Freeze Manual

Persistent Storage for Ice Objects

The Freeze persistence service allows you to store Ice objects in Oracle Berkeley DB, with all the features you expect from a robust database - transactions, hot backups, indexing, and more.

In C++, Freeze and Berkeley DB consist of a library that you link with your C++ application. In Java, Freeze is a JAR file that you include in your Java application. Together, Freeze and Berkeley DB give you a fully embedded solution: the databases are regular files on a local file system, there is no database server to setup, and there is no need for ongoing database administration.

Freeze lets you choose between two persistence models: evictors and maps. With Freeze evictors, Freeze persists the state of your Ice objects automatically; these Ice objects just need to define their persistent state in Slice classes. The alternative is to store key-value pairs in Freeze maps, where key and value are both Slice types.

> Freeze supports only the deprecated Slice to C++98 and Slice to Java Compat mappings. Freeze is likewise a **deprecated** service : we do not recommend using Freeze for new applications.

Getting Help with Freeze

If you have a question and you cannot find an answer in this manual, you can visit our developer forums to see if another developer has encountered the same issue. If you still need help, feel free to post your question on the forums, which ZeroC's developers monitor regularly. Note, however, that we can provide only limited free support in our forums. For guaranteed response and problem resolution times, you should subscribe to ZeroC's Priority Support.

Legal Notices

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this manual and ZeroC was aware of the trademark claim, the designations have been printed in initial caps or all caps. ZeroC has taken care in the preparation of this manual but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

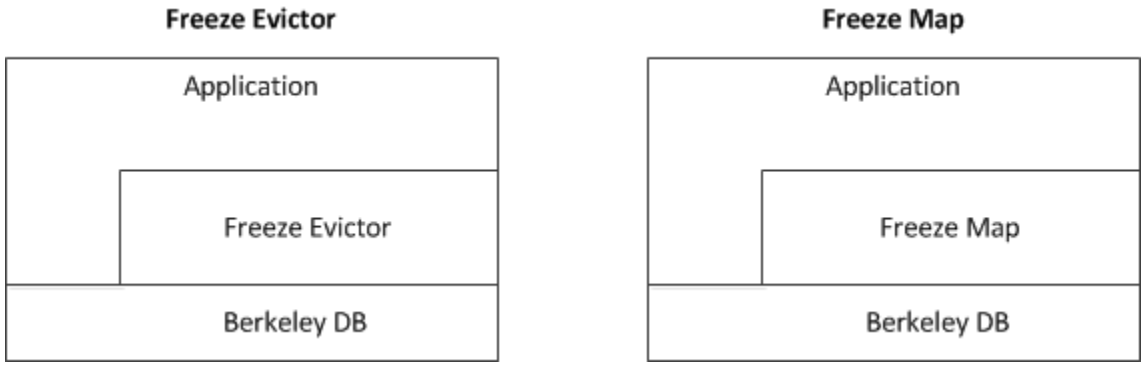
License

Copyright

mailto:info@zeroc.com
https://zeroc.com

Freeze

Freeze is a collection of services that simplify the use of persistence in Ice applications, as shown below:



Layer diagram for Freeze persistence services.

The Freeze map is an associative container mapping any Slice key and value types, providing a convenient and familiar interface to a persistent map. Freeze evictors are an especially powerful facility for supporting persistent Ice objects in a highly-scalable implementation.

The Freeze persistence services comprise:

- Freeze evictor
  A highly-scalable implementation of an Ice servant locator that provides automatic persistence and eviction of servants with only minimal application code.
- Freeze map
  A generic associative container. Code generators are provided that produce type-specific maps for Slice key and value types. Applications interact with a Freeze map just like any other associative container, except the keys and values of a Freeze map are persistent.

As you will see from the examples in this discussion, integrating a Freeze map or evictor into your Ice application is quite straightforward: once you define your persistent data in Slice, Freeze manages the mundane details of persistence.

Freeze is implemented using Berkeley DB, a compact and high-performance embedded database. The Freeze map and evictor APIs insulate applications from the Berkeley DB API, but do not prevent applications from interacting directly with Berkeley DB if necessary.

Topics

Evictors

Freeze evictors combine persistence and scalability features into a single facility that is easily incorporated into Ice applications.

As an implementation of a servant locator, a Freeze evictor takes advantage of the fundamental separation between Ice object and servant to activate servants on demand from persistent storage, and to deactivate them again using customized eviction constraints. Although an application may have thousands of Ice objects in its database, it is not practical to have servants for all of those Ice objects resident in memory simultaneously. The application can conserve resources and gain greater scalability by setting an upper limit on the number of active servants, and letting a Freeze evictor handle the details of servant activation, persistence, and deactivation.

Topics

See Also

- Servant Locators

Evictor Concepts

This page introduces the Freeze evictor.

On this page:

Describing Persistent State for an Evictor

The persistent state of servants managed by a Freeze evictor must be described in Slice. Specifically, every servant must implement a Slice class, and a Freeze evictor automatically stores and retrieves all the (Slice-defined) data members of these Slice classes. Data members that are not specified in Slice are not persistent.

A Freeze evictor relies on the Ice object factory facility to load persistent servants from disk: the evictor creates a brand new servant using the registered factory and then restores the servant's data members. Therefore, for every persistent servant class you define, you need to register a corresponding object factory with the Ice communicator. (For more details on object factories, refer to the C++98 mapping or the Java Compat mapping.)

Evictor Servant Semantics

With a Freeze evictor, each <*`object identity, facet`*> pair is associated with its own dedicated persistent object (servant). Such a persistent object cannot serve several identities or facets. Each servant is loaded and saved independently of other servants; in particular, there is no special grouping for the servants that serve the facets of a given Ice object.

Similar to the way you activate servants with an object adapter, the Freeze evictor provides operations named `add`, `addFacet`, `remove`, and `removeFacet`. They have the same signature and semantics, except that with the Freeze evictor, the mapping and the state of the mapped servants is stored in a database.

Evictor Types

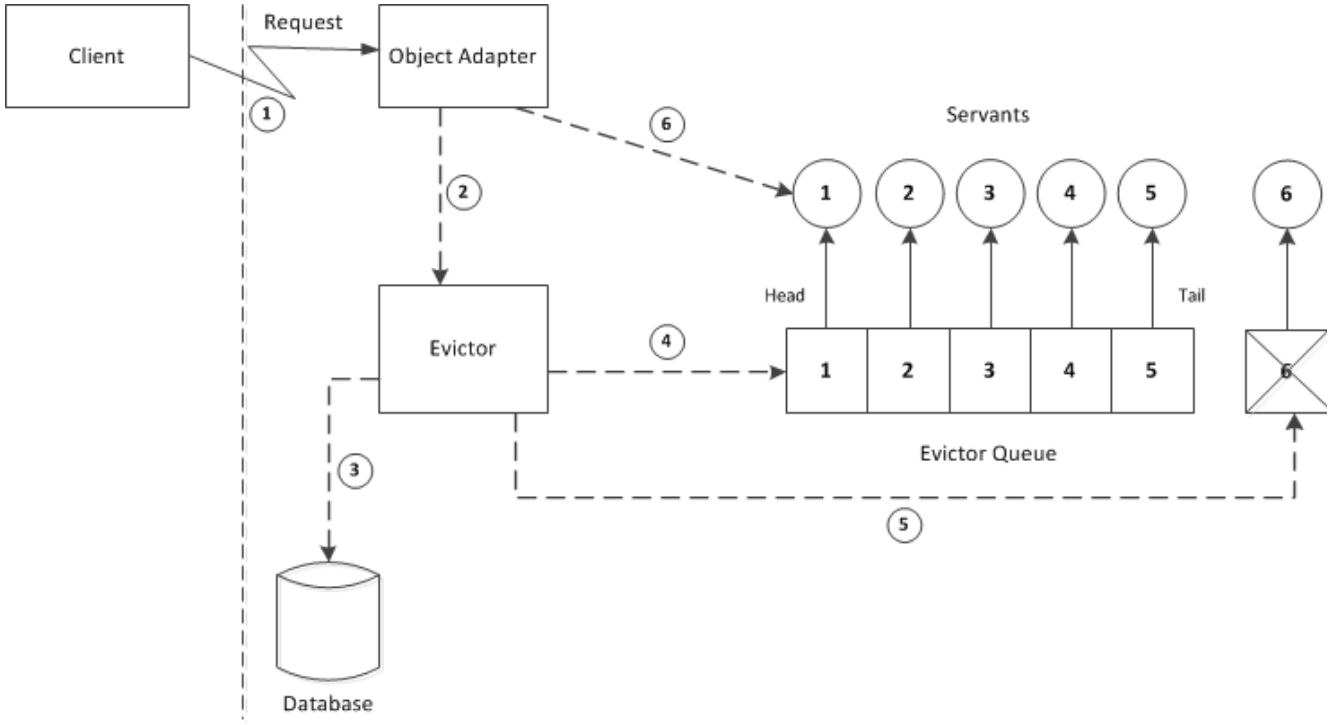
Freeze provides two types of evictors with different storage characteristics. The background save evictor records state changes to the database in a background thread, while the transactional evictor records all state changes immediately within the context of a transaction. You can choose the evictor that best fits the persistence requirements of your application.

