JRubyでなんちゃら

2009.12.5 サイロス誠

1.はじめに

本資料は、JAVA VM 上で動くRuby実行環境、JRubyを紹介します。2009年11月に公開されたバージョン1.4をもとに、実行方法などを講習形式で、とりとめなく紹介していきます。

2.想定している環境

本発表では、JRubyを体験していただくために、以下のソフトウェアをインストールし、利用可能になっていることを想定しています。

- JRuby本体
- JDK 6.0
- Apache ant

また、本発表では、エディタとコマンドラインを使用してJRubyなどを実行していきますので、コマンドラインで上記プログラムを実行できるように設定をお願いします。

3.jirbではじめてのJRuby

勉強会に来られた方の中には、JRuby? 何それ? という方もいらっしゃるかと思いますので、少し勉強がてらJRubyの世界になれていきましょう。
RubyにirbがあるようにJRubyにもjirbがあります。コマンドラインでjirbを実行してみましょう。

```
>jirb
```

すると、毎度おなじみの画面が出てきます。

```
>jirb
irb(main):001:0>
```

では、毎度おなじみ、Hello Worldを入れてみましょう。

```
irb(main):001:0> puts "Hello World"
Hello World
=> nil
```

はい、Rubyと同じですね。では、文字列を操作してみましょう。

```
irb(main):002:0> str = "Hello"
=> "Hello"
irb(main):003:0> puts str.reverse
olleH
=> nil
```

ちゃんとRubyしてますね。次は、モジュールを使って、ちょっとした数値演算をしてみましょう。角度60度のコサインは0.5(1/2)ですね。では、それを求めてみます。

```
irb(main):004:0> c = Math::cos(2 * Math::PI / 6)
=> 0.5
```

はい、うまくいきましたか? このように、JRubyでは、Rubyでできることはたいていできてしまいます。

4.JAVAの世界にようこそ!

JRubyはJAVA VM 上で動きます。ということは、JAVAのクラスライブラリも使えるのでは?
はい正解。それでは、JRubyでどうやってJAVAのクラスライブラリを扱うのか、jirbで試してみましょう。日付を、RubyのTimeクラスからではなく、JAVAのクラスライブラリから求めてみます。

まず最初に、クラスのインポートを行います。名前空間を面倒くさがらずに書けば使えるのですが、やはり面倒くさいのでインポートしましょう。
インポートは、**JAVAのimportと同じ**です。ただし、**ワイルドカードは使えません**。
今回は、java.util.Dateとjava.util.Calendarクラスを使うので、これら2つをインポートします。

```
irb(main):005:0> import "java.util.Date"
irb(main):006:0> import "java.util.Calendar"
```

次に、Dateクラスのインスタンス(dt)を生成します。**Rubyのnewメソッドで生成します**。

```
irb(main):007:0> dt = java.util.Date.new
=> #<Java::JavaUtil::Date:0x1b48392>
```

続いて、Calendarクラスのインスタンス(cal)を生成します。CalendarクラスのインスタンスはSingletonなので、getInstanceメソッドを使います。

```
irb(main):008:0> cal = java.util.Calendar.getInstance
=> #<Java::JavaUtil::GregorianCalendar:0x180cb01>
```

次に、dtをset_timeメソッドでcalに組み込みます。本来、JAVAでのメソッド名はsetTimeですが、JRubyでは、**自動的にメソッド名をRuby形式のメソッド名にエイリアスしています**。

```
irb(main):009:0> cal.set_time(dt)
=> nil
```

最後に年を表示してみましょう。Rubyではyearなんていうアクセサメソッドがあったりしますが、JAVAでは、指定の定数を引数にとってgetメソッドを使います。ややこしいですね。JAVAで定義されている定数は、Ruby流に**::を使ってアクセス**します。では、実行してみましょう。

```
irb(main):010:0> cal.get(java.util.Calendar::YEAR)
=> 2009
```

ちゃんと2009と出ましたね(来月実行したら2010ですね)。JRubyでは、簡単にJAVAのライブラリを**直接**利用できることがわかります。たとえば、JAVAだけで組むと非常に面倒な事例も簡単に実装できます。

・簡単なGUIプログラム(AWT,swing,JAVA 2D等)をJRubyで実装
・JAVAのライブラリのテストをRSpecで評価(JRubyでもRSpecが使えます)

次からは応用編です。

5.JRuby<=>JAVA間で配列を扱うコツ

JRubyとJAVA間でデータをやりとりするときにいちばんややこしいのが配列でしょう。Rubyではは Arrayクラスという動的な配列、JAVA ではC譲りの静的な配列です。
「ということは、java.util.Collection使わないと駄目なの?」とか聞こえそうですが、JRubyでは、その差を埋めてくれる便利なメソッドがあります。**to_java**と**to_a**です。

to_javaは、Rubyの配列をJAVAの配列に変換してくれる頼もしいメソッドです。引数として、返還後の要素の型(型名を元にしたシンボル)を指定します。

では…と、その前に、皆さんお待ちかねのJAVAプログラミングの時間です。byte[]型を使ったライブラリを作ります。一旦jirbを終了して、カレントディレクトリに、com、com/testディレクトリを作り、できたディレクトリ内にByteTest.javaを作成します。

```
package com.test;

public class ByteTest{
  public byte[] getByte(){
    byte[] bytes = {0x61, 0x62, 0x63};
    return bytes;
  }

  public void printByte(byte[] bytes){
    for(int i=0; i<bytes.length; i++){
      System.out.println(bytes[i]);
    }
  }
}
```

