Capítulo 43. Zend_XmlRpc

Traduzido por Flávio Gomes da Silva Lisboa

Sumário

43.1. Introdução

Em sua home page, XML-RPC é descrita como uma "...chamada a procedimento remoto usando HTTP como transporte e XML como codificação. XML-RPC é projetado para ser tão simples quanto possível, enquanto permite que estruturas de dados complexas sejam transmitidas, processadas e retornadas."

O Zend Framework provê suporte tanto para consumo de serviços XML-RPC remotos quanto a construção de novos servidores XML-RPC.

43.2. Zend_XmlRpc_Client

43.2.1. Introdução

O Zend Framework provê suporte para consumir serviços XML-RPC remotos como um cliente no pacote `Zend_XmlRpc_Client`. Suas principais características incluem conversão automática de tipos entre PHP e XML-RPC, um objeto servidor proxy[1], e acesso às capacidades de introspecção do servidor.

43.2.2. Chamadas de Método

O construtor de `Zend_XmlRpc_Client` recebe a URL do ponto final do servidor XML-RPC remoto como seu primeiro parâmetro. A nova instância retornada pode ser usada para chamar qualquer número de métodos remotos naquele ponto final.

Para chamar um método com o cliente XML-RPC, instancie-o e use o método de instância `call()`. O exemplo de código abaixo usa um servidor XML-RPC de demonstração no website do Zend Framework. Você pode usá-lo para testar ou explorar os componentes `Zend_XmlRpc`.

Exemplo 43.1. Chamada de Método XML-RPC

```
<?php
require_once 'Zend/XmlRpc/Client.php';
$client = new Zend_XmlRpc_Client('http://framework.zend.com/xmlrpc');
echo $client->call('test.sayHello');
// hello
```

O valor XML-RPC retornado pela chamada de método remoto irá automaticamente ser verificado e forçado a assumir um tipo nativo PHP equivalente. No exemplo acima, um tipo `string do` PHP é retornado e imediatamente está pronto para ser usado.

O primeiro parâmetro do método `call()` recebe o nome do método remoto a ser chamado. Se o método remoto requer quaisquer parâmetros, estes podem ser enviados fornecendo um segundo parâmetro, opcional, para `call()` com um `array` de valores para passar ao método remoto:

Exemplo 43.2. Chamada de Método XML-RPC com Parâmetros

1 Um servidor proxy é um servidor intermediário que recebe solicitações de recursos e conecta-se aos servidores que disponibilizam esses recursos.

```php
<?php
require_once 'Zend/XmlRpc/Client.php';
$client = new Zend_XmlRpc_Client('http://framework.zend.com/xmlrpc');
$arg1 = 1.1;
$arg2 = 'foo';
$result = $client->call('test.sayHello', array($arg1, $arg2));
// $result is a native PHP type
```

Se o método remoto não requer parâmetros, este parâmetro opcional pode ou ser descartado ou passado como um `array()` vazio. O vetor de parâmetros para o método remoto pode conter tipos nativos do PHP, objetos `Zend_XmlRpc_Value`, ou uma mistura de cada.

O método `call()` irá automaticamente converter a resposta XML-RPC e retornar seu tipo nativo PHP equivalente. Um objeto `Zend_XmlRpc_Response` para valor retornado ficará disponível pela chamando o método `getLastResponse()` depois da chamada ao método `call()`.

43.2.3. Tipos e Conversões

Algumas chamadas de método remoto requerem parâmetros. Estes são dados pelo método `call()` de `Zend_XmlRpc_Client` como um vetor no segundo parâmetro. Cada parâmetro pode ser dado ou como um tipo nativo PHP que será automaticamente convertido, ou como um objeto representando um tipo XML-RPC específico (um dos objetos `Zend_XmlRpc_Value`).

