Implementing Object Life Cycle in Java

The implementation of our life cycle design has the following characteristics:

- It uses UUIDs as the object identities for nodes to avoid object reincarnation problems.
- When `destroy` is called on a node, the node needs to destroy itself and inform its parent directory that it has been destroyed (because the parent directory is the node's factory and also acts as a collection manager for child nodes).

Note that, in contrast to the initial version, the entire implementation resides in a `FilesystemI` package instead of being part of the `Filesystem` package. Doing this is not essential, but is a little cleaner because it keeps the implementation in a package that is separate from the Slice-generated package.

On this page:

Object Life Cycle Changes for the `NodeI` Class in Java

Our `DirectoryI` and `FileI` servants derive from a common `NodeI` base interface. This interface is not essential, but useful because it allows us to treat servants of type `DirectoryI` and `FileI` polymorphically:

Java

```
package FilesystemI;

public interface NodeI
{
    Ice.Identity id();
}
```

The only method is the `id` method, which returns the identity of the corresponding node.

Object Life Cycle Changes for the `DirectoryI` Class in Java

As in the initial version, the `DirectoryI` class derives from the generated base class `_DirectoryDisp`. In addition, the class implements the `NodeI` interface. `DirectoryI` must implement each of the Slice operations, leading to the following outline:

Java

```
package FilesystemI;

import Ice.*;
import Filesystem.*;

public class DirectoryI extends _DirectoryDisp implements NodeI
{
    public Identity
    id();

    public synchronized String
    name(Current c);

    public synchronized NodeDesc[]
    list(Current c);

    public synchronized NodeDesc
    find(String name, Current c) throws NoSuchName;

    public synchronized FilePrx
    createFile(String name, Current c) throws NameInUse;

    public synchronized DirectoryPrx
    createDirectory(String name, Current c) throws NameInUse;

    public void
    destroy(Current c) throws PermissionDenied;

    // ...
}
```

To support the implementation, we also require a number of methods and data members:

Java

```
package FilesystemI;

import Ice.*;
import Filesystem.*;

public class DirectoryI extends _DirectoryDisp implements NodeI
{
    // ...

    public DirectoryI();
    public DirectoryI(String name, DirectoryI parent);

    public synchronized void
    removeEntry(String name);

    private String _name;       // Immutable
    private DirectoryI _parent; // Immutable
    private Identity _id;       // Immutable
    private boolean _destroyed;
    private java.util.Map<String, NodeI> _contents;
}
```

The `_name` and `_parent` members store the name of this node and a reference to the node's parent directory. (The root directory's `_parent` member is null.) Similarly, the `_id` member stores the identity of this directory. The `_name`, `_parent`, and `_id` members are immutable once they have been initialized by the constructor. The `_destroyed` member prevents a race condition; to interlock access to `_destroyed` (as well as the `_contents` member) we can use synchronized methods (as for the `name` method), or use a `synchronized(this)` block.

The `_contents` map records the contents of a directory: it stores the name of an entry, together with a reference to the child node.

Here are the two constructors for the class:

Java

```
public DirectoryI()
{
    this("/", null);
}

public DirectoryI(String name, DirectoryI parent)
{
    _name = name;
    _parent = parent;
    _id = new Identity();
    _destroyed = false;
    _contents = new java.util.HashMap<String, NodeI>();

    _id.name = parent == null ? "RootDir" : java.util.UUID.randomUUID().toString();
}
```

The first constructor is a convenience function to create the root directory with the fixed identity "RootDir" and a null parent.

The real constructor initializes the `_name`, `_parent`, `_id`, `_destroyed`, and `_contents` members. Note that nodes other than the root directory use a UUID as the object identity.

The `removeEntry` method is called by the child to remove itself from its parent's `_contents` map:

Java

```
public synchronized void
removeEntry(String name)
{
    _contents.remove(name);
}
```

The implementation of the Slice name operation simply returns the name of the node, but also checks whether the node has been destroyed:

Java

```
public synchronized String
name(Current c)
{
    if (_destroyed)
        throw new ObjectNotExistException();
    return _name;
}
```

Note that this method is synchronized, so the `_destroyed` member cannot be accessed concurrently.

Here is the `destroy` member function for directories:

Java

```
public void
destroy(Current c) throws PermissionDenied
{
    if (_parent == null)
        throw new PermissionDenied("Cannot destroy root directory");

    synchronized(this) {
        if (_destroyed)
            throw new ObjectNotExistException();

        if (_contents.size() != 0)
            throw new PermissionDenied("Cannot destroy non?empty directory");

        c.adapter.remove(id());
        _destroyed = true;
    }

    _parent.removeEntry(_name);
}
```

The code first prevents destruction of the root directory and then checks whether this directory was destroyed previously. It then acquires the lock and checks that the directory is empty. Finally, `destroy` removes the Active Servant Map (ASM) entry for the destroyed directory and removes itself from its parent's `_contents` map. Note that we call `removeEntry` outside the synchronization to avoid deadlocks.