できたByteTest.javaをコンパイルしてクラス**com.test.ByteTest**(.class)を作成します

```
javac com/test/ByteTest.java
```

試しに、文字列をJAVAのbyte型の配列(byte[])に変換してみましょう。再びjirbを起動してから、String#unpackメソッドを使って、配列に変換します。

```
irb(main):001:0> ary = "abcde".unpack("C*")
=> [97, 98, 99, 100, 101]
```

配列aryをbyte[]に変換します。引数として、シンボル:byteを使います。実行してみると、どうやらJAVAのオブジェクトになったようです。

```
irb(main):002:0> bytes = ary.to_java(:byte)
=> #<#<Class:01x12c4c57>:0x147e668>
```

続いて、ByteTestクラスのインスタンス(bt)を生成します。これは、これまでのやり方と同じですね。

```
irb(main):003:0> bt = com.test.ByteTest.new
=> #<Java::ComTest::ByteTest:0x14c5b37>
```

では、ByteTest.printByteメソッドを呼び出します。bytesがbyte[]になっていれば正常に動くはずですが…。

```
irb(main):004:0> bt.print_byte(bytes)
97
98
99
100
101
=> nil
```

おお、ちゃんと動きましたね。これで、Rubyの配列も、気にせずにJAVAのクラスライブラリに持って行けることが分かりました。

逆に、bytes[]を受け取ったときはどうでしょうか。こんどは、ByteTest.getByteを呼び出して、bytes[]を受け取ります。

```
irb(main):005:0> b2 = bt.get_byte
=> #<#<Class:01x9fdee>:0x2bfdff>
```

何が何だか分かりませんが、JAVAのオブジェクトを受け取ったようです。それでは、to_aメソッドで変換してみましょう。

```
irb(main):006:0> b3 = b2.to_a
=> [97, 98, 99]
```

おお、ちゃんと配列をゲットできました。

本当にこれが0x61,0x62,0x63の並びなのか、16進数に変換してみましょう。

```
irb(main):007:0> b3.each{ | byte |  puts sprintf("%x", byte)}
61
62
63
=> [97, 98, 99]
```

ちゃんと求まっていますね。これで、RubyとJAVAとの相互変換ができることが分かりました。

あと、数値など以外のオブジェクトでは道でしょうか。再びjirbを終了して、JAVAのプログラムを作ります。今度は、com/test/ObjectTest.javaを作成します。**静的メソッド**として定義されていることに注意してください。

```
package com.test;

public class ObjectTest{
  public static void printObject(Object[] objects){
    String name = objects[0].getClass().getName();
    System.out.println(name);
  }
}
```

書き終えたら、コンパイルしてcom.test.ObjectTest(.class)を生成しておきます。

```
javac com/test/ObjectTest.java
```

jirbを起動して、com.test.ObjectTest.printObjectメソッドを呼び出してみましょう。メソッドの呼び方は、**特異メソッドの呼び方**と同じです。一気にやっていきます。

```
irb(main):001:0> x = [Time.now]
=> [Fri Dec 04 01:17:40 +0900 2009]
irb(main):002:0> o = x.to_java
=> #<#<Class:01x309f9f>:0x3fa6cd>
irb(main):003:0> com.test.ObjectTest.print_object(o)
java.util.Date
=> nil
```

なんと、RubyのTimeクラスは、JAVAではjava.util.Dateクラスのオブジェクトになるようです。驚きましたね。JRubyでは、既存のクラスを

ただし、自作のクラスを扱うときは、非常にややこしくなりますので割愛します。どういう風にすれば良いかは、以下のブログをご参照ください。

N2 TOOLBOX
http://blog.goo.ne.jp/ikkoan/e/2b6fb0f27ccd9a8c45b64e83d913a8f7

Prepared Mind
http://d.hatena.ne.jp/funnystone/20071030/1193695074

プログラマメモ2
http://programamemo2.blogspot.com/search/label/jruby

ここで、ひととおりの講習は終わりです。次は、いくつかおまけを用意してみました。

おまけ1 .jirbでRuby1.9使ってみたい!