Eviction Strategy

Both types of evictors associate a queue with their servant map and manage this queue using a "least recently used" eviction algorithm: if the queue is full, the least recently used servant is evicted to make room for a new servant.

Here is the sequence of events for activating a servant as shown in the figure below. Let us assume that we have configured the evictor with a size of five, that the queue is full, and that a request has arrived for a servant that is not currently active. (With a transactional evictor, we also assume this request does not change any persistent state.)

1. A client invokes an operation.
2. The object adapter invokes on the evictor to locate the servant.
3. The evictor first checks its servant map and fails to find the servant, so it instantiates the servant and restores its persistent state from the database.
4. The evictor adds an item for the servant (servant 1) at the head of the queue.
5. The queue's length now exceeds the configured maximum, so the evictor removes servant 6 from the queue as soon as it is eligible for eviction. With a background save evictor, this occurs once there are no outstanding requests pending on servant 6, and once the servant's state has been safely stored in the database. With a transactional save, the servant is removed from the queue immediately.
6. The object adapter dispatches the request to the new servant.



An evictor queue after restoring servant 1 and evicting servant 6.

Detecting Updates to Persistent State

A Freeze evictor considers that a servant's persistent state has been modified when a read-write operation on this servant completes. To indicate whether an operation is read-only or read-write, you add metadata directives to the Slice definitions of the objects:

- The `["freeze:write"]` directive informs the evictor that an operation modifies the persistent state of the target servant.
- The `["freeze:read"]` directive informs the evictor that an operation does not modify the persistent state of the target.

If no metadata directive is present, an operation is assumed to not modify its target.

Here is how you could mark the operations on an interface with these metadata directives:

Slice

```
interface Example
{
    ["freeze:read"]  string readonlyOp();
    ["freeze:write"] void   writeOp();
}
```

This marks `readonlyOp` as an operation that does not modify its target, and marks `writeOp` as an operation that does modify its target. Because, without any directive, an operation is assumed to not modify its target, the preceding definition can also be written as follows:

Slice

```
interface Example
{
    string readonlyOp(); // ["freeze:read"] implied
    ["freeze:write"] void writeOp();
}
```

The metadata directives can also be applied to an interface or a class to establish a default. This allows you to mark an interface as `["freeze:write"]` and to only add a `["freeze:read"]` directive to those operations that are read-only, for example:

Slice

```
["freeze:write"]
interface Example
{
    ["freeze:read"] string readonlyOp();
                    void   writeOp1();
                    void   writeOp2();
                    void   writeOp3();
}
```

This marks `writeOp1`, `writeOp2`, and `writeOp3` as read-write operations, and `readonlyOp` as a read-only operation.

Note that it is important to correctly mark read-write operations with a `["freeze:write"]` metadata directive — without the directive, Freeze will not know when an object has been modified and may not store the updated persistent state to disk.

Also note that, if you make calls directly on servants (so the calls are not dispatched via the Freeze evictor), the evictor will have no idea when a servant's persistent state is modified; if any such direct call modifies the servant's data members, the update may be lost.

Iterating an Evictor

A Freeze evictor iterator provides the ability to iterate over the identities of the objects stored in an evictor. The operations are similar to Java iterator methods: `hasNext` returns true while there are more elements, and `next` returns the next identity:

Slice

```
local interface EvictorIterator
{
    bool hasNext();
    Ice::Identity next();
}
```

You create an iterator by calling `getIterator` on your evictor:

Slice

```
EvictorIterator getIterator(string facet, int batchSize);
```

The new iterator is specific to a facet (specified by the `facet` parameter). Internally, this iterator will retrieve identities in batches of `batchSize` objects; we recommend using a fairly large batch size to get good performance.

Indexing an Evictor Database

A Freeze evictor supports the use of indexes to quickly find persistent servants using the value of a data member as the search criteria. The types allowed for these indexes are the same as those allowed for Slice dictionary keys.

The `slice2freeze` and `slice2freezej` tools can generate an `Index` class when passed the `--index` option:

`--index` *`CLASS`*`,`*`TYPE`*`,`*`MEMBER`*`[,case-sensitive|case-insensitive]`

`CLASS` is the name of the class to be generated. *`TYPE`* denotes the type of class to be indexed (objects of different classes are not included in this index). *`MEMBER`* is the name of the data member in *`TYPE`* to index. When *`MEMBER`* has type `string`, it is possible to specify whether the index is case-sensitive or not. The default is case-sensitive.

The generated `Index` class supplies three methods whose definitions are mapped from the following Slice operations:

- `sequence<Ice::Identity> findFirst(`*`member-type`* `index, int firstN)`
  Returns up to `firstN` objects of *`TYPE`* whose *`MEMBER`* is equal to `index`. This is useful to avoid running out of memory if the potential number of objects matching the criteria can be very large.
- `sequence<Ice::Identity> find(`*`member-type`* `index)`
  Returns all the objects of *`TYPE`* whose *`MEMBER`* is equal to `index`.
- `int count(`*`member-type`* `index)`
  Returns the number of objects of *`TYPE`* having *`MEMBER`* equal to `index`.

Indexes are associated with a Freeze evictor during evictor creation. See the definition of the `createBackgroundSaveEvictor` and `createTransactionalEvictor` functions for details.

Indexed searches are easy to use and very efficient. However, be aware that an index adds significant write overhead: with Berkeley DB, every update triggers a read from the database to get the old index entry and, if necessary, replace it.

If you add an index to an existing database, by default existing facets are not indexed. If you need to populate a new or empty index using the facets stored in your Freeze evictor, set the property `Freeze.Evictor.`*`env-name.filename`*`.PopulateEmptyIndices` to a non-zero value, which instructs Freeze to iterate over the corresponding facets and create the missing index entries during the call to `createBackgroundSaveEvictor` or `createTransactionalEvictor`. When you use this feature, you must register the object factories for all of the facet types before you create your evictor.

Using a Servant Initializer

In some applications, it may be necessary to initialize a servant after the servant is instantiated by the evictor but before an operation is dispatched to the servant. The Freeze evictor allows an application to specify a servant initializer for this purpose.

To illustrate the sequence of events, let us assume that a request has arrived for a servant that is not currently active:

1. The evictor restores a servant for the target Ice object (and facet) from the database. This involves two steps:
   - The Ice run time locates and invokes the factory for the Ice facet's type, thereby obtaining a new instance with uninitialized data members.
   - The data members are populated from the persistent state.
2. The evictor invokes the application's servant initializer (if any) for the servant.
3. If the evictor is a background-save evictor, it adds the servant to its cache.
4. The evictor dispatches the operation.

With a background-save evictor, the servant initializer is called before the object is inserted into the evictor's internal cache, and *without* holding any internal lock, but in such a way that when the servant initializer is called, the servant is guaranteed to be inserted in the evictor cache.

There is only one restriction on what a servant initializer can do: it must not make a remote invocation on the object (facet) being initialized. Failing to follow this rule will result in deadlocks.

The file system example demonstrates the use of a servant initializer.

Application Design Considerations for Evictors

The Freeze evictor creates a snapshot of a servant's state for persistent storage by marshaling the servant, just as if the servant were being

sent "over the wire" as a parameter to a remote invocation. Therefore, the Slice definitions for an object type must include the data members comprising the object's persistent state.

For example, we could define a Slice class as follows:

Slice

```
class Stateless
{
    void calc();
}
```

However, without data members, there will not be any persistent state in the database for objects of this type, and hence there is little value in using the Freeze evictor for this type.

Obviously, Slice object types need to define data members, but there are other design considerations as well. For example, suppose we define a simple application as follows:

Slice

```
class Account
{
    ["freeze:write"] void withdraw(int amount);
    ["freeze:write"] void deposit(int amount);

    int balance;
}

interface Bank
{
    Account* createAccount();
}
```

In this application, we would use a Freeze evictor to manage `Account` objects that have a data member `balance` representing the persistent state of an account.

From an object-oriented design perspective, there is a glaring problem with these Slice definitions: implementation details (the persistent state) are exposed in the client-server contract. The client cannot directly manipulate the `balance` member because the `Bank` interface returns `Account` proxies, not `Account` instances. However, the presence of the data member may cause unnecessary confusion for client developers.

A better alternative is to clearly separate the persistent state as shown below:

Slice

```
interface Account
{
    ["freeze:write"] void withdraw(int amount);
    ["freeze:write"] void deposit(int amount);
}

interface Bank
{
    Account* createAccount();
}

class PersistentAccount implements Account
{
    int balance;
}
```

Now the Freeze evictor can manage `PersistentAccount` objects, while clients interact with `Account` proxies. (Ideally, `PersistentAccount` would be defined in a different source file and inside a separate Slice module.)

See Also

- Classes
- C++98 Mapping for Classes
- Java Compat Mapping for Classes
- Servant Activation and Deactivation
- Background Save Evictor
- Transactional Evictor
- Dictionaries
- Using an Evictor in the File System Server

Background Save Evictor

Freeze provides two types of evictors. This page describes the background save evictor.