43.2.3.1. Tipos Nativos PHP como Parâmetros

Parâmetros podem ser passados para `call()` como variáveis PHP nativas, isto é, `string`, `integer`, `float`, `boolean`, `array` ou `object`. Nesse caso, cada tipo nativo PHP será autodetectado e convertido para um dos tipos de acordo com esta tabela:

Tabela 43.1. Conversão de Tipos PHP e XML-RPC

Tipo Nativo PHP	Tipo XML-RPC
integer	int
double	double
boolean	boolean
string	string
array	array
associative array	struct
object	array

43.2.3.2. Objetos `Zend_XmlRpc_Value` como Parâmetros

Parâmetros podem também ser criados como instâncias `Zend_XmlRpc_Value` para especificar um tipo XML-RPC exato. As principais razões para fazer isso são:

- Quando você quiser certificar-se que o parâmetro correto foi passado para o procedimento (isto é, o procedimento requer um inteiro e você pode obtê-lo do banco de dados como string)
- Quando o procedimento requer um tipo `base64` ou `dateTime.iso8601` (que não existe como tipo nativo do PHP)
- Quando a autoconversão puder falhar (isto é, você quer passar uma estrutura XML-RPC como um parâmetro. Estruturas vazias são representadas como vetores vazios em PHP mas, se você der um vetor vazio como parâmetro ele será autoconvertido em um vetor XML-RPC desde que não seja um vetor associativo) .

Há dois modos de criar um objeto `Zend_XmlRpc_Value`: instanciar uma das subclasses `Zend_XmlRpc_Value` diretamente, ou usar o método de fabricação estático `Zend_XmlRpc_Value::getXmlRpcValue()`.

Tabela 43.2. Objetos `Zend_XmlRpc_Value` Objects para Tipos XML-RPC Types

Tipo XML-RPC	Constante `Zend_XmlRpc_Value`	Objeto `Zend_XmlRpc_Value`
int	`Zend_XmlRpc_Value::XMLRPC_TYPE_INTEGER`	`Zend_XmlRpc_Value_Integer`
double	`Zend_XmlRpc_Value::XMLRPC_TYPE_DOUBLE`	`Zend_XmlRpc_Value_Double`
boolean	`Zend_XmlRpc_Value::XMLRPC_TYPE_BOOLEAN`	`Zend_XmlRpc_Value_Boolean`
string	`Zend_XmlRpc_Value::XMLRPC_TYPE_STRING`	`Zend_XmlRpc_Value_String`
base64	`Zend_XmlRpc_Value::XMLRPC_TYPE_BASE64`	`Zend_XmlRpc_Value_Base64`
dateTime.iso8601	`Zend_XmlRpc_Value::XMLRPC_TYPE_DATETIME`	`Zend_XmlRpc_Value_DateTime`
array	`Zend_XmlRpc_Value::XMLRPC_TYPE_ARRAY`	`Zend_XmlRpc_Value_Array`
struct	`Zend_XmlRpc_Value::XMLRPC_TYPE_STRUCT`	`Zend_XmlRpc_Value_Struct`

Conversão Automática

Quando construímos um novo objeto `Zend_XmlRpc_Value`, seu valor é configurado para um tipo PHP. O tipo PHP será convertido para o tipo especificado usando *PHP type casting*[2]. Por exemplo, se um tipo string é dado como valor para o objeto `Zend_XmlRpc_Value_Integer`, ele será convertido usando `(int)$value`.

2 Quando se obriga uma variável a assumir um determinado tipo.

43.2.4. Objeto Server Proxy

Outro modo de chamar métodos remotos com o cliente XML-RPC é usar o proxy de servidor. Este é um objeto PHP que substitui um namespace[3] XML-RPC remoto, fazendo-o trabalhar o mais próximo possível de um objeto PHP nativo.

Para instanciar um proxy de servidor, chame o método de instância `getProxy()` de `Zend_XmlRpc_Client`. Este retornará uma instância de `Zend_XmlRpc_Client_ServerProxy`. Qualquer chamada de método de um objeto proxy de servidor será encaminhada para o remoto, e os parâmetros podem ser passados como qualquer outro método PHP.

Exemplo 43.3. Substituindo o Namespace Padrão

```
<?php
require_once 'Zend/XmlRpc/Client.php';
$client = new Zend_XmlRpc_Client('http://framework.zend.com/xmlrpc');
$server = $client->getProxy();           // Substitui o namespace padrão
$hello = $server->test->sayHello(1, 2);  // test.Hello(1, 2) retorna "hello"
```

O método `getProxy()` recebe um argumento opcional especificando qual namespace do servidor remoto deve ser substituído. Se ele não receber um namespace, o namespace padrão será substituído. No próximo exemplo, o namespace `test` será substituído:

Exemplo 43.4. Substituindo Qualquer Namespace

```
<?php
require_once 'Zend/XmlRpc/Client.php';
$client = new Zend_XmlRpc_Client('http://framework.zend.com/xmlrpc');
$test  = $client->getProxy('test');     // Substitui o namespace "test"
$hello = $test->sayHello(1, 2);         // test.Hello(1,2) retorna "hello"
```

Se o servidor remoto suporta namespaces aninhados de qualquer profundidade, estes podem ser também usados através do proxy do servidor. Por exemplo, se o servidor no exemplo acima tinha um método `test.foo.bar()`, ele poderia ser chamado como `$test->foo->bar()`.

