Using a Freeze Map in the File System Server

We can use a Freeze map to add persistence to the file system server, and we'll present implementations in both C++ and Java. However, a Freeze evictor is often a better choice for applications (such as the file system server) in which the persistent value is an Ice object.

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

1. Evaluate your existing Slice definitions for suitable key and value types.
2. If no suitable key or value types are found, define new (possibly derived) types that capture your persistent state requirements. Consider placing these definitions in a separate file: these types 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. Generate a Freeze map for your persistent types using the Freeze compiler.
4. Use the Freeze map in your operation implementations.

Choosing Key and Value Types for the File System

Our goal is to implement the file system using Freeze maps for all persistent storage, including files and their contents. There are various options for how to implement the server. For this example, the server is stateless; whenever a client invokes an operation, the server accesses the database to satisfy the request. Implementing the server in this way has the advantage that it scales very well: we do not need a separate servant for each node; instead two default servants, one for directories and one for files, are sufficient. This keeps the memory requirements of the server to a minimum and also allows us to rely on the database for transactions and locking. (This is a very common implementation technique for servers that act as a front end to a database: the server is a simple facade that implements each operation by accessing the database.)

Our first step is to select the Slice types we will use for the key and value types for our maps. For each file, we need to store the name of the file, its parent directory, and the contents of the file. For directories, we also store the name and parent directory, as well as a dictionary that keeps track of the subdirectories and files in that directory. This leads to Slice definitions (in file `FilesystemDB.ice`) as follows:

Slice

```
#include <Filesystem.ice>
#include <Ice/Identity.ice>

module FilesystemDB {
    struct FileEntry {
        string name;
        Ice::Identity parent;
        Filesystem::Lines text;
    };

    dictionary<string, Filesystem::NodeDesc> StringNodeDescDict;

    struct DirectoryEntry {
        string name;
        Ice::Identity parent;
        StringNodeDescDict nodes;
    };
};
```

Note that the definitions are placed into a separate module, so they do not affect the existing definitions of the non-persistent version of the application. For reference, here is the definition of `NodeDesc` once more:

Slice

```
module Filesystem {
    // ...

    enum NodeType { DirType, FileType };

    struct NodeDesc {
        string name;
        NodeType type;
        Node* proxy;
    };

    // ...
};
```

To store the persistent state for the file system, we use two Freeze maps: one map for files and one map for directories. For files, we map the identity of the file to its corresponding `FileEntry` structure and, similarly, for directories, we map the identity of the directory to its corresponding `DirectoryEntry` structure.

When a request arrives from a client, the object identity is available in the server. The server uses the identity to retrieve the state of the target node for the request from the database and act on that state accordingly.

Topics

- Adding a Freeze Map to the C++ File System Server
- Adding a Freeze Map to the Java File System Server

See Also

- Default Servants
- Freeze Evictors