Adding an Evictor to the Java File System Server

On this page:

The Server Main Program in Java

The server's `main` method is responsible for creating the evictor and initializing the root directory node. Many of the administrative duties, such as creating and destroying a communicator, are handled by the `Ice.Application` class. Our server `main` program has now become the following:

Java

```
import Filesystem.*;

public class Server extends Ice.Application
{
    public
    Server(String envName)
    {
        _envName = envName;
    }

    public int
    run(String[] args)
    {
        Ice.ObjectFactory factory = new NodeFactory();
        communicator().addObjectFactory(factory, PersistentFile.ice_staticId());
        communicator().addObjectFactory(factory, PersistentDirectory.ice_staticId());

        Ice.ObjectAdapter adapter = communicator().createObjectAdapter("EvictorFilesystem");

        Freeze.Evictor evictor =
            Freeze.Util.createTransactionalEvictor(adapter, _envName, "evictorfs",
                                                   null, null, null, true);
        DirectoryI._evictor = evictor;
        FileI._evictor = evictor;

        adapter.addServantLocator(evictor, "");

        Ice.Identity rootId = new Ice.Identity();
        rootId.name = "RootDir";
        if(!evictor.hasObject(rootId))
        {
            PersistentDirectory root = new DirectoryI();
            root.nodeName = "/";
            root.nodes = new java.util.HashMap<java.lang.String, NodeDesc>();
            evictor.add(root, rootId);
        }

        adapter.activate();

        communicator().waitForShutdown();

        return 0;
    }

    public static void
    main(String[] args)
    {
        Server app = new Server("db");
        int status = app.main("Server", args, "config.server");
        System.exit(status);
    }

    private String _envName;
}
```

Let us examine the changes in detail. First, we define the class `Server` as a subclass of `Ice.Application`, and provide a constructor taking a string argument:

Java

```
    public
    Server(String envName)
    {
        _envName = envName;
    }
```

The string argument represents the name of the database environment, and is saved for later use in `run`.

One of the first tasks `run` performs is installing the Ice object factories for `PersistentFile` and `PersistentDirectory`. Although these classes are not exchanged via Slice operations, they are marshalled and unmarshalled in exactly the same way when saved to and loaded from the database, therefore factories are required. A single instance of `NodeFactory` is installed for both types:

Java

```
        Ice.ObjectFactory factory = new NodeFactory();
        communicator().addObjectFactory(factory, PersistentFile.ice_staticId());
        communicator().addObjectFactory(factory, PersistentDirectory.ice_staticId());
```

After creating the object adapter, the program initializes a transactional evictor by invoking `createTransactionalEvictor`. The third argument to `createTransactionalEvictor` is the name of the database, the fourth is null to indicate that we do not use facets, the fifth is null to indicate that we do not use a servant initializer, the sixth argument (`null`) indicates no indexes are in use, and the `true` argument requests that the database be created if it does not exist. The evictor is then added to the object adapter as a servant locator for the default category:

Java

```
        Freeze.Evictor evictor =
            Freeze.Util.createTransactionalEvictor(adapter, _envName, "evictorfs",
                                                   null, null, null, true);
        DirectoryI._evictor = evictor;
        FileI._evictor = evictor;

        adapter.addServantLocator(evictor, "");
```

Next, the program creates the root directory node if it is not already being managed by the evictor:

Java

```
        Ice.Identity rootId = new Ice.Identity();
        rootId.name = "RootDir";
        if(!evictor.hasObject(rootId))
        {
            PersistentDirectory root = new DirectoryI();
            root.nodeName = "/";
            root.nodes = new java.util.HashMap<String, NodeDesc>();
            evictor.add(root, rootId);
        }
```

Finally, the `main` function instantiates the `Server` class, passing `db` as the name of the database environment:

Java

```
    public static void
    main(String[] args)
    {
        Server app = new Server("db");
        int status = app.main("Server", args, "config.server");
        System.exit(status);
    }
```

The Persistent Servant Class Definitions in Java

The servant classes must also be changed to incorporate the Freeze evictor. The `FileI` class now has a static state member `_evictor`:

Java

```
import Filesystem.*;

public final class FileI extends PersistentFile
{
    public
    FileI()
    {
        _destroyed = false;
    }

    // Slice operations...

    public static Freeze.Evictor _evictor;
    private boolean _destroyed;
}
```

The `DirectoryI` class has undergone a similar transformation:

Java

```
import Filesystem.*;

public final class DirectoryI extends PersistentDirectory
{
    public
    DirectoryI()
    {
        _destroyed = false;
        nodes = new java.util.HashMap<String, NodeDesc>();
    }

    // Slice operations...

    public static Freeze.Evictor _evictor;
    private boolean _destroyed;
}
```

Implementing a Persistent `FileI` in Java

The `FileI` methods are mostly trivial, because the Freeze evictor handles persistence for us.