> Freeze also provides a transactional evictor, with different persistence semantics. The on-disk format of these two types of evictors is the same: you can switch from one type of evictor to the other without any data transformation.

On this page:

Overview of the Background Save Evictor

A background save evictor keeps all its servants in a map and writes the state of newly-created, modified, and deleted servants to disk asynchronously, in a background thread. You can configure how often servants are saved; for example you could decide to save every three minutes, or whenever ten or more servants have been modified. For applications with frequent updates, this allows you to group many updates together to improve performance.

The downside of the background save evictor is recovery from a crash. Because saves are asynchronous, there is no way to force an immediate save to preserve a critical update. Moreover, you cannot group several related updates together: for example, if you transfer funds between two accounts (servants) and a crash occurs shortly after this update, it is possible that, once your application comes back up, you will see the update on one account but not on the other. Your application needs to handle such inconsistencies when restarting after a crash.

Similarly, a background save evictor provides no ordering guarantees for saves. If you update servant 1, servant 2, and then servant 1 again, it is possible that, after recovering from a crash, you will see the latest state for servant 1, but no updates at all for servant 2.

The background save evictor implements the local interface `Freeze::BackgroundSaveEvictor`, which derives from `Freeze::Evictor`.

Creating a Background Save Evictor

You create a background save evictor in C++ with the global function `Freeze::createBackgroundSaveEvictor`, and in Java with the static method `Freeze.Util.createBackgroundSaveEvictor`.

For C++, the signatures are as follows:

C++

```
BackgroundSaveEvictorPtr
createBackgroundSaveEvictor(
    const ObjectAdapterPtr& adapter,
    const string& envName,
    const string& filename,
    const ServantInitializerPtr& initializer = 0,
    const vector<IndexPtr>& indexes = vector<IndexPtr>(),
    bool createDb = true);

BackgroundSaveEvictorPtr
createBackgroundSaveEvictor(
    const ObjectAdapterPtr& adapter,
    const string& envName,
    DbEnv& dbEnv,
    const string& filename,
    const ServantInitializerPtr& initializer = 0,
    const vector<IndexPtr>& indexes = vector<IndexPtr>(),
    bool createDb = true);
```

For Java, the method signatures are:

Java

```
public static BackgroundSaveEvictor
createBackgroundSaveEvictor(
    Ice.ObjectAdapter adapter,
    String envName,
    String filename,
    ServantInitializer initializer,
    Index[] indexes,
    boolean createDb);

public static BackgroundSaveEvictor
createBackgroundSaveEvictor(
    Ice.ObjectAdapter adapter,
    String envName,
    com.sleepycat.db.Environment dbEnv,
    String filename,
    ServantInitializer initializer,
    Index[] indexes,
    boolean createDb);
```

Both C++ and Java provide two overloaded functions: in one case, Freeze opens and manages the underlying Berkeley DB environment; in the other case, you provide a `DbEnv` object that represents a Berkeley DB environment you opened yourself. (Usually, it is easiest to let Freeze take care of all interactions with Berkeley DB.)

The `envName` parameter represents the name of the underlying Berkeley DB environment, and is also used as the default Berkeley DB

home directory. (See `Freeze.DbEnv.env-name.DbHome`.)

The `filename` parameter represents the name of the Berkeley DB database file associated with this evictor. The persistent state of all your servants is stored in this file.

The `initializer` parameter represents the servant initializer. It is an optional parameter in C++; in Java, pass `null` if you do not need a servant initializer.

The `indexes` parameter is a vector or array of evictor indexes. It is an optional parameter in C++; in Java, pass `null` if your evictor does not define an index.

Finally, the `createDb` parameter tells Freeze what to do when the corresponding Berkeley DB database does not exist. When true, Freeze creates a new database; when false, Freeze raises a `Freeze::DatabaseException`.

The Background Saving Thread

All persistence activity of a background save evictor is handled in a background thread created by the evictor. This thread wakes up periodically and saves the state of all newly-registered, modified, and destroyed servants in the evictor's queue.

For applications that experience bursts of activity that result in a large number of modified servants in a short period of time, you can also configure the evictor's thread to begin saving as soon as the number of modified servants reaches a certain threshold.

Synchronization Semantics for the Background Save Evictor

When the saving thread takes a snapshot of a servant it is about to save, it is necessary to prevent the application from modifying the servant's persistent data members at the same time.

The Freeze evictor and the application need to use a common synchronization to ensure correct behavior. In Java, this common synchronization is the servant itself: the Freeze evictor synchronizes the servant (a Java object) while taking the snapshot. In C++, the servant is required to inherit from the class `IceUtil::AbstractMutex` described below: the background save evictor locks the servant through this interface while taking a snapshot. On the application side, the servant's implementation is required to use the same mechanism to synchronize all operations that access the servant's Slice-defined data members.

Preventing Servant Eviction

Occasionally, automatically evicting and reloading all servants can be inefficient. You can remove a servant from the evictor's queue by locking this servant "in memory" using the `keep` or `keepFacet` operation on the evictor:

Slice

```
local interface BackgroundSaveEvictor extends Evictor {
    void keep(Ice::Identity id);
    void keepFacet(Ice::Identity id, string facet);
    void release(Ice::Identity id);
    void releaseFacet(Ice::Identity id, string facet);
};
```

`keep` and `keepFacet` are recursive: you need to call `release` or `releaseFacet` for this servant the same number of times to put it back in the evictor queue and make it eligible again for eviction.

Servants kept in memory (using `keep` or `keepFacet`) do not consume a slot in the evictor queue. As a result, the maximum number of servants in memory is approximately the number of kept servants plus the evictor size. (It can be larger while you have many evictable objects that are modified but not yet saved.)

Handling Fatal Evictor Errors

Freeze allows you to register a callback for handling fatal errors encountered by a background save evictor. If no callback is registered, the evictor aborts the application by default.

The application should assume that an evictor will not continue to work properly after encountering a fatal error.

Use `Freeze.Util.registerFatalErrorCallback` (Java) or `Freeze::registerFatalErrorCallback` (C++) to register the callback.

For C++, `registerFatalErrorCallback` accepts a function pointer of type `FatalErrorCallback`:

C++

```
typedef void (*FatalErrorCallback)(const BackgroundSaveEvictorPtr&,
const Ice::CommunicatorPtr&);
```

In Java, `registerFatalErrorCallback` accepts a reference to an object that implements `Freeze.FatalErrorCallback`:

Java

```
package Freeze;

public interface FatalErrorCallback
{
    void handleError(Evictor evictor, Ice.Communicator communicator,
RuntimeException ex);
}
```

Note that the `RuntimeException` argument may be `null`.

Abstract Mutex

`IceUtil::AbstractMutex` defines a mutex base interface used by the Freeze background save evictor. The interface allows the evictor to synchronize with servants that are stored in a Freeze database. The class has the following definition:

C++

```
namespace IceUtil
{
    class AbstractMutex {
    public:
        typedef LockT<AbstractMutex> Lock;
        typedef TryLockT<AbstractMutex> TryLock;

        virtual ~AbstractMutex();

        virtual void lock() const = 0;
        virtual void unlock() const = 0;
        virtual bool tryLock() const = 0;
    };
}
```

This class is in namespace `IceUtil` for backwards compatibility with prior releases. It is however included through `Freeze/Freeze.h`.

The same header file also defines a few template implementation classes that specialize `AbstractMutex`, as described below.

AbstractMutexI

This template class implements `AbstractMutex` by forwarding all member functions to its template argument:

C++

```
namespace IceUtil
{
    template <typename T>
    class AbstractMutexI : public AbstractMutex, public T
    {
    public:
        typedef LockT<AbstractMutexI> Lock;
        typedef TryLockT<AbstractMutexI> TryLock;

        virtual void lock() const
        {
            T::lock();
        }

        virtual void unlock() const
        {
            T::unlock();
        }

        virtual bool tryLock() const
        {
            return T::tryLock();
        }

        virtual ~AbstractMutexI() {}
    };
}
```

AbstractMutexReadI

This template class implements a read lock by forwarding the `lock` and `tryLock` functions to the `readLock` and `tryReadLock` functions of its template argument:

C++

```
namespace IceUtil
{
    template <typename T>
    class AbstractMutexReadI : public AbstractMutex, public T
    {
    public:
        typedef LockT<AbstractMutexReadI> Lock;
        typedef TryLockT<AbstractMutexReadI> TryLock;

        virtual void lock() const
        {
            T::readLock();
        }

        virtual void unlock() const
        {
            T::unlock();
        }

        virtual bool tryLock() const
        {
            return T::tryReadLock();
        }

        virtual ~AbstractMutexReadI() {}
    };
}
```

AbstractMutexWriteI

This template class implements a write lock by forwarding the `lock` and `tryLock` functions to the `writeLock` and `tryWriteLock` function s of its template argument:

C++

```
namespace IceUtil
{
    template <typename T>
    class AbstractMutexWriteI : public AbstractMutex, public T
    {
    public:
        typedef LockT<AbstractMutexWriteI> Lock;
        typedef TryLockT<AbstractMutexWriteI> TryLock;

        virtual void lock() const
        {
            T::writeLock();
        }

        virtual void unlock() const
        {
            T::unlock();
        }

        virtual bool tryLock() const
        {
            return T::tryWriteLock();
        }

        virtual ~AbstractMutexWriteI() {}
    };
}
```

Apart from use with Freeze servants, these templates are also useful if, for example, you want to implement your own evictor.