43.2.5. Manipulação de Erros

Dois tipos de erros podem ocorrer durante uma chamada de método XML-RPC: erros HTTP e falhas XML-RPC. `Zend_XmlRpc_Client` reconhece cada um e provê a habilidade de detectar e criar ciladas para eles de forma independente.

3 Container abstrato que fornece contexto para os itens que armazena (nomes, termos técnicos, conceitos...), e que fornece desambiguação para itens que possuem o mesmo nome mas que residem em espaços de nomes diferentes

43.2.5.1. HTTP Errors

Se qualquer erro HTTP ocorre, tal como o servidor HTTP remoto retornar um `404 Not Found`, uma exceção `Zend_XmlRpc_Client_HttpException` será lançada.

Exemplo 43.5. Manipulando Erros HTTP

```
<?php
require_once 'Zend/XmlRpc/Client.php';
$client = new Zend_XmlRpc_Client('http://foo/404');
try {
    $client->call('bar', array($arg1, $arg2));
} catch (Zend_XmlRpc_Client_HttpException $e) {
    // $e->getCode() retorna 404
    // $e->getMessage() retorna "Not Found"
}
```

A despeito de como o cliente XML-RPC client é usado, `Zend_XmlRpc_Client_HttpException` será lançada sempre que um erro HTTP ocorrer.

43.2.5.2. Falhas XML-RPC

Uma falha XML-RPC é análoga à exceção PHP. É um tipo especial retornado de uma chamada de método XML-RPC que tem um código de erro e uma mensagem de erro. Falhas XML-RPC são manipuladas de forma diferente dependendo do contexto no qual `Zend_XmlRpc_Client` é usado.

Quando o método `call()` ou o objeto servidor do proxy é usado, uma falha XML-RPC resultará em uma exceção `Zend_XmlRpc_Client_FaultException` sendo lançada. O código e mensagem da exceção irá mapear diretamente para seus respectivos valores na resposta de falha XML-RPC original.

Exemplo 43.6. Manipulando Falhas XML-RPC

```
<?php
require_once 'Zend/XmlRpc/Client.php';
$client = new Zend_XmlRpc_Client('http://framework.zend.com/xmlrpc');
try {
    $client->call('badMethod');
} catch (Zend_XmlRpc_Client_FaultException $e) {
    // $e->getCode() retorna 1
    // $e->getMessage() retorna "Unknown method"
}
```

Quando o método `call()` é usado para fazer a requisição, a exceção `Zend_XmlRpc_Client_FaultException` será lançada em caso de falha. Um objeto `Zend_XmlRpc_Response` contendo a falha também estará disponível pela chamada a `getLastResponse()`.

Quando o método `doRequest()` é usado para fazer a requisição, não será lançada a exceção. Ao invés disso, ele retornará um objeto `Zend_XmlRpc_Response` contendo a falha. Isso pode ser verificado com o método de instância `isFault()` de `Zend_XmlRpc_Response`.

43.2.6. Introspecção de Servidor

Alguns servidores XML-RPC suportam os métodos de introspecção (de fato) debaixo de XML-RPC `system.namespace`. `Zend_XmlRpc_Client` provê suporte especial para servidores com essa capacidades.

Uma instância `Zend_XmlRpc_Client_ServerIntrospection` pode ser recuperada pela chamada ao método `getIntrospector()` de `Zend_XmlRpcClient`. Isso pode ser usado então para executar operações de introspecção no servidor.

43.2.7. Da Requisição a Resposta

Debaixo da coberta, o método de instância `call()` constrói um objeto de requisição (`Zend_XmlRpc_Request`) e envia-o para outro método, `doRequest()`, que retorna um objeto de resposta (`Zend_XmlRpc_Response`).

