Adding a Freeze Map to the Java File System Server

Here we present a Java implementation of the file system server.

On this page:

Generating the File System Maps in Java

Now that we have selected our key and value types, we can generate the maps as follows:

```
$ slice2freezej -I$(ICE_HOME)/slice -I. --ice --dict \
    FilesystemDB.IdentityFileEntryMap,Ice::Identity,\
    FilesystemDB::FileEntry \
    IdentityFileEntryMap FilesystemDB.ice \
    $(ICE_HOME)/slice/Ice/Identity.ice
$ slice2freezej -I$(ICE_HOME)/slice -I. --ice --dict \
    FilesystemDB.IdentityDirectoryEntryMap,Ice::Identity,\
    FilesystemDB::DirectoryEntry \
    IdentityDirectoryEntryMap FilesystemDB.ice \
    $(ICE_HOME)/slice/Ice/Identity.ice
```

The resulting map classes are named `IdentityFileEntryMap` and `IdentityDirectoryEntryMap`.

The Server Main Program in Java

The server's main program is very simple:

Java

```
import Filesystem.*;
import FilesystemDB.*;

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

    public int
    run(String[] args)
    {
        Ice.ObjectAdapter adapter = communicator().createObjectAdapter("MapFilesystem");

        Freeze.Connection connection = null;
        try {
            connection = Freeze.Util.createConnection(communicator(), _envName);
            IdentityFileEntryMap fileDB =
                new IdentityFileEntryMap(connection, FileI.filesDB(), true);
            IdentityDirectoryEntryMap dirDB =
                new IdentityDirectoryEntryMap(connection, DirectoryI.directoriesDB(), true);

            adapter.addDefaultServant(new FileI(communicator(), _envName), "file");
            adapter.addDefaultServant(new DirectoryI(communicator(), _envName), "");

            adapter.activate();

            communicator().waitForShutdown();
        } finally {
            connection.close();
        }

        return 0;
    }

    public static void
    main(String[] args)
    {
        Server app = new Server("db");
        app.main("MapServer", args, "config.server");
        System.exit(0);
    }

    private String _envName;
}
```

First, we import the `Filesystem` and `FilesystemDB` packages.

Next, we define the class `FilesystemApp` as a subclass of `Ice.Application`, and provide a constructor taking a string argument:

Java

```
    FilesystemApp(const string& envName)
        : _envName(envName) {}
```

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

The interesting part of `run` are the few lines of code that create the database connection and the two maps that store files and directories, plus the code to add the two default servants:

Java

```
            connection = Freeze.Util.createConnection(communicator(), _envName);
            IdentityFileEntryMap fileDB =
                new IdentityFileEntryMap(connection, FileI.filesDB(), true);
            IdentityDirectoryEntryMap dirDB =
                new IdentityDirectoryEntryMap(connection, DirectoryI.directoriesDB(), true);

            adapter.addDefaultServant(new FileI(communicator(), _envName), "file");
            adapter.addDefaultServant(new DirectoryI(communicator(), _envName), "");
```

`run` keeps the database connection open for the duration of the program for performance reasons. As we will see shortly, individual operation implementations will use their own connections; however, it is substantially cheaper to create second (and subsequent connections) than it is to create the first connection.

For the default servants, we use `file` as the category for files. For directories, we use the empty default category.

Implementing `FileI` with a Freeze Map in Java

The class definition for `FileI` is very simple:

Java

```
public class FileI extends _FileDisp
{
    public
    FileI(Ice.Communicator communicator, String envName)
    {
        _communicator = communicator;
        _envName = envName;
    }

    // Slice operations...

    public static String
    filesDB()
    {
        return "files";
    }

    private void
    halt(Freeze.DatabaseException e)
    {
        java.io.StringWriter sw = new java.io.StringWriter();
        java.io.PrintWriter pw = new java.io.PrintWriter(sw);
        e.printStackTrace(pw);
        pw.flush();
        _communicator.getLogger().error(
            "fatal database error\n" + sw.toString() + "\n*** Halting JVM ***");
        Runtime.getRuntime().halt(1);
    }

    private Ice.Communicator _communicator;
    private String _envName;
}
```

The `FileI` class stores the communicator and the environment name. These members are initialized by the constructor. The `filesDB` static method returns the name of the file map, and the `halt` member function is used to stop the server if it encounters a catastrophic error.

The Slice operations all follow the same implementation strategy: we create a database connection and the file map and place the body of the operation into an infinite loop:

Java

```
public String
someOperation(/* ... */ Ice.Current c)
{
    Freeze.Connection connection = Freeze.Util.createConnection(_communicator, _envName);
    try {
        IdentityFileEntryMap fileDB = new IdentityFileEntryMap(connection, filesDB());

        for (;;) {
            try {

                // Operation implementation here...

            } catch (Freeze.DeadlockException ex) {
                continue;
            } catch (Freeze.DatabaseException ex) {
                halt(ex);
            }
        }
    } finally {
        connection.close();
    }
}
```

Each operation creates its own database connection and map for concurrency reasons: the database takes care of all the necessary locking, so there is no need for any other synchronization in the server. If the database detects a deadlock, the code handles the corresponding `DeadlockException` and simply tries again until the operation eventually succeeds; any other database exception indicates that something has gone seriously wrong and terminates the server.

