Obtaining Proxies

This page describes the ways an application can obtain a proxy.

On this page:

Obtaining a Proxy from a String

Slice

```
module Ice
{
    local interface Communicator
    {
        Object* stringToProxy(string str);
        // ...
    }
}
```

The communicator operation `stringToProxy` creates a proxy from its stringified representation, as shown in the following C++ example:

C++11

```
auto p = communicator->stringToProxy("ident:tcp -p 5000"); // p is a std::shared_ptr<Ice::ObjectPrx>
```

C++98

```
Ice::ObjectPrx p = communicator->stringToProxy("ident:tcp -p 5000");
```

Obtaining a Proxy from Properties

Slice

```
module Ice
{
    local interface Communicator
    {
        Object* propertyToProxy(string property);
        // ...
    }
}
```

Rather than hard-coding a stringified proxy as the previous example demonstrated, an application can gain more flexibility by externalizing the proxy in a configuration property. For example, we can define a property that contains our stringified proxy as follows:

```
MyApp.Proxy=ident:tcp -p 5000
```

We can use the communicator operation `propertyToProxy` to convert the property's value into a proxy. A null proxy is returned if no property is found with the specified name. For example in Java:

Java

```
com.zeroc.Ice.ObjectPrx p = communicator.propertyToProxy("MyApp.Proxy");
```

Java Compat

```
Ice.ObjectPrx p = communicator.propertyToProxy("MyApp.Proxy");
```

As an added convenience, `propertyToProxy` allows you to define subordinate properties that configure the proxy's local settings. The properties below demonstrate this feature:

```
MyApp.Proxy=ident:tcp -p 5000
MyApp.Proxy.PreferSecure=1
MyApp.Proxy.EndpointSelection=Ordered
```

These additional properties simplify the task of customizing a proxy (as you can with proxy methods) without the need to change the application's code. The properties shown above are equivalent to the following statements:

Java

```
com.zeroc.Ice.ObjectPrx p = communicator.stringToProxy("ident:tcp -p 5000");
p = p.ice_preferSecure(true);
p = p.ice_endpointSelection(com.zeroc.Ice.EndpointSelectionType.Ordered);
```

Java Compat

```
Ice.ObjectPrx p = communicator.stringToProxy("ident:tcp -p 5000");
p = p.ice_preferSecure(true);
p = p.ice_endpointSelection(Ice.EndpointSelectionType.Ordered);
```

The list of supported proxy properties includes the most commonly-used proxy settings. The communicator prints a warning by default if it does not recognize a subordinate property. You can disable this warning using the property `Ice.Warn.UnknownProperties`.

Note that proxy properties can themselves have proxy properties. For example, the following sets the `PreferSecure` property on the default locator's router:

```
Ice.Default.Locator.Router.PreferSecure=1
```

Obtaining a Proxy using Factory Methods

Proxy factory methods allow you to modify aspects of an existing proxy. Since proxies are immutable, factory methods always return a new proxy if the desired modification differs from the proxy's current configuration. Consider the following C# example:

C#

```
Ice.ObjectPrx p = communicator.stringToProxy("...");
p = p.ice_oneway();
```

`ice_oneway` is considered a factory method because it returns a proxy configured to use oneway invocations. If the original proxy uses a different invocation mode, the return value of `ice_oneway` is a new proxy object.

The `checkedCast` and `uncheckedCast` methods can also be considered factory methods because they return new proxies that are narrowed to a particular Slice interface. A call to `checkedCast` or `uncheckedCast` typically follows the use of other factory methods, as shown below:

C#

```
Ice.ObjectPrx p = communicator.stringToProxy("...");
Ice.LocatorPrx loc = Ice.LocatorPrxHelper.checkedCast(p.ice_secure(true));
```

Note however that, once a proxy has been narrowed to a Slice interface, it is not normally necessary to perform another down-cast after using a factory method. For example, we can rewrite this example as follows:

C#

```
Ice.ObjectPrx p = communicator.stringToProxy("...");
Ice.LocatorPrx loc = Ice.LocatorPrxHelper.checkedCast(p);
loc = (Ice.LocatorPrx)p.ice_secure(true);
```

A language-specific cast may be necessary, as shown here for C#, because the factory methods are declared to return the type `ObjectPrx`, but the proxy object itself retains its narrowed type. 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.

Obtaining a Proxy by Invoking Operations

An application can also obtain a proxy as the result of an Ice invocation. Consider the following Slice definitions:

Slice

```
interface Account { ... }
interface Bank
{
    Account* findAccount(string id);
}
```

Invoking the `findAccount` operation returns a proxy for an `Account` object. There is no need to use `checkedCast` or `uncheckedCast` on this proxy because it has already been narrowed to the `Account` interface. The C++ code below demonstrates how to invoke `findAccount`:

C++11

```
std::shared_ptr<BankPrx> bank = ...
auto acct = bank->findAccount(id); // acct is a shared_ptr<AccountPrx>
```

C++98

```
BankPrx bank = ...
AccountPrx acct = bank->findAccount(id);
```

Of course, the application must have already obtained a proxy for the bank object using one of the techniques shown above.

See Also

- Communicator
- Proxy and Endpoint Syntax
- Proxy Methods
- Proxy Properties
- Ice.Warn.*