

## JSTL入門 第4回: SQLおよびXMLコンテンツへのアクセス

JSPページ内のXMLおよびデータベース・コンテンツを交換するためのカスタム・タグ・ライブラリー

Mark Kolb

Software Engineer

2003年 5月 20日

Webベース・アプリケーションには、複数のサブシステムの統合がつきものです。このようなサブシステム間でデータを交換するために一般的に使用されるメカニズムとして、SQLとXMLの2つがあります。今回、Mark Kolb氏は、JSTLに関するシリーズ記事の締めくくりとして、データベースおよびJSPページ内のXMLコンテンツにアクセスするための、`sql` ライブラリーおよび`xml` ライブラリーを紹介します。

Webベース・アプリケーションの典型的なアーキテクチャーは、要求を処理するためのWebサーバー、ビジネス・ロジックを実装するためのアプリケーション・サーバー、および永続的なデータを管理するためのデータベースの、3つの層を必要とします。アプリケーション層とデータベース層の間のリンケージは、一般には、リレーショナル・データベースへのSQL呼び出しの形をとります。ビジネス・ロジックがJava言語で書かれている場合には、これらの呼び出しを実装するためにJDBCが使用されます。

アプリケーションが (ローカルであれ、リモートであれ) 追加サーバーとの統合を必要とする場合、さまざまなサブシステムの間でデータを交換するためのメカニズムも必要になります。Webアプリケーション内およびWebアプリケーション間でデータをやり取りするための方法としては、XML文書の交換が一般的になりつつあります。

これまで、JSTLを概観してきた過程で、JSTLの**式言語** (EL)、および`core` と `fmt` の両方のタグ・ライブラリーについて検討してきました。最終回となる今回の記事では、`sql` ライブラリーおよび`xml` ライブラリーについて検討します。これらのライブラリーは、その名前から連想されるように、SQLデータベースおよびXML文書から検索されたデータにアクセスし、それを操作するための、カスタム・タグを提供します。

### 本シリーズの他の記事もお読みください

第1回「**式言語**」(2003年2月)

第2回「**核心(core)に触れる**」(2003年3月)

第3回「**プレゼンテーションがすべて**」(2003年4月)

## xmlライブラリー

XMLは、構造化されたデータを表示するための柔軟な手段として設計されていて、同時にまた、妥当性検査が簡単に行えるようになっています。したがって、緩やかに結合されたシステム間でのデータ交換に適しています。このことが、XMLをWebベース・アプリケーションのための魅力的な統合テクノロジーにしています。

XMLとして表現されたデータと相互作用するための最初のステップは、そのデータをXML文書として検索し、構文解析して、文書の内容にアクセスするためのデータ構造を得ることです。文書が構文解析された後で、オプションとして、それを変換して新しいXML文書を作り、その新規文書に対して同じ操作を再度適用することができます。最後に、文書に含まれるデータを抽出して、表示したり、追加操作を行うための入力として使用したりすることができます。

これらのステップは、XMLを操作するために使用されるJSTLタグにミラーリングされます。XML文書は、第2回の「[核心\(core\)に触れる](#)」で述べたように、`<c:import>` タグを使用してcore ライブラリーから検索されます。そして、`<x:parse>` タグを使用して、Document Object Model (DOM) や Simple API for XML (SAX) などの標準XML構文解析テクノロジーをサポートし、文書を構文解析します。`<x:transform>` タグを使用すると、XML文書を変換することができます。ただし、XMLデータを変換するための標準テクノロジーeXtensible Stylesheet Language (XSL) が必要です。最後に、構文解析済みのXMLデータにアクセスして操作するために、いくつかのタグが用意されています。これらはすべて、構文解析済みのXML文書の内容を参照するために、さらに別の標準、XML Path Language (XPath) を必要とします。

### XMLの構文解析

`<x:parse>` タグは、実際には必要な構文解析のタイプに応じて、いくつかの形式で使用されます。このアクションの最も基本的な形式では、以下の構文が使用されます。

```
<x:parse xml="expression" var="name" scope="scope"
filter="expression" systemId="expression"/>
```

これらの5つの属性のうち、必須属性は`xml` 属性のみです。この属性の値は、構文解析したいXML文書を含むString、または構文解析したい文書の読み取りに使用される`java.io.Reader` のインスタンスのいずれかにする必要があります。あるいは、次の構文を使用して、構文解析したい文書を`<x:parse>` タグの本体コンテンツとして指定することもできます。

```
<x:parse var="name" scope="scope"
filter="expression" systemId="expression">
body content
</x:parse>
```

