Java Web サービス: Metro の紹介

JAXB および JAX-WS リファレンス実装をベースとした Web サービス・フレームワークについて学ぶ

Dennis Sosnoski
Architecture Consultant and Trainer
Sosnoski Software Solutions, Inc.

2009年 11月 03日

Metro Web サービス・スタックは、Web サービスを実装したり、Web サービスにアクセスしたりするための包括的なソリューションを提供します。そのベースとなっているのは JAXB 2.x および JAX-WS 2.x Java™ 標準のリファレンス実装であり、この実装に WS-* SOAP 拡張技術と実際の Web サービス・デプロイメントをサポートするためのコンポーネントが追加されています。Dennis Sosnoski の連載「Java Web サービス」では今回、この Metro のクライアントおよびサーバー開発の基本原則を取り上げます。

このシリーズの他の記事を見る

Metro Web サービス・スタックは、Sun Microsystems によって開発されたオープンソースのツールです。Metro は、JAXB 2.x データ・バインディングおよび JAX-WS 2.x Web サービス標準のリファレンス実装をベースに、XML 関連のその他の Java 標準を採り入れています。また、基本 JAX-WS サービスの定義と使用のサポート、そして SOAP メッセージ交換に対するさまざまな WS-* 拡張のサポートを目的とした非標準コンポーネントも組み込まれています。

この連載について

Web サービスは、エンタープライズ・コンピューティングにおいて Java 技術が担う重大な役割の一部です。この連載では、XML および Web サービスのコンサルタントである Dennis Sosnoski が、Web サービスを使用する Java 開発者にとって重要になる主要なフレームワークと技術について説明します。この連載から、現場での最新の開発情報を入手して、それらを皆さんのプログラミング・プロジェクトにどのように利用できるかを知っておいてください。

Metro はスタンドアロンの Web サービス・スタックとして使用することも、オープンソースの Glassfish アプリケーション・サーバーの組み込みコンポーネントとして使用することもできます。オープンソースの NetBeans IDE で開発を行っている場合はなおさらのことですが、Glassfish を使用すると Web サービスの構成がいくぶん簡単になります。Glassfish には、基本 Web サービスと WS-* 拡張機能を構成するための GUI ツールが組み込まれているからです。しかしこの連

載の焦点は Web サービスにあるので、この記事では IDE とは関係なく、スタンドアロンとして Metro を使用する場合のみを検討します。これは以前の Apache Axis2 に関する記事で、Axis2 を組み込んで GUI ツールをサポートするアプリケーション・サーバーについてではなく、Axis2 をスタンドアロンで使用する場合について検討したのと同じことです。

Metro の基本と Axis2 との比較

連載の以前の記事で Axis2 について詳しく説明してあるので、まずはその Axis2 と Metro との類似点と相違点について説明するところから始めることにしましょう。この 2 つの類似点は限られており、そのほとんどは Web サービスを使用したコードを開発する上での共通の要件に関するものです。どちらのフレームワークでも、既存の Java コードから Web サービスを作成するという方法 (ただし Axis2 の場合、Jibx2Wsdl などのツールを別途使わない限り、この方法のサポートは制限されます)、または WSDL Web サービスを記述するところから始めて、サービスを使用または実装する Java コードを生成するという方法を使用することができます。また、この 2 つのフレームワークは両方とも、サービス操作 (メソッド呼び出しなど) とサービス・ポート・タイプをインターフェースとしてモデル化します。

Metro と Axis2 の相違点は、類似点と比べると遥かに顕著です。根本的なところでは、Metro の設計は JAXB 2.x と JAX-WS 2.x を中心としており、これらの技術に代わる技術に対するサポートは一切考慮されていません (ただし、レガシー JAX-RPC の使用は例外です)。一方 Axis2 は、特に XML データ・バインディングの分野をはじめ、あらゆる技術をサポートするように設計されています。Axis2 にも JAXB 2.x および JAX-WS 2.x のサポートは組み込まれていますが、この 2 つに特別な地位を与えているということはありません (どちらかと言うと、JAX-WS は Axis2 では優先されない代替案となっています。その理由は、「JAXB and JAX-WS in Axis2」でも説明したように、JAX-WS サービスを対象とした WS-Security やその他の機能は構成できないためです)。

構造面では、どちらのスタックもリクエストとレスポンスの処理の一部としてハンドラーを使用します。Axis2 はこのハンドラー方式をベースにモジュールを実装します (モジュールとは基本 SOAP メッセージ交換に対するプラガブルな拡張機能のことで、WS-* 技術を極めて柔軟に構成できるような形で実装するために使用されます)。Metro は広範な WS-* 技術をハンドラーによってサポートしますが、これらの技術は個別のコンポーネントにされるのではなく、Metro エンジンに統合されます。Metro が使用する統合手法は Axis2 モジュールほど柔軟ではないものの、WS-* 拡張機能の構成および使用に関しては、いくつかの利点をもたらします。