The `createDirectory` implementation acquires the lock before checking whether the directory already contains a node with the given name (or an invalid empty name). If not, it creates a new servant, adds it to the ASM and the `_contents` map, and returns its proxy:

Java

```
public synchronized DirectoryPrx
createDirectory(String name, Current c) throws NameInUse
{
    if (_destroyed)
        throw new ObjectNotExistException();

    if (name.length() == 0 || _contents.containsKey(name))
        throw new NameInUse(name);

    DirectoryI d = new DirectoryI(name, this);
    ObjectPrx node = c.adapter.add(d, d.id());
    _contents.put(name, d);
    return DirectoryPrxHelper.uncheckedCast(node);
}
```

The `createFile` implementation is identical, except that it creates a file instead of a directory:

Java

```
public synchronized FilePrx
createFile(String name, Current c) throws NameInUse
{
    if (_destroyed)
        throw new ObjectNotExistException();

    if (name.length() == 0 || _contents.containsKey(name))
        throw new NameInUse(name);

    FileI f = new FileI(name, this);
    ObjectPrx node = c.adapter.add(f, f.id());
    _contents.put(name, f);
    return FilePrxHelper.uncheckedCast(node);
}
```

Here is the implementation of `list`:

Java

```
public synchronized NodeDesc[]
list(Current c)
{
    if(_destroyed)
        throw new ObjectNotExistException();

    NodeDesc[] ret = new NodeDesc[_contents.size()];
    java.util.Iterator<java.util.Map.Entry<String, NodeI> > pos =
        _contents.entrySet().iterator();
    for(int i = 0; i < _contents.size(); ++i) {
        java.util.Map.Entry<String, NodeI> e = pos.next();
        NodeI p = e.getValue();
        ret[i] = new NodeDesc();
        ret[i].name = e.getKey();
        ret[i].type = p instanceof FileI ? NodeType.FileType : NodeType.DirType;
        ret[i].proxy = NodePrxHelper.uncheckedCast(c.adapter.createProxy(p.id()));
    }
    return ret;
}
```

After acquiring the lock, the code iterates over the directory's contents and adds a `NodeDesc` structure for each entry to the returned array.

The `find` operation proceeds along similar lines:

Java

```
public synchronized NodeDesc
find(String name, Current c) throws NoSuchName
{
    if (_destroyed)
        throw new ObjectNotExistException();

    NodeI p = _contents.get(name);
    if (p == null)
        throw new NoSuchName(name);

    NodeDesc d = new NodeDesc();
    d.name = name;
    d.type = p instanceof FileI ? NodeType.FileType : NodeType.DirType;
    d.proxy = NodePrxHelper.uncheckedCast(c.adapter.createProxy(p.id()));
    return d;
}
```

Object Life Cycle Changes for the `FileI` Class in Java

The `FileI` class is similar to the `DirectoryI` class. The data members store the name, parent, and identity of the file, as well as the `_destroyed` flag and the contents of the file (in the `_lines` member). The constructor initializes these members:

Java

```
package FilesystemI;

import Ice.*;
import Filesystem.*;
import FilesystemI.*;

public class FileI extends _FileDisp implements NodeI
{
    // ...

    public FileI(String name, DirectoryI parent)
    {
        _name = name;
        _parent = parent;
        _destroyed = false;
        _id = new Identity();
        _id.name = Util.generateUUID();
    }

    private String _name;
    private DirectoryI _parent;
    private boolean _destroyed;
    private Identity _id;
    private String[] _lines;
}
```

The implementation of the remaining member functions of the `FileI` class is trivial, so we present all of them here:

Java

```
public synchronized String
name(Current c)
{
    if (_destroyed)
        throw new ObjectNotExistException();
    return _name;
}

public Identity
id()
{
    return _id;
}

public synchronized String[]
read(Current c)
{
    if (_destroyed)
        throw new ObjectNotExistException();

    return _lines;
}

public synchronized void
write(String[] text, Current c)
{
    if (_destroyed)
        throw new ObjectNotExistException();

    _lines = (String[])text.clone();
}

public void
destroy(Current c)
{
    synchronized(this) {
        if (_destroyed)
            throw new ObjectNotExistException();

        c.adapter.remove(id());
        _destroyed = true;
    }

    _parent.removeEntry(_name);
}
```

Object Life Cycle Concurrency Issues in Java

The preceding implementation is provably deadlock free. All methods hold only one lock at a time, so they cannot deadlock with each other or themselves. While the locks are held, the methods do not call other methods that acquire locks, so any potential deadlock can only arise by concurrent calls to another mutating method, either on the same node or on different nodes. For concurrent calls on the same node, deadlock is impossible because such calls are strictly serialized on the instance; for concurrent calls to `destroy` on different nodes, each node locks itself, releases itself again, and then acquires and releases a lock on its parent (by calling `removeEntry`), also making deadlock impossible.

See Also

- Example of a File System Server in Java
- Life Cycle and Normal Operations
- Object Identity and Uniqueness
- Acquiring Locks without Deadlocks
- The Active Servant Map