`var` 属性と`scope` 属性は、構文解析済みの文書を保管するための、有効範囲付き変数を指定します。この変数は、追加操作を行うために`xml` ライブラリー内で他のタグによって使用することができます。`var` 属性と`scope` 属性が存在する場合、構文解析済み文書を表すためにJSTLによって使用されるデータ構造は、実装に特有のタイプになりますので、ベンダーによって最適化されることがあります。

ご使用のアプリケーションが、JSTLによって提供される構文解析済み文書で操作を行う必要がある場合、代替形式の`<x:parse>` を使用することができます。この形式を使用する場合には、構文

解析済みの文書が標準インターフェースに準拠している必要があります。この場合のタグの構文は次のとおりです。

```
<x:parse xml="expression" varDom="name" scopeDom="scope"
filter="expression" systemId="expression"/>
```

このバージョンの<x:parse>を使用する場合には、構文解析済みのXML文書を表すオブジェクトがorg.w3c.dom.Document インターフェースを実装していなければなりません。また、XML文書が<x:parse>の本体コンテンツとして指定されている場合には、次のように、var およびscope 属性の代わりにvarDom およびscopeDom 属性を使用することもできます。

```
<x:parse varDom="name" scopeDom="scope"
filter="expression" systemId="expression">
body content
</x:parse>
```

## RSSについて

RDF Site Summary (RSS) は、多くのニュース指向サイトのパブリッシュに使用されているXML文書フォーマットであり、現行ヘッドラインをリストし、対応する記事にリンクするためのURLを提供します。したがって、RSSは、ニュース項目をWebで発表するための簡単なメカニズムを備えています。RSSの詳細については、[参考文献](#)を参照してください。

残りの2つの属性filter とsystemId は、構文解析の制御をよりきめ細かに行うための属性です。filter 属性では、構文解析の前に文書をフィルターに掛けるためのorg.xml.sax.XMLFilter クラスのインスタンスを指定します。この属性は、構文解析する文書が非常に大きく、当面のタスクのためにはそのうちの一部だけが必要である場合に、特に便利です。systemId 属性は、構文解析される文書のURIを示し、文書内に相対パスが示されている場合にはそれを絶対パスに変換します。この属性は、構文解析するXML文書で、構文解析プロセス中にアクセスする必要のある他の文書またはリソースを参照するために、相対URLが使用されている場合に必要です。

リスト1は、<x:parse> タグの使用法を、<c:import> との相互作用を含めて示しています。ここでは、<c:import> タグは、Slashdotという既知のWebサイトに関するRDF Site Summary (RSS) フィードを検索するために使用されています。RSSフィードを表すXML文書は<x:parse> によって構文解析され、構文解析済みの文書を表す実装に特有のデータ構造は、有効範囲としてpage が設定されたrss という名前の変数に保管されます。

### リスト1. <x:parse> アクションと <c:import> アクションの相互作用

```
<c:import var="rssFeed" url="http://slashdot.org/slashdot.rdf"/>
<x:parse var="rss" xml="{rssFeed}"/>
```

## XMLの変換

XMLは、XSLスタイルシートを使用して変換されます。JSTLは、<x:transform> タグを使用することによって、この操作をサポートします。<x:parse> の場合と同様に、<x:transform> タグは、いくつかの異なる形式をサポートします。最も基本的な形式の<x:transform> の構文は、次のとおりです。

```
<x:transform xml="expression" xslt="expression"
var="name" scope="scope"
xmlSystemId="expression" xsltSystemId="expression">
<x:param name="expression" value="expression"/>
...
</x:transform>
```

ここで、`xml` 属性では変換対象の文書が指定され、`xslt` 属性ではその変換を定義するスタイルシートが指定されています。これら2つの属性は必須属性であり、そのほかはオプションの属性です。

`<x:parse>` の`xml` 属性と同じように、`<x:transform>` の`xml` 属性の値は、XML文書を含むString、またはそのような文書にアクセスするためのReaderのいずれかです。ただし、`org.w3c.dom.Document` クラスまたは`javax.xml.transform.Source` クラスのいずれかのインスタンスの形式をとることもあります。また、`<x:parse>` アクションの`var` または`varDom` 属性のいずれかを使用して割り当てられた、変数の値となる場合もあります。