この 2 つのスタックは、クライアント・コードでの WSDL サービス定義の使い方という点でも異なります。Axis2 は WSDL サービス定義を、主としてクライアント・サイドのコードを生成するために使用します。つまり、WSDL からサービス構成情報を抽出した上で、実行時にそれに対応するAxis2 クライアント構成を組み立てるコードを生成するという使い方です (ただし、WSDL サービス定義を実行時に構文解析することもできます)。JAX-WS 2.x、したがって Metro では、実行時にサービス構成を作成するために WSDL サービス定義が必要となります。このように実行時に WSDL を使用するということは、(最初のサービス呼び出しに限られるとは言うものの) 起動時のオーバーヘッドを追加するだけで、明らかなメリットは何もありません。

サーバー・サイドにも違いがあります。一般的な HTTP トランスポートの場合、Axis2 は個別の Web アプリケーション (WAR ファイル) としてセットアップされるのが通常です。この Axis2 Web アプリケーションには、任意の数の個別サービスをデプロイすることができます (ただし、アプ

リケーション WAR の一部としてパッケージすることも可能です)。サービスをデプロイするには、Web ページのアップロードを使用するか、または Axis2 サービスの AAR ファイルを、Axis2 Web アプリケーションを展開して該当するディレクトリーに直接ドロップするという方法をとれます。個々のサービス構成情報は、通常、Axis2 がビルド時に WSDL サービス定義から生成してサービス AAR ファイルに組み込みます。標準的な Axis2 Web アプリケーションは、Web ページ・インターフェースによってさまざまなモニター・ツールや制御ツールも提供します。

それとは対照的に Metro では、開発者がそれぞれの Web サービス・アプリケーションごとに個別の WAR ファイルを作成しなければなりません。その際に使用するのは Metro ライブラリーの JAR ファイルと、WAR の中に含まれる WEB-INF/web.xml ファイルです。前者の JAR ファイルは WAR に組み込まれるものか、もしくは (HTTP サーバー・システムの一部として) クラス・パスに組み込まれているもののいずれかであり、後者の WEB-INF/web.xml はサービスと Metro サーブレットの両方を参照する WAR に含まれているものです。Metro をスタンドアロンとして使用する場合には、サービス構成に関する追加情報を提供する sun-jaxws.xml 構成ファイルを作成する必要もあります。これらの構成ファイルからの情報が、実際の Web サービス・クラスで JAX-WS アノテーションと組み合わせられて、作成したサービスに対応した Metro が完全に構成されるというわけです。Metro はこのように組み込み式で使用するように設計されているため、直接的なモニター・ツールや制御ツールは提供していません。

Axis2 と Metro は、どちらも統合 HTTP サーバー・サポートを提供します。Metro の場合、このサポートの提供手段となっているのは JAX-WS 機能の 1 つである `javax.xml.ws.Endpoint` クラスです。Axis2 の統合 HTTP サーバーにしても、Metro/JAX-WS の統合 HTTP サーバーにしても、テストで使用したり、非同期レスポンス・ポートとして使用したりするには適していますが、本番 Web サービスをホストする場合には Servlet API をサポートする Java アプリケーション・サーバーを使用するほうが適切です。

サンプル・アプリケーション

コードのダウンロードには、連載の以前の記事で使用した単純なライブラリー管理サービスを、Metro の使用方法を説明するために変更したバージョンが用意されています。以前の記事と同じく、WSDL サービス定義では以下の 4 つの操作を定義しています。

- `getBook`。ISBN (International Standard Book Number) で識別される特定の本についての詳細を取得するために使用します。
- `getBooksByType`。特定のタイプのすべての本についての詳細を取得するために使用します。
- `getTypes`。入手可能な本のタイプを検索するために使用します。
- `addBook`。新しい本をライブラリーに追加するために使用します。

「JAXB and JAX-WS in Axis2」ではこのアプリケーションが Axis2 でどのように機能するかを説明するために、最初に JAXB 2.x データ・バインディングを使用して通常の Axis2 コード生成を行い、次に JAX-WS 2.x サービスの構成を行いました。そのときに説明した内容の大部分は、Metro を使用する場合にも当てはまります。WSDL はサービス名とエンドポイント・アドレス以外は Metro でもまったく同じで、生成される JAXB データ・モデルも同じです。さらには生成されるサービス・クラスも、Java パッケージと、JAX-WS アノテーションで使用するサービス名を除けば、異なっているところはありません。

クライアント・サイドの使用方法

Metro サンプル・アプリケーションのクライアント・サイドのコードは、Axis2 で JAX-WS を使用するためのコードと同じで、ビルド・ステップも同じです。このコードとその処理方法についての詳細は、「JAXB and JAX-WS in Axis2」を参照してください。