Here is the implementation of the `name` method:

Java

```
public String
name(Ice.Current c)
{
    Freeze.Connection connection = Freeze.Util.createConnection(_communicator, _envName);
    try
    {
        IdentityFileEntryMap fileDB = new IdentityFileEntryMap(connection, filesDB());

        for (;;) {
            try {
                FileEntry entry = fileDB.get(c.id);
                if (entry == null) {
                    throw new Ice.ObjectNotExistException();
                }
                return entry.name;
            } catch (Freeze.DeadlockException ex) {
                continue;
            } catch (Freeze.DatabaseException ex) {
                halt(ex);
            }
        }
    } finally {
        connection.close();
    }
}
```

The implementation could hardly be simpler: the default servant uses the identity in the `Current` object to index into the file map. If a record with this identity exists, it returns the name of the file as stored in the `FileEntry` structure in the map. Otherwise, if no such entry exists, it throws `ObjectNotExistException`. This happens if the file existed at some time in the past but has since been destroyed.

The `read` implementation is almost identical. It returns the text that is stored by the `FileEntry`:

Java

```
public String[]
read(Ice.Current c)
{
    Freeze.Connection connection =
        Freeze.Util.createConnection(_communicator, _envName);
    try {
        IdentityFileEntryMap fileDB = new IdentityFileEntryMap(connection, filesDB());

        for (;;) {
            try {
                FileEntry entry = fileDB.get(c.id);
                if (entry == null) {
                    throw new Ice.ObjectNotExistException();
                }
                return entry.text;
            } catch (Freeze.DeadlockException ex) {
                continue;
            } catch (Freeze.DatabaseException ex) {
                halt(ex);
            }
        }
    } finally {
        connection.close();
    }
}
```

The `write` implementation updates the file contents and calls `put` on the iterator to update the map with the new contents:

Java

```
public void
write(String[] text, Ice.Current c)
    throws GenericError
{
    Freeze.Connection connection = Freeze.Util.createConnection(_communicator, _envName);
    try {
        IdentityFileEntryMap fileDB = new IdentityFileEntryMap(connection, filesDB());

        for (;;) {
            try {
                FileEntry entry = fileDB.get(c.id);
                if (entry == null) {
                    throw new Ice.ObjectNotExistException();
                }
                entry.text = text;
                fileDB.put(c.id, entry);
                break;
            } catch (Freeze.DeadlockException ex) {
                continue;
            } catch (Freeze.DatabaseException ex) {
                halt(ex);
            }
        }
    } finally {
        connection.close();
    }
}
```

Finally, the `destroy` implementation for files must update two maps: it needs to remove its own entry in the file map as well as update the `nodes` map in the parent to remove itself from the parent's map of children. This raises a potential problem: if one update succeeds but the other one fails, we end up with an inconsistent file system: either the parent still has an entry to a non-existent file, or the parent lacks an entry to a file that still exists.

To make sure that the two updates happen atomically, `destroy` performs them in a transaction:

Java

```
    public void
    destroy(Ice.Current c)
        throws PermissionDenied
    {
        Freeze.Connection connection = Freeze.Util.createConnection(_communicator, _envName);
        try {
            IdentityFileEntryMap fileDB = new IdentityFileEntryMap(connection, filesDB());
            IdentityDirectoryEntryMap dirDB =
                new IdentityDirectoryEntryMap(connection, DirectoryI.directoriesDB());

            for (;;) {
                Freeze.Transaction txn = null;
                try {
                    txn = connection.beginTransaction();

                    FileEntry entry = fileDB.get(c.id);
                    if (entry == null) {
                        throw new Ice.ObjectNotExistException();
                    }

                    DirectoryEntry dirEntry = (DirectoryEntry)dirDB.get(entry.parent);
                    if (dirEntry == null) {
                        halt(new Freeze.DatabaseException(
                                 "consistency error: file without parent"));
                    }

                    dirEntry.nodes.remove(entry.name);
                    dirDB.put(entry.parent, dirEntry);

                    fileDB.remove(c.id);

                    txn.commit();
                    txn = null;
                    break;
                } catch (Freeze.DeadlockException ex) {
                    continue;
                } catch (Freeze.DatabaseException ex) {
                    halt(ex);
                } finally {
                    if (txn != null) {
                        txn.rollback();
                    }
                }
            }
        }
        finally
        {
            connection.close();
        }
    }
```

As you can see, the code first establishes a transaction and then locates the file in the parent directory's map of nodes. After removing the file from the parent, the code updates the parent's persistent state by calling `put` on the parent iterator and then removes the file from the file map before committing the transaction.

