# 多忙な Java 開発者のための Scala ガイド: Scala とサーブレット

## サーブレットで Scala を動作させる

Ted Neward
Principal, ThoughtWorks
Neward & Associates

2008年 12月 22日

ある言語が「現実的なもの」そして「実際に使用する準備ができている」と見なされるためには、その言語は実際の環境とアプリケーションに対応できなければなりません。「多忙な Java 開発者のための Scala ガイド」シリーズの今回の記事では、Ted Neward が現実の世界での Scala のツアーを開始し、コアとなるサーブレット API と Scala がどのようにやり取りするか、そしてさらに少しばかりサーブレット API を改善する方法についても説明します。

このシリーズの他の記事を見る

Scala は確かに興味深い言語であり、言語の理論や革新における新たな素晴らしい考えを示すのには非常に適しています。しかし最終的に「現実の」使用に耐えるものであるためには、実際に業務を行う開発者の要求を多少なりとも満足し、また「現実の世界」に応用できる何かがなければなりません。

前回までに Scala 言語のコアとなる機能をいくつか調べ、Scala の言語上の柔軟性を学び、また DSL を作成しながら Scala の実際の動作も見てきたので、今度は実際のアプリケーションで使われる環境について調べ、そこで Scala がどんな役割を果たすのかを示すことにしましょう。では、このシリーズの新しいフェーズとして、ほとんどの Java™ アプリケーションの核心部分であるサーブレット API から始めることにします。

## サーブレットの復習

サーブレットの基本となるクラスの説明やチュートリアルを思い出してください。サーブレット環境の核心は、基本的に HTTP プロトコルを使用してソケット (通常はポート 80) 上でクライアントとサーバーが交換を行うところにあります。クライアントは任意の「ユーザー・エージェント」であり、これは HTTP 仕様での定義のとおりです。一方、サーバーは、最終的に `javax.servlet.Servlet` インターフェースを実装するために作成したクラスのメソッドを見つけて、ロードし、実行するサーブレット・コンテナーです。

## このシリーズについて

このシリーズでは、Ted Neward が皆さんと共に Scala プログラミング言語を深く掘り下げます。この、developerWorks の新シリーズでは、Scala が最近もてはやされている理由を調べ、Scala の言語機能の実際の動作を調べます。Scala のコードと Java のコードの比較が重要な場合には両者のコードを並べて示しますが、(これから学ぶように) Scala の機能のうちの多くは、Java には直接対応するものがありません。そして Scala の魅力の多くがあるのはそこなのです。結局のところ、Java で可能ならば、手間をかけて Scala を学ぶ必要はないのです。

通常、実際に作業を行う Java 開発者はそうしたインターフェースを直接実装するクラスは作成しません。というのも、初期のサーブレット仕様では HTTP 以外のプロトコルに対して汎用の API を提供することを想定していたため、サーブレット名前空間は以下の 2 つの部分に分割されていました。

- 「汎用」パッケージ (`javax.servlet`)
- HTTP 専用パッケージ (`javax.servlet.http`)

その結果、一部の基本機能は `javax.servlet.GenericServlet` という抽象基底クラスの汎用パッケージの中に実装されており、HTTP 専用の機能が `javax.servlet.http.HttpServlet` という派生クラスの中で実装されていました (`javax.servlet.http.HttpServlet` は通常、サーブレットの実際の「中身」に対する基底クラスとして使われました)。`HttpServlet` は `Servlet` の完全な実装を提供しており、オーバーライドされるはずの `doGet` メソッドや `doPut` メソッドに、それぞれ `GET` リクエストや `POST` リクエストを委譲したりする、といった処理を行います。

## Scala でのサーブレット

当然、誰もが最初に作成するサーブレットはおなじみの「Hello, World」サーブレットであり、Scala での最初のサーブレットの場合も例外ではありません。何年も前のサーブレットの入門チュートリアルを思い出して欲しいのですが、Java の基本的な「Hello, World」サーブレットは単純に HTML のレスポンスを出力します (リスト 1)。

### リスト 1. 想定される HTML のレスポンス

```
<HTML>
   <HEAD><TITLE>Hello, Scala!</TITLE></HEAD>
   <BODY>Hello, Scala! This is a servlet.</BODY>
</HTML>
```

これを行う単純なサーブレットは、Scala では極めて容易に作成することができ、また Scala で作成したものは Java で作成したものとほとんど同じように見えます (リスト 2)。