サーバー・サイドの使用方法

Metro サンプル・アプリケーションのサーバー・サイドのコードについても、Axis2 で JAX-WS を使用するためのコードと同じですが、ビルド・ステップは多少異なります。Axis2 では、サービスをデプロイするための準備として、サービスおよびデータ・モデル・クラスが含まれる JAR ファイルを作成し、その JAR を Axis2 サーバー・システムの WEB-INF/servicejars ディレクトリーにドロップすることによってサービスをデプロイしました。

Metro の場合は、サービスおよびデータ・モデル・クラスが含まれる WAR ファイル、Metro ライブラリー JAR、そして構成ファイルのペアを作成する必要があります (Metro ライブラリー JAR に関しては、Web サーバーに直接インストールすることもできます。Tomcat のユーザー用に、Metro ダウンロードには Metro JAR をインストールするための metro-on-tomcat.xml Ant ビルド・ファイルが含まれています。インストール方法は、付属のドキュメントを参照してください)。実際のサーブレット処理を構成する場所は、WEB-INF/web.xml ファイルです。リスト 1 に、このファイルのサンプル・アプリケーション用バージョンを記載します。

リスト 1. サンプル・アプリケーションの web.xml

```
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee">
  <display-name>MetroLibrary</display-name>
  <description>Metro Library Service</description>
  <listener>
    <listener-class>com.sun.xml.ws.transport.http.servlet.WSServletContextListener
    </listener-class>
  </listener>
  <servlet>
    <servlet-name>MetroLibraryPort</servlet-name>
    <display-name>MetroLibraryService</display-name>
    <description>Endpoint for Metro Library Service</description>
    <servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>MetroLibraryPort</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>60</session-timeout>
  </session-config>
</web-app>
```

Java Web アプリケーションに取り組んだ経験があれば、リスト 1 の WEB-INF/web.xml ファイルは (少なくとも構造の点で) お馴染みのはずです。上記では、WAR ファイルをデプロイするサーブレット・エンジンに対し、`com.sun.xml.ws.transport.http.servlet.WSServletContextListener` クラスをサーブレット・コンテキストのイベント・リスナーとして使用するよう指示するエントリー、そして`com.sun.xml.ws.transport.http.servlet.WSServlet` クラスを実際のサーブレットとして使用するように指示するエントリーが使用されています。この 2 つのクラスは、Sun の

Metro スタックに特有のもので、Metro と連動するには、これらのクラスへの参照が必要となります。サーブレットは、この Web アプリケーションに送られてくるすべてのリクエストを受け取るように構成されています (`<url-pattern>/</url-pattern>` エントリー)。

リスト 1 の WEB-INF/web.xml ファイル自体は、Metroが提供するリスナーとサーブレットを使用するようにサーブレット・エンジンを構成しているだけにすぎません。サーブレットが受信したリクエストをサービス実装コードにルーティングするように Metro を構成するためのファイルは別にあります。それが、リスト 2 に記載する WEB-INF/sun-jaxws.xml です。

リスト 2. サンプル・アプリケーションの sun-jaxws.xml

```
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime" version="2.0">

    <endpoint name="MetroLibraryPort"
        implementation="com.sosnoski.ws.library.metro.MetroLibraryImpl"
        url-pattern="/"
        wsdl-location="WEB-INF/wsdl/library.wsdl"/>

</endpoints>
```

リスト 2 の WEB-INF/sun-jaxws.xml ファイルはこの上なく単純なもので、たった 1 つのエンドポイント定義によって、ポート名、実装クラス、リクエストに突き合わせるパターン、そして WSDL 文書の場所を指定しています。WSDL 文書の場所は、このエンドポイントの定義で唯一のオプションです。sun-jaxws.xml ファイルでサービス・エンドポイントの WSDL 文書を指定しなければ、Metro が実行時に、この文書を自動的に生成します。

バンドルの問題

Java SE 6 の時点で、JAXB 2.x および JAX-WS 2.x リファレンス実装のランタイム (ベンダー拡張機能を除く) は、標準 JRE (Java Runtime Environment) ライブラリーの一部となっています。その目的は、Java 標準としてのこの2 つの技術の使用を広めることでしたが、残念なことに、これには副次作用が伴います。それは、これらの技術の新しいバージョンを使用するには、お使いの JRE を変更しなければならないことです。

サンプル・アプリケーションのダウンロードで使用している build.xml は、Metro JAR ファイルをサービス WAR ファイルに直接コピーします。この動作は、著者のシステムでアプリケーションを Apache Tomcat 6.0.20 Web サーバーにデプロイしたときには、Java SE 5 および Java SE 6 JDK/JRE の両方で上手くいきました。しかし、Java SE 6 以降を使用する場合、クラス・ロードが競合すると (例えば、`ClassCastException`s がスローされる、`com.sun.xml...` クラスが見つからないなど) 問題が発生します。その場合の問題を修正する方法は以下のとおりです。

