of key-value pairs ("key=value").

These options are available in the Groovy script via the `getAdditionalOptions()` method, returning an `adams.flow.core.AdditionalOptions` container object. This container object offers retrieval of the options via their key:

- `getBoolean(String)` and `getBoolean(String,Boolean)`
- `getInteger(String)` and `getInteger(String,Integer)`
- `getDouble(String)` and `getDouble(String,Double)`
- `getString(String)` and `getString(String,String)`

The second method listed allows you to specify a default value, in case the option was not supplied.

Assuming that we require an additional option called `add`, we can use this parameter to add to our incoming integer value in order to generate output:

```
..
protected String doExecute() {
  Integer input = (Integer) m_InputToken.getPayload()
  m_OutputToken = new Token(new Integer(input + getAdditionalOptions().getInteger("add", 1)))
  return null
}
..
```

Bibliography

[1] *ADAMS* – Advanced Data mining and Machine learning System
http://adams.cms.waikato.ac.nz/

[2] *Groovy* – An agile dynamic language for the Java Platform
http://groovy.codehaus.org/