JavaScript 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 Interfaces in JavaScript

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

Slice

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

The Slice compiler generates code for use by the client similar to the following:

JavaScript

JavaScript

```
class SimplePrx extends Ice.ObjectPrx
{
    op(context){ ... }
    static uncheckedCast(prx, facet) { ... }
    static checkedCast(prx, facet, contex) { ... }
}
```

TypeScript

TypeScript

```
abstract class SimplePrx extends Ice.ObjectPrx
{
    op(context?:Map<string, string>):Ice.AsyncResult<void>;
    static uncheckedCast(prx:Ice.ObjectPrx, facet?:string):SimplePrx;
    static checkedCast(prx:Ice.ObjectPrx, facet?:string, contex?:Map<string, string>):Ice.
AsyncResult<SimplePrx>;
}
```

As you can see, the compiler generates a *proxy type* `SimplePrx`. In general, the generated name is `<`*`interface-name`*`>Prx`. If an interface is nested in a module `M`, the fully-qualified name is `M.<`*`interface-name`*`>Prx`.

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 the prototype for `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 type defines a function with the same name. For the preceding example, we find that the operation `op` has been mapped to the function `op`. The function has an optional parameter `context` that is used by the Ice run time to store information about how to deliver a request. You normally do not need to use it. Legal values for this parameter are `null` or an instance of `Ice.Context` (which is a JavaScript Map). (We examine the `context` parameter in detail in Request Contexts. The parameter is also used by IceStorm.)

Asynchronous Operations

Unlike most other Ice language mappings, where a Slice operation generates both synchronous and asynchronous proxy methods, the JavaScript language mapping generates only an asynchronous method. Given the event-driven nature of JavaScript and its networking API, it is not feasible to provide the traditional blocking RPC model.

Client code must not create an instance of a `<`*`interface-name`*`>Prx` type directly. Instead, 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 `null` denotes the null proxy. The null proxy is a dedicated value that indicates that a proxy points "nowhere" (denotes no object).

Interface Inheritance in JavaScript

Inheritance relationships among Slice interfaces are maintained in the generated JavaScript code. For example:

Slice

```
interface A { ... }
interface B { ... }
interface C extends A, B { ... }
```

Given a proxy for `C`, a client can invoke any operation defined for interface `C`, as well as any operation inherited from `C`'s base interfaces.

The `Ice.ObjectPrx` Interface in JavaScript

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

JavaScript

JavaScript

```
class ObjectPrx
{
    equals(r) { ... }
    ice_getIdentity() { ... }
    ice_isA(id, context) { ... }
    ice_ids(context) { ... }
    ice_id(context) { ... }
    ice_ping(context) { ... }
}
```

TypeScript

TypeScript

```
class ObjectPrx
{
    equals(rhs:any):boolean;
    ice_getIdentity():Identity;
    ice_isA(id:string, context?:Map<string, string>):AsyncResult<boolean>;
    ice_ids(context?:Map<string, string>):AsyncResult<string[]>;
    ice_id(context?:Map<string, string>):AsyncResult<string>;
    ice_ping(context?:Map<string, string>):AsyncResult<void>;
}
```

The functions behave as follows:

- **`ObjectPrx.prototype.equals`**
  This function compares two proxies for equality. Note that all aspects of proxies are compared by this function, 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.

- **`ObjectPrx.prototype.ice_getIdentity`**
  This function 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:

  JavaScript

  ```
  const prx1 = ...;
  const prx2 = ...;
  const ident1 = prx1.ice_getIdentity();
  const ident2 = prx2.ice_getIdentity();

  if(i1.equals(i2))
  {
      // o1 and o2 denote the same object
  }
  else
  {
      // o1 and o2 denote different objects

  }
  ```

- **`ObjectPrx.prototype.ice_isA`**
  The `ice_isA` function determines whether the object 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:

JavaScript

```
(async function()
{
    const prx = ...;
    try
    {
        if(await prx.ice_isA("::Printer"))
        {
            // Target object supports the Printer interface!
        }
        else
        {
            // Oops, target object is a different type
        }
    }
    catch(ex)
    {
        // Handle failure
    }
}());
```

- **`ObjectPrx.prototype.ice_ids`**
  The `ice_ids` function obtains an array of strings representing all of the type IDs that the object supports.