O método `doRequest()` também está disponível para uso direto:

Exemplo 43.7. Processando Requisição para Resposta

```
<?php
require_once 'Zend/XmlRpc/Client.php';
$client = new Zend_XmlRpc_Client('http://framework.zend.com/xmlrpc');
$request = new Zend_XmlRpc_Request();
$request->setMethod('test.sayHello');
$request->setParams(array('foo', 'bar'));
$client->doRequest($request);
// $server->getLastRequest() retorna instância de Zend_XmlRpc_Request
// $server->getLastResponse() retorna instância de Zend_XmlRpc_Response
```

Sempre que uma chamada de método XML-RPC é feita pelo cliente através de qualquer meio, ou o método `call()`, ou o método `doRequest()`, ou proxy de servidor, o último objeto de requisição e seu objeto de resposta resultante estará sempre disponível através dos métodos `getLastRequest()` e `getLastResponse()` respectivamente.

43.2.8. Cliente HTTP e Teste

Em todos os exemplos anteriores, um cliente HTTP nunca foi especificado. Quando este é o caso, uma nova instância de `Zend_Http_Client` será criada com suas opções padrão e usada por `Zend_XmlRpc_Client` automaticamente.

O cliente HTTP pode ser recuperado a qualquer momento com o método `getHttpClient()`. Para a maioria dos casos, o cliente HTTP padrão será suficiente. Entretanto , o método `setHttpClient()` permite que uma instância de cliente HTTP diferente seja injetada.

O método `setHttpClient()` é particularmente útil para uso em testes. Quando combinado com `Zend_Http_Client_Adapter_Test`, serviços remotos podem ser simulados para teste. Veja os testes unitários de `Zend_XmlRpc_Client` ilustrando como fazer isso.

43.3. Zend_XmlRpc_Server

43.3.1. Introdução

Zend_XmlRpc_Server tem como intenção ser um servidor XML-RPC com características completas, seguindo as especificações descritas em www.xml.rpc. Adicionamente, ele implementa o método system.multicall(), permitindo o encarrilamento de requisições.

43.3.2. Uso Básico

Um exemplo do caso de uso mais básico:

```
<?php
require_once 'Zend/XmlRpc/Server.php';
require_once 'My/Service/Class.php';
$server = new Zend_XmlRpc_Server();
$server->setClass('My_Service_Class');
echo $server->handle();
```

43.3.3. Estrutura do Servidor

A classe Zend_XmlRpc_Server é composta de uma variedade de componentes, variando do próprio servidor para requisição, resposta e objetos de falha.

Para iniciar Zend_XmlRpc_Server, o desenvolvedor deve anexar uma ou mais classes ou funções ao servidor, através dos métodos `setClass()` e `addFunction()`.

Uma vez feito isso, você pode ou passar um objeto `Zend_XmlRpc_Request` para o método `Zend_XmlRpc_Server::handle()`, ou instanciar um objeto `Zend_XmlRpc_Request_Http` se nenhum foi fornecido – apoderando-se assim da requisição

de `php://input`.

O método `Zend_XmlRpc_Server::handle()` tenta então despachar para o manipulador apropriado baseado no método requisitado. Então retorna ou um objeto baseado em `Zend_XmlRpc_Response` ou um objeto `Zend_XmlRpc_Server_Fault`. Esses objetos têm ambos os métodos `__toString()` que criam respostas XML-RPC XML válidas, permitindo a eles serem diretamente ecoados.

43.3.4. Convenções

Zend_XmlRpc_Server permite ao desenvolvedor anexar funções e chamadas de métodos de classe como métodos XML-RPC despacháveis.. Por meio de Zend_Server_Reflection, ele realiza introspecção em todos os métodos anexados, usando blocos de documentação de métodos e funções para determinar o texto de ajuda do método e as asssinaturas dos métodos.

Tipos XML-RPC não mapeiam necessariamente tipos PHP um a um. Entretanto, o código fará o melhor para encontrar o tipo apropriado baseado nos valores listados nas linhas @param e @return. Alguns tipos XML-RPC não tem um equivalente PHP imediato, contudo, e devem ser sinalizados usando o tipo XML-RPC no phpdoc. Estes incluem:

- dateTime.iso8601, um tipo string formatado como YYYYMMDDTHH:mm:ss
- base64, dados condificados base64
- struct, qualquer vetor associativo

Um exemplo de como sinalizar a seguir:

```
<?php
/**
* Esta é um função de exemplo
*
* @param base64 $val1 Base64-encoded data
* @param dateTime.iso8601 $val2 An ISO date
* @param struct $val3 An associative array
* @return struct
*/
function myFunc($val1, $val2, $val3)
{
}
```

PhpDocumentor não faz validação dos tipos especificados para parâmetros ou valores de retorno, assim isso não terá impacto na documentação de sua API. O fornecimento da sinalização, entretanto, é necessária quando o servidor está validando os parâmetros fornecidos pela chamada do método.

