Using a Freeze Evictor in the File System Server

In this section, we present file system implementations that use a transactional evictor. The implementations are based on the ones discussed in Object Life Cycle, and in this section we only discuss code that illustrates use of the Freeze evictor.

In general, incorporating a Freeze evictor into your application requires the following steps:

1. Evaluate your existing Slice definitions for a suitable persistent object type.
2. If no suitable type is found, you typically define a new derived class that captures your persistent state requirements. Consider placing these definitions in a separate file: they are only used by the server for persistence, and therefore do not need to appear in the "public" definitions required by clients. Also consider placing your persistent types in a separate module to avoid name clashes.
3. If you use indexes with your evictor, generate code (using `slice2freeze` or `slice2freezej`) for your new definitions.
4. Create an evictor and register it as a servant locator with an object adapter.
5. Create instances of your persistent type and register them with the evictor.

Persistent Types for File System Evictor

Fortunately, it is unnecessary for us to change any of the existing file system Slice definitions to incorporate the Freeze evictor. However, we do need to add metadata definitions to inform the evictor which operations modify object state:

Slice

```
module Filesystem {
    // ...

    interface Node {
        idempotent string name();

        ["freeze:write"]
        void destroy() throws PermissionDenied;
    };

    // ...

    interface File extends Node {
        idempotent Lines read();

        ["freeze:write"]
        idempotent void write(Lines text) throws GenericError;
    };

    // ...

    interface Directory extends Node {
        idempotent NodeDescSeq list();

        idempotent NodeDesc find(string name) throws NoSuchName;

        ["freeze:write"]
        File* createFile(string name) throws NameInUse;

        ["freeze:write"]
        Directory* createDirectory(string name) throws NameInUse;
    };
};
```

These definitions are identical to the original ones, with the exception of the added `["freeze:write"]` directives.

The remaining definitions are in derived classes:

Slice

```
#include <Filesystem.ice>

module Filesystem {
    class PersistentDirectory;

    class PersistentNode implements Node {
        string nodeName;
        PersistentDirectory* parent;
    };

    class PersistentFile extends PersistentNode implements File {
        Lines text;
    };

    dictionary<string, NodeDesc> NodeDict;

    class PersistentDirectory extends PersistentNode implements Directory {
        ["freeze:write"]
        void removeNode(string name);

        NodeDict nodes;
    };
};
```

As you can see, we have sub-classed all of the file system interfaces. Let us examine each one in turn.

The `PersistentNode` class adds two data members: `nodeName` and `parent`.

We used `nodeName` instead of `name` because `name` is already used as an operation in the `Node` interface.

The file system implementation requires that a child node knows its parent node in order to properly implement the `destroy` operation. Previous implementations had a state member of type `DirectoryI`, but that is not workable here. It is no longer possible to pass the parent node to the child node's constructor because the evictor may be instantiating the child node (via a factory), and the parent node will not be known. Even if it were known, another factor to consider is that there is no guarantee that the parent node will be active when the child invokes on it, because the evictor may have evicted it. We solve these issues by storing a proxy to the parent node. If the child node invokes on the parent node via the proxy, the evictor automatically activates the parent node if necessary.

The `PersistentFile` class is very straightforward, simply adding a `text` member representing the contents of the file. Notice that the class extends `PersistentNode`, and therefore inherits the state members declared by the base class.

Finally, the `PersistentDirectory` class defines the `removeNode` operation, and adds the `nodes` state member representing the immediate children of the directory node. Since a child node contains only a proxy for its `PersistentDirectory` parent, and not a reference to an implementation class, there must be a Slice-defined operation that can be invoked when the child is destroyed.

If we had followed our earlier advice, we would have defined `Node`, `File`, and `Directory` classes in a separate `PersistentFilesystem` module, but in this example we use the existing `Filesystem` module for the sake of simplicity.

Topics

- Adding an Evictor to the C++ File System Server
- Adding an Evictor to the Java File System Server

See Also

- Object Life Cycle
- Freeze Evictors
- Freeze Evictor Concepts