あるいは、変換したいXML文書を`<x:transform>` アクションの本体コンテンツとして組み込むこともできます。この場合の`<x:transform>` の構文は次のとおりです。

```
<x:transform xslt="expression"
var="name" scope="scope"
xmlSystemId="expression" xsltSystemId="expression">
body content
<x:param name="expression" value="expression"/>
...
</x:transform>
```

いずれの場合にも、XSLスタイルシートを指定する`xslt` 属性の値は、String またはReader、あるいは`javax.xml.transform.Source` のインスタンスのいずれかになっている必要があります。

`var` 属性が存在する場合、変換されたXML文書は、対応する有効範囲付き変数に`org.w3c.dom.Document` クラスのインスタンスとして割り当てられます。通常の場合と同様に、`scope` 属性ではこの変数割り当ての有効範囲を指定します。

`<x:transform>` タグは変換の結果を、`org.w3c.dom.Document` のインスタンスではなく、`javax.xml.transform.Result` クラスのインスタンスに保管するためのサポートも行います。`var` および`scope` 属性を省略し、`result` 属性の値としてResult オブジェクトを指定した場合、`<x:transform>` タグは、スタイルシートの適用結果を保持するためにそのオブジェクトを使用します。`<x:transform>` の`result` 属性を使用するための構文の2つのバリエーションを、リスト2に示します。

## リスト2. javax.xml.transform.Resultインスタンスを提供するために結果属性を使用する場合の、<x:transform> アクションの構文バリエーション

```
<x:transform xml="expression" xslt="expression"
result="expression"
xmlSystemId="expression" xsltSystemId="expression">
<x:param name="expression" value="expression"/>
...
</x:transform>
<x:transform xslt="expression"
result="expression"
xmlSystemId="expression" xsltSystemId="expression">
body content
<x:param name="expression" value="expression"/>
...
</x:transform>
```

これらの2つの形式の<x:transform> のいずれかを利用する場合、javax.xml.transform.Result オブジェクトが、カスタム・タグとは独立して作成されていなければなりません。このオブジェクト自体は、result 属性の値として提供されます。

var 属性もresult 属性も指定されていない場合、変換の結果は、<x:transform> アクションの処理結果としてJSPページにそのまま挿入されます。これは、リスト3に示すように、データをXMLからHTMLに変換するためにスタイルシートが使用されている場合に、特に便利です。

## リスト3. 変換済みXMLデータをJSPページで直接表示する

```
<c:import var="rssFeed" url="http://slashdot.org/slashdot.rdf"/>
<c:import var="rssToHtml" url="/WEB-INF/xslt/rss2html.xsl"/>
<x:transform xml="{rssFeed}" xslt="{rssToHtml}"/>
```

この例では、RSSフィードと該当のスタイルシートの両方が、<c:import> タグを使用して読み取られます。スタイルシートの出力はHTMLであり、これは、<x:transform> のvar 属性とresult 属性の両方を省略することにより、直接表示されます。図1は、サンプルの結果を表しています。

## 図1. リスト3の出力



<x:parse> のsystemId 属性と同じように、XML文書の相対パスを絶対パスに変換するために、<x:transform> のxmlSystemId 属性とxsltSystemId 属性が使用されています。この場合、xmlSystemId 属性は、タグのxml 属性の値として提供される文書に適用され、xsltSystemId



属性は、タグの`xslt` 属性によって指定されたスタイルシート内の相対パスを絶対パスに変換するために使用されます。

文書変換を行うスタイルシートがパラメーターを使用する場合、それらのパラメーターは`<x:param>` タグを使用して指定されます。これらのタグは、存在する場合には、`<x:transform>` タグの本体内になければなりません。変換されるXML文書も本体コンテンツとして指定されている場合、この文書はどの`<x:param>` タグよりも前になければなりません。

`<x:param>` タグには、このシリーズの[第2回](#)および[第3回](#)で述べた`<c:param>` タグおよび`<fmt:param>` タグと同じように、`name` と `value` の2つの必須属性があります。

## XMLコンテンツを使用して行う作業

構文解析および変換は、すべて、XML文書に対して行われます。ただし、文書を使用可能なフォームに変換した後では、特定のアプリケーションで利用されるのは、多くの場合、その文書に含まれているデータの特定の要素のみです。このような理由により、`xml` ライブラリーには、XML文書の個々の内容にアクセスして操作するために、複数のタグが組み込まれています。

