Parameter Passing in C-Sharp

Previous

Next

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

The parameter-passing rules change somewhat when using the asynchronous mapping.

On this page:

Server-Side Mapping for Parameters in C#

The servant mapping for operations is consistent with the proxy mapping. To illustrate the rules for the C# mapping, consider the following interface:

Slice

```
module M
{
    interface Example
    {
        string op(string sin, out string sout);
    }
}
```

The generated method for op looks as follows:

C#

```
public interface ExampleOperations_
{
    string op(string sin, out string sout, Ice.Current current = null);
}
```

As you can see, there are no surprises here. For example, we could implement op as follows:

C#

```
using System;

public class ExampleI : ExampleDisp_
{
    public override string op(string sin, out string sout, Ice.Current current = null)
    {
        Console.WriteLine(sin);      // In params are initialized
        sout = "Hello World!";       // Assign out param
        return "Done";
    }
}
```

This code is in no way different from what you would normally write if you were to pass strings to and from a method; the fact that remote procedure calls are involved does not affect 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 C# rules and do not require special-purpose API calls.

Back to Top ^

Thread-Safe Marshaling in C#

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

The potential for corruption occurs whenever a servant returns data by reference, yet continues to hold a reference to that data. For example, consider the following servant implementation:

C#

```
public class GridI : GridDisp_
{
    GridI()
    {
        _grid = // ...
    }

    public override int[][] getGrid(Current current = null)
    {
        return _grid;
    }

    public override void setValue(int x, int y, int val, Current current = null)
    {
        _grid[x][y] = val;
    }

    private int[][] _grid;
}
```

Suppose that a client invoked the `getGrid` operation. While the Ice run time marshals the returned array in preparation to send a reply message, it is possible for another thread to dispatch the `setValue` operation on the same servant. This race condition can result in several unexpected outcomes, including a failure during marshaling or inconsistent data in the reply to `getGrid`. Synchronizing the `getGrid` and `setValue` operations 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:

C#

```
public class GridI : GridDisp_
{
    public override int[][] getGrid(Current current = null)
    {
        lock(this)
        {
            return _grid;
        }
    }

    public override void setValue(int x, int y, int val, Current current = null)
    {
        lock(this)
        {
            int[][] newGrid = // shallow copy...
            newGrid[x][y] = val;
            _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 directive changes the signature of the corresponding servant method, but only if that operation returns mutable types. The metadata directive has the following effects:

- For an operation op from an interface `Intf` that returns one or multiple values and at least one of those values has a mutable type, the Slice compiler generates an `Intf_OpMarshaledResult` class and the return type of the servant method becomes `OpMarshaledResult`.
- The constructor for `Intf_OpMarshaledResult` takes an extra argument of type `Current`. The servant must supply the `Current` in order for the results to be marshaled correctly.

The metadata directive also affects the asynchronous mapping but has no effect on the proxy mapping, nor does it affect the servant mapping of 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:

C#

```
public class GridI : GridDisp_
{
   public override Grid_GetGridMarshaledResult getGrid(Current current)
   {
        lock(this)
        {
            return new Grid_GetGridMarshaledResult(_grid, current); // _grid is marshaled immediately
        }
    }

    public override void setValue(int x, int y, int val, Current current)
    {
        lock(this)
        {
            _grid[x][y] = val; // this is safe
        }
    }

    ...
}
```

Here are more examples to demonstrate the mapping:

Slice

```
class C { ... }
struct S { ... }
sequence<string> Seq;

interface Example
{
    C getC();
    ["marshaled-result"]
    C getC2();

    void getS(out S val);

    ["marshaled-result"]
    void getS2(out S val);

    string getValues(string name, out Seq val);

    ["marshaled-result"]
    string getValues2(string name, out Seq val);

    ["amd", "marshaled-result"]
    string getValuesAMD(string name, out Seq val);
}
```

Review the generated code below to see the changes that the presence of the metadata causes in the servant method signatures:

C#

```
public struct Example_GetC2MarshaledResult : Ice.MarshaledResult
{
    public Example_GetC2MarshaledResult(C returnValue, Current current);
    ...
}

public struct Example_GetS2MarshaledResult : Ice.MarshaledResult
{
    public Example_GetS2MarshaledResult(S returnValue, Current current);
    ...
}

public struct Example_GetValues2MarshaledResult : Ice.MarshaledResult
{
    public Example_GetValues2MarshaledResult(string returnValue, string[] val, Current current);
    ...
}

public struct Example_GetValuesAMDResult
{
    public Example_GetValuesAMDResult(string returnValue, string[] val);
    ...
}

public struct Example_GetValuesAMDMarshaledResult : Ice.MarshaledResult
{
    public Example_GetValuesAMDMarshaledResult(string ret, string[] val, Ice.Current current);
    ...
}

public interface ExampleOperations_
{
    C getC(Ice.Current current = null);
    Example_GetC2MarshaledResult getC2(Ice.Current current = null);
    S getS(Ice.Current current = null);
    Example_GetS2MarshaledResult getS2(Ice.Current current = null);
    string getValues(string name, out string[] val, Ice.Current current = null);
    Example_GetValues2MarshaledResult getValues2(string name, Ice.Current current = null);
    System.Threading.Tasks.Task<M.Example_GetValuesAMDMarshaledResult> getValuesAMDAsync(string name, Ice.
Current current = null);
}
```

Back to Top ^

See Also

- Server-Side C-Sharp Mapping for Interfaces
- Raising Exceptions in C-Sharp
- Tie Classes in C-Sharp
- The Current Object

Previous

Next