See Also

- Transactional Evictor
- Evictor Concepts

Transactional Evictor

Freeze provides two types of evictors. This page describes the transactional evictor.

> Freeze also provides a background save evictor, with different persistence semantics. The on-disk format of these two types of evictors is the same: you can switch from one type of evictor to the other without any data transformation.

On this page:

Overview of the Transactional Evictor

A transactional evictor maintains a servant map, but only keeps read-only servants in this map. The state of these servants corresponds to the latest data on disk. Any servant creation, update, or deletion is performed within a database transaction. This transaction is committed (or rolled back) immediately, typically at the end of the dispatch of the current operation, and the associated servants are then discarded.

With such an evictor, you can ensure that several updates, often on different servants (possibly managed by different transactional evictors) are grouped together: either all or none of these updates occur. In addition, updates are written almost immediately, so crash recovery is a lot simpler: few (if any) updates will be lost, and you can maintain consistency between related persistent objects.

However, an application based on a transactional evictor is likely to write a lot more to disk than an application with a background save evictor, which may have an adverse impact on performance.

Creating a Transactional Evictor

You create a transactional evictor in C++ with the global function `Freeze::createTransactionalEvictor`, and in Java with the static method `Freeze.Util.createTransactionalEvictor`.

For C++, the signatures are as follows:

C++

```
typedef map<string, string> FacetTypeMap;

TransactionalEvictorPtr
createTransactionalEvictor(
    const ObjectAdapterPtr& adapter,
    const string& envName,
    const string& filename,
    const FacetTypeMap& facetTypes = FacetTypeMap(),
    const ServantInitializerPtr& initializer = 0,
    const vector<IndexPtr>& indexes = vector<IndexPtr>(),
    bool createDb = true);

TransactionalEvictorPtr
createTransactionalEvictor(
    const ObjectAdapterPtr& adapter,
    const string& envName,
    DbEnv& dbEnv,
    const string& filename,
    const FacetTypeMap& facetTypes = FacetTypeMap(),
    const ServantInitializerPtr& initializer = 0,
    const vector<IndexPtr>& indexes = vector<IndexPtr>(),
    bool createDb = true);
```

For Java, the method signatures are:

Java

```
public static TransactionalEvictor
createTransactionalEvictor(
    Ice.ObjectAdapter adapter,
    String envName,
    String filename,
    java.util.Map facetTypes,
    ServantInitializer initializer,
    Index[] indexes,
    boolean createDb);

public static TransactionalEvictor
createTransactionalEvictor(
    Ice.ObjectAdapter adapter,
    String envName,
    com.sleepycat.db.Environment dbEnv,
    String filename,
    java.util.Map facetTypes,
    ServantInitializer initializer,
    Index[] indexes,
    boolean createDb);
```

Both C++ and Java provide two overloaded functions: in one case, Freeze opens and manages the underlying Berkeley DB environment; in the other case, you provide a `DbEnv` object that represents a Berkeley DB environment you opened yourself. (Usually, it is easier to let Freeze take care of all interactions with Berkeley DB.)

The `envName` parameter represents the name of the underlying Berkeley DB environment, and is also used as the default Berkeley DB home directory. (See `Freeze.DbEnv.`*`env-name`*`.DbHome`.)

The `filename` parameter represents the name of the Berkeley DB database file associated with this evictor. The persistent state of all your servants is stored in this file.

The `facetTypes` parameter allows you to specify a single class type (Slice type ID string) for each facet in your new evictor (see below). Most applications use only the default facet, represented by an empty string. This parameter is optional in C++; in Java, pass `null` if you do not want to specify such a facet-to-type mapping.

The `initializer` parameter represents the servant initializer. It is an optional parameter in C++; in Java, pass `null` if you do not need a servant initializer.

The `indexes` parameter is a vector or array of evictor indexes. It is an optional parameter in C++; in Java, pass `null` if your evictor does not define an index.

Finally, the `createDb` parameter tells Freeze what to do when the corresponding Berkeley DB database does not exist. When true, Freeze creates a new database; when false, Freeze raises a `Freeze::DatabaseException`.

Read and Write Operations

When a transactional evictor processes an incoming request without an associated transaction, it first needs to find out whether the corresponding operation is read-only or read-write (as specified by the `"freeze:read"` and `"freeze:write"` operation metadata). This is straightforward if the evictor knows the target's type; in this case, it simply instantiates and keeps a "dummy" servant to look up the attributes of each operation.

However, if the target type can vary, the evictor needs to look up and sometimes load a read-only servant to find this information. For read-write requests, it will then load the servant from disk a second time (within a new transaction). Once the transaction commits, the read-only servant — sometimes freshly loaded from disk — is discarded.

When you create a transactional evictor with `createTransactionalEvictor`, you can pass a facet name to type ID map to associate a single servant type with each facet and speed up the retrieval of these operation attributes.

Synchronization Semantics for the Transactional Evictor

With a transactional evictor, there is no need to perform any synchronization on the servants managed by the evictor:

- For read-only operations, the application must not modify any data member, and hence there is no need to synchronize. (Many threads can safely read the same data members concurrently.)
- For read-write operations, each operation dispatch gets its own private servant or servants (see transaction propagation below).

Not having to worry about synchronization can dramatically simplify your application code.

Transaction Propagation

Without a distributed transaction service, it is not possible to invoke several remote operations within the same transaction. Nevertheless, Freeze supports transaction propagation for collocated calls: when a request is dispatched within a transaction, the transaction is associated with the dispatch thread and will propagate to any other servant reached through a collocated call. If the target of a collocated call is managed by a transactional evictor associated with the same database environment, Freeze reuses the propagated transaction to load the servant and dispatch the request. This allows you to group updates to several servants within a single transaction.

You can also control how a transactional evictor handles an incoming transaction through optional metadata added after `"freeze:write"` and `"freeze:read"`. There are six valid directives:

- `freeze:read:never`
  Verify that no transaction is propagated to this operation. If a transaction is present, the transactional evictor raises a `Freeze::DatabaseException`.
- `freeze:read:supports`
  Accept requests with or without a transaction, and re-use the transaction if present. `"supports"` is the default for `"freeze:read"` operations.
- `freeze:read:mandatory` and `freeze:write:mandatory`
  Verify that a transaction is propagated to this operation. If there is no transaction, the transactional evictor raises a `Freeze::DatabaseException`.
- `freeze:read:required` and `freeze:write:required`
  Accept requests with or without a transaction, and re-use the transaction if present. If no transaction is propagated, the transactional evictor creates a new transaction before dispatching the request. `"required"` is the default for `"freeze:write"` operations.

Commit or Rollback on User Exception

When a transactional evictor processes an incoming read-write request, it starts a new database transaction, loads a servant within the transaction, dispatches the request, and then either commits or rolls back the transaction depending on the outcome of this dispatch. If the dispatch does not raise an exception, the transaction is committed just before the response is sent back to the client. If the dispatch raises an Ice run-time exception, the transaction is rolled back. If the dispatch raises a user exception, by default, the transaction is committed. However, you can configure Freeze to rollback on user-exceptions by setting `Freeze.Evictor.`*`env-name.fileName`*`.RollbackOnUserException` to a non-zero value.

Database Deadlocks and Automatic Retries

When reading and writing in separate concurrent transactions, deadlocks are likely to occur. For example, one transaction may lock pages in a particular order while another transaction locks the same pages in a different order; the outcome is a deadlock. Berkeley DB automatically detects such deadlocks, and "kills" one of the transactions.

With a Freeze transactional evictor, the application does not need to catch any deadlock exceptions or retry when deadlock occurs because the transactional evictor automatically retries its transactions whenever it encounters a deadlock situation.

However, this can affect how you implement your operations: for any operation called within a transaction (mainly read-write operations), you must anticipate the possibility of several calls for the same request, all in the same dispatch thread.

AMD and the Transactional Evictor

When a transactional evictor dispatches a read-write operation implemented using AMD, it starts a transaction before dispatching the request, and commits or rolls back the transaction when the dispatch is done. Two threads are involved here: the *dispatch thread* and the *callback thread*. The dispatch thread is a thread from an Ice thread pool tasked with dispatching a request, and the callback thread is the thread that invokes the AMD callback to send the response to the client. These threads may be one and the same if the servant invokes the AMD callback from the dispatch thread.

It is important to understand the threading semantics of an AMD request with respect to the transaction:

- If a successful AMD response is sent from the dispatch thread, the transaction is committed *after* the response is sent. If a deadlock occurs during this commit, the request is not retried and the client receives no indication of the failure.
- If a successful AMD response is sent from another thread, the evictor commits its transaction when the dispatch thread completes, regardless of whether the servant has sent the AMD response. The callback thread waits until the transaction has been committed by the dispatch thread before sending the response.
- If a commit results in a deadlock and the AMD response has not yet been sent, the evictor cancels the original AMD callback and automatically retries the request again with a new AMD callback. Invocations on the original AMD callback are ignored (`ice_response` and `ice_exception` on this callback do nothing).
- Otherwise, if the servant sends an exception via the AMD callback, the response is sent directly to the client.

