Python Mapping for Interfaces

The mapping of Slice interfaces revolves around the idea that, to invoke a remote operation, you call a member function on a local class instance that is a proxy for the remote object. This makes the mapping easy and intuitive to use because making a remote procedure call is no different from making a local procedure call (apart from error semantics).

On this page:

- Proxy Classes in Python
- Ice.ObjectPrx Class in Python
- Casting Proxies in Python
- Using Proxy Methods in Python
- Object Identity and Proxy Comparison in Python

Proxy Classes in Python

On the client side, a Slice interface maps to a Python class with methods that correspond to the operations on that interface. Consider the following simple interface:

Slice

```
interface Simple {
    void op();
};
```

The Python mapping generates the following definition for use by the client:

Python

```
class SimplePrx(Ice.ObjectPrx):
    def op(self, _ctx=None):
        # ...

    def ice_staticId():
        return '::M::Simple'
    ice_staticId = staticmethod(ice_staticId)

    # ...
```

In the client's address space, an instance of `SimplePrx` is the local ambassador for a remote instance of the `Simple` interface in a server and is known as a *proxy instance*. All the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance.

Note that `SimplePrx` inherits from `Ice.ObjectPrx`. This reflects the fact that all Ice interfaces implicitly inherit from `Ice::Object`.

For each operation in the interface, the proxy class has a method of the same name. In the preceding example, we find that the operation `op` has been mapped to the method `op`. Note that `op` accepts an optional trailing parameter `_ctx` representing the operation context. This parameter is a Python dictionary for use by the Ice run time to store information about how to deliver a request. You normally do not need to use it. (We examine the context parameter in detail in Request Contexts. The parameter is also used by IceStorm.)

Proxy instances are always created on behalf of the client by the Ice run time, so client code never has any need to instantiate a proxy directly.

A value of `None` denotes the null proxy. The null proxy is a dedicated value that indicates that a proxy points "nowhere" (denotes no object).

Another method defined by every proxy class is `ice_staticId`, which returns the type ID string corresponding to the interface. As an example, for the Slice interface `Simple` in module `M`, the string returned by `ice_staticId` is `"::M::Simple"`.

`Ice.ObjectPrx` Class in Python

Copyright 2017, ZeroC, Inc.

All Ice objects have `Object` as the ultimate ancestor type, so all proxies inherit from `Ice.ObjectPrx`. `ObjectPrx` provides a number of methods:

Python

```
class ObjectPrx(object):
    def equals(self, other):
    def ice_getIdentity(self):
    def ice_isA(self, id):
    def ice_ids(self):
    def ice_id(self):
    def ice_ping(self):
    # ...
```

The methods behave as follows:

- **`equals`**
  This method compares two proxies for equality. Note that all aspects of proxies are compared by this method, such as the communication endpoints for the proxy. This means that, in general, if two proxies compare unequal, that does *not* imply that they denote different objects. For example, if two proxies denote the same Ice object via different transport endpoints, `equals` returns `false` even though the proxies denote the same object.

- **`ice_getIdentity`**
  This method returns the identity of the object denoted by the proxy. The identity of an Ice object has the following Slice type:

  Slice

  ```
  module Ice {
      struct Identity {
          string name;
          string category;
      };
  };
  ```

  To see whether two proxies denote the same object, first obtain the identity for each object and then compare the identities:

  Python

  ```
  proxy1 = ...
  proxy2 = ...
  id1 = proxy1.ice_getIdentity()
  id2 = proxy2.ice_getIdentity()

  if id1 == id2:
      # proxy1 and proxy2 denote the same object
  else:
      # proxy1 and proxy2 denote different objects
  ```

- **`ice_isA`**
  The `ice_isA` method determines whether the object denoted by the proxy supports a specific interface. The argument to `ice_isA` is a type ID. For example, to see whether a proxy of type `ObjectPrx` denotes a `Printer` object, we can write:

Copyright 2017, ZeroC, Inc.

