Dynamic Invocation and Dispatch in C-Sharp

This page describes the C# mapping for the `ice_invoke` proxy function and the `Blobject` class.

On this page:

`ice_invoke` in C#

The mapping for `ice_invoke` is shown below:

C#

```
namespace Ice
{
    public interface ObjectPrx
    {
        bool ice_invoke(string operation,
                        OperationMode mode,
                        byte[] inParams,
                        out byte[] outParams);
        // ...
    }
}
```

Another overloading of `ice_invoke` (not shown) adds a trailing argument of type `Ice.Context`.

As an example, the code below demonstrates how to invoke the operation `op`, which takes no `in` parameters:

C#

```
Ice.ObjectPrx proxy = ...
try {
    byte[] outParams;
    if (proxy.ice_invoke("op", Ice.OperationMode.Normal, null, outParams)) {
        // Handle success
    } else {
        // Handle user exception
    }
} catch (Ice.LocalException ex) {
    // Handle exception
}
```

As a convenience, the Ice run time accepts a null or empty byte sequence when there are no input parameters and internally translates it into an empty encapsulation. In all other cases, the value for `inParams` must be an encapsulation of the encoded parameters.

Using Streams with `ice_invoke` in C#

The streaming interfaces provide the tools an application needs to dynamically invoke operations with arguments of any Slice type. Consider the following Slice definition:

Slice

```
module Calc {
    exception Overflow {
        int x;
        int y;
    };
    interface Compute {
        idempotent int add(int x, int y)
            throws Overflow;
    };
};
```

Now let's write a client that dynamically invokes the `add` operation:

C#

```
Ice.ObjectPrx proxy = ...
try {
    Ice.OutputStream outStream = Ice.Util.createOutputStream(communicator);
    outStream.startEncapsulation();
    outStream.writeInt(100); // x
    outStream.writeInt(-1);  // y
    outStream.endEncapsulation();
    byte[] inParams = outStream.finished();

    byte[] outParams;
    if (proxy.ice_invoke("add", Ice.OperationMode.Idempotent, inParams, out outParams)) {
        // Handle success
        Ice.InputStream inStream = Ice.Util.createInputStream(communicator, outParams);
        inStream.startEncapsulation();
        int result = inStream.readInt();
        inStream.endEncapsulation();
        System.Diagnostics.Debug.Assert(result == 99);
    } else {
        // Handle user exception
    }
} catch (Ice.LocalException ex) {
    // Handle exception
}
```

You can see here that the input and output parameters are enclosed in encapsulations.

We neglected to handle the case of a user exception in this example, so let's implement that now. We assume that we have compiled our program with the Slice-generated code, therefore we can call `throwException` on the input stream and catch `Overflow` directly:

C#

```
    if (proxy.ice_invoke("add", Ice.OperationMode.Idempotent, inParams, out outParams)) {
        // Handle success
        ...
    } else {
        // Handle user exception
        Ice.InputStream inStream = Ice.Util.createInputStream(communicator, outParams);
        try {
            inStream.startEncapsulation();
            inStream.throwException();
        } catch (Calc.Overflow ex) {
            System.Console.WriteLine("overflow while adding " +
                                     ex.x + " and " + ex.y);
        } catch (Ice.UserException) {
            // Handle unexpected user exception
        }
    }
```

This is obviously a contrived example: if the Slice-generated code is available, why bother using dynamic dispatch? In the absence of Slice-generated code, the caller would need to manually unmarshal the user exception, which is outside the scope of this book.

As a defensive measure, the code traps `Ice.UserException`. This could be raised if the Slice definition of `add` is modified to include another user exception but this segment of code did not get updated accordingly.

Subclassing `Blobject` in C#

Implementing the dynamic dispatch model requires writing a subclass of `Ice.Blobject`. We continue using the `Compute` interface to demonstrate a `Blobject` implementation:

C#

```
public class ComputeI : Ice.Blobject {
    public bool ice_invoke(byte[] inParams, out byte[] outParams, Ice.Current current);
    {
        ...
    }
}
```

An instance of `ComputeI` is an Ice object because `Blobject` derives from `Object`, therefore an instance can be added to an object adapter like any other servant.

For the purposes of this discussion, the implementation of `ice_invoke` handles only the `add` operation and raises `OperationNotExistExceptior` for all other operations. In a real implementation, the servant must also be prepared to receive invocations of the following `Object` operations:

- `string ice_id()`
  Returns the Slice type ID of the servant's most-derived type.
- `StringSeq ice_ids()`
  Returns a sequence of strings representing all of the Slice interfaces supported by the servant, including `"::Ice::Object"`.
- `bool ice_isA(string id)`
  Returns `true` if the servant supports the interface denoted by the given Slice type ID, or `false` otherwise. This operation is invoked by the proxy function `checkedCast`.
- `void ice_ping()`
  Verifies that the object denoted by the identity and facet contained in `Ice::Current` is reachable.

With that in mind, here is our simplified version of `ice_invoke`:

C#

```
public bool ice_invoke(byte[] inParams, out byte[] outParams, Ice.Current current);
{
    if (current.operation.Equals("add")) {
        Ice.Communicator communicator = current.adapter.getCommunicator();
        Ice.InputStream inStream = Ice.Util.createInputStream(communicator, inParams);
        inStream.startEncapsulation();
        int x = inStream.readInt();
        int y = inStream.readInt();
        inStream.endEncapsulation();

        Ice.OutputStream outStream = Ice.Util.createOutputStream(communicator);
        try {
            if (checkOverflow(x, y)) {
                Calc.Overflow ex = new Calc.Overflow();
                ex.x = x;
                ex.y = y;
                outStream.startEncapsulation();
                outStream.writeException(ex);
                outStream.endEncapsulation();
                outParams = outStream.finished();
                return false;
            } else {
                outStream.startEncapsulation();
                outStream.writeInt(x + y);
                outStream.endEncapsulation();
                outParams = outStream.finished();
                return true;
            }
        } finally {
            outStream.destroy();
        }
    } else {
        Ice.OperationNotExistException ex = new Ice.OperationNotExistException();
        ex.id = current.id;
        ex.facet = current.facet;
        ex.operation = current.operation;
        throw ex;
    }
}
```

If an overflow is detected, the code "raises" the `Calc::Overflow` user exception by calling `writeException` on the output stream and returning `false`, otherwise the return value is encoded and the function returns `true`.

See Also

- Object Adapters
- Request Contexts
- Streaming Interfaces
- Type IDs
- Object Identity
- Facets and Versioning