今まで知らなかった 5 つの事項: Java スクリプト API

Java プラットフォームでスクリプトを使うための容易な方法

Ted Neward
Principal
Neward & Associates

2010年 7月 27日

Java™ 言語はプロジェクトによっては必要以上のものを提供しますが、スクリプト言語はパフォーマンスがあまり高くないことで有名です。Java スクリプト API (`javax.script`) を利用すると、Java プログラムからスクリプトを呼び出すことも、スクリプトから Java プログラムを呼び出すこともでき、Java 言語とスクリプト言語の両方の最高の部分を活用できることを学びましょう。

このシリーズの他の記事を見る

今日の多くの Java 開発者は Java プラットフォームでスクリプト言語を使うことに関心を持っています。しかし動的言語が Java バイトコードの中にコンパイルされてしまうと、必ずしもその動的言語を利用できるとは限りません。また、Java アプリケーションの一部をスクリプト化したり、アプリケーションに必要な特定の Java オブジェクトをスクリプトの中から呼び出したりした方が、簡単で効率的な場合があります。

そこに `javax.script` が登場します。Java 6 で導入された Java スクリプト API は、手軽で便利なスクリプト言語と堅牢な Java エコシステムの橋渡しをするものです。Java スクリプト API を使用すると、自分で作成したコードをほとんどすべてのスクリプト言語によるコードと簡単に統合できるため、誰か他の人が作成したコードを利用して簡単な問題を解決しようとする際の選択肢の幅が大きく広がります。

1. jrunscript を使って JavaScript を実行する

この連載について

皆さんは自分が Java プログラミングについて知っていると思うかもしれません。しかし実際には、ほとんどの開発者は Java プラットフォームの表面的な部分しか扱っておらず、当面の作業を完了するために十分なことしか学んでいません。この連載では、Ted Neward が Java プラットフォームのコア機能を深く掘り下げ、非常に厄介な Java プログラミングの難題の解決にも役立つ、ほとんど知られていない事実を紹介します。

Java プラットフォームの新しいリリースが公開されるごとに、JDK の bin ディレクトリーの中に新しい一連のコマンドライン・ツールが追加されています。Java 6 の場合も例外ではなく、なかでも

`jrunscript` は Java プラットフォームのユーティリティーに追加された単なる追加機能以上のものを備えています。

例えば、パフォーマンスをモニタリングするためのコマンドライン・スクリプトを作成する、という基本的な問題を考えてみてください。そのツールに (この連載の「前回の記事」で紹介した) `jmap` を使用し、5 秒ごとに Java プロセスに対して jmap を実行することで、そのプロセスの実行状態を知ることができます。通常、そうしたことをする場合にはコマンドラインのシェル・スクリプトを使いますが、この問題の場合では、サーバー・アプリケーションは Windows® や Linux® など、さまざまなプラットフォームにデプロイされています。システム管理者ならば、複数のプラットフォームで実行可能なシェル・スクリプトを作成するのは骨の折れる作業であることを証明してくれるでしょう。これに対する通常のソリューションは、Windows のバッチ・ファイルと UNIX® のシェル・スクリプトを作成し、その 2 つを同期させて使い続けることです。

しかし『The Pragmatic Programmer』を読んだ人ならば誰もが知っているように、そうした方法は DRY (Don't Repeat Yourself) 原則に大きく違反し、バグや欠陥の元になります。本来の望ましい姿としては、すべてのプラットフォームで実行可能な、OS に依存しない何らかのスクリプトを作成することです。

もちろん、Java 言語はプラットフォームに依存しませんが、「システム」言語に関してはそうとも言えません。私達に必要なものは、例えば JavaScript のようなスクリプト言語です。

リスト 1 は、私達に必要な基本的なシェルで開始されています。

リスト 1. periodic.js

```
while (true)
{
    echo("Hello, world!");
}
```

私達が否応なく Web ブラウザーを使っているおかげで、(大部分とは言えないまでも) 多くの Java 開発者は既に JavaScript (つまり ECMAScript) のことを知っています (JavaScript は ECMAScript の方言の 1 つであり、Netscape が所有しています)。問題は、システム管理者はこのスクリプト言語をどのように実行すればよいのか、という点です。

その答えはもちろん、JDK に付属している `jrunscript` ユーティリティーです (リスト 2)。

リスト 2. jrunscript

```
C:\developerWorks\5things-scripting\code\jssrc>jrunscript periodic.js
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
...
```

`for` ループを使用してスクリプトを指定回数実行してから終了するという方法もあります。基本的に、`jrunscript` を使うことで、通常 JavaScript で実行可能なことをほとんどすべて実行するこ