Python

```
proxy = ...
if proxy != None and proxy.ice_isA("::Printer"):
    # proxy denotes a Printer object
else:
    # proxy denotes some other type of object
```

Note that we are testing whether the proxy is `None` before attempting to invoke the `ice_isA` method. This avoids getting a run-time error if the proxy is `None`.

- **`ice_ids`**
  The `ice_ids` method returns an array of strings representing all of the type IDs that the object denoted by the proxy supports.
- **`ice_id`**
  The `ice_id` method returns the type ID of the object denoted by the proxy. Note that the type returned is the type of the actual object, which may be more derived than the static type of the proxy. For example, if we have a proxy of type `BasePrx`, with a static type ID of `::Base`, the return value of `ice_id` might be `::Base`, or it might something more derived, such as `::Derived`.
- **`ice_ping`**
  The `ice_ping` method provides a basic reachability test for the object. If the object can physically be contacted (that is, the object exists and its server is running and reachable), the call completes normally; otherwise, it throws an exception that indicates why the object could not be reached, such as `ObjectNotExistException` or `ConnectTimeoutException`.

The `ice_isA`, `ice_ids`, `ice_id`, and `ice_ping` methods are remote operations and therefore support an optional trailing parameter representing a request context. Also note that there are other methods in `ObjectPrx`, not shown here. These methods provide different ways to dispatch a call and also provide access to an object's facets.

Casting Proxies in Python

The Python mapping for a proxy also generates two static methods:

Python

```
class SimplePrx(Ice.ObjectPrx):
    # ...

    def checkedCast(proxy, facet=''):
        # ...
    checkedCast = staticmethod(checkedCast)

    def uncheckedCast(proxy, facet=''):
        # ...
    uncheckedCast = staticmethod(uncheckedCast)
```

The method names `checkedCast` and `uncheckedCast` are reserved for use in proxies. If a Slice interface defines an operation with either of those names, the mapping escapes the name in the generated proxy by prepending an underscore. For example, an interface that defines an operation named `checkedCast` is mapped to a proxy with a method named `_checkedCast`.

For `checkedCast`, if the passed proxy is for an object of type `Simple`, or a proxy for an object with a type derived from `Simple`, the cast returns a reference to a proxy of type `SimplePrx`; otherwise, if the passed proxy denotes an object of a different type (or if the passed proxy is `None`), the cast returns `None`.

Given a proxy of any type, you can use a `checkedCast` to determine whether the corresponding object supports a given type, for example:

Copyright 2017, ZeroC, Inc.

Python

```
obj = ...       # Get a proxy from somewhere...

simple = SimplePrx.checkedCast(obj)
if simple != None:
    # Object supports the Simple interface...
else:
    # Object is not of type Simple...
```

Note that a `checkedCast` contacts the server. This is necessary because only the implementation of an object in the server has definite knowledge of the type of an object. As a result, a `checkedCast` may throw a `ConnectTimeoutException` or an `ObjectNotExistException`.

In contrast, an `uncheckedCast` does not contact the server and unconditionally returns a proxy of the requested type. However, if you do use an `uncheckedCast`, you must be certain that the proxy really does support the type you are casting to; otherwise, if you get it wrong, you will most likely get a run-time exception when you invoke an operation on the proxy. The most likely error for such a type mismatch is `OperationNotExistException`. However, other exceptions, such as a marshaling exception are possible as well. And, if the object happens to have an operation with the correct name, but different parameter types, no exception may be reported at all and you simply end up sending the invocation to an object of the wrong type; that object may do rather nonsensical things. To illustrate this, consider the following two interfaces:

Slice

```
interface Process {
    void launch(int stackSize, int dataSize);
};

// ...

interface Rocket {
    void launch(float xCoord, float yCoord);
};
```

Suppose you expect to receive a proxy for a `Process` object and use an `uncheckedCast` to down-cast the proxy:

Python

```
obj = ...                               # Get proxy...
process = ProcessPrx.uncheckedCast(obj) # No worries...
process.launch(40, 60)                  # Oops...
```

If the proxy you received actually denotes a `Rocket` object, the error will go undetected by the Ice run time: because `int` and `float` have the same size and because the Ice protocol does not tag data with its type on the wire, the implementation of `Rocket::launch` will simply misinterpret the passed integers as floating-point numbers.