Transactions and Freeze Maps

A transactional evictor uses the same transaction objects as Freeze maps, which allows you to update a Freeze map within a transaction managed by a transactional evictor.

You can get the current transaction created by a transactional evictor by calling `getCurrentTransaction`. Then, you would typically retrieve the associated Freeze connection (with `getConnection`) and construct a Freeze map using this connection:

Slice

```
local interface TransactionalEvictor extends Evictor
{
    Transaction getCurrentTransaction();
    void setCurrentTransaction(Transaction tx);
}
```

A transactional evictor also gives you the ability to associate your own transaction with the current thread, using `setCurrentTransaction`. This is useful if you want to perform many updates within a single transaction, for example to add or remove many servants in the evictor. (A less convenient alternative is to implement all such updates within a read-write operation on some object.)

See Also

- Background Save Evictor
- Type IDs
- Evictor Concepts
- Maps

Using an 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
- Evictors
- Evictor Concepts

Adding an Evictor to the C++ File System Server

On this page:

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 : 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 <Ice/Ice.h>
#include <Freeze/Freeze.h>

namespace Filesystem
{

class FileI : public virtual PersistentFile
{
public:

    FileI();

    // Slice operations...

    static Freeze::EvictorPtr _evictor;

private:

    bool _destroyed;
    IceUtil::Mutex _mutex;
};

class DirectoryI : public virtual 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 : 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__);
    }

    return nodeName;
}

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

        if(_destroyed)
        {
            throw Ice::ObjectNotExistException(__FILE__, __LINE__);
        }
        _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__);
    }

    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__);
    }
```

```
    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__);
    }

    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->crea
teProxy(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__);
    }

    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->cre
ateProxy(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

- C++98 Mapping for Classes
- Transactional Evictor

Adding an Evictor to the Java File System Server

On this page:

The Server Main Program in Java

The server's `main` method 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:

Java

```
import Filesystem.*;

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

    public int
    run(String[] args)
    {
        Ice.ObjectFactory factory = new NodeFactory();
        communicator().addObjectFactory(factory, PersistentFile.ice_sta
ticId());
        communicator().addObjectFactory(factory, PersistentDirectory.ic
e_staticId());

        Ice.ObjectAdapter adapter = communicator().createObjectAdapter(
"EvictorFilesystem");

        Freeze.Evictor evictor =
            Freeze.Util.createTransactionalEvictor(adapter, _envName, "
evictorfs",

null, null, null, true);
        DirectoryI._evictor = evictor;
        FileI._evictor = evictor;

        adapter.addServantLocator(evictor, "");

        Ice.Identity rootId = new Ice.Identity();
        rootId.name = "RootDir";
        if(!evictor.hasObject(rootId))
```

```
        {
            PersistentDirectory root = new DirectoryI();
            root.nodeName = "/";
            root.nodes =
new java.util.HashMap<java.lang.String, NodeDesc>();
            evictor.add(root, rootId);
        }

        adapter.activate();

        communicator().waitForShutdown();

        return 0;
    }

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

```
    private String _envName;
}
```

Let us examine the changes in detail. First, we define the class `Server` as a subclass of `Ice.Application`, and provide a constructor taking a string argument:

Java

```
    public
    Server(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:

Java

```
        Ice.ObjectFactory factory = new NodeFactory();
        communicator().addObjectFactory(factory, PersistentFile.ice_sta
ticId());
        communicator().addObjectFactory(factory, PersistentDirectory.ic
e_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, the fourth is null to indicate that we do not use facets, the fifth is null to indicate that we do not use a servant initializer, the sixth argument (`null`) indicates no indexes are in use, and the `true` argument requests that the database be created if it does not exist. The evictor is then added to the object adapter as a servant locator for the default category:

Java

```
        Freeze.Evictor evictor =
            Freeze.Util.createTransactionalEvictor(adapter, _envName, "
evictorfs",

null, null, null, true);
        DirectoryI._evictor = evictor;
        FileI._evictor = evictor;

        adapter.addServantLocator(evictor, "");
```

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

Java

```
        Ice.Identity rootId = new Ice.Identity();
        rootId.name = "RootDir";
        if(!evictor.hasObject(rootId))
        {
            PersistentDirectory root = new DirectoryI();
            root.nodeName = "/";
            root.nodes = new java.util.HashMap<String, NodeDesc>();
            evictor.add(root, rootId);
        }
```

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

Java

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

The Persistent Servant Class Definitions in Java

The servant classes must also be changed to incorporate the Freeze evictor. The `FileI` class now has a static state member `_evictor`:

Java

```
import Filesystem.*;

public final class FileI extends PersistentFile
{
    public
    FileI()
    {
        _destroyed = false;
    }

    // Slice operations...

    public static Freeze.Evictor _evictor;
    private boolean _destroyed;
}
```

The `DirectoryI` class has undergone a similar transformation:

Java

```
import Filesystem.*;

public final class DirectoryI extends PersistentDirectory
{
    public
    DirectoryI()
    {
        _destroyed = false;
        nodes = new java.util.HashMap<String, NodeDesc>();
    }

    // Slice operations...

    public static Freeze.Evictor _evictor;
    private boolean _destroyed;
}
```

Implementing a Persistent `FileI` in Java

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

Java

```
    public synchronized String
    name(Ice.Current current)
    {
        if(_destroyed)
        {
            throw new Ice.ObjectNotExistException();
        }

        return nodeName;
    }

    public void
    destroy(Ice.Current current)
        throws PermissionDenied
    {
        synchronized(this)
        {
            if(_destroyed)
            {
                throw new Ice.ObjectNotExistException();
            }
            _destroyed = true;
        }
```

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

    public synchronized String[]
    read(Ice.Current current)
    {
        if(_destroyed)
        {
            throw new Ice.ObjectNotExistException();
        }

        return (String[])text.clone();
    }

    public synchronized void
    write(String[] text, Ice.Current current)
        throws GenericError
    {
        if(_destroyed)
        {
            throw new Ice.ObjectNotExistException();
        }
```

```
        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 Java

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

Java

```
    public synchronized DirectoryPrx
    createDirectory(String name, Ice.Current current)
        throws NameInUse
    {
        if(_destroyed)
        {
            throw new Ice.ObjectNotExistException(current.id, current.f
acet, current.operation);
        }

        if(name.length() == 0 || nodes.containsKey(name))
        {
            throw new NameInUse(name);
        }

        Ice.Identity id = current.adapter.getCommunicator().stringToIde
ntity(
            java.util.UUID.randomUUID().toString());
        PersistentDirectory dir = new DirectoryI();
        dir.nodeName = name;
        dir.parent = PersistentDirectoryPrxHelper.uncheckedCast(
            current.adapter.createProxy(current.id));
        DirectoryPrx proxy = DirectoryPrxHelper.uncheckedCast(_evictor.
add(dir, id));

        NodeDesc nd = new NodeDesc();
        nd.name = name;
        nd.type = NodeType.DirType;
        nd.proxy = proxy;
        nodes.put(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`:

Java

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

        if(name.length() == 0 || nodes.containsKey(name))
        {
            throw new NameInUse(name);
        }

        Ice.Identity id = current.adapter.getCommunicator().stringToIde
ntity(
            java.util.UUID.randomUUID().toString());
        PersistentFile file = new FileI();
        file.nodeName = name;
        file.parent = PersistentDirectoryPrxHelper.uncheckedCast(
            current.adapter.createProxy(current.id));
        FilePrx proxy = FilePrxHelper.uncheckedCast(_evictor.add(file,
id));

        NodeDesc nd = new NodeDesc();
        nd.name = name;
        nd.type = NodeType.FileType;
        nd.proxy = proxy;
        nodes.put(name, nd);

        return proxy;
    }
```

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

Implementing `NodeFactory` in Java

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.

Java

```
package Filesystem;

public class NodeFactory implements Ice.ObjectFactory
{
    public Ice.Object
    create(String type)
    {
        if(type.equals(PersistentFile.ice_staticId()))
        {
            return new FileI();
        }
        else if(type.equals(PersistentDirectory.ice_staticId()))
        {
            return new DirectoryI();
        }
        else
        {
            assert(false);
            return null;
        }
    }

    public void
    destroy()
    {
    }
}
```

See Also

- Java Mapping for Classes
- Transactional Evictor

Cache Helper Class for Evictor Implementation

Topics

Cache Helper Class for C++

This `Freeze::Cache` template class allows you to efficiently maintain a cache that is backed by secondary storage, such as a Berkeley DB database, without holding a lock on the entire cache while values are being loaded from the database. If you want to create evictors for servants that store their state in a database, the `Cache` class can simplify your evictor implementation considerably.

> The Freeze evictors are implemented using `Freeze::Cache`, but `Freeze::Cache` is not tied to Berkeley DB - you can use this template class for any evictor implementation.

The `Cache` class has the following interface:

C++

