Parameter Passing in Python

Previous Next

Parameter passing on the server side follows the rules for the client side. Additionally, every operation has a trailing parameter of type `Ice.Current`. For example, the `name` operation of the `Node` interface has no parameters, but the `name` method in a Python servant has a `current` parameter. We will ignore this parameter for now.

On this page:

Server-Side Mapping for Parameters in Python

For each `in` parameter of a Slice operation, the Python mapping generates a corresponding parameter for the method in the skeleton. An operation returning multiple values returns them in a tuple consisting of a non-`void` return value, if any, followed by the `out` parameters in the order of declaration. An operation returning only one value simply returns the value itself.

> An operation returns multiple values when it declares multiple out parameters, or when it declares a non-`void` return type and at least one out parameter.

To illustrate these rules, consider the following interface that passes string parameters in all possible directions:

Slice

```
interface Example
{
    string op1(string sin);
    void op2(string sin, out string sout);
    string op3(string sin, out string sout);
}
```

The generated skeleton class for this interface looks as follows:

Python

```
class Example(Ice.Object):
    def __init__(self):
        # ...

    #
    # Operation signatures.
    #
    # def op1(self, sin, current=None):
    # def op2(self, sin, current=None):
    # def op3(self, sin, current=None):
```

The signatures of the Python methods are identical because they all accept a single in parameter, but their implementations differ in the way they return values. For example, we could implement the operations as follows:

Python

```
class ExampleI(Example):
    def op1(self, sin, current=None):
        print sin             # In params are initialized
        return "Done"         # Return value

    def op2(self, sin, current=None):
        print sin             # In params are initialized
        return "Hello World!" # Out parameter

    def op3(self, sin, current=None):
        print sin             # In params are initialized
        return ("Done", "Hello World!")
```

Notice that `op1` and `op2` return their string values directly, whereas `op3` returns a tuple consisting of the return value followed by the out parameter.

This code is in no way different from what you would normally write if you were to pass strings to and from a function; the fact that remote procedure calls are involved does not impact your code in any way. The same is true for parameters of other types, such as proxies, classes, or dictionaries: the parameter passing conventions follow normal Python rules and do not require special-purpose API calls.

Back to Top ^

Thread-Safe Marshaling in Python

The marshaling semantics of the Ice run time and the Python interpreter present a subtle thread safety issue that arises when an operation returns data by reference. For Python applications, this can affect servant methods that return instances of Slice classes, structures, sequences, or dictionaries.

In the C-based implementation of Python ("Cython"), only one thread at a time can be executing in the interpreter. However, depending on your t hread pool configuration, there may be multiple Ice threads waiting to enter the interpreter. You should write your code to assume that the interpreter can switch to a different thread at any time.

The potential for corruption occurs whenever a servant returns an instance of one of these types, yet continues to hold a reference to that data. For example, consider the following servant implementation:

Python

```
class GridI(Grid):
    def __init__(self):
        self._grid = # ...

    def getGrid(self, current):
        return self._grid

    def setValue(self, x, y, val, current):
        self._grid[x][y] = val
```

Suppose that a client invoked the `getGrid` operation, and another client invoked the `setValue` operation. The interpreter allows a thread to dispatch the call to `getGrid`, but before control returns to the Ice run time, the interpreter switches threads to allow the call to `setValue` to proceed. The problem is that `setValue` can modify the data before the thread that invoked `getGrid` has a chance to marshal its results. In most cases this won't cause a failure, but it does mean that an invocation might return different results than it intended. Furthermore, adding synchronization to the `getGrid` and `setValue` oper ations would not fix the race condition because the Ice run time performs its marshaling outside of this synchronization.

Solution 1: Copying

One solution is to implement accessor operations, such as `getGrid`, so that they return copies of any data that might change. There are several drawbacks to this approach:

- Excessive copying can have an adverse affect on performance.
- The operations must return deep copies in order to avoid similar problems with nested values.
- The code to create deep copies is tedious and error-prone to write.

Solution 2: Copy on Write

Another solution is to make copies of the affected data only when it is modified. In the revised code shown below, `setValue` replaces `_grid` with a copy that contains the new element, leaving the previous contents of `_grid` unchanged:

Python

```
class GridI(Grid):
    def __init__(self):
        self._lock = threading.Lock()

    def getGrid(self, current):
        with self._lock:
            return self._grid

    def setValue(self, x, y, val, current):
        with self._lock:
            newGrid = # shallow copy...
            newGrid[x][y] = val
            self._grid = newGrid
```

This allows the Ice run time to safely marshal the return value of `getGrid` because the array is never modified again. For applications where data is read more often than it is written, this solution is more efficient than the previous one because accessor operations do not need to make copies. Furthermore, intelligent use of shallow copying can minimize the overhead in mutating operations.

Solution 3: Marshal Immediately

Finally, a third approach is to modify the servant mapping using metadata in order to force the marshaling to occur immediately within your synchronization. Annotating a Slice operation with the `marshaled-result` metadata causes additional code to be generated, but only if that operation returns one or more of the mutable types listed earlier. The metadata directive has the following effects:

- For an operation `op` that returns at least one mutable type, the Slice compiler generates a static method named `OpMarshaledResult`. This method takes two parameters: the result value (or result tuple, if the operation returns multiple values), and a `Current`. The method marshals the results immediately, and the servant must supply the `Current` in order for the results to be marshaled correctly. Your servant must return the result of this method as its return value.
- A servant method can still optionally return its results using the regular mapping instead, as if the `marshaled-result` metadata was not present. Use caution to ensure no unexpected behavior can occur.

The metadata directive has no effect on the proxy mapping, nor does it generate a `MarshaledResult` method for Slice operations that return `void` or return only immutable values.

You can also annotate an interface with the `marshaled-result` metadata and it will be applied to all of the interface's operations.

After applying the metadata, we can now implement the `Grid` servant as follows:

Python

```
class GridI(Grid):
    def __init__(self):
        self._lock = threading.Lock()

    def getGrid(self, current):
        with self._lock:
            return Grid.GetGridMarshaledResult(self._grid, current) # _grid is marshaled immediately

    def setValue(self, x, y, val, current):
        with self._lock:
            self._grid[x][y] = val # this is safe
```

Back to Top ^

See Also

- Server-Side Python Mapping for Interfaces
- Python Mapping for Operations
- Raising Exceptions in Python
- The Current Object

Previous

Next