Object Scala Found

a JSR223-compliant version of the scala interpreter

Raphaël Jolly
Databeans, Vélizy-Villacoublay, France
raphael.jolly@free.fr

Abstract

The interpreter that comes together with the Scala bundle lacks JSR223 compliance. It has done so for a few years now, even though such compatibility is deemed useful by a growing number of people. We aim to highlight the impeding issues and to propose some solutions. A working prototype is provided, up-to-date with Scala version 2.8.1.

Categories and Subject Descriptors D.3.4 [*Interpreters*]

General Terms scripting, class path, class loader

Keywords JSR223

1. Introduction

Three years ago, an enhancement request was submitted to make the Scala interpreter JSR223 compliant [5]. Since then, some progress has been made, but it has still not made it to the generally available Scala bundle. In his presentation at Scala Days 2010 [3], Michael Dürig outlined three main impediments to a satisfactory implementation :

- the Scala compiler needs a class path to load class files, whereas usually only a class loader is provided,
- JSR223's dynamic approach for passing arguments does not play well with Scala's static types,
- performance constraints require caching of pre-compiled scripts.

The present work aims to address these three issues, in order to enable an eagerly awaited impementation, for uses ranging from server side and web applications to client side applets, command-line execution or IDE-plateform integration.

Scala Days '11 June 2-3, Stanford University, California.

2. Passing statically typed arguments to the scripting engine

Passing arguments to scripts is mainly a problem for server side applications where e.g. a servlet engine needs to communicate data to Scala web templates. The javax.script API [8] doesn't allow to specify a type for passed arguments, whereas the Scala compiler needs it. One is then conducted to write such things as:

```
val request = bindings.get("request")
  .asInstanceOf[HttpServletRequest]
```

, which is a little verbose compared to what is needed in a web template. However, we can first see that this is not so bad an issue for the many other uses of the Scala interpreter, which

- don't all require passed arguments
- don't mind a little added boilerplate when they do

Second, the solution devised in [3] is departing from a strict scripting approach by requiring an enclosing declaration, what we see as not completely satisfactory, even though it allows IDE to understand the code.

So we decided to put this issue aside, and we haven't followed up on the existing attempts to solve it.

3. Caching pre-compiled scripts

The issue of pre-compilation and caching is solved in our implementation [6], by using the optional `Compilable` and `CompiledScript` interface and class of the javax.script API. This is especially useful for web applications, because in such a case one can not afford to re-compile the dynamic pages at every web request.

4. Providing a class path to the scala compiler

This last issue is the serious one, holding back a general availability of a JSR223 implementation of Scala. Having `scala.tools.nsc.Interpreter` to implement `javax.script.ScriptEngine` is easy and has been done a few times now. However such a naive implementation

quickly meets some tricky issues. An easy test to make is to run the adapted interpreter through the jrunscript utility that comes with the JDK. So, whereas the proprietary interpreter execution yields:

```
$ scala
Welcome to Scala version 2.8.1.final (Java
  HotSpot(TM) Client VM, Java 1.6.0_21).
Type in expressions to have them evaluated.
Type :help for more information.

scala> "hello world"
res0: java.lang.String = hello world

scala> $
```

, in JSR223 compatible mode one would write in contrast:

```
$ jrunscript -classpath scala-compiler.jar
  -l scala
```

However one gets the following outcome then:

```
scala> "hello world"

Failed to initialize compiler: object
  scala not found.
** Note that as of 2.8 scala does not
  assume use of the java classpath.
** For the old behavior pass -usejavacp
  to scala, or if using a Settings
** object programatically,
  settings.usejavacp.value = true.
java.lang.NullPointerException
scala> $
```

However, a working command can be recovered, like so:

```
$ jrunscript -Djava.class.path=
  scala-library.jar -Dscala.usejavacp=
  true -classpath scala-compiler.jar
  -l scala
scala> "hello world"
hello world
scala> $
```

The issue here is that not every environment will allow to give the compiler all its needed arguments. As said above, some environments only provide a class loader and not a class path, for instance : applet containers. In [2], Michael Dürig says “all languages which I looked at and which do symbol resolution at compile time use the same ’classloader.getResource(”foo.class”)’ hack to get access to a class file. Languages which require to browse the classes available to them at compile time need to resort to even more esoteric hacks”

And indeed with Scala we are in the latter case : not only do we have to take class files as resources, but in addition we need a list a all such available files. But this is not allowed by the JDK implementation.