É perfeitamente válido especificar múltiplos tipos tanto para os parâmetros quanto para os valores de retorno; a especificação XML-RPC sugere que system.methodSignature deva retornar um vetor de todas as assinaturas de método possíveis (isto é, todas as combinações possíveis de parâmetros e

valores de retorno). Você pode fazer isso tão fácil quanto você normalmente faria com PhpDocumentor, usando o operador '|':

```
<?php
/**
* Esta é uma função de exemplo
*
* @param string|base64 $val1 String ou dado codificado base64
* @param string|dateTime.iso8601 $val2 String ou um ISO date
* @param array|struct $val3 Vetor indexado normal ou um vetor associativo
* @return boolean|struct
*/
function myFunc($val1, $val2, $val3)
{
}
```

Uma nota, porém: permitir múltiplas assinaturas pode levar a confusão por parte de desenvolvedores que usam os serviços; de modo genérico, um método XML-RPC deve tão somente ter uma única assinatura.

43.3.5. Utilizando Namespaces

XML-RPC tem um conceito de namespacing; basicamente, ele permite o agrupamento de métodos XML-RPC por namespaces delimitados por ponto. Isso ajuda a prevenir colisões de nomeação entre métodos servidos por diferentes classes. Como um exemplo, é esperado que o servidor XML-RPC sirva diversos métodos no namespace 'system':

- system.listMethods
- system.methodHelp
- system.methodSignature

Internalmente, eles mapeiam para métodos do mesmo nome em Zend_XmlRpc_Server.

Se você quer adicionar namespaces para os métodos que você serve, simplesmente forneça um namespace para o método apropriado quando anexar uma função ou classe:

```
<?php
// Todos os métodos públicos em My_Service_Class serão acessíveis como
// myservice.METHODNAME
$server->setClass('My_Service_Class', 'myservice');
// A função 'somefunc' será acessível como funcs.somefunc
$server->addFunction('somefunc', 'funcs');
```

43.3.6. Objetos de Requisição Customizados

A maior parte do tempo, você irá simplesmente usar o tipo de requisição padrão incluído com

Zend_XmlRpc_Server e Zend_XmlRpc_Request_Http. Entretanto, há vezes em que você precisa que XML-RPC esteja disponível pela linha de comando, uma GUI, ou outro ambiente, ou quer fazer log de requisições de entrada. Para fazer isso, você precisa criar um objeto de requisição customizado que estenda Zend_XmlRpc_Request. A coisa mais importante a ser lembrada é garantir que os métodos getMethod() e getParams() sejam implementados assim que o servidor XML-RPC possa recuperar essa informação de modo a despachar a requisição.

43.3.7. Respostas Customizadas

Similar aos objetos de requisição, Zend_XmlRpc_Server pode retornar objetos de resposta customizados; por padrão, um objeto Zend_XmlRpc_Response_Http object é retornado, o qual envia um cabeçalho HTTP Content-Type apropriado para usar com XML-RPC. Possíveis usos de um objeto customizado seriam logar respostas ou enviar respostas de volta a STDOUT.

Para usar uma classe de resposta customizada, use Zend_XmlRpc_Server::setResponseClass() antes de chamar handle().

43.3.8. Manipulação de Exceções via Falhas

Zend_XmlRpc_Server captura exceções geradas por um método despachante, e gera uma resposta de falha XML-RPC quando tal exceção é capturada. Por padrão, contudo, as mensagens de exceção e códigos não são usados em uma resposta de falha. Isso é uma decisão intencional para proteger seu código; muitas exceções expõem mais informações sobre o código ou ambiente do que um desenvolvedor necessariamente pretendidas (um exemplo primário inclui abstração de banco de dados ou exceções de camada de acesso).

Classes de exceção podem se tornar listas brancas para serem usadas como respostas de falha, entretanto. Para fazer isso, simplesmente utilize Zend_XmlRpc_Server_Fault::attachFaultException() para passar uma classe de exceção para a lista branca:

```
<?php
Zend_XmlRpc_Server_Fault::attachFaultException('My_Project_Exception');
```

Se você utilizar uma classe de exceção que suas exceções de outros projetos herdam, você pode então tornar uma família inteira de exceções de uma vez em listas brancas. Zend_XmlRpc_Server_Exceptions são sempre listas brancas, para permitir a reportagem de erros internos específicos (métodos indefinidos, etc).