このシリーズの第2回の記事「[核心\(core\)に触れる](#)」をお読みになった読者にとっては、これらの`xml` タグの名前はなじみがあると思います。これらのタグは、JSTLcore ライブラリーにある対応するタグを基にしたものです。これらの`core` ライブラリー・タグは、EL式を使用することにより、`value` 属性を介してJSPコンテナのデータにアクセスしますが、それに対応する`xml` ライブラリー内のタグは、XPath式を使用して、`select` 属性を介してXML文書のデータにアクセスします。

XPathは、XML文書とその属性および本体コンテンツの要素を参照するための、標準化された表記です。XPathという名前から想像されるとおり、この表記は、XPathステートメントのコンポーネントがスラッシュで区切られる点で、ファイル・システム・パスと似ています。これらのコンポーネントはXML文書のノードにマップされ、後続のコンポーネントが、ネストされた要素を突き合わせします。また、アスタリスクをワイルドカードとして使用して、複数のノードの突き合わせを行ったり、大括弧で囲んだ式を使用して、属性値の突き合わせやインデックスの指定を行ったりすることができます。XPathとその用法については、いくつかのオンライン参考文献で説明されています ([参考文献](#)を参照)。

XML文書のデータの要素を表示するには、`<x:out>` アクションを使用してください。これは、`core` ライブラリーの`<c:out>` タグに相当する、XMLにおけるアクションです。`<c:out>` には、`value` および`escapeXml` という名前の属性がありますが、`<x:out>` の属性は`select` と`escapeXml` です。

```
<x:out select="XPathExpression" escapeXml="boolean"/>
```

もちろん、`select` 属性の値がXPath式でなければならないのに対し、`<c:out>` 属性の`value` 属性はEL式でなければならない、という違いがあります。どちらのタグでも、`escapeXml` 属性の意味は同じです。

リスト4は、`<x:out>` アクションの使用法を示しています。`select` 属性に指定されたXPath式の前に、有効範囲付き変数`$rss` のためのEL式があることに注意してください。このEL式により、XPathステートメントの評価に使用される構文解析済みのXML文書が識別されます。このス

テートメントは、ここでは、親ノードの名前がchannel になっている、title という名前の要素を文書から検索し、(式の最後にある[1] インデックスの指定に従い) 検出された最初の該当要素を選択しています。<x:out> アクションは、XML文字のエスケープをオフにして、この要素の本体コンテンツを表示させます。

## リスト4. XML要素の本体コンテンツを表示するための <x:out> アクションの使用

```
<c:import var="rssFeed" url="http://slashdot.org/slashdot.rdf"/>
<x:parse var="rss" xml="{rssFeed}"/>
<x:out select="$rss//*[name()='channel']/*[name()='title']"
escapeXml="false"/>
```

xml ライブラリーには、<x:out> のほかに、XMLデータを操作するために以下のタグが組み込まれています。

- <x:set> (XPath式の値をJSTL有効範囲付き変数に割り当てます)
- <x:if> (XPath式のブール値を基に、コンテンツの条件を指定します)
- <x:choose>、<x:when>、および<x:otherwise> (XPath式を基に、相互に排他的な条件指定を実装します)
- <x:forEach> (XPath式で突き合わせされる複数の要素を繰り返します)

これらのそれぞれのタグは、core ライブラリーにおける対応するタグと同じように振る舞います。例として、<x:forEach> の使用方法をリスト5に示しておきますが、ここでは、RSSフィードを表すXML文書内のitem という名前のすべての項目で反復を行うために、<x:forEach> アクションが使用されています。<x:forEach> の本体コンテンツ内でネストされている2つの<x:out> アクションの中のXPath式は、<x:forEach> タグの反復が行われるノードを基準とした、相対的なものです。これらは、それぞれのitem 要素の下位ノードであるlink およびtitle を検索するために使用されます。

## リスト5. XMLデータの選択および表示を行うための <x:out> および <x:forEach> アクションの使用

```
<c:import var="rssFeed" url="http://slashdot.org/slashdot.rdf"/>
<x:parse var="rss" xml="{rssFeed}"/>
<a href="<x:out select="$rss//*[name()='channel']/*[name()='link']" />"
><x:out select="$rss//*[name()='channel']/*[name()='title']"
escapeXml="false"/></a>
<x:forEach select="$rss//*[name()='item']">
<li> <a href="<x:out select="."/ * [name()='link']" />"
><x:out select="."/ * [name()='title']" escapeXml="false"/></a>
</x:forEach>
```