```
template<typename Key, typename Value>
class Cache
{
public:
    typedef typename std::map</* ... */, /* ... */>::iterator Position;

    bool pin(const Key& k, const IceUtil::Handle<Value>& v);
    IceUtil::Handle<Value> pin(const Key& k);
    void unpin(Position p);

    IceUtil::Handle<Value> putIfAbsent(const Key& k, const IceUtil::Han
dle<Value>& v);

    IceUtil::Handle<Value> getIfPinned(const Key&, bool = false) const;

    void clear();
    size_t size() const;


protected:
    virtual IceUtil::Handle<Value> load(const Key& k) = 0;
    virtual void pinned(const IceUtil::Handle<Value>& v, Position p);

    virtual ~Cache();
};
```

Note that `Cache` is an abstract base class — you must derive a concrete implementation from `Cache` and provide an implementation of the `load` and, optionally, of the `pinned` member function.

Internally, a `Cache` maintains a map of name-value pairs. The key and value type of the map are supplied by the `Key` and `Value` template arguments, respectively. The implementation of `Cache` takes care of maintaining the map; in particular, it ensures that concurrent lookups by callers are possible without blocking even if some of the callers are currently loading values from the backing store. In turn, this is useful for evictor implementations, such as the Freeze evictors . The `Cache` class does not limit the number of entries in the cache — it is the job of the evictor implementation to limit the map size by calling `unpin` on elements of the map that it wants to evict.

Your concrete implementation class must implement the `load` function, whose job it is to load the value for the key `k` from the backing store and to return a IceUtil::Handle to that value. Note that `load` returns a value of type `IceUtil::IceUtil::Handle`, that is, the value must be heap-allocated and support the usual reference-counting functions for smart pointers. (The easiest way to achieve this is to derive the value from `IceUtil::Shared`.)

If `load` cannot locate a record for the given key because no such record exists, it must return a null IceUtil::Handle. If `load` fails for some other reason, it can throw an exception, which is propagated back to the application code.

Your concrete implementation class typically will also override the `pinned` function (unless you want to have a cache that does not limit the number of entries; the provided default implementation of `pinned` is a no-op). The `Cache` implementation calls `pinned` whenever it has added a value to the map as a result of a call to `pin`; the `pinned` function is therefore a callback that allows the derived class to find out when a value has been added to the cache and informs the derived class of the value and its position in the cache.

The `Position` parameter is a `std::iterator` into the cache's internal map that records the position of the corresponding map entry. (Note that the element type of map is opaque, so you should not rely on knowledge of the cache's internal key and value types.) Your implementation of `pinned` must remember the position of the entry because that position is necessary to remove the corresponding entry from the cache again.

The public member functions of `Cache` behave as follows:

```
bool pin(const Key& k, const IceUtil::Handle<Value>& v);
```

To add a key-value pair to the cache, your evictor can call `pin`. The return value is true if the key and value were added; a false return value indicates that the map already contained an entry with the given key and the original value for that key is unchanged.

`pin` calls `pinned` if it adds an entry.

This version of `pin` does *not* call `load` to retrieve the entry from backing store if it is not yet in the cache. This is useful when you add a newly-created object to the cache.

Once an entry is in the cache, it is guaranteed to remain in the cache at the same position in memory, and without its value being overwritten by another thread, until that entry is unpinned by a call to `unpin`.

```
IceUtil::Handle<Value> pin(const Key& k);
```

A second version of `pin` looks for the entry with the given key in the cache. If the entry is already in the cache, `pin` returns the entry's value. If no entry with the given key is in the cache, `pin` calls `load` to retrieve the corresponding entry. If `load` returns an entry, `pin` adds it to the cache and returns the entry's value. If the entry cannot be retrieved from the backing store, `pin` returns null.

`pin` calls `pinned` if it adds an entry.

The function is thread-safe, that is, it calls `load` only once all other threads have unpinned the entry.

Once an entry is in the cache, it is guaranteed to remain in the cache at the same position in memory, and without its value being overwritten by another thread, until that entry is unpinned by a call to `unpin`.

```
IceUtil::Handle<Value> putIfAbsent(const Key& k, const IceUtil::Handle<Value>& v);
```

This function adds a key-value pair to the cache and returns a smart pointer to the value. If the map already contains an entry with the given key, that entry's value remains unchanged and `putIfAbsent` returns its value. If no entry with the given key is in the cache, `putIfAbsent` calls `load` to retrieve the corresponding entry. If `load` returns an entry, `putIfAbsent` adds it to the cache and returns the entry's value. If the entry cannot be retrieved from the backing store, `putIfAbsent` returns null.

`putIfAbsent` calls `pinned` if it adds an entry.

The function is thread-safe, that is, it calls `load` only once all other threads have unpinned the entry.

Once an entry is in the cache, it is guaranteed to remain in the cache at the same position in memory, and without its value being overwritten by another thread, until that entry is unpinned by a call to `unpin`.

```
IceUtil::Handle<Value> getIfPinned(const Key& k, bool wait = false) const;
```

This function returns the value stored for the key `k`.

- If an entry for the given key is in the map, the function returns the value immediately, regardless of the value of `wait`.
- If no entry for the given key is in the map and the `wait` parameter is false, the function returns a null IceUtil::Handle.
- If no entry for the given key is in the map and the `wait` parameter is true, the function blocks the calling thread if another thread is currently attempting to load the same entry; once the other thread completes, `getIfPinned` completes and returns the value added by the other thread.

```
void unpin(Position p);
```

This function removes an entry from the map. The iterator p determines which entry to remove. (It must be an iterator that previously was passed to `pinned`.) The iterator p is invalidated by this operation, so you must not use it again once `unpin` returns. (Note that the `Cache` im

plementation ensures that updates to the map never invalidate iterators to existing entries in the map; `unpin` invalidates only the iterator for the removed entry.)

```
void clear();
```

This function removes all entries in the map.

```
size_t size() const;
```

This function returns the number of entries in the map.

See Also

- Servant Evictors
- The C++ IceUtil::Handle Template
- The C++ Shared and SimpleShared Classes

Cache Helper Class for Java

The Freeze.Cache class allows you to efficiently maintain a cache that is backed by secondary storage, such as a Berkeley DB database, without holding a lock on the entire cache while values are being loaded from the database. If you want to create evictors for servants that store their state in a database, the Cache class can simplify your evictor implementation considerably.

> The Freeze evictors are implemented using Freeze.Cache, but Freeze.Cache is not tied to Berkeley DB - you can use this template class for any evictor implementation.

The Cache class has the following interface:

Java

```
package Freeze;

public class Cache
{
    public Cache(Store store);

    public Object pin(Object key);
    public Object pin(Object key, Object o);
    public Object unpin(Object key);

    public Object putIfAbsent(Object key, Object newObj);
    public Object getIfPinned(Object key);

    public void clear();
    public int size();
}
```

Internally, a Cache maintains a map of name-value pairs. The implementation of Cache takes care of maintaining the map; in particular, it ensures that concurrent lookups by callers are possible without blocking even if some of the callers are currently loading values from the backing store. In turn, this is useful for evictor implementations. The Cache class does not limit the number of entries in the cache — it is the job of the evictor implementation to limit the map size by calling unpin on elements of the map that it wants to evict.

The Cache class works in conjunction with a Store interface for which you must provide an implementation. The Store interface is trivial:

Java

```
package Freeze;