5. Providing a class path : problem and solution

The problem is that the Scala compiler, which acts behind the scene for every instruction issued to the interpreter, needs to know what classes are available to it and what classes are not, be it only to resolve such wildcard imports as for instance:

```
import scala._
```

The solution we have devised to provide such list of classes is through the jar manifest file:

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.7.0
Created-By: 1.5.0_22-b03 (Sun Microsystems
Inc.)

Name: scala/xml/parsing/TokenTests.class

Name: scala/reflect/NamedType.class

...
```

That way, the needed list comes at no cost in the case of signed jar files , mandated in the case of applets for instance. In other cases, one can manually create the list and include it in the manifest. This is required if the target platform refuses signed jar files, as is the case for the Google App Engine, as far as our tests can tell.

In addition to its own classes which we saw are typically missing (“object scala not found”), the compiler can be given access to any number of third party libraries, provided their classes are listed in the manifest as explained. As scripting is often meant to make library access easy, this feature is of course a must.

6. Related work and conclusion

We have designed a modified Scala interpreter, which can build its classpath from its libraries’ jar manifest files. This provides for full independance w.r.t. the hosting plateform. The engine has been tested with success on the following environments, both on the client and server side:

- the jrunscript command
- the Google App Engine
- the Tomcat web server version 5.5 (with security enbled)
- a Java web start applet
- a Netbeans module

An existing Scala applet project [1] uses another technology : the URL of the jar file is derived from the one of a class

file resource, and it is added as-is to the classpath. It works well on the server side, but for a client side applet it means that the jar file will be downloaded again and again each time the applet is run, instead of being cached as is normally the case for applet libraries. In fact this problem prompted the present work, in the context of a scala computer algebra application which we wanted to make available through java web start [4, 7].

Acknowledgments

This paper is dedicated to the memory of my brother Patrice Jolly (Sep. 18, 1976 - Feb. 3, 2011)

References

[1] A. Bagwell. Simply scala. Technical report, http://www.simplyscala.com/, 2010-.

[2] M. Dürig. Introduce a classreader for scripts which need to do symbol resolution at compile time. Technical report, Apache Sling, 20O9. URL `http://issues.apache.org/jira/browse/SLING-945`.

[3] M. Dürig. Scala for scripting. Technical report, Day Software AG, 2010. URL `http://days2010.scala-lang.org/sites/days2010/files/15-5-E - Scripting - Dürig.pdf`.

[4] R. Jolly. jscl-meditor - Java symbolic computing library and mathematical editor. Technical report, http://jscl-meditor.sourceforge.net/, 2003-.

[5] R. Jolly. make scala jsr 223 compliant. Technical report, http://lampsvn.epfl.ch/trac/scala/ticket/874, May 2008.

[6] R. Jolly. Jsr223-compliant version of the scala interpreter. Technical report, http://github.com/rjolly/scala-compiler, 2011.

[7] H. Kredel and R. Jolly. Generic, Type-Safe and Object Oriented Computer Algebra Software. In *Computer Algebra in Scientific Computing*. Springer Berlin / Heidelberg, 2010. URL `http://krum.rz.uni-mannheim.de/kredel/oocas-casc2010-slides.pdf`.

[8] Sun Microsystems, Inc. JSR 223: Scripting for the Java platform. Technical report, http://scripting.dev.java.net/, 2003-2006.