### リスト 2. Hello, Scala サーブレット

```
import javax.servlet.http.{HttpServlet,
  HttpServletRequest => HSReq, HttpServletResponse => HSResp}

class HelloScalaServlet extends HttpServlet
{
  override def doGet(req : HSReq, resp : HSResp) =
    resp.getWriter().print("<HTML>" +
      "<HEAD><TITLE>Hello, Scala!</TITLE></HEAD>" +
      "<BODY>Hello, Scala! This is a servlet.</BODY>" +
      "</HTML>")
}
```

インポート用のエイリアスを適切に配置することでリクエストとレスポンスの型の名前を短縮していることに注目してください。それを除くと、この Scala によるサーブレットは Java によるサーブレットとほとんど同じように見えます。これをコンパイルする際には、servlet-api.jar への参照を含めることを忘れないでください(servlet-api.jar は通常、サーブレット・コンテナーに同梱されており、Tomcat 6.0 リリースでは lib サブディレクトリーの中に隠れています)。そうしないとサーブレット API の型が見つかりません。

ただし、まだこれをそのまま使うことはできません。サーブレット仕様に従って、このサーブレットを Web アプリケーションのディレクトリーの中に (つまり .war ファイルの中に) web.xml デプロイメント記述子と共にデプロイする必要があります (web.xml デプロイメント記述子は、このサーブレットをどんな URL と組み合わせるかを記述します)。こうした単純な例では、非常に単純な URL と組み合わせることが最も簡単です (リスト 3)。

## リスト 3. web.xml デプロイメント記述子

```
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <servlet>
    <servlet-name>helloWorld</servlet-name>
    <servlet-class>HelloScalaServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>helloWorld</servlet-name>
    <url-pattern>/sayHello</url-pattern>
  </servlet-mapping>
</web-app>
```

ここから先では、読者が必要に応じてデプロイメント記述子を変更するものと想定することにします。デプロイメント記述子は Scala とは何も関係がないからです。

当然のことですが、整形式の HTML は整形式の XML と非常によく似ています。このため、Scala が XML リテラルをサポートしていることによって、このサーブレットを極めて容易に作成することができます。(「参考文献」に挙げた「Scala and XML」を参照してください。) また、`HttpServletResponse` に渡されるストリングの中にメッセージを直接埋め込む代わりに、Scala は XML リテラルのサポートを利用してメッセージそのものを XML インスタンスの中に置き、そのインスタンスを返送します。こうすることで、ロジックとプレゼンテーションの分離を (単純に、おそらく極めて単純に) 実現することができるのです。

## リスト 4. Hello, Scala サーブレット

```
import javax.servlet.http.{HttpServlet,
  HttpServletRequest => HSReq, HttpServletResponse => HSResp}

class HelloScalaServlet extends HttpServlet
{
  def message =
    <HTML>
      <HEAD><TITLE>Hello, Scala!</TITLE></HEAD>
      <BODY>Hello, Scala! This is a servlet.</BODY>
    </HTML>

  override def doGet(req : HSReq, resp : HSResp) =
    resp.getWriter().print(message)
}
```

これはつまり、Scala でのインライン式の評価機能 (ここにXML リテラルが関係します) を利用すると、サーブレットにちょっとした機能を追加することが非常に容易になるということです。例えばメッセージに現在の日付を追加することは、XML そのものの中に Calendar 式を追加することと同じくらい容易になります (例えば `{ Text(java.util.Calendar.getInstance().getTime().toString() ) }` など)。あるいは、これが少し冗長すぎると感じられるのであれば、リスト 5 のようにすることもできます。

## リスト 5. 時計付きの Hello, Scala サーブレット

```
import javax.servlet.http.{HttpServlet,
  HttpServletRequest => HSReq, HttpServletResponse => HSResp}

class HelloScalaServlet extends HttpServlet
{
  def message =
    <HTML>
      <HEAD><TITLE>Hello, Scala!</TITLE></HEAD>
      <BODY>Hello, Scala! It's now { currentDate }</BODY>
    </HTML>
  def currentDate = java.util.Calendar.getInstance().getTime()

  override def doGet(req : HSReq, resp : HSResp) =
    resp.getWriter().print(message)
}
```

要するに、Scala コンパイラーは XML オブジェクトのメッセージを 1 つの `scala.xml.Node` の中につなぎ合わせ、それをレスポンスの `Writer` の `print` メソッドに渡す際にストリングに変換するのです。

