JavaプログラミングでのXML-RPC

アプリケーション間通信のための最も簡単な手段

Roy Miller
Independent consultant

2004年 1月 13日

アプリケーション間通信は、プログラマにとって扱いにくい問題であるかもしれません。JNIのような利用可能なオプションの多くは、使用が困難になる可能性があります。XML-RPCははるかに簡単な解決策を提供します。それはクリーンで、実装が簡単で、最もポピュラーな(Java言語およびC++のような)プログラミング言語のオープンソースライブラリーによって十分にサポートされています。例えば、あなたがJavaアプリケーションを持っており、それはC++で書かれたアプリケーションと通信する必要がある場合、XML-RPCはちょうどよい、最も単純なアプローチかもしれません。この記事では、ソフトウェア開発者であり指南役であるRoy Millerが、XML-RPCとは何か、そして効果的に使用する方法について話します。

私は開発者仲間から、最新の流行技術がソフトウェア開発業界を悩ますものに対する解決策であると、何度聞かされたかわかりません。XMLがデビューした時も、多くの人々がそう言いました。私はその時それほど興奮せず、またその時以来、態度はあまり変わっていません。私は常に、XMLはリレーショナル構造へフラット化する必要なしに、構造化データを定義するための優れた方法であると思っていました(それは不適当かもしれませんが)。しかし、XMLはプログラミング言語ではありません——XLSTは構文上面倒であり、（少なくとも私にとっては）異質なたぐいでした。したがって私は、構造化データの交換(まさにXMLはそのために作成されました)が要求される課題がやってくる時を待ちました。その明確な課題は最近のプロジェクト上に浮び上り、そしてXMLは、XML-RPCによって使用される時に、そのジョブのための適切な正しいツールとなりました。

プログラミングの課題

私たちのクライアントはハードウェアデバイスを作りました。プロジェクト上で私たちが関与する以前に、ユーザが各デバイスを設定することができる唯一の方法は、コマンドライン・インターフェースによるものでした。それは、各顧客がこれらのハードウェアデバイスを、各ネットワーク上に20以上(何百または何千でさえあるかもしれません)持っている場合を除いては、必ずしも悪くはありません。顧客にコマンドライン・インターフェースで一つずつ各デバイスを設定することを強いれば、確実に売上げは落ち込んだでしょう。納品後、顧客が多数のデバイスの初期セットアップおよび設定をしなければならなかった場合は、その問題は特に深刻になったことでしょう。各デバイスの設定は、デバイスがスタートアップで読込むXMLファイルに含まれることになりました。

私たちのクライアントは、1台以上の中心に位置する管理マシン上で動作可能な設定アプリケーションを作成するために、私たちを雇いました。そのアプリケーションは、全デバイスの初期セットアップを単純化し、必要なときに (ファームウェアの更新、エラーの修正など) それらを再設定し、かつ、既存のデバイスをモニターする必要がありました。それをいささか面倒な課題にしたのは、デバイス上のソフトウェアがC言語で書かれており、Javaプログラミング言語で私たちのデスクトップ・アプリケーションを書く必要があるということでした。

私たちは一時的にJNIを検討しました、しかし、推測することには、より単純なものでなければならないと判断しました——そしてそこには、XML-RPCと呼ばれる素晴らしい小さなものがありました。

XML-RPCの登場

XML-RPCウェブサイト(参考文献を参照)では、以下のように記述されています。

> 異種のオペレーティングシステム上、および異なる環境で動作するソフトウェアが、インターネット上のプロシージャコールの実行を可能にするのは、１つの仕様および1組のインプリメンテーションです。それは、トランスポートとしてHTTPを、そしてエンコーディングとしてXMLを使用してプロシジャーをリモートで呼び出します。複雑なデータ構造が送信され、処理され、そして返されることを可能にすると同時に、XML-RPCは可能な限り単純であるように設計されています。

私たちはそれを読んだとき、私達自身の答えを得ました。各デバイスの設定は、ファイルの中にあります(内容は同様にXMLですが、それはこの議論では重要ではありません)。それは私たちが既に、各デバイスにデバイス自体の設定方法を教えるためにセマンティクスを持っていることを意味していました。もし私たちが、それを設定ファイルに送信すれば、それは満足するでしょう。しかし、私たちはどのようにそれを送信するでしょうか。私たちはバイトのみを送信することができましたが、それはセキュリティリスクをもたらしました。また、すべてをそのバイト操作することは、実際に誰も望まないことでした。私たちは、十分に定義されたXML-RPCメッセージでの文字列ペイロードの送信が可能なことに気がつきました(それは私たちが各デバイス上のソフトウェアの、非常に制限されているパブリック・インターフェースで、Cファンクションを起動すること可能にします)。