リスト5のJSPコードの結果として得られる出力は、[図1](#) で示されている [リスト3](#) の出力と同じです。このようにxml ライブラリーのXPath指向のタグは、特に最終出力がHTMLである場合には、XMLコンテンツを変換するためのスタイルシートに変わるものです。

## sqlライブラリー

JSTLアクションの4番目の、そしてまた最後のセットは、sql カスタム・タグ・ライブラリーです。このライブラリーは、その名前から想像できるように、リレーショナル・データベースと相互作用するためのタグを提供します。より具体的には、sql ライブラリーは、データ・ソースの

指定、照会および更新の発行、およびトランザクションへの照会および更新のグループ化のための、タグを定義します。

## データ・ソース

データ・ソースは、データベース接続を獲得するためのファクトリーです。データ・ソースは、多くの場合、接続の作成および初期化に関連したオーバーヘッドを最小化するために、なんらかの形式の接続プールを実装します。Java 2 Enterprise Edition (J2EE) アプリケーション・サーバーは、一般には、データ・ソースのための組み込みサポートを備えています。J2EEアプリケーションは、Java Naming and Directory Interface (JNDI) を介してこのサポートを使用することができます。

JSTLの`sql` タグは、データ・ソースを利用して接続を獲得します。実際には、いくつかの`sql` タグには、接続ファクトリーを`javax.sql.DataSource` インターフェースのインスタンスまたはJNDI名として明示的に指定するために、オプションの`dataSource` 属性が組み込まれています。

`<sql:setDataSource>` タグを使用して`javax.sql.DataSource` のインスタンスを獲得することができます。これは、次の2つの形式で行うことができます。

```
<sql:setDataSource dataSource="expression"
var="name" scope="scope"/>
<sql:setDataSource url="expression" driver="expression"
user="expression" password="expression"
var="name" scope="scope"/>
```

最初の形式の場合には、必須属性は`dataSource` のみです。2番目の形式の場合、必須属性は`url` のみです。

JNDI名に関連したデータ・ソースにアクセスするためには、`dataSource` 属性の値としてその名前を指定して、最初の形式を使用してください。2番目の形式では、`url` 属性の値として提供されたJDBC URLを使用して、新規データ・ソースが作成されます。必要に応じ、オプションの`driver` 属性で、データベース・ドライバを実装するクラスの名前を指定し、`user` 属性および`password` 属性で、データベースにアクセスするためのログイン証明を行います。

2つのいずれの形式の`<sql:setDataSource>` の場合にも、オプションの`var` 属性と`scope` 属性は、指定されたデータ・ソースを有効範囲付き変数に割り当てます。ただし、`var` 属性が含まれていない場合には、`<sql:setDataSource>` アクションは、明示的なデータ・ソースが指定されていない`sql` タグによって使用される、デフォルトのデータ・ソースを設定することになります。

`javax.servlet.jsp.jstl.sql.dataSource` コンテキスト・パラメーターを使用して、`sql` ライブラリーのデフォルト・データ・ソースを構成することもできます。実際には、デフォルト・データ・ソースを指定するためには、リスト6に示すような項目をアプリケーションの`web.xml` ファイルに含めるのが、最も簡単な方法です。`<sql:setDataSource>` を使用してこれを行うには、アプリケーションを初期化するためにJSPページを使用する必要があり、したがってまた、そのページを自動的に実行するためになんらかの手段が必要です。



## リスト6. web.xmlデプロイメント記述子内でJSTLのデフォルト・データ・ソースを設定するためのJNDI名の使用

```
<context-param>
<param-name>javax.servlet.jsp.jstl.sql.dataSource</param-name>
<param-value>jdbc/blog</param-value>
</context-param>
```

### 照会および更新のサブミット

データ・ソースへのアクセスが確立された後で、`<sql:query>` アクションを使用して照会を実行することができ、`<sql:update>` アクションを使用してデータベースの更新を行うことができます。照会および更新はSQLステートメントとして指定されます。これらは、JDBCの`java.sql.PreparedStatement` インターフェースを基にした方法でパラメーター化することができます。パラメーター値は、ネストされた`<sql:param>` タグと`<sql:dateParam>` タグを使用して指定されます。

