Object Existence and Non-Existence

Before we talk about how to create and destroy objects, we need to look at a more basic concept, namely that of object existence. What does it mean for an object to "exist" and, more fundamentally, what do we mean by the term "object"?

As mentioned in Ice Objects, an *Ice object* is a conceptual entity, or abstraction that does not really exist. On the client side, the concrete representation of an Ice object is a proxy and, on the server side, the concrete representation of an Ice object is a servant. Proxies and servants are the concrete programming-language artifacts that represent Ice objects.

Because Ice objects are abstract, conceptual entities, they are invisible to the Ice run time and to the application code — only proxies and servants are real and visible. It follows that, to determine whether an *Ice* object exists, any determination must rely on proxies and servants, because they are the only tangible entities in the system.

On this page:

Object Non-Existence

Here is the definitive statement of what it means for an Ice object to *not* exist:

> *An Ice object does not exist if an invocation on the object raises an `ObjectNotExistException`.*

This may seem self-evident but, on closer examination, is a little more subtle than you might expect. In particular, Ice object existence has meaning *only within the context of a particular invocation*. If that invocation raises `ObjectNotExistException`, the object is known to not exist. Note that this says nothing about whether concurrent or future requests to that object will also raise `ObjectNotExistException` — they may or may not, depending on the semantics that are implemented by the application.

Also note that, because all the Ice run time knows about are servants, an `ObjectNotExistException` really indicates that a *servant* for the request could not be found at the time the request was made. This means that, ultimately, it is the application that attaches the meaning "the Ice object does not exist" to this exception.

In theory, the application can attach any meaning it likes to `ObjectNotExistException` and a server can throw this exception for whatever reason it sees fit; in practice, however, we recommend that you do not do this because it breaks with existing convention and is potentially confusing. You should reserve this exception for its intended meaning and not abuse it for other purposes.

Object Existence

The preceding definition does not say anything about object existence if something other than `ObjectNotExistException` is returned in response to a particular request. So, here is the definitive statement of what it means for an Ice object to exist:

> *An Ice object exists if a twoway invocation on the object either succeeds, raises a user exception, or raises `FacetNotExistException` or `OperationNotExistException`.*

It is self-evident that an Ice object exists if a twoway invocation on it succeeds: obviously, the object received the invocation, processed it, and returned a result. However, note the qualification: this is true only for *twoway* invocations; for oneway and datagram invocations, nothing can be inferred about the existence of the corresponding Ice object by invoking an operation on it: because there is no reply from the server, the client-side Ice run time has no idea whether the request was dispatched successfully in the server or not. This includes user exceptions, `ObjectNotExistException`, `FacetNotExistException`, and `OperationNotExistException` — these exceptions are never raised by oneway and datagram invocations, regardless of the actual state of the target object.

If a twoway invocation raises a user exception, the Ice object obviously exists: the Ice run time never raises user exceptions so, for an invocation to raise a user exception, the invocation was dispatched successfully in the server, and the operation implementation in the servant raised the exception.

If a twoway invocation raises `FacetNotExistException`, we do know that the corresponding Ice object indeed exists: the Ice run time raises `FacetNotExistException` only if it can find the identity of the target object in the Active Servant Map (ASM), but cannot find the facet that was specified by the client.

> Note that, if you use servant locators, for these semantics to hold, your servant locator must correctly raise `FacetNotExistException` in the `locate` operation (instead of returning null or raising `ObjectNotExistException`) if an Ice object exists, but the particular target facet does not exist.

Copyright © 2017, ZeroC, Inc.

As a corollary to the preceding two definitions, we can state:

> *A facet does not exist if a twoway invocation on the object raises `ObjectNotExistException` or `FacetNotExistException`.*
>
> *A facet exists if a twoway invocation on the object either succeeds, or raises `OperationNotExistException`.*

These definitions simply capture the fact that a facet is a "sub-object" of an Ice object: if an invocation raises `ObjectNotExistException`, we know that the facet does not exist either because, for a facet to exist, its Ice object must exist.

If an operation raises `OperationNotExistException`, we know that both the target Ice object and the target facet exist. However, the operation that the client attempted to invoke does not. (This is possible only if you use dynamic invocation or if you have mis-matched Slice definitions for client and server.)