- お使いのシステムで有効な最新バージョンの JRE を確実に使用するようにします。これは、更新に含まれている JAXB 2.x と JAX-WS 2.x のバージョンのほうが新しい可能性があるためです。
- `java.endorsed.dirs` システム・プロパティーを使用して、Metro の lib ディレクトリーからコピーされた webservices-api.jar ファイルが含まれているディレクトリー (ここに含めるのは、このファイルのみです。webservices-api.jar 以外の JAR はクラス・ロードの競合の原因となります) を更新後のライブラリーのソースとして指定します (Tomcat 6.0.x は、`JAVA_ENDORSED_DIRS` 環境変数を探し、この環境変数をシステム・プロパティーの値として使用することによって、このメカニズムをサポートします)。
- 上記の方法が上手くいかない場合は、インストールしてある JRE の lib ディレクトリーのなかに endorsed ディレクトリーを作成し (まだ存在しない場合)、そのディレクトリーに Metro の webservices-api.jar ファイルをコピーします。

最後の 2 つの方法では、いずれもサービス WAR ファイルに Metro の webservices-api.jar を組み込む必要はありません。この JAR ファイルは、Web サーバーのクラス・パスで直接使用可能になるからです。

サンプル・コードのビルドおよび実行

サンプル・コードを試すには、最新バージョンの Metro (この記事では、1.5 リリースでサンプル・コードをテストしました) をダウンロードしてシステムにインストールする必要があります (「参考文献」を参照)。さらに、解凍したサンプル・コード・ダウンロードの root ディレクトリー内にある build.properties ファイルを編集し、`metro-home` プロパティーの値を Metro システムへのパスに変更してください。サーバーを異なるシステムまたはポートでテストする予定であれば、`host-name` と `host-port` も変更する必要があります。

提供されている Ant build.xml を使用してサンプル・アプリケーションをビルドするには、コンソールでダウンロード・コードの root ディレクトリーを開き、`ant` と入力します。これにより、最初に JAX-WS `wsimport` ツール (Metro ディストリビューションに付属) が起動された後、クライアントとサーバーがコンパイルされ、最後にサーバー・コードが WAR としてパッケージ化されます。生成されたこの metro-library.war ファイルをテスト・サーバーにデプロイし、コンソールで `ant run` と入力すれば、サンプル・クライアントを実行してみることができます。サンプル・クライアントはサーバーに対して一連のリクエストを行い、それぞれのリクエストごとに簡潔な結果を出力します。

Metro の今後

この記事では、Metro Web サービス・スタックを使用する際の基本を説明しました。Metro は構成に JAX-WS 2.x アノテーションを使用するため、「JAXB and JAX-WS in Axis2」で使用したサンプル・アプリケーションのコードも Metro で動作します。このサンプル・アプリケーションを Metro で使用する際に必要な変更は、コードをパッケージ化してサーバー・サイドにデプロイする方法に関する部分だけであり、そこには Metro と Axis2 との間で大きな違いがあります。Metro が使用する組み込み方式では、開発者がサービスまたはサービスのグループごとに Web アプリケーションを作成します (制御やモニター機能は提供されません)。Axis2 では一般に、個々のあらゆるサービスに共通のホストとして、1 つの専用 Web アプリケーションを使用します (基本的な制御およびモニター機能は、Web ページ・インターフェースから直接提供されます)。

Web サービス・メッセージ交換の基本機能だけでなく、Metro は WS-Security などの SOAP 拡張機能もサポートします。サービスのパッケージ化に関する問題と同様、この分野でも Metro と Axis2 は非常によく似た要件に対して異なる手法をとります。次回の記事では、この連載で前に Axis2 で使用した WS-Security のサンプルを、Metro ではどのように処理するかを説明します。

ダウンロード

内容	ファイル名	サイズ
Source code for this article	j-jws9.zip	13KB

著者について

Dennis Sosnoski

Dennis Sosnoski は Java ベースの XML および Web サービスを専門とするコンサルタント兼トレーナーです。専門家としてのソフトウェア開発経験は 30 年以上に渡り、この 10 年間はサーバー・サイドの XML 技術や Java 技術に注力しています。オープンソースのJiBX XML Data Binding フレームワークや、それに関連した JiBX/WS Web サービス・フレームワークの開発リーダーを務め、さらに Apache Axis2 Web サービス・フレームワークのコミッターでもあります。彼は JAX-WS 2.0 および JAXB 2.0 仕様のエキスパート・グループの一員でもありました。