Java

```
public synchronized String
name(Ice.Current current)
{
    if (_destroyed) {
        throw new Ice.ObjectNotExistException(current.id, current.facet, current.operation);
    }

    return nodeName;
}

public void
destroy(Ice.Current current)
    throws PermissionDenied
{
    synchronized(this) {
        if (_destroyed) {
            throw new Ice.ObjectNotExistException(current.id, current.facet, current.operation);
        }
        _destroyed = true;
    }

    //
    // Because we use a transactional evictor,
    // these updates are guaranteed to be atomic.
    //
    parent.removeNode(nodeName);
    _evictor.remove(current.id);
}

public synchronized String[]
read(Ice.Current current)
{
    if (_destroyed) {
        throw new Ice.ObjectNotExistException(current.id, current.facet, current.operation);
    }

    return (String[])text.clone();
}

public synchronized void
write(String[] text, Ice.Current current)
    throws GenericError
{
    if (_destroyed) {
        throw new Ice.ObjectNotExistException(current.id, current.facet, current.operation);
    }

    this.text = text;
}
```

The code checks that the node has not been destroyed before acting on the invocation by updating or returning state. Note that `destroy` must update two separate nodes: as well as removing itself from the evictor, the node must also update the parent's node map. Because we are using a transactional evictor, the two updates are guaranteed to be atomic, so it is impossible to the leave the file system in an inconsistent state.

Implementing a Persistent `DirectoryI` in Java

The `DirectoryI` implementation requires more substantial changes. We begin our discussion with the `createDirectory` operation:

Java

```
public synchronized DirectoryPrx
createDirectory(String name, Ice.Current current)
    throws NameInUse
{
    if (_destroyed) {
        throw new Ice.ObjectNotExistException(current.id, current.facet, current.operation);
    }

    if (name.length() == 0 || nodes.containsKey(name)) {
        throw new NameInUse(name);
    }

    Ice.Identity id = current.adapter.getCommunicator().stringToIdentity(
        java.util.UUID.randomUUID().toString());
    PersistentDirectory dir = new DirectoryI();
    dir.nodeName = name;
    dir.parent = PersistentDirectoryPrxHelper.uncheckedCast(
        current.adapter.createProxy(current.id));
    DirectoryPrx proxy = DirectoryPrxHelper.uncheckedCast(_evictor.add(dir, id));

    NodeDesc nd = new NodeDesc();
    nd.name = name;
    nd.type = NodeType.DirType;
    nd.proxy = proxy;
    nodes.put(name, nd);

    return proxy;
}
```

After validating the node name, the operation obtains a unique identity for the child directory, instantiates the servant, and registers it with the Freeze evictor. Finally, the operation creates a proxy for the child and adds the child to its node table.

The implementation of the `createFile` operation has the same structure as `createDirectory`:

Java

```
public synchronized FilePrx
createFile(String name, Ice.Current current)
    throws NameInUse
{
    if (_destroyed) {
        throw new Ice.ObjectNotExistException(current.id, current.facet, current.operation);
    }

    if (name.length() == 0 || nodes.containsKey(name)) {
        throw new NameInUse(name);
    }

    Ice.Identity id = current.adapter.getCommunicator().stringToIdentity(
        java.util.UUID.randomUUID().toString());
    PersistentFile file = new FileI();
    file.nodeName = name;
    file.parent = PersistentDirectoryPrxHelper.uncheckedCast(
        current.adapter.createProxy(current.id));
    FilePrx proxy = FilePrxHelper.uncheckedCast(_evictor.add(file, id));

    NodeDesc nd = new NodeDesc();
    nd.name = name;
    nd.type = NodeType.FileType;
    nd.proxy = proxy;
    nodes.put(name, nd);

    return proxy;
}
```

The remaining Slice operations have trivial implementations, so we do not show them here.

Implementing `NodeFactory` in Java

We use a single factory implementation for creating two types of Ice objects: `PersistentFile` and `PersistentDirectory`. These are the only two types that the Freeze evictor will be restoring from its database.

Java

```
package Filesystem;

public class NodeFactory implements Ice.ObjectFactory
{
    public Ice.Object
    create(String type)
    {
        if (type.equals(PersistentFile.ice_staticId()))
            return new FileI();
        else if (type.equals(PersistentDirectory.ice_staticId()))
            return new DirectoryI();
        else {
            assert(false);
            return null;
        }
    }

    public void
    destroy()
    {
    }
}
```

See Also

- The Server-Side main Method in Java
- Java Mapping for Classes
- Transactional Evictor