XML-RPCのハイライト

一言で言えば、XML-RPCは、単純化されたSOAPと見なすことができます。それは、あなたが必要とする唯一のアプリケーション間通信かもしれません。XML-RPCウェブサイト上に、いくらかの歴史および様々な言語でのサンプルを提供する、優れた「how-to」ドキュメントがあります。その反面、あなたはただ仕様だけ読みたいのかもしれません。6ページ以前は、簡単なモデルです。私たちがプロジェクト上でどのようにXML-RPCを使用したかの基本を提供するために、私たちはこのセクションのいくつかのハイライトを取り上げます。

XML-RPCメッセージはXML本文を持つHTTP-POSTリクエストです。メッセージを作成するXML-RPCクライアントおよびそれを受け取るXML-RPCサーバーが必要となります。一旦サーバーがリクエストを完了すると、さらにXMLで、XML-RPCレスポンス・メッセージを返信します。そのリクエストはパラメーター(integer、string、date、および必要ならばarrayおよび複雑なレコードを含む

その他の型)を含むことができます。各リクエストの形式は、リスト1に示すように非常に簡単です。

リスト１．XML-RPCリクエストのサンプル

```
POST /RPC2 HTTP/1.0
User-Agent: Frontier/5.1.2 (WinNT)
Host: betty.userland.com
Content-Type: text/xml
Content-length: 181
<?xml version="1.0"?>
<methodCall>
   <methodName>examples.getStateName</methodName>
   <params>
      <param>
         <value><i4>41</i4></value>
      </param>
   </params>
</methodCall>
```

「ハンドラ」名を指定するmethodName文字列(リスト1内のexamples)、およびそのハンドラを要求するメソッド(リスト1内のgetStateName) が必要となります。必要であれば、サーバーはこの名前文字列を解釈することができます。私たちが使用したJavaサーバーは、（ それについては少し後で記します ）examplesのハンドラ名を持つオブジェクトを見つけ、その上でgetStateNameメソッドを呼びます。

リスト2に示されるように、レスポンスは実に簡単です。

リスト2．XML-RPCレスポンスのサンプル

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 158
Content-Type: text/xml
Date: Fri, 17 Jul 1998 19:55:08 GMT
Server: UserLand Frontier/5.1.2-WinNT
<?xml version="1.0"?>
<methodResponse>
   <params>
      <param>
         <value><string>South Dakota</string></value>
      </param>
   </params>
</methodResponse>
```

XML-RPCを呼出す時、あなたはXMLレスポンスを得るでしょう(それは1つの<params>要素を含み、<params>要素は次に1つの<params>要素を含み、<params>要素は1つの< value >要素を含み、<value>要素は処理する必要のある返り値を含んでいます)。たいていの場合、これが受け取りたいレスポンスです。しかし、人生はそれほど簡単ではありません。何かが間違っている場合、サーバーは、リスト3に見られるような「fault」レスポンスを返すはずです。このfaultはRPCに送信されたパラメータが多すぎることを示しています。

リスト3．XML-RPC faultレスポンスのサンプル

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 426
```

```
Content-Type: text/xml
Date: Fri, 17 Jul 1998 19:55:02 GMT
Server: UserLand Frontier/5.1.2-WinNT

<?xml version="1.0"?>
<methodResponse>
   <fault>
      <value>
         <struct>
            <member>
               <name>faultCode</name>
               <value><int>4</int></value>
               </member>
            <member>
               <name>faultString</name>
               <value><string>Too many parameters.</string>
               </value>
               </member>
         </struct>
      </value>
   </fault>
</methodResponse>
```

<fault> 要素の<value> 要素は、faultCodeメンバおよび<faultString> メンバを持つ１つの<struct> 要素を含んでいます。これはJavaクラスのtoString()に似ています。何かが間違っている場合、toString()は (あなたがそれを行う為にコード化したと仮定すれば) 、エラーコードおよびエラーメッセージを完備しており、それが何かをあなたに教えます。XML-RPC faultレスポンスはそれと同じことを行います。