`<sql:query>` アクションのバリエーションとして、次の3つがサポートされます。

```
<sql:query sql="expression" dataSource="expression"
var="name" scope="scope"
maxRows="expression" startRow="expression"/>
<sql:query sql="expression" dataSource="expression"
var="name" scope="scope"
maxRows="expression" startRow="expression">
<sql:param value="expression"/>
...
</sql:query>
<sql:query dataSource="expression"
var="name" scope="scope"
maxRows="expression" startRow="expression">
SQL statement
<sql:param value="expression"/>
...
</sql:query>
```

最初の2つの形式では、`sql` 属性と`var` 属性は必須です。3番目の形式の場合、`var` 属性のみが必須です。

`var` 属性と`scope` 属性は、照会の結果を保管するための、有効範囲付き変数を指定します。`maxRows` 属性を使用すると、照会によって戻される行数を制限することができ、`startRow` 属性を使用すると、(例えば、結果セットをデータベースによって構成するときに行を飛ばす場合などに) 最初の何行かを無視することができます。

照会を実行した後で、結果セットは、`javax.servlet.jsp.jstl.sql.Result` インターフェースのインスタンスとして有効範囲付き変数に割り当てられます。このオブジェクトでは、表1にまとめたように、照会の結果セットの行、列名、およびサイズにアクセスするためのプロパティを指定します。

表1. `javax.servlet.jsp.jstl.sql.Result` インターフェースによって定義されるプロパティ

プロパティ	説明
-------	----

rows	SortedMap オブジェクトの配列であり、それぞれのオブジェクトは、列名を結果セット内の1つの行にマップします。
rowsByIndex	配列の配列であり、それぞれの配列は、結果セット内の1つの行に対応しています。
columnNames	結果セット内の列の名前を付けるストリングの配列であり、それぞれの列は、rowsByIndexプロパティの場合と同じ順序になっています。
rowCount	照会結果に含まれる行の総数です。
limitedByMaxRows	照会がmaxRows 属性の値によって制限されていた場合には、trueになります。

これらのプロパティの中で、rows は、結果セットの中での繰り返しを行ったり、名前を使用して列データにアクセスしたりするために使用できるので、特に便利です。これについては、[リスト7](#)で示してあります。このリストでは、照会の結果はqueryResults という有効範囲付き変数に割り当てられ、その行は、core ライブラリーの<c:forEach> タグを使用して繰り返されます。ネストされた<c:out> タグは、Map コレクションのためのELの組み込みサポートを利用して、列名に対応する行データを検索します。(第1回で述べたように、`${row.title}` と `${row["title"]}` は等価な式です)

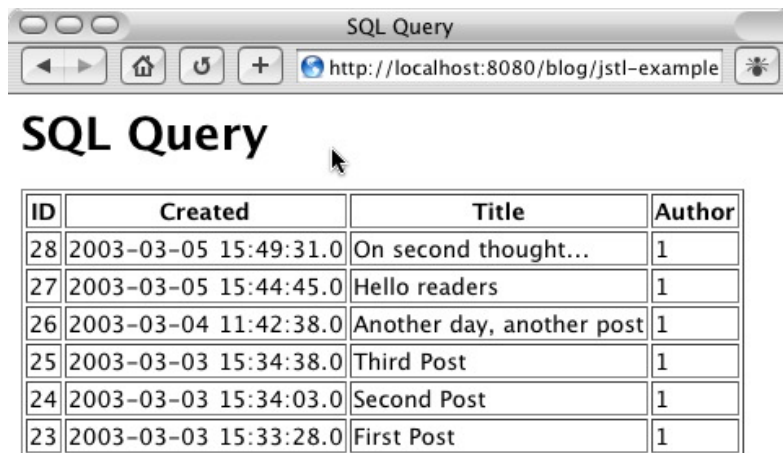
リスト7には、<sql:setDataSource> を使用してデータ・ソースを有効範囲付き変数に関連付けておき、後で<sql:query> アクションでdataSource 属性を使用してアクセスする方法が示されています。

**リスト7. データベースを照会するために <sql:query> を使用し、結果セットでそれを繰り返すために <c:forEach> を使用する方法**

```
<sql:setDataSource var="dataSrc"
url="jdbc:mysql:///taglib" driver="org.gjt.mm.mysql.Driver"
user="admin" password="secret"/>
<sql:query var="queryResults" dataSource="${dataSrc}">
select * from blog group by created desc limit ?
</sql:query>
<table border="1">
<tr>
<th>ID</th>
<th>Created</th>
<th>Title</th>
<th>Author</th>
</tr>
<c:forEach var="row" items="${queryResults.rows}">
<tr>
<td><c:out value="${row.id}"/></td>
<td><c:out value="${row.created}"/></td>
<td><c:out value="${row.title}"/></td>
<td><c:out value="${row.author}"/></td>
</tr>
</c:forEach>
</table>
```