現在のJRubyでは、標準では1.8.7が動作するようになっていますが、JRubyにオプションを渡すことで1.9ベースの動作になります。

```
jruby --1.9 hoge.rb
```

では、jirbではどうでしょうか。jirbのヘルプを見ても、そのようなオプションはなさそうです。では、ijrbは1.9ベースにならないのでしょうか。そんなことはありません。-Sオプションと併用することで、1.9ベースのjirbを起動できるのです。

```
jruby --1.9 -S jirb
```

試しに、バージョン番号を表示してみましょう。

```
>jruby --1.9 -S jirb
irb(main):001:0> RUBY_VERSION
=> "1.9.2dev"
```

ちゃんとRuby1.9になっているようです。本当に層でしょうか。Ruby1.9で追加されたハッシュの定義で試してみましょう(標準のjirbではエラーが出ます)

```
irb(main):002:0> {a: 1, b: 2, c: 3}
=> {:a=>1, :b=>2, :c=>3}
```

ちゃんと動いているようです。Encodingクラスはどうでしょうか。実際は無意味ですが、文字列に対してforce_encodingメソッドを使ってみます。

```
irb(main):003:0> v = "12345"
=> "12345"
irb(main):004:0> v.encoding
=> #<Encoding:ASCII-8BIT>
irb(main):005:0> v2 = v.force_encoding(Encoding::Shift_JIS)
=> "12345"
irb(main):006:0> v2.encoding
=> #<Encoding:Shift_JIS>
```

上手いこと動いていますね。JRubyの公式サイト(http://jruby.org/)では、1.9サポートは100%コンパチではないと述べていますが、十分に使えるようですね。

おまけ2.jruby-complete.jar

さて、Rubyを別の環境に移動させるときに問題となるのは、移動先の環境でもRubyをセットアップする必要があると言うことです。しかも、Rubyの実行環境にファイルがたくさんあるので、管理も大変です。しかし、JRubyでは、**jruby-complete.jar**を使えばその問題を解決できます。ここでは、簡単なjruby-complete.jarの作り方を解説します。

(注:ここでは、JRubyのサイトから、JRuby1.4.0のソースをダウンロードして展開したと想定します)

最初に、jruby本体のバイナリを作成します。ソースのディレクトリに移ってantを実行します。

```
>cd jruby-1.4.0
>ant
```

エラー無く終了すれば、JRubyのバイナリが手に入ります。次に、jruby-complete.jarを生成します。再びantを起動します。

```
>ant jar-complete
```

これで、libディレクトリにjruby-complete.jarができあがります。あとは、適当な環境に持って行って、javaコマンドから起動すれば、**お手軽JRuby環境**のできあがりです。

```
>java -jar jruby-complete.jar hoge.rb
```

あと、jruby-complete.jarを作るときに便利なのが、**自動gemインストール**です。
build.xml(makefileのようなもの)を編集して、gemファイルを指定しておくと、jruby-complete.jarを生成するときに、一緒にそのgemファイルをインストールしてくれます。つまり、**いちいちgemコマンドでインストールしなくても、gemライブラリを利用できます**(バージョンアップ毎にjruby-complete.jarを作り直す手間はありますが…)。

おまけ3.jrubyからgを使う

最近話題になっている(KOF2009やRubyConf2009)Rubyのライブラリと言えば、jugyoさんのgです。MacOSXのGrowl通知を使い、メッセージを表示する機能です。

g自体は、ネイティブなライブラリを使っていないため、JRubyから起動できます。やりかたは、ほかの環境と同じく、**gemコマンド**を使います。

```
>gem install g
JRuby limited openssl loaded. gem install jruby-openssl for full support.
http://jruby.kenai.com/pages/JRuby_Builtin_OpenSSL
Successfully installed ruby-growl-1.0.1
Successfully installed g-1.1.0
2 gems installed
Installing ri documentation for ruby-growl-1.0.1...
Installing ri documentation for g-1.1.0...
Installing RDoc documentation for ruby-growl-1.0.1...
Installing RDoc documentation for g-1.1.0...
```

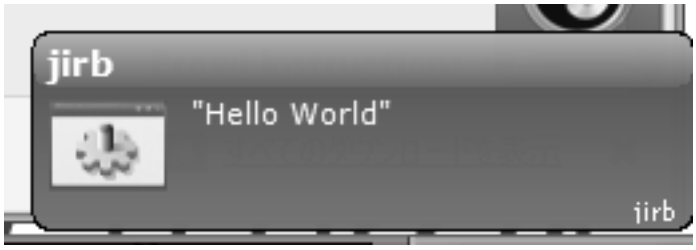
では、jirbからgを使ってみましょう。

```
>jirb
irb(main):001:0> require 'rubygems'
=> true
irb(main):002:0> require 'g'
=> true
irb(main):003:0> g "Hello World"
=> "Hello World"
```

ちなみに、標準でポップアップが表示されるのはMacOSXだけですが、Windowsでも、WindowsでGrowlと同等の機能が得られるソフトがフリーで配布されていますので、これをインストールして使ってみましょう。

Growl for Windows
http://www.growlforwindows.com/gfw/

インストール後、一旦jirbを再起動して、gメソッドを呼び出してみましょう。以下のポップアップが現れましたか?



最後に

ここまでお疲れ様でした。このように、JRubyでは、JAVAだけでは非常に煩雑なプログラムになるところを、Rubyの資産を生かして簡潔に記述できるようになります。もし、何かJAVAでプロジェクトを起こすことがあれば、JRubyとの併用も是非検討してみてください。