Adding an Evictor to the C++ File System Server

On this page:

- The Server main Program in C++
- The Persistent Servant Class Definitions in C++
- Implementing a Persistent FileI in C++
- Implementing a Persistent DirectoryI in C++
- Implementing NodeFactory in C++

The Server `main` Program in C++

The server's `main` program 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:

C++

```
#include <PersistentFilesystemI.h>

using namespace std;
using namespace Filesystem;

class FilesystemApp : virtual public Ice::Application
{
public:

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

    virtual int run(int, char*[])
    {
        Ice::ObjectFactoryPtr factory = new NodeFactory;
        communicator()->addObjectFactory(factory, PersistentFile::ice_staticId());
        communicator()->addObjectFactory(factory, PersistentDirectory::ice_staticId());

        Ice::ObjectAdapterPtr adapter = communicator()->createObjectAdapter("EvictorFilesystem");

        Freeze::EvictorPtr evictor =
            Freeze::createTransactionalEvictor(adapter, _envName, "evictorfs");
        FileI::_evictor = evictor;
        DirectoryI::_evictor = evictor;

        adapter->addServantLocator(evictor, "");

        Ice::Identity rootId;
        rootId.name = "RootDir";
        if(!evictor->hasObject(rootId))
        {
            PersistentDirectoryPtr root = new DirectoryI;
            root->nodeName = "/";
            evictor->add(root, rootId);
        }

        adapter->activate();

        communicator()->waitForShutdown();
        if(interrupted())
        {
            cerr << appName() << ": received signal, shutting down" << endl;
        }

        return 0;
    }

private:

    string _envName;
};

int
main(int argc, char* argv[])
{
    FilesystemApp app("db");
    return app.main(argc, argv, "config.server");
}
```

Let us examine the changes in detail. First, we are now including `PersistentFilesystemI.h`. This header file includes all of the other Freeze (and Ice) header files this source file requires.

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

C++

```
    FilesystemApp(const 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:

C++

```
        Ice::ObjectFactoryPtr 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 file, which by default is created if it does not exist. The new evictor is then added to the object adapter as a servant locator for the default category:

C++

```
        NodeI::_evictor = Freeze::createTransactionalEvictor(adapter, _envName, "evictorfs");
        adapter->addServantLocator(NodeI::_evictor, "");
```

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

C++

```
        Ice::Identity rootId;
        rootId.name = "RootDir";
        if(!evictor->hasObject(rootId))
        {
            PersistentDirectoryPtr root = new DirectoryI;
            root->nodeName = "/";
            evictor->add(root, rootId);
        }
```

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

C++

```
int
main(int argc, char* argv[])
{
    FilesystemApp app("db");
    return app.main(argc, argv, "config.server");
}
```

The Persistent Servant Class Definitions in C++

The servant classes must also be changed to incorporate the Freeze evictor. We no longer derive the servants from a common base class. Instead, `FileI` and `DirectoryI` each have their own `_destroyed` and `_mutex` members, as well as a static `_evictor` smart pointer that points at the transactional evictor:

C++

```
#include <PersistentFilesystem.h>
#include <IceUtil/IceUtil.h>
#include <Freeze/Freeze.h>

namespace Filesystem {

class FileI : virtual public PersistentFile {
public:

    FileI();

    // Slice operations...

    static Freeze::EvictorPtr _evictor;

private:

    bool _destroyed;
    IceUtil::Mutex _mutex;
};

class DirectoryI : virtual public PersistentDirectory {
public:

    DirectoryI();

    // Slice operations...

    virtual void removeNode(const std::string&, const Ice::Current&);

    static Freeze::EvictorPtr _evictor;

public:
    bool _destroyed;
    IceUtil::Mutex _mutex;
};
```

In addition to the node implementation classes, we have also declared an object factory:

C++

```
namespace Filesystem {
    class NodeFactory : virtual public Ice::ObjectFactory {
    public:
        virtual Ice::ObjectPtr create(const std::string&);
        virtual void destroy();
    };
```

