PHP Mapping for Operations

On this page:

Basic PHP Mapping for Operations

As we saw in the PHP mapping for interfaces, for each operation on an interface, a proxy object narrowed to that type supports a corresponding method with the same name. To invoke an operation, you call it via the proxy. For example, here is part of the definitions for our file system:

Slice

```
module Filesystem {
    interface Node {
        idempotent string name();
    };
    // ...
};
```

The `name` operation returns a value of type `string`. Given a proxy to an object of type `Node`, the client can invoke the operation as follows:

PHP

```
$node = ...             // Initialize proxy
$name = $node->name();  // Get name via RPC
```

Normal and `idempotent` Operations in PHP

You can add an `idempotent` qualifier to a Slice operation. As far as the signature for the corresponding proxy method is concerned, `idempotent` has no effect.

Passing Parameters in PHP

In-Parameters in PHP

The PHP mapping for `in` parameters guarantees that the value of a parameter will not be changed by the invocation.

Here is an interface with operations that pass parameters of various types from client to server:

Slice

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ClientToServer {
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
};
```

A proxy object narrowed to the `ClientToServer` interface supports the following methods:

PHP

```
function op1($i, $f, $b, $s, $_ctx=null);
function op2($ns, $ss, $st, $_ctx=null);
function op3($proxy, $_ctx=null);
```

Given a proxy to a `ClientToServer` interface, the client code can pass parameters as in the following example:

PHP

```
$p = ...                                // Get proxy...

$p->op1(42, 3.14, true, "Hello world!"); // Pass simple literals

$i = 42;
$f = 3.14;
$b = true;
$s = "Hello world!";
$p->op1($i, $f, $b, $s);                // Pass simple variables

$ns = new NumberAndString;
$ns->x = 42;
$ns->str = "The Answer";
$ss = array("Hello world!");
$st = array();
$st[0] = $ns;
$p->op2($ns, $ss, $st);                 // Pass complex variables

$p->op3($p);                            // Pass proxy
```

Out-Parameters in PHP

Out parameters are passed by reference. Here is the same Slice definition we saw earlier, but this time with all parameters being passed in the out direction:

Slice

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ServerToClient {
    int op1(out float f, out bool b, out string s);
    void op2(out NumberAndString ns,
             out StringSeq ss,
             out StringTable st);
    void op3(out ServerToClient* proxy);
};
```

The PHP mapping looks the same as it did for the in-parameters version:

PHP

```
function op1($i, $f, $b, $s, $_ctx=null);
function op2($ns, $ss, $st, $_ctx=null);
function op3($proxy, $_ctx=null);
```

Given a proxy to a `ServerToClient` interface, the client code can receive the results as in the following example:

PHP

```
$p = ...                // Get proxy...
$p->op1($i, $f, $b, $s);
$p->op2($ns, $ss, $st);
$p->op3($stcp);
```

Note that it is not necessary to use the reference operator (`&`) before each argument because the Ice run time forces each `out` parameter to have reference semantics.

Parameter Type Mismatches in PHP

The Ice run time performs validation on the arguments to a proxy invocation and reports any type mismatches as `InvalidArgumentException`.

Null Parameters in PHP

Some Slice types naturally have "empty" or "not there" semantics. Specifically, sequences, dictionaries, and strings all can be `null`, but the corresponding Slice types do not have the concept of a null value. To make life with these types easier, whenever you pass `null` as a parameter or return value of type sequence, dictionary, or string, the Ice run time automatically sends an empty sequence, dictionary, or string to the receiver.

This behavior is useful as a convenience feature: especially for deeply-nested data types, members that are sequences, dictionaries, or strings automatically arrive as an empty value at the receiving end. This saves you having to explicitly initialize, for example, every string element in a large sequence before sending the sequence in order to avoid a run-time error. Note that using null parameters in this way does *not* create null semantics for Slice sequences, dictionaries, or strings. As far as the object model is concerned, these do not exist (only *empty* sequences, dictionaries, and strings do). For example, it makes no difference to the receiver whether you send a string as `null` or as an empty string: either way, the receiver sees an empty string.

Exception Handling in PHP

Any operation invocation may throw a run-time exception and, if the operation has an exception specification, may also throw user exceptions. Suppose we have the following simple interface:

Slice

```
exception Tantrum {
    string reason;
};

interface Child {
    void askToCleanUp() throws Tantrum;
};
```

Slice exceptions are thrown as PHP exceptions, so you can simply enclose one or more operation invocations in a `try-catch` block:

PHP

```
$child = ...        // Get child proxy...

try
{
    $child->askToCleanUp();
}
catch(Tantrum $t)
{
    echo "The child says: " . $t->reason . "\n";
}
```

Typically, you will catch only a few exceptions of specific interest around an operation invocation; other exceptions, such as unexpected run-time errors, will usually be handled by exception handlers higher in the hierarchy. For example:

PHP

```
function run()
{
    $child = ...          // Get child proxy...
    try
    {
        $child->askToCleanUp();
    }
    catch(Tantrum $t)
    {
        echo "The child says: " . $t->reason . "\n";
        $child->scold();  // Recover from error...
    }
    $child->praise();     // Give positive feedback...
}

try
{
    // ...
    run();
    // ...
}
catch(Ice_Exception $ex)
{
    echo $ex->__toString() . "\n";
}
```

This code handles a specific exception of local interest at the point of call and deals with other exceptions generically. (This is also the strategy we used for our first simple application in Hello World Application.)

See Also

- Operations
- Hello World Application
- Slice for a Simple File System
- PHP Mapping for Interfaces
- PHP Mapping for Exceptions