この方法は非常に現実的な方法であり、軽く見てはいけません。これはロジックとプレゼンテーションを強力に分離する方法であり、しかもこれがすべて 1 つのクラスの中で行われるのです。XML メッセージはコンパイル時にチェックされ、構文が適切なこと、そして整形式であることが確認されますが、これは標準的なサーブレット (または JSP) では実現できないことです。Scala の型推論機能のおかげで `message` と `currentDate` の前後にある実際の型情報は省略することができるため、このコードはほとんど Groovy/Grails 風の動的言語を読むのと同様に読むことができます。最初の結果としては悪くありません。

もちろん、読み取り専用のサーブレットでは何も面白味がありません。

## Hello, Scala のパラメーター

大部分のサーブレットは、単に静的なコンテンツや現在の日付と時刻などの単純なメッセージを返すだけではありません。`POST` されたフォームの内容を取得し、その内容を検証し、そしてその内容に従って処理を行います。例えば、Web アプリケーションがユーザーの素性を知ろうとして名前 (first name) と苗字 (last name) を要求するとします。

## リスト 6. チャレンジ

```
<HTML>
  <HEAD><TITLE>Who are you?</TITLE></HEAD>
  <BODY>
Who are you? Please answer:
<FORM action="/scalaExamples/sayMyName" method="POST">
Your first name: <INPUT type="text" name="firstName" />
Your last name: <INPUT type="text" name="lastName" />
<INPUT type="submit" />
</FORM>
  </BODY>
</HTML>
```

もちろん、これではユーザー・インターフェースの設計コンテストに優勝できないかもしれませんが、目的は達成することができます。これは HTML フォームであり、フォームのデータを (sayMyName という相対 URL にバインドされた) 新しい Scala サーブレットに送信しようとしています。これらのデータは (サーブレット仕様に従って) 名前と値のペアの集合に保存されます。データを取得するためには `HttpServletRequest.getParameter()` という API 呼び出しを使い、この API 呼び出しのパラメーターとして `FORM` 要素の名前が渡されます。

リスト 7 のサーブレットで行っているように、Java コードからの直接変換は比較的容易です。

## リスト 7. レスポンス (バージョン 1)

```
class NamedHelloWorldServlet1 extends HttpServlet
{
  def message(firstName : String, lastName : String) =
    <HTML>
      <HEAD><TITLE>Hello, {firstName} {lastName}!</TITLE></HEAD>
      <BODY>Hello, {firstName} {lastName}! It is now {currentTime}.</BODY>
    </HTML>
  def currentTime =
    java.util.Calendar.getInstance().getTime()

  override def doPost(req : HSReq, resp : HSResp) =
  {
    val firstName = req.getParameter("firstName")
    val lastName = req.getParameter("lastName")

    resp.getWriter().print(message(firstName, lastName))
  }
}
```

しかしこうすると、先ほど触れたメッセージの分離というメリットが一部失われることになります。なぜなら、この場合はメッセージの定義が明示的に `firstName` と `lastName` というパラメーターを取る必要があるからです。レスポンスの中で使われる要素の数が 3 から 4 個よりも大きくなると、これは面倒になります。さらに、表示用のメッセージに渡す前に `doPost` メソッド自身がすべてのパラメーターを 1 つ 1 つ抽出しなければなりません。そうした種類のコーディングは退屈であり、間違いを起こしやすくなります。

これに対する 1 つの方法は、パラメーターの抽出と、`doPost` メソッドそのものの呼び出しとに分けて、基底クラスの中に入れる方法です (リスト 8 に示すバージョン 2)。

## リスト 8. レスポンス (バージョン 2)

```
abstract class BaseServlet extends HttpServlet
```

```
{
  import scala.collection.mutable.{Map => MMap}

  def message : scala.xml.Node;

  protected var param : Map[String, String] = Map.empty
  protected var header : Map[String, String] = Map.empty

  override def doPost(req : HSReq, resp : HSResp) =
  {
    // Extract parameters
    //
    val m = MMap[String, String]()
    val e = req.getParameterNames()
    while (e.hasMoreElements())
    {
      val name = e.nextElement().asInstanceOf[String]
      m += (name -> req.getParameter(name))
    }
    param = Map.empty ++ m

    // Repeat for headers (not shown)
    //

    resp.getWriter().print(message)
  }
}
class NamedHelloWorldServlet extends BaseServlet
{
  override def message =
    <HTML>
      <HEAD><TITLE>Hello, {param("firstName")} {param("lastName")}!</TITLE></HEAD>
      <BODY>Hello, {param("firstName")} {param("lastName")}! It is now {currentTime}.
      </BODY>
    </HTML>

  def currentTime = java.util.Calendar.getInstance().getTime()
}
```

