Object Incarnation in Python

Having created a servant class such as the rudimentary `NodeI` class, you can instantiate the class to create a concrete servant that can receive invocations from a client. However, merely instantiating a servant class is insufficient to incarnate an object. Specifically, to provide an implementation of an Ice object, you must take the following steps:

1. Instantiate a servant class.
2. Create an identity for the Ice object incarnated by the servant.
3. Inform the Ice run time of the existence of the servant.
4. Pass a proxy for the object to a client so the client can reach it.

On this page:

Instantiating a Python Servant

Instantiating a servant means to allocate an instance:

Python

```
servant = NodeI("Fred")
```

This statement creates a new `NodeI` instance and assigns its reference to the variable `servant`.

Creating an Identity in Python

Each Ice object requires an identity. That identity must be unique for all servants using the same object adapter.

The Ice object model assumes that all objects (regardless of their adapter) have a globally unique identity.

An Ice object identity is a structure with the following Slice definition:

Slice

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

The full identity of an object is the combination of both the `name` and `category` fields of the `Identity` structure. For now, we will leave the `categor` field as the empty string and simply use the `name` field. (The `category` field is most often used in conjunction with servant locators.)

To create an identity, we simply assign a key that identifies the servant to the `name` field of the `Identity` structure:

Python

```
id = Ice.Identity()
id.name = "Fred" # Not unique, but good enough for now
```

Note that the mapping for structures allows us to write the following equivalent code:

Python

```
id = Ice.Identity("Fred") # Not unique, but good enough for now
```

Activating a Python Servant

Merely creating a servant instance does nothing: the Ice run time becomes aware of the existence of a servant only once you explicitly tell the object adapter about the servant. To activate a servant, you invoke the `add` operation on the object adapter. Assuming that we have access to the object adapter in the `adapter` variable, we can write:

Python

```
adapter.add(servant, id)
```

Note the two arguments to `add`: the servant and the object identity. Calling `add` on the object adapter adds the servant and the servant's identity to the adapter's servant map and links the proxy for an Ice object to the correct servant instance in the server's memory as follows:

1. The proxy for an Ice object, apart from addressing information, contains the identity of the Ice object. When a client invokes an operation, the object identity is sent with the request to the server.
2. The object adapter receives the request, retrieves the identity, and uses the identity as an index into the servant map.
3. If a servant with that identity is active, the object adapter retrieves the servant from the servant map and dispatches the incoming request into the correct member function on the servant.

Assuming that the object adapter is in the active state, client requests are dispatched to the servant as soon as you call `add`.

UUIDs as Identities in Python

The Ice object model assumes that object identities are globally unique. One way of ensuring that uniqueness is to use UUIDs (Universally Unique Identifiers) as identities. The `Ice.generateUUID` function creates such identities:

Python

```
import Ice
print Ice.generateUUID()
```

When executed, this program prints a unique string such as `5029a22c-e333-4f87-86b1-cd5e0fcce509`. Each call to `generateUUID` creates a string that differs from all previous ones.

You can use a UUID such as this to create object identities. For convenience, the object adapter has an operation `addWithUUID` that generates a UUID and adds a servant to the servant map in a single step. Using this operation, we can create an identity and register a servant with that identity in a single step as follows:

Python

```
adapter.addWithUUID(NodeI("Fred"))
```

Creating Proxies in Python

Once we have activated a servant for an Ice object, the server can process incoming client requests for that object. However, clients can only access the object once they hold a proxy for the object. If a client knows the server's address details and the object identity, it can create a proxy from a string, as we saw in our first example in Hello World Application. However, creation of proxies by the client in this manner is usually only done to allow the client access to initial objects for bootstrapping. Once the client has an initial proxy, it typically obtains further proxies by invoking operations.

The object adapter contains all the details that make up the information in a proxy: the addressing and protocol information, and the object identity. The Ice run time offers a number of ways to create proxies. Once created, you can pass a proxy to the client as the return value or as an out-parameter of an operation invocation.

Proxies and Servant Activation in Python

The `add` and `addWithUUID` servant activation operations on the object adapter return a proxy for the corresponding Ice object. This means we can write:

Python

```
proxy = adapter.addWithUUID(NodeI("Fred"))
nodeProxy = Filesystem.NodePrx.uncheckedCast(proxy)

# Pass nodeProxy to client...
```

Here, `addWithUUID` both activates the servant and returns a proxy for the Ice object incarnated by that servant in a single step.

Note that we need to use an `uncheckedCast` here because `addWithUUID` returns a proxy of type `Ice.ObjectPrx`.

Direct Proxy Creation in Python

The object adapter offers an operation to create a proxy for a given identity:

Slice

```
module Ice {
    local interface ObjectAdapter {
        Object* createProxy(Identity id);
        // ...
    };
};
```

Note that `createProxy` creates a proxy for a given identity whether a servant is activated with that identity or not. In other words, proxies have a life cycle that is quite independent from the life cycle of servants:

Python

```
id = Ice.Identity()
id.name = Ice.generateUUID()
proxy = adapter.createProxy(id)
```

This creates a proxy for an Ice object with the identity returned by `generateUUID`. Obviously, no servant yet exists for that object so, if we return the proxy to a client and the client invokes an operation on the proxy, the client will receive an `ObjectNotExistException`. (We examine these life cycle issues in more detail in Object Life Cycle.)

See Also

- Hello World Application
- Python Mapping for Structures
- Server-Side Python Mapping for Interfaces
- Object Adapter States
- Servant Locators
- Object Life Cycle