Qualquer exceção não definida especificamente como lista branca irá gerar uma resposta de falha com um código '404' e uma mensagem de 'Unknown error'.

43.3.9. Cacheando Definições de Servidor Entre Requisições

Anexar muitas classes a uma instância de servidor XML-RPC pode utilizar muitos recursos; cada classe deve examinar a si mesma usando a API de reflexão (via Zend_Server_Reflection), que por

sua vez gera uma lista de todos as possíveis assinaturas de métodos para fornecer à classe servidora.

Para reduzir esse pico de performance de alguma forma, Zend_XmlRpc_Server_Cache pode ser usada para cachear a definição do servidor entre requisições. Quando combinado com __autoload(), pode aumentar consideravelmente a performance.

Um exemplo de uso a seguir:

```
<?php
require_once 'Zend/Loader.php';
require_once 'Zend/XmlRpc/Server.php';
require_once 'Zend/XmlRpc/Server/Cache.php';
function __autoload($class)
{
    Zend_Loader::loadClass($class);
}
$cacheFile = dirname(__FILE__) . '/xmlrpc.cache';
$server = new Zend_XmlRpc_Server();
if (!Zend_XmlRpc_Server_Cache::get($cacheFile, $server)) {
    require_once 'My/Services/Glue.php';
    require_once 'My/Services/Paste.php';
    require_once 'My/Services/Tape.php';
    $server->setClass('My_Services_Glue', 'glue');   // glue. namespace
    $server->setClass('My_Services_Paste', 'paste'); // paste. namespace
    $server->setClass('My_Services_Tape', 'tape');   // tape. namespace
    Zend_XmlRpc_Server_Cache::save($cacheFile, $server);
}
echo $server->handle();
```

O exemplo acima tenta receber uma definição de servidor de xmlrpc.cache no mesmo diretório do script. Se fracassar, ele carrega as classes de serviço que precisa, anexa as mesmas à instância do servidor e então tenta criar um novo arquivo de cache com a definição do servidor.

43.3.10. Exemplos de Uso

Abaixo estão vários exemplos de uso, mostrando o espectro completo de opções disponíveis para desenvolvedores. Os exemplos de uso irão, cada um, ser construídos sobre o exemplo fornecido anteriormente.

43.3.10.1. Uso Básico

O exemplo abaixo anexa uma função como um método XML-RPC despachável e manipula as chamadas de entrada.

```
<?php
require_once 'Zend/XmlRpc/Server.php';
/**
 * Retorna a soma MD5 de um valor
 *
 * @param string $value Valor para md5sum
```

```
 * @return string soma MD5 do valor
 */
function md5Value($value)
{
    return md5($value);
}
$server = new Zend_XmlRpc_Server();
$server->addFunction('md5Value');
echo $server->handle();
```

43.3.10.2. Anexando uma classe

O exemplo abaixo ilustra como anexar métodos públicos de uma classe como métodos XML-RPC despacháveis.

```
<?php
require_once 'Zend/XmlRpc/Server.php';
require_once 'Services/Comb.php';
$server = new Zend_XmlRpc_Server();
$server->setClass('Services_Comb');
echo $server->handle();
```

43.3.10.3. Anexando várias classes usando namespaces

O exemplo abaixo ilustra a junção de várias classes, cada uma com seu próprio namespace.

```
<?php
require_once 'Zend/XmlRpc/Server.php';
require_once 'Services/Comb.php';
require_once 'Services/Brush.php';
require_once 'Services/Pick.php';
$server = new Zend_XmlRpc_Server();
$server->setClass('Services_Comb', 'comb');   // métodos chamados como comb.*
$server->setClass('Services_Brush', 'brush'); // métodos chamados como brush.*
$server->setClass('Services_Pick', 'pick');   // métodos chamados como pick.*
echo $server->handle();
```