そして、それは何がXML-RPCで起こっているか理解するのに必要な全てです。事実、実際にメッセージ用のXMLレイアウトの詳細を知る必要はありません。有効な入力値を与えれば、あなたが選んだXML-RPCインプリメンテーション・ライブラリは、その仕事をすべて行うでしょう。したがって、仕様を読んだ後に足りないただ一つのツールは、クライアントとサーバーのインプリメンテーションです。このアプリケーションでは、私たちは、クライアントのJavaインプリメンテーションおよびサーバーのCインプリメンテーションが必要となりました。

動作させましょう

XML-RPCウェブサイトには、Javaプログラミング言語、Ruby、Python、C/C++、およびPerlを含む多数の言語の仕様のクライアントおよびサーバーインプリメンテーションのためのリンクがあります。

そこにはApacheチームによって書かれた１つのインプリメンテーションがあり、また、個々の開発者によって書かれたものがいくつかあります。これらのうちのいくつかのコードを検討した後に、私たちはGreger Ohlsonによって書かれた、Marquee XML-RPCクライアント・インプリメンテーションを選びました。Ohlsonは同様にサーバーについても書きましたが、私たちが設定するハードウェアデバイスのためのコードを書く同僚は、Apache XML-RPCサーバーを選びました。しかし、入出力が予測可能な限り、それは実際に重要ではありません。

私たちは、すべてEclipseで開発を行いました。したがって、単にMarqueeライブラリをダウンロードし、そのためのプロジェクトを作成し、ワークスペースにロードしました。私たちのアプリケーションのクラスパスにそれを配置することで、Marquee インターフェースへアクセスできるようになりました。私たちがその時点でしなければならなかったのは、それを使用することで

した。私たちのアプローチを単純化するために、私たちは、デバイスのためのラッパーを作成しました(それは、XML-RPCの詳細を意識せずにドメインオブジェクトでアプリケーションの処理の残りを行わせ、そしてリクエストをパッケージにし、レスポンスをデコードするためにXML-RPCオブジェクトを作成しました)。

アプリケーションがある理由のために(例えばステータスをチェックするため)デバイスと通信する必要があった時は、単純にそのデバイス用のラッパー上でメソッドを呼びました (それはXML-RPCマジックを行うためにXML-RPCオブジェクトと交信をしました) 。リスト４は、ラッパーの外観の単純化したサンプルを示します。

リスト4. Deviceラッパークラス

```
public class Device {
    protected DeviceConfiguration configuration;
    protected Status status = Status.UNREACHABLE;
    protected Device(DeviceConfiguration configuration) {
        this.configuration = configuration;
    }
    public Status getStatus() {
        return obtainRpcClient().getStatus();
    }
    public void setStatus(Status status) {
        if (this.status != status) {
            this.status = status;
        }
    }
    public void reboot() {
        Status status;
        try {
            status = obtainRpcClient().reboot();
        } finally {
            makeUnreachable();
        }
        setStatus(status);
    }
    public DeviceConfiguration getConfiguration() {
        this.configuration =
            DeviceConfigurationBuilder.toConfig(
                obtainRpcClient().getDeviceConfiguration());
        makeOk();
        return this.configuration;
    }
    public Status putConfiguration() {
        Status status =
            obtainRpcClient().replaceDeviceConfiguration(
                DeviceConfigurationBuilder.toData(this.configuration));
        setStatus(status);
        return status;
    }
    protected RpcClient obtainRpcClient() {
        return new RpcClient(
            this.configuration.getIpAddress(),
            80,
            this.configuration.getUserPassword(),
            100);
    }
    public void makeOk() {
        setStatus(Status.OK);
    }
    public void makeUnreachable() {
        setStatus(Status.UNREACHABLE);
    }
}
```

Deviceクラスは3つのヘルパークラスを使用します。DeviceConfiguration、Status、およびDeviceConfigurationBuilderです。

これらの3つのクラスの詳細については、この記事の範囲外ですが、DeviceConfigurationクラスのインスタンスが、各ハードウェアデバイスのXML設定から抽出された値を保持することを私は述べたいと思います。それは単に、それらの値を追跡する一つの便利な方法です。ご覧の通り、Deviceクラスは、生の設定データからDeviceConfigurationインスタンスに変換するためにDeviceConfigurationBuilderを使用し、またその逆も行います。