Implementing `DirectoryI` with a Freeze Map in Java

The `DirectoryI.directoriesDB` implementation returns the string `directories`, and the `halt` implementation is the same as for `FileI`, so we do not show them here.

Turning to the constructor, we must cater for two different scenarios:

- The server is started with a database that already contains a number of nodes.
- The server is started for the very first time with an empty database.

This means that the root directory (which must always exist) may or may not be present in the database. Accordingly, the constructor looks for the root directory (with the fixed identity `RootDir`); if the root directory does not exist in the database, it creates it:

Java

```
public
DirectoryI(Ice.Communicator communicator, String envName)
{
    _communicator = communicator;
    _envName = envName;

    Freeze.Connection connection = Freeze.Util.createConnection(_communicator, _envName);
    try {
        IdentityDirectoryEntryMap dirDB =
            new IdentityDirectoryEntryMap(connection, directoriesDB());

        for (;;) {
            try {
                Ice.Identity rootId = new Ice.Identity("RootDir", "");
                DirectoryEntry entry = dirDB.get(rootId);
                if (entry == null) {
                    dirDB.put(rootId, new DirectoryEntry("/", new Ice.Identity("", ""), null));
                }
                break;
            } catch (Freeze.DeadlockException ex) {
                continue;
            } catch (Freeze.DatabaseException ex) {
                halt(ex);
            }
        }
    } finally {
        connection.close();
    }
}
```

Next, let us examine the implementation of `createDirectory`. Similar to the `FileI::destroy` operation, `createDirectory` must update both the parent's nodes map and create a new entry in the directory map. These updates must happen atomically, so we perform them in a separate transaction:

Java

```
public DirectoryPrx
createDirectory(String name, Ice.Current c)
    throws NameInUse
{
    Freeze.Connection connection = Freeze.Util.createConnection(_communicator, _envName);
    try {
        IdentityDirectoryEntryMap dirDB =
            new IdentityDirectoryEntryMap(connection, directoriesDB());

        for (;;) {
            Freeze.Transaction txn = null;
            try {
                txn = connection.beginTransaction();

                DirectoryEntry entry = dirDB.get(c.id);
                if (entry == null) {
                    throw new Ice.ObjectNotExistException();
                }
                if(name.length() == 0 || entry.nodes.get(name) != null) {
                    throw new NameInUse(name);
                }

                DirectoryEntry newEntry = new DirectoryEntry(name, c.id, null);
                Ice.Identity id = new Ice.Identity(java.util.UUID.randomUUID().toString(), "");
                DirectoryPrx proxy = DirectoryPrxHelper.uncheckedCast(c.adapter.createProxy(id));

                entry.nodes.put(name, new NodeDesc(name, NodeType.DirType, proxy));
                dirDB.put(c.id, entry);
                dirDB.put(id, newEntry);

                txn.commit();
                txn = null;

                return proxy;
            } catch (Freeze.DeadlockException ex) {
                continue;
            } catch (Freeze.DatabaseException ex) {
                halt(ex);
            } finally {
                if(txn != null) {
                    txn.rollback();
                }
            }
        }
    } finally {
        connection.close();
    }
}
```

After establishing the transaction, the code ensures that the directory does not already contain an entry with the same name and then initializes a new `DirectoryEntry`, setting the name to the name of the new directory, and the parent to its own identity. The identity of the new directory is a UUID, which ensures that all directories have unique identities. In addition, the UUID prevents the accidental rebirth of a file or directory in the future.

The code then initializes a new `NodeDesc` structure with the details of the new directory and, finally, updates its own map of children as well as adding the new directory to the map of directories before committing the transaction.

The `createFile` implementation is almost identical, so we do not show it here. Similarly, the `name` and `destroy` implementations are almost identical to the ones for `FileI`, so let us move to `list`:

Java

```
public NodeDesc[]
list(Ice.Current c)
{
    Freeze.Connection connection =
        Freeze.Util.createConnection(_communicator, _envName);
    try {
        IdentityDirectoryEntryMap dirDB =
            new IdentityDirectoryEntryMap(connection, directoriesDB());

        for (;;) {
            try {
                DirectoryEntry entry = dirDB.get(c.id);
                if (entry == null) {
                    throw new Ice.ObjectNotExistException();
                }
                NodeDesc[] result = new NodeDesc[entry.nodes.size()];
                java.util.Iterator<NodeDesc> p = entry.nodes.values().iterator();
                for (int i = 0; i < entry.nodes.size(); ++i) {
                    result[i] = p.next();
                }
                return result;
            } catch (Freeze.DeadlockException ex) {
                continue;
            } catch (Freeze.DatabaseException ex) {
                halt(ex);
            }
        }
    } finally {
        connection.close();
    }
}
```

Again, the code is very simple: it iterates over the `nodes` map, adding each `NodeDesc` structure to the returned sequence.

The `find` implementation is even simpler, so we do not show it here.

See Also

- Freeze Maps
- The Current Object
- Object Identity and Uniqueness