図2は、リスト7のJSTLコードに対応するサンプル・ページ出力を示しています。リスト7に示す<sql:query> アクションの本体で使用されているSQLステートメントがパラメーター化されていることにも、注意してください。

## 図2. リスト7の出力



ID	Created	Title	Author
28	2003-03-05 15:49:31.0	On second thought...	1
27	2003-03-05 15:44:45.0	Hello readers	1
26	2003-03-04 11:42:38.0	Another day, another post	1
25	2003-03-03 15:34:38.0	Third Post	1
24	2003-03-03 15:34:03.0	Second Post	1
23	2003-03-03 15:33:28.0	First Post	1

`<sql:query>` アクション内では、本体コンテンツとして、あるいは`sql` 属性を介して指定されたSQLステートメントは、`?` という文字を使用してパラメーター化することができます。SQLステートメントのそれぞれのパラメーターごとに、対応する`<sql:param>` または`<sql:dateParam>` アクションが、`<sql:query>` タグの本体内にネストされている必要があります。`<sql:param>` タグは、`value` という単一属性を使用して、パラメーター値を指定します。あるいは、パラメーターの値を文字ストリングにする必要がある場合には、`value` 属性を省略して、`<sql:param>` タグの本体コンテンツとしてパラメーター値を指定することもできます。

日付、時刻、またはタイム・スタンプを表すパラメーター値は、次の構文で`<sql:dateParam>` タグを使用して指定します。

```
<sql:dateParam value="expression" type="type"/>
```

`<sql:dateParam>` の`value` 属性の式の評価結果は、`java.util.Date` クラスのインスタンスでなければならない、`type` 属性の値は、SQLステートメントで要求されている時間関係の値のタイプに応じ、`date`、`time`、または`timestamp` のいずれかでなければなりません。

`<sql:query>` と同じように、`<sql:update>` アクションでは次の3つの形式がサポートされます。

```
<sql:update sql="expression" dataSource="expression"
var="name" scope="scope"/>
<sql:update sql="expression" dataSource="expression"
var="name" scope="scope">
  <sql:param value="expression"/>
  ...
</sql:update>
<sql:update dataSource="expression"
var="name" scope="scope">
  SQL statement
  <sql:param value="expression"/>
  ...
</sql:update>
```

`sql` 属性と`dataSource` 属性における`<sql:update>` の意味は、`<sql:query>` の場合の意味と同じです。同様に、`var` 属性と`scope` 属性も、有効範囲付き変数の指定に使用されますが、この場合に

は、有効範囲付き変数に割り当てられる値は、データベース更新を実行した結果として変更された行の数を示す `java.lang.Integer` のインスタンスになります。

## トランザクションの管理

トランザクションは、グループ全体として成功または失敗のいずれかの結果となる、一連のデータベース操作を保護するために使用されます。トランザクション・サポートはJSTLの`sql` ライブラリーに組み込まれているため、一連の照会および更新をトランザクションにラップする作業は、対応する`<sql:query>` アクションと`<sql:update>` アクションを`<sql:transaction>` タグの本体コンテンツ内でネストさせるだけで、簡単に行うことができます。

`<sql:transaction>` の構文は次のとおりです。

```
<sql:transaction dataSource="expression" isolation="isolationLevel">
<sql:query .../>or<sql:update .../>
...
```

`<sql:transaction>` アクションには、必須属性はありません。 `dataSource` 属性を省略した場合には、JSTLデフォルト・データ・ソースが使用されます。 `isolation` 属性は、トランザクションの分離レベルを指定するために使用されるもの

で、`read_committed`、`read_uncommitted`、`repeatable_read`、または`serializable` のいずれかを指定することができます。この属性を指定しなかった場合、トランザクションはデータ・ソースのデフォルト分離レベルを使用します。

もちろん、ネストされたすべての照会および更新は、トランザクション自体と同じデータ・ソースを使用しなければなりません。実際には、`<sql:transaction>` アクション内でネストされた`<sql:query>` または`<sql:update>` では、`dataSource` 属性を指定することができないようになっています。これらのアクションを囲んでいる`<sql:transaction>` タグに (明示的または暗黙的に) 関連付けられたデータ・ソースが、自動的に使用されます。

