Dynamic Invocation and Dispatch in C++

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++

```
bool ice_invoke(
    const std::string& operation,
    Ice::OperationMode mode,
    const std::vector< Ice::Byte >& inParams,
    std::vector< Ice::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 {
    std::vector<Ice::Byte> inParams, outParams;
    if (proxy->ice_invoke("op", Ice::Normal, inParams, outParams)) {
        // Handle success
    } else {
        // Handle user exception
    }
} catch (const Ice::LocalException& ex) {
    // Handle exception
}
```

As a convenience, the Ice run time accepts an 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 {
    std::vector< Ice::Byte > inParams, outParams;

    Ice::OutputStreamPtr out = Ice::createOutputStream(communicator);
    out->startEncapsulation();
    out->writeInt(100); // x
    out->writeInt(-1);  // y
    out->endEncapsulation();
    out->finished(inParams);

    if (proxy->ice_invoke("add", Ice::Idempotent, inParams, outParams)) {
        // Handle success
        Ice::InputStreamPtr in = Ice::createInputStream(communicator, outParams);
        in->startEncapsulation();
        int result = in->readInt();
        in->endEncapsulation();
        assert(result == 99);
    } else {
        // Handle user exception
    }
} catch (const 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::Idempotent, inParams, outParams)) {
        // Handle success
        // ...
    } else {
        // Handle user exception
        Ice::InputStreamPtr in = Ice::createInputStream(communicator, outParams);
        try {
            in->startEncapsulation();
            in->throwException();
        } catch (const Calc::Overflow& ex) {
            cout << "overflow while adding " << ex.x << " and " << ex.y << endl;
        } catch (const Ice::UserException& ex) {
            // 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 manual.

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++

```
class ComputeI : public Ice::Blobject {
public:
    virtual bool ice_invoke(
        const std::vector<Ice::Byte>& inParams,
        std::vector<Ice::Byte>& outParams,
        const 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++

```
bool ComputeI::ice_invoke(
    const std::vector<Ice::Byte>& inParams,
    std::vector<Ice::Byte>& outParams,
    const Ice::Current& current)
{
    if (current.operation == "add") {
        Ice::CommunicatorPtr communicator = current.adapter->getCommunicator();
        Ice::InputStreamPtr in = Ice::createInputStream(communicator, inParams);
        in->startEncapsulation();
        int x = in->readInt();
        int y = in->readInt();
        in->endEncapsulation();

        Ice::OutputStreamPtr out = Ice::createOutputStream(communicator);
        if (checkOverflow(x, y)) {
            Calc::Overflow ex;
            ex.x = x;
            ex.y = y;
            out->startEncapsulation();
            out->writeException(ex);
            out->endEncapsulation();
            out->finished(outParams);
            return false;
        } else {
            out->startEncapsulation();
            out->writeInt(x + y);
            out->endEncapsulation();
            out->finished(outParams);
            return true;
        }
    } else {
        Ice::OperationNotExistException ex(__FILE__, __LINE__);
        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`.

Using the Array Mapping for `ice_invoke` and `Blobject` in C++

Ice for C++ supports an alternative mapping for `sequence` input parameters that avoids the overhead of extra copying. Since the `ice_invoke` functions treat the encoded input parameters as a value of type `sequence<byte>`, the dynamic invocation and dispatch facility includes interfaces that use the array mapping for the input parameter blob.

Ice provides two overloaded versions of the proxy function `ice_invoke` that use the array mapping. The version that omits the trailing `Ice::Context` argument is shown below:

C++

```
bool ice_invoke(
    const std::string& operation,
    Ice::OperationMode mode,
    const std::pair< const Ice::Byte*, const Ice::Byte* >& in,
    std::vector< Ice::Byte >& out
);
```

A `Blobject` servant uses the array mapping by deriving its implementation class from `Ice::BlobjectArray` and overriding its `ice_invoke` function:

C++

```
class BlobjectArray {
public:
    virtual bool ice_invoke(
        const std::pair<const Ice::Byte*, const Ice::Byte*>& in,
        std::vector<Ice::Byte>& out,
        const Ice::Current& current) = 0;
};
```

See Also

- C++ Mapping for Sequences
- Object Adapters
- Request Contexts
- Streaming Interfaces
- Type IDs
- Object Identity
- Facets and Versioning