このサンプルは、デバイスにステータスを求め、デバイス自体のリブートを命じ、設定データを取得、配置するメソッドを含んでいます。しかし、Deviceインスタンスは、XML-RPCのエキサイトメントが実際に起こったところで、代わりにXML-RPCラッパークラスに委託してモデル化したデバイスへの通信を処理しませんでした。リスト5は、私たちのXML-RPCラッパークラスの外観を単純化したサンプルを示します。私たちは、それをMarqueeXmlRpcClientと識別するためにRpcClientと呼びました。

リスト5. XML-RPCクライアントラッパークラス

```
import java.util.Hashtable;
import marquee.xmlrpc.XmlRpcClient;
import marquee.xmlrpc.XmlRpcException;
public class RpcClient {
    protected static final Object[] EMPTY_ARRAY = new Object[0];
    protected XmlRpcClient xmlRpcClient;
    protected String ipAddress;
    protected String password;
    protected int port;
    protected int timeout;
    public RpcClient(
        String ipAddress,
        int port,
        String password,
        int timeout) {
        super();
        this.ipAddress = ipAddress;
        this.port = port;
        this.password = password;
        this.timeout = timeout;
        xmlRpcClient = new XmlRpcClient(ipAddress, port, "/RPC2");
    }
    protected Object invoke(final String rpcMethodName) {
        return invoke(rpcMethodName, EMPTY_ARRAY);
    }
    protected Object invoke(final String rpcMethodName, Object[] parameters) {
        try {
            Object result = xmlRpcClient.invoke(rpcMethodName, parameters);
            if (result instanceof Hashtable) {
                Hashtable fault = (Hashtable) result;
                int faultCode = ((Integer) fault.get("faultCode")).intValue();
                throw new RuntimeException(
                    "Unable to connect to device via XML-RPC. \nFault Code: "
                        + faultCode
                        + "\nFault Message: "
                        + (String) fault.get("faultString"));
            }
            if (result instanceof Integer) {
                return Status.getStatus((Integer) result);
            }
```

```
            return result.toString();
        } catch (XmlRpcException e) {
            throw new RuntimeException(e);
        }
    }
    public Status getStatus() {
        return (Status) invoke("Device.getStatus");
    }
    public String getDeviceConfiguration() {
        return (String) invoke("Device.getConfiguration");
    }
    public Status replaceDeviceConfiguration(String configurationData) {
        return (Status) invoke(
            "Device.replaceConfiguration",
            new Object[] { configurationData });
    }
    public Status reboot() {
        return (Status) invoke("Device.reboot");
    }
}
```

RpcClientのパブリックインターフェースが、コンストラクタを含む5つのメソッドを持っていることに注目してください。各メソッドはinvoke()の一種を呼びます。ある種類はパラメーターをとらず(例えばreboot ()から呼ばれたもの)、また他方は、パラメーターのオブジェクト配列をとります(例えばreplaceDeviceConfiguration()から呼ばれたもの)。

パラメーターをとらない種類は、空のオブジェクト配列を持つ別の種類を呼びます。パラメーターをとるinvoke()メソッドは、私たちがMarquee ライブラリと通信をする唯一の場所です。メソッドは、RpcClientに含まれるXmlRpcClientインスタンス上でinvoke()を呼出します。Marquee ライブラリは、XMLでパラメーターの私たちのメソッド文字列 (仕様によるとそれはhandlerName.methodNameのようなものだと、思い出してください) およびオブジェクト配列をラッピングすることで、そのマジックを行います(その後、それはサーバーのもとへ送ります)。それが戻す結果は、1つのHashtable(XML-RPCレスポンスの「fault」用のMarquee の選択)、1つのIntegerラッパー(数値の返り値用)、あるいは1つのString(XML-RPCメッセージ用のデフォルト・リターンタイプ)です。

faultを戻された場合、私たちはfault のHashtableから抽出された詳細を持つRuntimeExceptionを投げます。数値のステータス値を取得する場合 (例えばreboot()を呼ぶ場合) 、それを保持するために、UI内で表示する優れたテキストへの変換と共に、ステータス・オブジェクトのインスタンスを作成します。Stringの返り値を取得した場合 (getDeviceConfiguration()を呼んだ場合) 、私たちは単にそれを返します。

aDeviceのリブート

すべての断片がそろったところで、点をつなげていきましょう。私たちのアプリケーションがそれ自体をリブートするために、特別のデバイス(私たちはそれをaDeviceと呼びます)に命令する事について話しましょう。UIの世界のどこかのあるクラスは、aDeviceの上のreboot ()を呼びます。以下は、次に起こることです。