とができます。唯一の例外は、環境がブラウザーではないため DOM がありません。そのため、最上位レベルで利用可能な関数やオブジェクトは少し異なります。

Java 6 には JDK の一部として Rhino ECMAScript エンジンが用意されているため、`jrunscript` は渡される任意の ECMAScript コードを実行することができます。`jrunscript` は、(この場合のように) ファイルとして渡される ECMAScript コードを実行することや、より対話型のシェル環境である REPL ("Read-Evaluate-Print-Loop") で ECMAScript コードを実行することもできます。jrunscript を何も指定せずに単独で実行すると REPL シェルを利用できるようになります。

2. スクリプトから Java オブジェクトにアクセスする

JavaScript/ECMAScript コードを作成できることは素晴らしいのですが、Java 言語で使用するものをすべてゼロから作成し直したくはありません (それでは本来の目的に反します)。幸いなことに、Java スクリプト API エンジンを使用するものはすべて、Java のエコシステム全体をフルに利用することができます。なぜなら、核心部分ではすべてが相変わらず Java のバイトコードだからです。そこで先ほどの問題に戻ると、Java プラットフォームから従来の `Runtime.exec()` 呼び出しを使ってプロセスを起動することができます (リスト 3)。

リスト 3. Runtime.exec() によって jmap を起動する

```
var p = java.lang.Runtime.getRuntime().exec("jmap", [ "-histo", arguments[0] ])
p.waitFor()
```

`arguments` 配列は ECMAScript 標準に組み込まれており、この関数に渡された引数を参照します。最上位レベルのスクリプト環境では、arguments 配列はスクリプト自体に渡された引数 (コマンドライン・パラメーター) の配列です。つまりリスト 3 のスクリプトは、マッピング対象の Java プロセスの VMID を含む引数が渡されることを想定しています。

あるいは、`jmap` が Java クラスであるという事実を利用し、単純に jmap の `main()` メソッドを呼び出すこともできます (リスト 4)。この方法では、`Process` オブジェクトの `in/out/err` ストリームを「パイプ」する必要がなくなります。

リスト 4. JMap.main()

```
var args = [ "-histo", arguments[0] ]
Packages.sun.tools.jmap.JMap.main(args)
```

`Packages` 構文は Rhino ECMAScript の表記方法であり、Rhino の中に既に設定されているコアの `java.*` パッケージ以外の Java パッケージを参照するために使われます。

3. Java コードからスクリプトを呼び出す

スクリプトからの Java オブジェクトの呼び出しは話題の半分にすぎません。Java スクリプト環境には、Java コードの中からスクリプトを呼び出す機能も用意されています。そのためには、`ScriptEngine` をインスタンス化してから、スクリプトをロードして評価すればよいのです (リスト 5)。

リスト 5. Java プラットフォームでスクリプトを実行する

```
import java.io.*;
```

```
import javax.script.*;

public class App
{
    public static void main(String[] args)
    {
        try
        {
            ScriptEngine engine =
                new ScriptEngineManager().getEngineByName("javascript");
            for (String arg : args)
            {
                FileReader fr = new FileReader(arg);
                engine.eval(fr);
            }
        }
        catch(IOException ioEx)
        {
            ioEx.printStackTrace();
        }
        catch(ScriptException scrEx)
        {
            scrEx.printStackTrace();
        }
    }
}
```

`eval()` メソッドを直接 `String` に対して実行することもできます。つまりスクリプトは必ずしもファイルシステム上のファイルである必要はなく、データベースに保管されたスクリプトやユーザーが作成するスクリプトも可能であり、さらにはアプリケーションの中で状況やユーザー・アクションに基づいて作成されるスクリプトでも構わないのです。

4. Java オブジェクトをスクリプト空間にバインドする

単にスクリプトを呼び出すだけでは十分ではありません。Java 環境内で作成されたオブジェクトをスクリプトによって操作したい場合がよくあります。このような場合には、Java ホスト環境がオブジェクトを作成してバインドし、それらのオブジェクトをスクリプトが容易に発見、使用できるようにする必要があります。それを行うのが `ScriptContext` オブジェクトです (リスト 6)。