In fairness, this example is somewhat contrived. For such a mistake to go unnoticed at run time, both objects must have an operation with the same name and, in addition, the run-time arguments passed to the operation must have a total marshaled size that matches the number of bytes that are expected by the unmarshaling code on the server side. In practice, this is extremely rare and an incorrect `uncheckedCast` typically results in a run-time exception.

Using Proxy Methods in Python

The base proxy class `ObjectPrx` supports a variety of methods for customizing a proxy. Since proxies are immutable, each of these "factory methods" returns a copy of the original proxy that contains the desired modification. For example, you can obtain a proxy configured with a ten second timeout as shown below:

Copyright 2017, ZeroC, Inc.

Python

```
proxy = communicator.stringToProxy(...)
proxy = proxy.ice_timeout(10000)
```

A factory method returns a new proxy object if the requested modification differs from the current proxy, otherwise it returns the current proxy. With few exceptions, factory methods return a proxy of the same type as the current proxy, therefore it is generally not necessary to repeat a down-cast after using a factory method. The example below demonstrates these semantics:

Python

```
base = communicator.stringToProxy(...)
hello = Demo.HelloPrx.checkedCast(base)
hello = hello.ice_timeout(10000) # Type is preserved
hello.sayHello()
```

The only exceptions are the factory methods `ice_facet` and `ice_identity`. Calls to either of these methods may produce a proxy for an object of an unrelated type, therefore they return a base proxy that you must subsequently down-cast to an appropriate type.

Object Identity and Proxy Comparison in Python

Proxy objects support comparison using the built-in relational operators as well as the `cmp` function. Note that proxy comparison uses *all* of the information in a proxy for the comparison. This means that not only the object identity must match for a comparison to succeed, but other details inside the proxy, such as the protocol and endpoint information, must be the same. In other words, comparison tests for *proxy* identity, *not* object identity. A common mistake is to write code along the following lines:

Python

```
p1 = ...        # Get a proxy...
p2 = ...        # Get another proxy...

if p1 != p2:
    # p1 and p2 denote different objects       # WRONG!
else:
    # p1 and p2 denote the same object         # Correct
```

Even though `p1` and `p2` differ, they may denote the same Ice object. This can happen because, for example, both `p1` and `p2` embed the same object identity, but each uses a different protocol to contact the target object. Similarly, the protocols may be the same, but denote different endpoints (because a single Ice object can be contacted via several different transport endpoints). In other words, if two proxies compare equal, we know that the two proxies denote the same object (because they are identical in all respects); however, if two proxies compare unequal, we know absolutely nothing: the proxies may or may not denote the same object.

To compare the object identities of two proxies, you can use helper functions in the `Ice` module:

Python

```
def proxyIdentityCompare(lhs, rhs)
def proxyIdentityAndFacetCompare(lhs, rhs)
```

`proxyIdentityCompare` allows you to correctly compare proxies for identity:

Copyright 2017, ZeroC, Inc.

Python

```
p1 = ...        # Get a proxy...
p2 = ...        # Get another proxy...

if Ice.proxyIdentityCompare(p1, p2) != 0:
    # p1 and p2 denote different objects       # Correct
else:
    # p1 and p2 denote the same object         # Correct
```

The function returns 0 if the identities are equal, `1` if `p1` is less than `p2`, and `1` if `p1` is greater than `p2`. (The comparison uses `name` as the major sort key and `category` as the minor sort key.)

The `proxyIdentityAndFacetCompare` function behaves similarly, but compares both the identity and the facet name.

See Also

- Interfaces, Operations, and Exceptions
- Proxies for Ice Objects
- Python Mapping for Operations
- Operations on Object
- Proxy Methods
- Versioning
- IceStorm

Copyright 2017, ZeroC, Inc.