1. aDeviceは、物理的なデバイス上のXML-RPCサーバーに接続するために、RpcClientインスタンス(anRpcClient)を作成します(aDeviceのDeviceConfigurationインスタンスからIPアドレスおよびユーザ・パスワードを使用します)。
2. aDeviceは、anRpcClientの上でreboot ()を呼出します。

3. anRpcClientは、空のパラメーター・リストを渡して、MarqueeXmlRpcClientインスタンス(xmlRpcClient)上のinvoke()を呼出します。
4. xmlRpcClientは、サーバーからfault のXML-RPCメッセージを取得した場合には１つのHashtableを、数値の返り値を得た場合には１つのIntegerを、あるいは他のすべての場合には、１つのStringを返します。
5. anRpcClientは、xmlRpcClientから得たものを返します。この場合、単純にすべてがうまくいったことを示すリターンコードとなるでしょう(すなわち、サーバーは本来私たちにどんなデータも返しません)。
6. 何かが間違っていた場合、UIがそれを報告するように、aDeviceはそのStatusインスタンスを、UNREACHABLEに設定します。
7. すべてがうまくいった場合、aDeviceはただそのStatusインスタンスを、anRpcClientがそれに与えたもの全てで更新します。

現在のステータスをaDeviceに求める

私たちがあるデータをaDeviceに求める場合は、それほど困難ではありませんでした。最も基本的な例は、私たちのアプリケーションが現在のステータスをいつaDeviceに求めるかでした。この場合、UIの世界のどこかのあるクラスは、aDeviceの上でgetStatus()を呼びます。以下は、次に起こることです。

1. aDeviceは、物理的なデバイス上のXML-RPCサーバーに接続するためにRpcClientインスタンス(anRpcClient)を作成します。
2. aDeviceは、anRpcClient上でgetStatus()を呼びます。
3. anRpcClientは、空のパラメーター・リストを渡して、そのMarqueeXmlRpcClientインスタンス(xmlRpcClient)上のinvoke()を呼出します。
4. xmlRpcClientは、サーバーからfault のXML-RPCメッセージを取得した場合には１つのHashtable、数値の返り値を得た場合には１つのInteger、あるいは他のすべての場合にはStringを返します。
5. anRpcClientは、それがxmlRpcClientから得たステータスを返します。
6. 何かが間違っていた場合、UIがそれを報告するように、aDeviceはそのStatusインスタンスをUNREACHABLEに設定します。
7. すべてがうまくいった場合、aDeviceはただそのStatusインスタンスを、anRpcClientがそれに与えたもの全てで更新します。

私たちが現在の設定データをaDeviceに求めた場合は、事実上同じでした。ただ一つの違いは、私たちは、XML-RPCコールによって返された生の設定データを取得し(Stringとして)、DeviceConfigurationインスタンスへそれを適応させなければならないことでした。aDeviceに新しい設定データを送信した時、私たちは、DeviceConfigurationインスタンスから設定データを抽出し、次に、その外部の文字列ペイロードを構築することにより、反対のことを行いました。

このサンプル・コードの中で、私たちがXML操作を全く行う必要がなかったことに注意してください。全くです。Marquee ライブラリは私たちのためにそれをすべて行いました。さて、XML-RPCの仕様は非常に簡単です、したがって、たぶんあなたは自分のクライアントに取り掛かることができました。しかし、そうする必要はほとんどありません--Marquee ライブラリは極めて優れ

ており、私がこの記事で探求しなかった機能があります。説明書は直感的で完全なものです。私の意見としては、XMLを解析する必要が全くないことが満足です。

サーバーサイド

この時点まで、私は、サーバーサイドの方式に関してあまり言及していません。それはチームの他の誰かがこのアプリケーションのXML-RPCサーバーサイドを作成したからです(ApacheからCインプリメンテーションを使用して)。それは素晴らしいものだが、万が一私たちが同様にJava言語でXML-RPCサーバーを開発しなければならなかったならば、どうでしょう？実際には、たやすいことでした。私たちのプロジェクトにおいては、私たちが、Java言語で書かれた、Marquee XML-RPCサーバー・インプリメンテーションを使用することができました(私たちが実際にどのように行ったか知るため、あるいはその他の目的のために、XML-RPCのその他の用途を参照してください)。