43.3.10.4. Especificando exceções para usar como respostas de falha válidas

O exemplo abaixo permite que qualquer classe derivada de Services_Exception reporte seu código e mensagem na resposta de falha.

```
<?php
require_once 'Zend/XmlRpc/Server.php';
require_once 'Zend/XmlRpc/Server/Fault.php';
require_once 'Services/Exception.php';
require_once 'Services/Comb.php';
require_once 'Services/Brush.php';
```

```
require_once 'Services/Pick.php';
// Permite a Services_Exceptions reportarem como respostas de falha
Zend_XmlRpc_Server_Fault::attachFaultException('Services_Exception');
$server = new Zend_XmlRpc_Server();
$server->setClass('Services_Comb', 'comb');   // métodos chamados como comb.*
$server->setClass('Services_Brush', 'brush'); // métodos chamados como brush.*
$server->setClass('Services_Pick', 'pick');   // métodos chamados como pick.*
echo $server->handle();
```

43.3.10.5. Utilizando um objeto de requisição customizado

O exemplo abaixo instancia um objeto de requisição customizado e o passa para o servidor manipular.

```
<?php
require_once 'Zend/XmlRpc/Server.php';
require_once 'Zend/XmlRpc/Server/Fault.php';
require_once 'Services/Request.php';
require_once 'Services/Exception.php';
require_once 'Services/Comb.php';
require_once 'Services/Brush.php';
require_once 'Services/Pick.php';
// Permite a Services_Exceptions reportarem como respostas de falha
Zend_XmlRpc_Server_Fault::attachFaultException('Services_Exception');
$server = new Zend_XmlRpc_Server();
$server->setClass('Services_Comb', 'comb');   // métodos chamados como comb.*
$server->setClass('Services_Brush', 'brush'); // métodos chamados como brush.*
$server->setClass('Services_Pick', 'pick');   // métodos chamados como pick.*
// Cria um objeto de requisição
$request = new Services_Request();
echo $server->handle($request);
```

43.3.10.6. Utilizando um objeto de resposta customizado

O exemplo abaixo ilustra especificamente uma classe de resposta customizada para a resposta retornada.

```
<?php
require_once 'Zend/XmlRpc/Server.php';
require_once 'Zend/XmlRpc/Server/Fault.php';
require_once 'Services/Request.php';
require_once 'Services/Response.php';
require_once 'Services/Exception.php';
require_once 'Services/Comb.php';
require_once 'Services/Brush.php';
require_once 'Services/Pick.php';
// Permite a Services_Exceptions reportarem como respostas de falha
Zend_XmlRpc_Server_Fault::attachFaultException('Services_Exception');
$server = new Zend_XmlRpc_Server();
$server->setClass('Services_Comb', 'comb');   // métodos chamado como comb.*
$server->setClass('Services_Brush', 'brush'); // métodos chamado como brush.*
$server->setClass('Services_Pick', 'pick');   // métodos chamado como pick.*
```

```
// Cria um objeto de requisição
$request = new Services_Request();
// Utiliza uma resposta customizada
$server->setResponseClass('Services_Response');
echo $server->handle($request);
```

43.3.10.7. Definições do servidor de cache entre requisições

O exemplo abaixo ilustra definições de servidor de cache entre requisições.

```
<?php
require_once 'Zend/XmlRpc/Server.php';
require_once 'Zend/XmlRpc/Server/Fault.php';
require_once 'Zend/XmlRpc/Server/Cache.php';
require_once 'Services/Request.php';
require_once 'Services/Response.php';
require_once 'Services/Exception.php';
require_once 'Services/Comb.php';
require_once 'Services/Brush.php';
require_once 'Services/Pick.php';
// Especifica um arquivo de cache
$cacheFile = dirname(__FILE__) . '/xmlrpc.cache';
// Permite a Services_Exceptions reportarem como respostas de falha
Zend_XmlRpc_Server_Fault::attachFaultException('Services_Exception');
$server = new Zend_XmlRpc_Server();
// Tenta recuperar a definição do servidor do cache
if (!Zend_XmlRpc_Server_Cache::get($cacheFile, $server)) {
    $server->setClass('Services_Comb', 'comb');   // métodos chamados como
 comb.*
    $server->setClass('Services_Brush', 'brush'); // métodos chamados
como brush.*
    $server->setClass('Services_Pick', 'pick');   // métodos chamados
como pick.*
    // Grava o cache
    Zend_XmlRpc_Server_Cache::save($cacheFile, $server));
}
// Cria um objeto de requisição
$request = new Services_Request();
// Utiliza uma resposta customizada
$server->setResponseClass('Services_Response');
echo $server->handle($request);
```