リスト 6. オブジェクトをスクリプトにバインドする

```
import java.io.*;
import javax.script.*;

public class App
{
    public static void main(String[] args)
    {
        try
        {
            ScriptEngine engine =
                new ScriptEngineManager().getEngineByName("javascript");

            for (String arg : args)
            {
                Bindings bindings = new SimpleBindings();
                bindings.put("author", new Person("Ted", "Neward", 39));
                bindings.put("title", "5 Things You Didn't Know");

                FileReader fr = new FileReader(arg);
                engine.eval(fr, bindings);
```

```
            }
        }
        catch(IOException ioEx)
        {
            ioEx.printStackTrace();
        }
        catch(ScriptException scrEx)
        {
            scrEx.printStackTrace();
        }
    }
}
```

バインドされたオブジェクトの利用方法は単純です。バインドされたオブジェクトの名前はスクリプトのレベルで、グローバルな名前空間のメンバーとして導入されています。そのため、Rhino の `Person` オブジェクトはリスト 7 のように簡単に使用することができます。

リスト 7. この記事の執筆者は誰でしょう

```
println("Hello from inside scripting!")

println("author.firstName = " + author.firstName)
```

これを見るとわかるように、JavaBeans スタイルのプロパティーが、あたかもフィールドであるかのように、名前のアクセサーにまで単純化されています。

5. 頻繁に使用されるスクリプトをコンパイルする

スクリプト言語の欠点は昔から、パフォーマンスが低いことでした。これは多くの場合、スクリプト言語が「オンザフライ」で解釈されるためです。スクリプト言語はテキストを実行しながらそのテキストを構文解析して妥当性検証しなければならないため、時間と CPU サイクルを浪費します。JVM 上で実行される多くのスクリプト言語は、入力されるコードを最終的に Java バイトコードに変換します (少なくとも、スクリプトを最初に構文解析して妥当性検証する際には Java バイトコードに変換します)。ただし、こうしてオンザフライで行われるコンパイルは Java プログラムが終了すると消えてしまいます。頻繁に使用されるスクリプトをバイトコード形式で保持しておくと、大きくパフォーマンスを改善することができます。

Java スクリプト API を使用すると、それを自然な意味のある形で行うことができます。返された `ScriptEngine` が `Compilable` インターフェースを実装している場合には、そのインターフェースの complie メソッドを使用することで、その (`String` または `Reader` として渡された) スクリプトを `CompiledScript` インスタンスにコンパイルすることができます。そしてその `CompiledScript` インスタンスと多様なバインディングを使用して、コンパイルされたコードに何度も `eval()` を実行することができます。それを示したものがリスト 8 です。

リスト 8. 解釈済みのコードをコンパイルする

```
import java.io.*;
import javax.script.*;

public class App
{
    public static void main(String[] args)
    {
```

```
        try
        {
            ScriptEngine engine =
                new ScriptEngineManager().getEngineByName("javascript");

            for (String arg : args)
            {
                Bindings bindings = new SimpleBindings();
                bindings.put("author", new Person("Ted", "Neward", 39));
                bindings.put("title", "5 Things You Didn't Know");

                FileReader fr = new FileReader(arg);
                if (engine instanceof Compilable)
                {
                    System.out.println("Compiling....");
                    Compilable compEngine = (Compilable)engine;
                    CompiledScript cs = compEngine.compile(fr);
                    cs.eval(bindings);
                }
                else
                    engine.eval(fr, bindings);
            }
        }
        catch(IOException ioEx)
        {
            ioEx.printStackTrace();
        }
        catch(ScriptException scrEx)
        {
            scrEx.printStackTrace();
        }
    }
}
```

ほとんどの場合、長期間保持されるストレージ (`servlet-context` など) に `CompiledScript` インスタンスを保持し、同じスクリプトを繰り返し再コンパイルすることがないようにする必要があります。ただしスクリプトの内容が変更された場合には、その変更を反映した新しい `CompiledScript` を作成する必要があります。いったん `CompiledScript` をコンパイルすると、その CompiledScript は元のスクリプト・ファイルの内容を実行することができません。

まとめ

Java スクリプト API は Java プログラムの範囲と機能を拡張する上で大きな前進であり、Java 環境にスクリプト言語を導入することで生産性を高めます。`jrunscript` (それほど作成が複雑なプログラムではありません) と `javax.script` を組み合わせることで、Ruby (JRuby) や ECMAScript (Rhino) といったスクリプト言語を使用する利点を活かすことができ、Java 環境のエコシステムやスケーラビリティーを犠牲にする必要がなくなります。

次回の「今まで知らなかった 5 つの事項」では JDBC を取り上げます。

著者について

Ted Neward

Ted Neward has written over 250 articles and a dozen books across many different technologies, including .NET, iOS, Java, Android, and JavaScript. He resides in Seattle with his wife, two kids, nine laptops, fourteen mobile devices, and two cats. Email him if you're interested in having him or his company work with you.