Implementing a Persistent `FileI` in C++

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

C++

```
Filesystem::FileI::FileI() : _destroyed(false)
{
}

string
Filesystem::FileI::name(const Ice::Current& c)
{
    IceUtil::Mutex::Lock lock(_mutex);

    if (_destroyed) {
        throw Ice::ObjectNotExistException(__FILE__, __LINE__, c.id, c.facet, c.operation);
    }

    return nodeName;
}

void
Filesystem::FileI::destroy(const Ice::Current& c)
{
    {
        IceUtil::Mutex::Lock lock(_mutex);

        if (_destroyed) {
            throw Ice::ObjectNotExistException(__FILE__, __LINE__, c.id, c.facet, c.operation);
        }
        _destroyed = true;
    }

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

Filesystem::Lines
Filesystem::FileI::read(const Ice::Current& c)
{
    IceUtil::Mutex::Lock lock(_mutex);

    if (_destroyed) {
        throw Ice::ObjectNotExistException(__FILE__, __LINE__, c.id, c.facet, c.operation);
    }

    return text;
}

void
Filesystem::FileI::write(const Filesystem::Lines& text,
                         const Ice::Current& c)
{
    IceUtil::Mutex::Lock lock(_mutex);

    if (_destroyed) {
        throw Ice::ObjectNotExistException(__FILE__, __LINE__, c.id, c.facet, c.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 C++

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

C++

```
Filesystem::DirectoryPrx
Filesystem::DirectoryI::createDirectory(const string& name, const Ice::Current& c)
{
    IceUtil::Mutex::Lock lock(_mutex);

    if (_destroyed) {
        throw Ice::ObjectNotExistException(__FILE__, __LINE__, c.id, c.facet, c.operation);
    }

    if (name.empty() || nodes.find(name) != nodes.end()) {
        throw NameInUse(name);
    }

    Ice::Identity id;
    id.name = IceUtil::generateUUID();
    PersistentDirectoryPtr dir = new DirectoryI;
    dir?>nodeName = name;
    dir?>parent = PersistentDirectoryPrx::uncheckedCast(c.adapter?>createProxy(c.id));
    DirectoryPrx proxy = DirectoryPrx::uncheckedCast(_evictor?>add(dir, id));

    NodeDesc nd;
    nd.name = name;
    nd.type = DirType;
    nd.proxy = proxy;
    nodes[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`:

C++

```
Filesystem::FilePrx
Filesystem::DirectoryI::createFile(const string& name,
                                   const Ice::Current& c)
{
    IceUtil::Mutex::Lock lock(_mutex);

    if (_destroyed) {
        throw Ice::ObjectNotExistException(__FILE__, __LINE__, c.id, c.facet, c.operation);
    }

    if (name.empty() || nodes.find(name) != nodes.end()) {
        throw NameInUse(name);
    }

    Ice::Identity id;
    id.name = IceUtil::generateUUID();
    PersistentFilePtr file = new FileI;
    file?>nodeName = name;
    file?>parent = PersistentDirectoryPrx::uncheckedCast(c.adapter?>createProxy(c.id));
    FilePrx proxy = FilePrx::uncheckedCast(_evictor?>add(file, id));

    NodeDesc nd;
    nd.name = name;
    nd.type = FileType;
    nd.proxy = proxy;
    nodes[name] = nd;

    return proxy;
}
```

Implementing `NodeFactory` in C++

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.

C++

```
Ice::ObjectPtr
Filesystem::NodeFactory::create(const string& type)
{
    if (type == PersistentFile::ice_staticId())
        return new FileI;
    else if (type == PersistentDirectory::ice_staticId())
        return new DirectoryI;
    else {
        assert(false);
        return 0;
    }
}

void
Filesystem::NodeFactory::destroy()
{
}
```

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

See Also

- The Server-Side main Function in C++
- C++ Mapping for Classes
- Transactional Evictor