Indeterminate Object State

The preceding definitions clearly state under what circumstances we can conclude that an Ice object (or its facet) does or does not exist. However, the preceding definitions are incomplete because operation invocations can have outcomes other than success or failure with `ObjectNotExistException`, `FacetNotExistException`, or `OperationNotExistException`. For example, a client might receive a `MarshalException`, `UnknownLocalException`, `UnknownException`, or `TimeoutException`. In that case, the client cannot draw any conclusions about whether the Ice object on which it invoked a twoway operation exists or not — the exceptions simply indicate that something went wrong while the invocation was processed. So, to complete our definitions, we can state:

> *If a twoway invocation raises a run-time exception other than `ObjectNotExistException`, `FacetNotExistException`, or `OperationNotExistException`, nothing is known about the existence or non-existence of the Ice object that was the target of the invocation. Furthermore, it is impossible to determine the state of existence of an Ice object with a oneway or datagram invocation.*

Authoritative Object Existence Semantics

The preceding definitions capture the fact that, to make a determination of object existence or non-existence, the client-side Ice run time must be able to contact the server and, moreover, receive a reply from the server:

- If the server can be contacted and returns a successful reply for an invocation, the Ice object exists.
- If the server can be contacted and returns an `ObjectNotExistException` (or `FacetNotExistException`), the Ice object (or facet) does not exist. If the server returns an `OperationNotExistException`, the Ice object (and its facet) exists, but does not provide the requested operation, which indicates a type mismatch due to client and server using out-of-sync Slice definitions or due to incorrect use of dynamic invocation.
- If the server cannot be contacted, does not return a reply (as for oneway and datagram invocations), or if anything at all goes wrong with the process of sending an invocation, processing it in the server, and returning the reply, nothing is known about the state of the Ice object, including its existence or non-existence.

Another way of looking at this is that a decision as to whether an object exists or not is *never* made by the Ice run time and, instead, is *always* made by the server-side *application* code:

- If an invocation completes successfully, the server-side application code was involved because it processed the invocation.
- If an invocation returns `ObjectNotExistException` or `FacetNotExistException`, the server-side application code was also involved:
  - either the Ice run time could not find a servant for the invocation in the ASM, in which case the application code was involved by virtue of not having added a servant to the ASM in the first place, or
  - the Ice run time consulted a servant locator that explicitly returned null or raised `ObjectNotExistException` or `FacetNotExistException`.

This means that `ObjectNotExistException` and `FacetNotExistException` are *authoritative*: when you receive these exceptions, you can always believe what they tell you — the Ice run time never raises these exceptions without consulting your code, either implicitly (via an ASM lookup) or explicitly (by calling a servant locator's `locate` operation).

These semantics are motivated by the need to keep the Ice run time stateless with respect to object existence. For example, it would be nice to have stronger semantics, such as a promise that "once an Ice object has existed and been destroyed, all future requests to that Ice object also raise `ObjectNotExistException`". However, to implement these semantics, the Ice run time would have to remember all object identities that were used in the past, and prevent their reuse for new Ice objects. Of course, this would be inherently non-scalable. In addition, it would prevent applications from controlling object identity; allowing such control for applications is important however, for example, to link the identity of an Ice object to its persistent state in a database.

Note that, if the implementation of an operation calls another operation, dealing with `ObjectNotExistException` may require some care. For example, suppose that the client holds a proxy to an object of type `Service` and invokes an operation `provideService` on it:

Copyright © 2017, ZeroC, Inc.

C++

```
ServicePrx service = ...;

try {
    service?>provideService();
} catch (const ObjectNotExistException&) {
    // Service does not exist.
}
```

Here is the implementation of `provideService` in the server, which makes a call on a helper object to implement the operation:

C++

```
void
ServiceI::provideService(const Ice::Current&)
{
    // ...
    proxyToHelper?>someOp();
    // ...
}
```

If `proxyToHelper` happens to point at an object that was destroyed previously, the call to `someOp` will throw `ObjectNotExistException`. If the implementation of `provideService` does not intercept this exception, the exception will propagate all the way back to the client, who will conclude that the service has been destroyed when, in fact, the service still exists but the helper object used to implement `provideService` no longer exists.

Usually, this scenario is not a serious problem. Most often, the helper object cannot be destroyed while it is needed by `provideService` due to the way the application is structured. In that case, no special action is necessary because `someOp` will never throw `ObjectNotExistException`. On the other hand, if it is possible for the helper object to be destroyed, `provideService` can wrap a try-catch block for `ObjectNotExistException` around the call to `someOp` and throw an appropriate user exception from the exception handler (such as `ResourceUnavailable` or similar).

See Also

- Terminology
- Servant Locators
- Oneway Invocations
- Datagram Invocations
- Facets and Versioning
- Dynamic Ice
- Freeze

Copyright © 2017, ZeroC, Inc.