public interface Store
{
    Object load(Object key);
}
```

You must implement the load method in a class that you derive from Store. The Cache implementation calls load when it needs to retrieve the value for the passed key from the backing store. If load cannot locate a record for the given key because no such record exists, it must return null. If load fails for some other reason, it can throw an exception derived from java.lang.RuntimeException, which is propagated back to the application code.

The public member functions of `Cache` behave as follows:

`Cache(Store s)`

The constructor initializes the cache with your implementation of the `Store` interface.

`Object pin(Object key, Object val)`

To add a key-value pair to the cache, your evictor can call `pin`. The return value is null if the key and value were added; otherwise, if the map already contains an entry with the given key, the entry is unchanged and `pin` returns the original value for that key.

This version of `pin` does *not* call `load` to retrieve the entry from backing store if it is not yet in the cache. This is useful when you add a newly-created object to the cache.

`Object pin(Object key)`

This version of `pin` returns the value stored in the cache for the given key if the cache already contains an entry for that key. If no entry with the given key is in the cache, `pin` calls `load` to retrieve the corresponding value (if any) from the backing store. `pin` returns the value returned by `load`, that is, the value if `load` could retrieve it, null if `load` could not retrieve it, or any exception thrown by `load`.

`Object unpin(Object key)`

`unpin` removes the entry for the given key from the cache. If the cache contained an entry for the key, the return value is the value for that key; otherwise, the return value is null.

`Object putIfAbsent(Object key, Object val)`

This function adds a key-value pair to the cache. If the cache already contains an entry for the given key, `putIfAbsent` returns the original value for that key. If no entry with the given key is in the cache, `putIfAbsent` calls `load` to retrieve the corresponding entry (if any) from the backing store and returns the value returned by `load`.

If the cache does not contain an entry for the given key and `load` does not retrieve a value for the key, the method adds the new entry and returns null.

`Object getIfPinned(Object key)`

This function returns the value stored for the given key. If an entry for the given key is in the map, the function returns the corresponding value; otherwise, the function returns null. `getIfPinned` does not call `load`.

`void clear()`

This function removes all entries in the map.

`int size()`

This function returns the number of entries in the map.

Maps

A Freeze map is a persistent, associative container in which the key type must be a legal dictionary key type and the value type can be any primitive or user-defined Slice type. For each pair of key and value types, the developer uses a code-generation tool to produce a language-specific class that conforms to the standard conventions for maps in that language. For example, in C++, the generated class resembles a `std::map`, and in Java it implements the `java.util.SortedMap` interface. Most of the logic for storing and retrieving state to and from the database is implemented in a Freeze base class. The generated map classes derive from this base class, so they contain little code and therefore are efficient in terms of code size.

You can only store data types that are defined in Slice in a Freeze map. Types without a Slice definition (that is, arbitrary C++ or Java types) cannot be stored because a Freeze map reuses the Ice-generated marshaling code to create the persistent representation of the data in the database. This is especially important to remember when defining a Slice class whose instances will be stored in a Freeze map; only the "public" (Slice-defined) data members will be stored, not the private state members of any derived implementation class.

Topics

See Also

- Classes

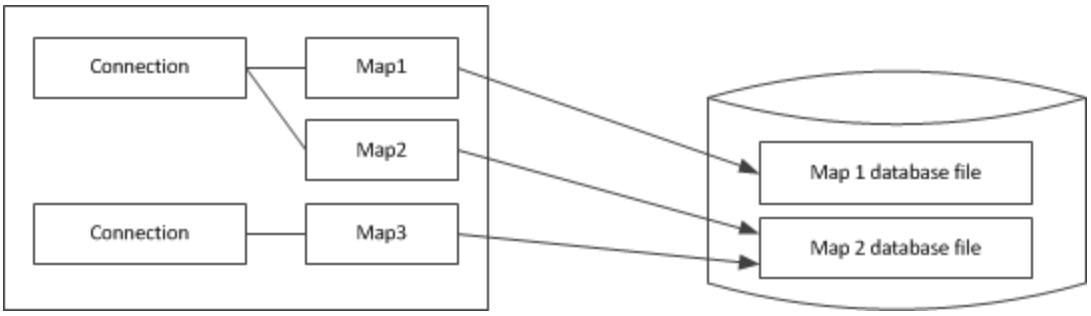
Map Concepts

On this page:

Freeze Connections

In order to create a Freeze map object, you first need to obtain a Freeze `Connection` object by connecting to a database environment.

As illustrated in the following figure, a Freeze map is associated with a single connection and a single database file. Connection and map objects are not thread-safe: if you want to use a connection or any of its associated maps from multiple threads, you must serialize access to them. If your application requires concurrent access to the same database file (persistent map), you must create several connections and associated maps.



Freeze connections and maps.

Freeze connections provide operations that allow you to begin a transaction, access the current transaction, get the communicator associated with a connection, close a connection, and remove a map index. See the Slice API reference for more information on these operations.

Using Transactions with Freeze Maps

You may optionally use transactions with Freeze maps. Freeze transactions provide the usual ACID (atomicity, concurrency, isolation, durability) properties. For example, a transaction allows you to group several database updates in one atomic unit: either all or none of the updates within the transaction occur.

You start a transaction by calling `beginTransaction` on the `Connection` object. Once a connection has an associated transaction, all operations on the map objects associated with this connection use this transaction. Eventually, you end the transaction by calling `commit` or `rollback`: `commit` saves all your updates while `rollback` undoes them. The `currentTransaction` operation returns the transaction associated with a connection, if any; otherwise, it returns nil.

Slice

```
module Freeze
{
    local interface Transaction
    {
        void commit();
        void rollback();
    }

    local interface Connection
    {
        Transaction beginTransaction();
        idempotent Transaction currentTransaction();
        // ...
    }
}
```

If you do not use transactions, every non-iterator update is enclosed in its own internal transaction, and every read-write iterator has an associated internal transaction that is committed when the iterator is closed.

Using Transactions with C++

You must ensure that you either commit or roll back each transaction that you begin (otherwise, locks will be held by the database until they time out):

C++

```
ConnectionPtr connection = ...;

TransactionPtr tx = connection->beginTransaction();
try
{
    // DB updates that might throw here...

    tx->commit();

    // More code that might throw here...

}
catch(...)
{
    try
    {
        tx->rollback();
    }
    catch(...)
    {
    }
    throw;
}
```

The outer try-catch blocks are necessary because, if the code encounters an exception, we must roll back any updates that were made. In turn, the attempt to roll back might throw itself, namely, if the code following the commit throws an exception (in which case the transaction cannot be rolled back because it is already committed).

Code such as this is difficult to maintain: for example, an early return statement can cause the transaction to be neither committed nor rolled back. The `TransactionHolder` class ensures that such errors cannot happen:

C++

```
namespace Freeze
{
    class TransactionHolder
    {
    public:
        TransactionHolder(const ConnectionPtr&);
        ~TransactionHolder();

        void commit();
        void rollback();

    private:
        // Copy and assignment are forbidden.
        TransactionHolder(const TransactionHolder&);
        TransactionHolder& operator=(const TransactionHolder&);
    };
}
```

The constructor calls `beginTransaction` if the connection does not already have a transaction in progress, so instantiating the holder also starts a transaction. When the holder instance goes out of scope, its destructor calls `rollback` on the transaction and suppresses any exceptions that the rollback attempt might throw. This ensures that the transaction is rolled back if it was not previously committed or rolled back and ensures that an early return or an exception cannot cause the transaction to remain open:

C++

```
ConnectionPtr connection = ...;

{ // Open scope

    TransactionHolder tx(connection); // Begins transaction

    // DB updates that might throw here...

    tx.commit();

    // More code that might throw here...

} // Transaction rolled back here if not previously
  // committed or rolled back.
```

If you instantiate a `TransactionHolder` when a transaction is already in progress, it does nothing: the constructor notices that it could not begin a new transaction and turns `commit`, `rollback`, and the destructor into no-ops. For example, the nested `TransactionHolder` inst ance in the following code is benign and does nothing:

C++

```
ConnectionPtr connection = ...;

{ // Open scope

    TransactionHolder tx(connection); // Begins transaction

    // DB updates that might throw here...

    { // Open nested scope

        TransactionHolder tx2(connection); // Does nothing

        // DB updates that might throw here...

        tx2.commit(); // Does nothing

        // More code that might throw here...

    } // Destructor of tx2 does nothing

    tx.commit();

    // More code that might throw here...

} // Transaction rolled back here if not previously
  // committed or rolled back.
```

Using Transactions with Java

You must ensure that you either commit or roll back each transaction that you begin (otherwise, locks will be held by the database until they time out):

Java

```
Connection connection = ...;

Transaction tx = connection.beginTransaction();
try
{

    // DB updates that might throw here...

    tx.commit();

    // More code that might throw here...

}
catch(java.lang.RuntimeException ex)
{
    try
    {
        tx.rollback();
    }
    catch(DatabaseException e)
    {
    }
    throw ex;
}
```

The catch handler ensures that the transaction is rolled back before re-throwing the exception. Note that the nested try-catch blocks are necessary: if the transaction committed successfully but the code following the commit throws an exception, the rollback attempt will fail therefore we need to suppress the corresponding `DatabaseException` that is raised in that case.

Also use caution with early `return` statements:

Java

```
Connection connection = ...;

Transaction tx = connection.beginTransaction();
try
{

    // DB updates that might throw here...

    if(error)
    {
        // ...
        return; // Oops, bad news!
    }

    // ...

    tx.commit();

    // More code that might throw here...

}
catch(java.lang.RuntimeException ex)
{
    try
    {
        tx.rollback();
    }
    catch(DatabaseException e)
    {
    }
    throw ex;
}
```

The early `return` statement in the preceding code causes the transaction to be neither committed nor rolled back. To deal with this situation, avoid early return statements or ensure that you either commit or roll back the transaction before returning. Alternatively, you can use a `finally` block to ensure that the transaction is rolled back:

Java

```
Connection connection = ...;

try
{
    Transaction tx = connection.beginTransaction();

    // DB updates that might throw here...

    if(error)
    {
        // ...
        return; // No problem, see finally block.
    }

    // ...

    tx.commit();

    // More code that might throw here...