リスト8は、`<sql:transaction>` の使用方法を示す例です。

## リスト8. 複数のデータベース更新を結合してトランザクションにするための `<sql:transaction>` の使用方法

```
<sql:transaction>
<sql:update sql="update blog set title = ? where id = ?">
<sql:param value="New Title"/>
<sql:param value="{23}"/>
</sql:update>
<sql:update sql="update blog set last_modified = now() where id = ?">
<sql:param value="{23}"/>
</sql:update>
</sql:transaction>
```

## 注意すべきこと

JSTLの`xml` ライブラリーと`sql` ライブラリーを使用すると、カスタム・タグを使用して複雑な機能をJSPページに実装することができます。しかしながら、こうした機能をプレゼンテーション層に実装することは、必ずしも最良の方法とは限りません。

大規模なアプリケーションを複数の開発者が長期間かけて作成する場合、ユーザー・インターフェース、基礎になるビジネス・ロジック、およびデータ・リポジトリを厳密に区別することが、長期にわたるソフトウェア保守を容易にすることが明らかになっています。一般に人気のあるModel-View-Controller (MVC) 設計パターンは、こうした「最良実例」を公式化したものです。J2EE Webアプリケーションの分野では、アプリケーションのビジネス・ロジックがモデルに相当し、プレゼンテーション層を構成するJSPページがビューに当たります。(コントローラーに相当するのは、ブラウザー・アクションでモデルの変更を開始し、さらにビューを更新できるようにするための、フォーム・ハンドラーその他のサーバー・サイド・メカニズムです。)MVC設計パターンでは、アプリケーションの3つの主要な要素(モデル、ビュー、およびコントローラー)が、相互に最低限の依存関係しか持たず、それぞれの間での相互作用が、一貫性のある、十分に定義されたインターフェースのみに限定されています。

アプリケーションがXML文書に依存してデータ交換を行い、リレーショナル・データベースに依存してデータの永続性を実現することは、アプリケーションのビジネス・ロジック(つまりそのモデル)の特徴となっています。したがって、MVC設計パターンに準拠するためには、このような実装の詳細がアプリケーションのプレゼンテーション層(つまりそのビュー)に反映しないようにする必要があります。JSPを使用してプレゼンテーション層を実装する場合、`xml` ライブラリーや`sql` ライブラリーを使用すると、プレゼンテーション層における、基礎となるビジネス・ロジックの要素が開示されるため、MVCに違反することになります。

このような理由により、`xml` ライブラリーおよび`sql` ライブラリーは、小規模なプロジェクトやプロトタイプ作業に最も適しています。アプリケーション・サーバーによるJSPページの動的なコンパイルを行う際にも、これらのライブラリーの中のカスタム・タグがデバッグ・ツールとして役に立ちます。

## まとめ

このシリーズでは、4つのJSTLカスタム・タグ・ライブラリーの機能と、それらの使用法を検討しました。[第1回](#)および[第2回](#)では、ELおよび`core` ライブラリーのタグを使用することにより、一般的な多くの状況でJSPスクリプト要素の使用を回避できることを示しました。[第3回](#)では、`fmt` ライブラリーを使用してWebコンテンツをローカライズすることに焦点を当てました。

最終回に当たるこの記事では、`xml` ライブラリーおよび`sql` ライブラリーの機能を検討しました。プレゼンテーション層にビジネス・ロジックを組み込むことによって生じる結果を受け入れても構わないという場合には、これら2つのライブラリーに含まれるタグを使用すると、XML文書およびリレーショナル・データベースの内容を簡単にJSPページに組み込むことができます。また、これら2つのライブラリーは、JSTLのライブラリーが相互に依存し合って構築されていて、`<sql:query>` と`<c:forEach>` を統合する際に相互操作されることの実例となります。また、`<c:import>` アクションを活用するための、`xml` ライブラリーの機能も示しました。

---



## 著者について

Mark Kolb

Mark Kolbは、テキサス州オースチンで働くソフトウェア・エンジニアです。サーバー・サイドのJavaトピックについての講演多数のほか、[Web Development with JavaServer Pages, 2nd Edition](#) の共著者でもあります。

© Copyright IBM Corporation 2003

([www.ibm.com/legal/copytrade.shtml](http://www.ibm.com/legal/copytrade.shtml))

商標

([www.ibm.com/developerworks/jp/ibm/trademarks/](http://www.ibm.com/developerworks/jp/ibm/trademarks/))