- **`ObjectPrx.prototype.ice_id`**
  The `ice_id` function obtains the type ID of the object. 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 actual type ID might be `::Base`, or it might something more derived, such as `::Derived`.

- **`ObjectPrx.prototype.ice_ping`**
  The `ice_ping` function 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`.

As remote operations, the `ice_isA`, `ice_ids`, `ice_id`, and `ice_ping` functions support an optional trailing argument 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.

Down-casting Proxies in JavaScript

In addition to the methods corresponding to Slice operations, the Slice-to-JavaScript compiler also generates two helper methods that support down-casting:

JavaScript

JavaScript

```
class SimplePrx extends Ice.ObjectPrx
{
   static checkedCast(prx, facet, context) { ... }
   static uncheckedCast(prx, facet) { ... }
}
```

TypeScript

TypeScript

```
class SimplePrx extends Ice.ObjectPrx
{
    static uncheckedCast(prx:ObjectPrx, facet?:string):SimplePrx;
    static checkedCast(prx:ObjectPrx, facet?:string, contex?:Map<string, string>):AsyncResult<SimplePrx>;
}
```

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 result of the cast is a non-null 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 null), the result of the cast is a null reference.

Note that a checked cast 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. Consequently, a checked cast is implemented as the asynchronous method `checkedCast`, which may result in a `ConnectTimeoutException` or an `ObjectNotExistException`.

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

JavaScript

```
(async function()
{
    const prx = ...;  // Get a proxy from somewhere...
    try
    {
        const simple = await SimplePrx.checkedCast(prx);
        if(simple !== null)
        {
            // Object supports the Simple interface...
        }
        else
        {
            // Object is not of type Simple...
        }
    }
    catch(ex)
    {
        // Handle failure
    }
}());
```

In contrast, an unchecked cast does not contact the server and unconditionally returns a proxy of the requested type. However, if you do use `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:

JavaScript

```
const obj = ...;                                // Get proxy...
const 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.

Type IDs in JavaScript

You can discover the type ID string corresponding to an interface by calling the `ice_staticId` function on the proxy type:

JavaScript

```
const id = SimplePrx.ice_staticId();
```

As an example, for an interface `Simple` in module `M`, the `ice_staticId` function returns the string `"::M::Simple"`.

Using Proxy Methods in JavaScript

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 invocation timeout as shown below:

JavaScript

```
const proxy = communicator.stringToProxy(...);
proxy = proxy.ice_invocationTimeout(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 checked or unchecked cast after using a factory method. 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 JavaScript

Proxies provide an `equals` function that compares proxies:

JavaScript

```
class Ice.ObjectPrx
{
    equals(other) { ... }
}
```

Note that proxy comparison with `equals` 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 with `equals` tests for *proxy* identity, *not* object identity. A common mistake is to write code along the following lines:

JavaScript

```
const p1 = ...;        // Get a proxy...
const p2 = ...;        // Get another proxy...

if(!p1.equals(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 use 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 with `equals`, we know that the two proxies denote the same object (because they are identical in all respects); however, if two proxies compare unequal with `equals`, 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 a helper function:

JavaScript

```
Ice.proxyIdentityCompare = function(lhs, rhs) { ... }
Ice.proxyIdentityAndFacetCompare = function(lhs, rhs) { ... }
```

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

JavaScript

```
const p1 = ...;        // Get a proxy...
const 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 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
- Type IDs
- JavaScript Mapping for Operations
- Request Contexts
- Operations on Object
- Proxy Methods
- Versioning