このバージョンでは、実際の表示サーブレットは (この前のものに比べ) 比較的単純に保たれ、また `param` マップと `header` マップが不変であるというさらなるメリットがあります。(リクエスト・オブジェクトを参照するメソッドとして `param` を定義することもできたことに注意してください。しかしその場合はそのリクエスト・オブジェクトをフィールドとして定義する必要があり、そうすると並行性が非常に問題になってきます。なぜならサーブレット・コンテナーは本質的に各 `do` メソッドをリエントラントと見なすからです。)

もちろん、Web アプリケーションの `FORM` を処理する上でエラー処理は一般的に重要な部分であり、また関数型言語である Scala ではすべてのものが式であるという事実から、(入力が好ましいものである場合には) メッセージを結果ページとして作成するか、あるいは (入力が好ましくない場合には) メッセージをエラー・ページとして作成することになります。従って、`firstName` と `lastName` が空の状態ではないことをチェックする検証関数はリスト 9 のようになります。

## リスト 9. レスポンス (バージョン 3)

```
class NamedHelloWorldServlet extends BaseServlet
{
  override def message =
    if (validate(param))
   <HTML>
  <HEAD><TITLE>Hello, {param("firstName")} {param("lastName")}!
```

```
            </TITLE></HEAD>
  <BODY>Hello, {param("firstName")} {param("lastName")}!
               It is now {currentTime}.</BODY>
   </HTML>
    else
      <HTML>
        <HEAD><TITLE>Error!</TITLE></HEAD>
        <BODY>How can we be friends if you don't tell me your name?!?</BODY>
      </HTML>

  def validate(p : Map[String, String]) : Boolean =
  {
    p foreach {
      case ("firstName", ") => return false
      case ("lastName", ") => return false
   //case ("lastName", v) => if (v.contains("e")) return false
   case (_, _) => ()
    }
    true
  }

  def currentTime = java.util.Calendar.getInstance().getTime()
}
```

パターン・マッチングのおかげで、非常に単純な検証ルールを容易に作成できることに注目してください (パターン・マッチングを使うと、この前の場合のように生の値に対してバインドすることもできれば、この前のコメントのように名前の中に「e」がある人を除外したい場合にはローカル変数にバインドすることもできます)。

もちろん、ここでさらに他のことをすることもできます。Java による Web アプリケーションを苦しめる昔からの問題の 1 つに SQL インジェクション攻撃があります。この攻撃では、`FORM` を介して渡されるエスケープされていない SQL コマンド文字を利用して攻撃が仕組まれ、SQL 構造を含むストリングそのものに連結することで、データベースに対して攻撃が実行されるのです。`FORM` 入力が適切かどうかの検証は、scala.regex パッケージの正規表現サポートを使って行うこともでき、さらにはこのシリーズの前回までの 3 回の記事で説明したパーサー・コンビネーターの概念を使って行うこともできます。実際、単純にデフォルトで真を返すデフォルトの検証実装を使って基底クラスの中に検証プロセス全体を入れることもできます。(単に Scala が関数型言語だからといって適切なオブジェクト設計手法を軽視しないようにしましょう。)

## まとめ

ここで作成した取るに足らない Scala サーブレット・フレームワークは、世の中にある他の Java Web フレームワークと比べるとフル機能を備えるには程遠いですが、それでも以下の 2 つの基本的な目的を果たすことができます。

- Scala の機能を興味深い方法で活用できるため、JVM プログラミングが容易になることを示すことができます。
- Web アプリケーションに Scala を使うという考え方を無理なく導入することができます。これによって自然に「lift」フレームワーク (「参考文献」を参照) を参照することになります。

今回はこれで終わりです。では次回をお楽しみに。

# ダウンロード

| 内容 | ファイル名 | サイズ |
| --- | --- | --- |
| Sample Scala code for this article | j-scala12238.zip | 179KB |

# 著者について

Ted Neward

Ted Neward は世界規模でコンサルティングを行う ThoughtWorks のコンサルタントであり、また Neward & Associates の代表として、Java や .NET、XML サービスなどのプラットフォームに関するコンサルティング、助言、指導、講演を行っています。彼はワシントン州シアトルの近郊に在住です。