リスト6は、前記のXML-RPCクライアントからのリクエストを扱う、簡単なXML-RPCサーバーを示します。それは単純にサンプル目的で、実際の物理的なデバイスをシミュレートします。XML-RPCサーバーの詳細を知るためにこのコードを分析しましょう。

リスト6. 単純なXML-RPCサーバー

```
import java.io.IOException;
import marquee.xmlrpc.XmlRpcServer;
import marquee.xmlrpc.handlers.ReflectiveInvocationHandler;
public class DeviceServer {
    public static final String USERNAME = "username";
    public static final String PASSWORD = "password";
    protected boolean isShuttingDown;
    protected String configuration = "initial configuration";
    protected String host;
    protected String password = PASSWORD;
    protected XmlRpcServer rpcServer;
    protected Thread rpcThread;
    protected int port;
    protected Status status = Status.OK;
    public DeviceServer(String theHost, int thePort) {
        host = theHost;
        port = thePort;
        createRpcServer();
        startRpcServer();
    }
    public void shutDown() {
        isShuttingDown = true;
        if (rpcServer != null)
            rpcServer.shutDown();
    }
    public Object getStatus() {
        return new Integer(status.getCode());
    }
    public Object getConfiguration() {
        return "valid configuration data";
    }
    public Object replaceConfiguration(String xml) {
        configuration = xml;
        return new Integer(status.getCode());
    }
    public Object reboot() {
        return new Integer(status.getCode());
    }
    public void setStatus(Status newStatus) {
        status = newStatus;
```

```
    }
    protected void startRpcServer() {
        rpcThread = new Thread(new Runnable() {
            public void run() {
                try {
                    rpcServer.runAsService(port);
                } catch (IOException ioe) {
                    if (!isShuttingDown)
                        ioe.printStackTrace();
                }
            }
        });
        rpcThread.setName("DeviceServer[" + this.host + "] on " + this.port);
        rpcThread.start();
    }
    protected void createRpcServer() {
        rpcServer = new XmlRpcServer();
        rpcServer.registerInvocationHandler(
            "Device",
            new ReflectiveInvocationHandler(this));
    }
}
```

私たちのサーバー用のコンストラクタは、これから起こることの概要を与えてくれます。私たちは、渡されたホストおよびポート情報を保存します。その次に、Marquee`XmlRpcServer`をインスタンス化し、そしてそれをもつ`ReflectiveInvocationHandler`(間もなく詳細を示します)を登録するために`createRpcServer()`を呼びます。その後、サーバーに有用な名前を与え、そしてそれ自身のスレッドの中で実行するために`startRpcServer()`を呼びます。`Marquee XmlRpcServer`は、スレッドを開始する前に`int`のポート番号を持つ`runAsService()`を呼ぶ必要があります。一旦それを開始すると、サーバーは、そのポートに接続された任意のクライアントからそのポートに入るリクエストのために待機します。

そのうちのほとんどは簡単ですが、この`ReflectiveInvocationHandler`はどうでしょう? XML-RPCの仕様では、クライアントあるいはサーバーのどちらの実装方法も詳しく述べていません。それは実に、XML-RPCサーバーが`<methodcall>`要素を持って入って来るリクエストを扱わなければならないと述べています。その要素の中身は、`handlername.methodname`の形式の呼出し文字列です。Marquee クライアント上で`invoke()`を呼出すとき、それはあなたのメソッド文字列と任意のパラメーターを、的確なXML-RPCメッセージでサーバーへ渡すために、正確なXMLを生成します。サーバーサイドの、Marquee XML-RPCサーバー

- ハンドラ名およびそのハンドラを要求するメソッドにメソッド文字列を解析します。
- 示された名前をもつ登録されたハンドラを見つけます。
- リクエストで送信した任意のパラメーターを渡して、その上でメソッドを呼びます。
- XML-RPCレスポンス内に結果をパッケージし、クライアントに返信します。

Marquee XML-RPCサーバーを使用するために、実行するサーバーインスタンスは、メソッド文字列をデコードする方法を知らなければなりません。それは、どのオブジェクトが`handlername`キーの値に相当するか知らなければなりません。そうは言っても、もしサーバーに「デバイス」という名前が与えられたオブジェクトに相当することを教えていなければ、サーバーがその`handlername`をデコードする方法が分からないということは明白なはずです。それはどんなインスタンスでもありえます。リスト7で示すように、私たちは単にサーバーに、入ってくる全てのメソッドリクエストを扱わせました。私たちは、`createRpcServer()`のこのコードでそれを行いました。

