Ruby Mapping for Operations

On this page:

Basic Ruby Mapping for Operations

As we saw in the Ruby mapping for interfaces, for each operation on an interface, the proxy class contains 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:

Ruby

```
node = ...          # Initialize proxy
name = node.name()  # Get name via RPC
```

Normal and `idempotent` Operations in Ruby

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. For example, consider the following interface:

Slice

```
interface Example {
                string op1();
    idempotent  string op2();
};
```

The proxy class for this is:

Ruby

```
class ExamplePrx < Ice::ObjectPrx
    def op1(_ctx=nil)

    def op2(_ctx=nil)
end
```

Because `idempotent` affects an aspect of call dispatch, not interface, it makes sense for the two methods to look the same.

Passing Parameters in Ruby

In-Parameters in Ruby

All parameters are passed by reference in the Ruby mapping; it is guaranteed 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);
};
```

The Slice compiler generates the following proxy for this definition:

Ruby

```
class ClientToServerPrx < Ice::ObjectPrx
    def op1(i, f, b, s, _ctx=nil)

    def op2(ns, ss, st, _ctx=nil)

    def op3(proxy, _ctx=nil)
end
```

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

Ruby

```
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 = NumberAndString.new()
ns.x = 42
ns.str = "The Answer"
ss = [ "Hello world!" ]
st = {}
st[0] = ns
p.op2(ns, ss, st)                       # Pass complex variables

p.op3(p)                                # Pass proxy
```

Out-Parameters in Ruby

As in Java, Ruby functions do not support reference arguments. That is, it is not possible to pass an uninitialized variable to a Ruby function in order to have its value initialized by the function. The Java mapping overcomes this limitation with the use of *holder classes* that represent each `out` parameter. The Ruby mapping takes a different approach, one that is more natural for Ruby users.

The semantics of `out` parameters in the Ruby mapping depend on whether the operation returns one value or multiple values. An operation returns multiple values when it has declared multiple `out` parameters, or when it has declared a non-`void` return type and at least one `out` parameter.

If an operation returns multiple values, the client receives them in the form of a *result array*. A non-`void` return value, if any, is always the first element in the result array, followed by the `out` parameters in the order of declaration.

If an operation returns only one value, the client receives the value itself.

Here again are the same Slice definitions 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 Ruby mapping generates the following code for this definition:

Ruby

```
class ClientToServerPrx < Ice::ObjectPrx
    def op1(_ctx=nil)

    def op2(_ctx=nil)

    def op3(_ctx=nil)
end
```

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

Ruby

```
p = ...              # Get proxy...
i, f, b, s = p.op1()
ns, ss, st = p.op2()
stcp = p.op3()
```

The operations have no `in` parameters, therefore no arguments are passed to the proxy methods. Since `op1` and `op2` return multiple values, their result arrays are unpacked into separate values, whereas the return value of `op3` requires no unpacking.

Parameter Type Mismatches in Ruby

Although the Ruby compiler cannot check the types of arguments passed to a method, the Ice run time does perform validation on the arguments to a proxy invocation and reports any type mismatches as a `TypeError` exception.

Null Parameters in Ruby

Some Slice types naturally have "empty" or "not there" semantics. Specifically, sequences, dictionaries, and strings all can be `nil`, but the corresponding Slice types do not have the concept of a null value. To make life with these types easier, whenever you pass `nil` 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 `nil` or as an empty string: either way, the receiver sees an empty string.

Optional Parameters in Ruby

Optional parameters use the same mapping as required parameters. The only difference is that `Ice::Unset` can be passed as the value of an optional parameter or return value. Consider the following operation:

Slice

```
optional(1) int execute(optional(2) string params, out optional(3) float value);
```

A client can invoke this operation as shown below:

Ruby

```
i, v = proxy.execute("--file log.txt")
i, v = proxy.execute(Ice::Unset)

if v != Ice::Unset
    puts "value = " + v.to_s
end
```

A well-behaved program must always compare an optional parameter to `Ice::Unset` prior to using its value.

Exception Handling in Ruby

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 Ruby exceptions, so you can simply enclose one or more operation invocations in a `begin-rescue` block:

Ruby

```

child = ...        # Get child proxy...

begin
    child.askToCleanUp()
rescue Tantrum => t
    puts "The child says: #{t.reason}"
end
```

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:

Ruby

```
def run()
    child = ...        # Get child proxy...
    begin
        child.askToCleanUp()
    rescue Tantrum => t
        puts "The child says: #{t.reason}"
        child.scold()  # Recover from error...
    end
    child.praise()     # Give positive feedback...
end

begin
    # ...
    run()
    # ...
rescue Ice::Exception => ex
    print ex.backtrace.join("\n")
end
```

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
- Ruby Mapping for Operations
- Ruby Mapping for Interfaces
- Ruby Mapping for Exceptions