リスト7. 単純なXML-RPCサーバー

```
rpcServer.registerInvocationHandler("Device", new
ReflectiveInvocationHandler(this));
```

さて、私たちのサーバーは、`Device.someMethod`のようなメソッド文字列を持つリクエストを取得した場合は常に、そのリクエストを扱うためにそれ自体の上で`someMethod()`を求めるということを承知しています。私たちのサンプル・サーバーでは、私たちが必要としたすべてのことは、基本的な「要求されたオブジェクト上で要求されたメソッドを求める」という振る舞いであり、それで私たちはMarqueeの`ReflectiveInvocationHandler`を使用しました。Marquee に付属の基本的なものはそれをちゃんと行うので、私自身で書く必要がありませんでした。そのハンドラは単純に、ハンドラがインスタンス化されたオブジェクト上で、要求されたメソッドを求めます。コードを見れば、Java Reflection logic Marqueeがあなたから記述の困難を取り除いたということすべてを理解するでしょう。

ハンドラの概念はXML-RPCの仕様によって要求されてはいませんが、Marquee はそれに近い形で構築され、またそれが有効に働きます。基本的な`ReflectiveInvocationHandler`があなたのジョブに合わない場合、あなたのものに合うように`XmlRpcInvocationHandler`をサブクラスにすることができます。

XML-RPCのその他の用途

この記事に記述したプロジェクトにおいて、私たちはテスト目的で、アプリケーションの外部スクリプトの記述を容易にするために、XML-RPCを使用しました。私たちはRubyで簡単なテストをするフレームワークを書き、それに私たちのアプリケーション(それはMarquee Java XML-RPCサーバーを含んでいました)へのXML-RPCリクエストを行なわせました。アプリケーションを納品する時がやって来たとき、私たちはただXML-RPCサーバーを停止させました。

要約

SWTとRuby

私が作業したもう一つのプロジェクトにおいては、私たちはバックエンドで、SWT UIとRubyでアプリケーションを作成するためにXML-RPCを使用しました。私たちが書いていたアプリケーションのためにRubyを使用することはより簡単でした。しかし、Ruby用のUIライブラリはSWTにとてもかないません。XML-RPCは、余計なトラブルなしでそれらの世界と結合することを可能にしました。

この記事でレビューしたサンプルは明らかに単純化してあります。Marquee XML-RPCライブラリは、私が記さなかった多くの機能を含んでいます(プリプロセッサの起動やJavaオブジェクトをXML-RPC送信用のstructに変換するシリアライザのような)。しかしながら、それらの追加機能はおまけです。XML-RPCから見事な値を取り出すためにはそれらは必要ではありません。XML-RPCは本当に単純ですが、それは分散アプリケーションのために考慮する価値があります。一般論として、あなたが2つのアプリケーション間で通信する必要がある場合、特にそれらのアプリケーションが異なる言語で書かれている場合、XML-RPCは一見の価値があります。XML-RPCウェブサイトが引用した Byteマガジンの評論家の「分散コンピューティングはこれより困難でなければなりませんか？私はそうは思いません。」という発言に、私は少なくともここで話したプロジェクトを考えると同意します。この場合、XML-RPCは、邪魔にならずに、私たちに必要なことをさせてくれました。よいツールとはそういうものです。

著者について

Roy Miller

Roy W. Miller は最初に Andersen Consulting (現在の Accenture) にて、10 年以上技術コンサルタント、ソフトウェア開発者、および指南役を務め、ノースキャロライナの RoleModel Software 社にてほぼ 3 年を過ごしました(ここでは彼は、エクストリームプログラミング (XP) を使用した Java 言語アプリケーションの構築に注力しました)。現在、彼は独立コンサルタントであり指南役です。彼は、XP を含む重要なメソッドおよび機動的なメソッドを使用しており、Addison-Wesley XP Series (Extreme Programming Applied: Playing to Win) 中で本を共同執筆しました。彼の最も最近の本である Managing Software for Growth: Without Fear, Control, and the Manufacturing Mindset は、どのように、複雑な技術がソフトウェア開発を支援することができ、そして他の IT マネージャー達が、プログラマをコントロールしたり枯らすことなく、彼らのチームが実際の人々が使用して楽しいような素晴らしいソフトウェアを作成することの、支援方法を理解するかを述べています。