}
finally
{
    if(connection.currentTransaction() != null)
    {
       connection.currentTransaction().rollback();
    }
}
```

Iterating over a Freeze Map

Iterators allow you to traverse the contents of a Freeze map. Iterators are implemented using Berkeley DB cursors and acquire locks on the underlying database page files. In C++, both read-only (`const_iterator`) and read-write iterators (`iterator`) are available. In Java, an iterator is read-write if it is obtained in the context of a transaction and read-only if it is obtained outside a transaction.

Locks held by an iterator are released when the iterator is closed (if you do not use transactions) or when the enclosing transaction ends. Releasing locks held by iterators is very important to let other threads access the database file through other connection and map objects. Occasionally, it is even necessary to release locks to avoid self-deadlock (waiting forever for a lock held by an iterator created by the same thread).

To improve ease of use and make self-deadlocks less likely, Freeze often closes iterators automatically. If you close a map or connection, associated iterators are closed. Similarly, when you start or end a transaction, Freeze closes all the iterators associated with the corresponding maps. If you do not use transactions, any write operation on a map (such as inserting a new element) automatically closes all iterators opened on the same map object, except for the current iterator when the write operation is performed through that iterator. In Java, Freeze also closes a read-only iterator when no more elements are available.

There is, however, one situation in C++ where an explicit iterator close is needed to avoid self-deadlock:

- you do not use transactions, and
- you have an open iterator that was used to update a map (it holds a write lock), and
- in the same thread, you read that map.

Read operations in C++ never close iterators automatically: you need to either use transactions or explicitly close the iterator that holds the write lock. This is not an issue in Java because you cannot use an iterator to update a map outside of a transaction.

Recovering from Freeze Map Deadlocks

If you use multiple threads to access a database file, Berkeley DB may acquire locks in conflicting orders (on behalf of different transactions or iterators). For example, an iterator could have a read-lock on page P1 and attempt to acquire a write-lock on page P2, while another iterator (on a different map object associated with the same database file) could have a read-lock on P2 and attempt to acquire a write-lock on P1.

When this occurs, Berkeley DB detects a deadlock and resolves it by returning a "deadlock" error to one or more threads. For all non-iterator operations performed outside any transaction, such as an insertion into a map, Freeze catches such errors and automatically retries the operation until it succeeds. (In that case, the most-recently acquired lock is released before retrying.) For other operations, Freeze reports this deadlock by raising `Freeze::DeadlockException`. In that case, the associated transaction or iterator is also automatically rolled back or closed. A properly written application must expect to catch deadlock exceptions and retry the transaction or iteration.

Key Sorting for Freeze Maps

Keys in Freeze maps and indexes are always sorted. By default, Freeze sorts keys according to their Ice-encoded binary representation; this is very efficient but the resulting order is rarely meaningful for the application. Starting with Ice 3.0, Freeze offers the ability to specify your own comparator objects so that you can customize the traversal order of your maps. Note however that the comparator of a Freeze map should remain the same throughout the life of the map. Berkeley DB stores records according to the key order provided by this comparator; switching to another comparator will cause undefined behavior.

Key Sorting for Freeze Maps in C++

In C++, you specify the name of your comparator objects during code generation. The generated map provides the standard features of `std::map`, so that iterators return entries according to the order you have defined for the main key with your comparator object. The `lower_bound`, `upper_bound`, and `equal_range` functions provide range-searches (see the definition of these functions on `std::map`).

Apart from these standard features, the generated map provides additional functions and methods to perform range searches using secondary keys. The additional functions are `lowerBoundFor`*`Member`*, `upperBoundFor`*`Member`*, and `equalRangeFor`*`Member`*, where *`Member`* is the name of the secondary-key member. These functions return regular iterators on the Freeze map.

Key Sorting for Freeze Maps in Java

In Java, you supply comparator objects (instances of the standard Java interface `java.util.Comparator`) at run time when instantiating the generated map class. The map constructor accepts a comparator for the main key and optionally a collection of comparators for secondary keys. The map also provides a number of methods for performing range searches on the main key and on secondary keys.

Indexing a Freeze Map

Freeze maps support efficient reverse lookups: if you define an index when you generate your map (with `slice2freeze` or `slice2freezej`), the generated code provides additional methods for performing reverse lookups. If your value type is a structure or a class, you can also index on a member of the value, and several such indices can be associated with the same Freeze map.

Indexed searches are easy to use and very efficient. However, be aware that an index also adds significant write overhead: with Berkeley DB, every update triggers a read from the database to get the old index entry and, if necessary, replace it.

If you add an index to an existing map, Freeze automatically populates the index the next time you open the map. Freeze populates the index by instantiating each map entry, so it is important that you register the object factories for any class types in your map before you open the map.

Note that the index key comparator of a Freeze map index should remain the same throughout the life of the index. Berkeley DB stores records according to the key order provided by this comparator; switching to another comparator will cause undefined behavior.

See Also

- Using a Map in C++
- Using a Map in Java

Using a Map in C++

This page describes the C++ code generator and demonstrates how to use a Freeze map in a C++ program.

On this page:

Generating a Simple Map for C++

We can use slice2freeze to generate a simple Freeze map:

```
$ slice2freeze --dict StringIntMap,string,int StringIntMap
```

This command directs the compiler to create a map named `StringIntMap`, with the Slice key type `string` and the Slice value type `int`. The final argument is the base name for the output files, to which the compiler appends the `.h` and `.cpp` suffixes. As a result, this command produces two C++ source files, `StringIntMap.h` and `StringIntMap.cpp`.

The Freeze Map Class in C++

If you examine the contents of the header file created by the example in the previous section, you will discover that a Freeze map is a subclass of the template class `Freeze::Map`:

C++

```
// StringIntMap.h

class StringIntMap : public Freeze::Map< std::string, Ice::Int, ... >
{
public:
    StringIntMap(const Freeze::ConnectionPtr& connection,
                 const std::string& dbName,
                 bool createDb = true,
                 const Freeze::IceEncodingCompare& compare = ...);

    template <class _InputIterator>
    StringIntMap(const Freeze::ConnectionPtr& connection,
                 const std::string& dbName,
                 bool createDb,
                 _InputIterator first, _InputIterator last,
                 const Freeze::IceEncodingCompare& compare = ...);

    ...
};
```

The `Freeze::Map` template class closely resembles the standard container class `std::map`, as shown in the following class definition:

C++

```
namespace Freeze
{
    template<...> class Map {
    public:
        typedef ... value_type;
        typedef ... iterator;
        typedef ... const_iterator;

        typedef size_t size_type;
        typedef ptrdiff_t difference_type;

        static void recreate(const Freeze::ConnectionPtr& connection,
                             const std::string& dbName,

               const Freeze::IceEncodingCompare& compare = ...);

        bool operator==(const Map& rhs) const;
        bool operator!=(const Map& rhs) const;

        void swap(Map& rhs);

        iterator begin();
        const_iterator begin() const;

        iterator end();
        const_iterator end() const;

        bool empty() const;
        size_type size() const;
        size_type max_size() const;

        iterator insert(iterator /*position*/, const value_type& elem);

        std::pair<iterator, bool> insert(const value_type& elem);

        template <typename InputIterator>
        void insert(InputIterator first, InputIterator last);

        void put(const value_type& elem);

        template <typename InputIterator>
        void put(InputIterator first, InputIterator last);

        void erase(iterator position);
        size_type erase(const key_type& key);
        void erase(iterator first, iterator last);

        void clear();

        void destroy(); // Non-standard.
```

```
iterator find(const key_type& key);
const_iterator find(const key_type& key) const;

size_type count(const key_type& key) const;

iterator lower_bound(const key_type& key);
const_iterator lower_bound(const key_type& key) const;
iterator upper_bound(const key_type& key);
const_iterator upper_bound(const key_type& key) const;

std::pair<iterator, iterator>
equal_range(const key_type& key);

std::pair<const_iterator, const_iterator>
equal_range(const key_type& key) const;

const Ice::CommunicatorPtr&
communicator() const;

...
```

```
    };
}
```

The semantics of the `Freeze::Map` methods are identical to those of `std::map` unless otherwise noted. In particular, the overloaded `insert` method shown below ignores the `position` argument:

C++

```
iterator insert(iterator /*position*/, const value_type& elem);
```

A Freeze map class supports only those methods shown above; other features of `std::map`, such as allocators and overloaded array operators, are not available.

Non-standard methods that are specific to Freeze maps are discussed below:

- Constructors
  The following overloaded constructors are provided:

  C++

  ```
  Map(const Freeze::ConnectionPtr& connection,
      const std::string& dbName,
      bool createDb = true,
      const Freeze::IceEncodingCompare& compare = ...);

  template<class _InputIterator>
  Map(const Freeze::ConnectionPtr& connection,
      const std::string& dbName,
      bool createDb,
      _InputIterator first, _InputIterator last,
      const Freeze::IceEncodingCompare& compare = ...);
  ```

  The first constructor accepts a connection, the database name, a flag indicating whether to create the database if it does not exist, and an object used to compare keys. The second constructor accepts all of the parameters of the first, with the addition of iterators from which elements are copied into the map.

  Note that a database can only contain the persistent state of one map type. Any attempt to instantiate maps of different types on the same database results in undefined behavior.

- Map copy
  The `recreate` function copies an existing database:

  C++

  ```
  static void recreate(const Freeze::ConnectionPtr& connection,
                       const std::string& dbName,
                       const Freeze::IceEncodingCompare& compare = .
  ..)
  ```

  The `dbName` parameter specifies an existing database name. The copy has the name <*dbName*>`.old-`<*uuid*>. For example, if the database name is `MyDB`, the copy might be named `MyDB.old-edefd55a-e66a-478d-a77b-f6d53292b873`. (Obviously, a different UUID is used each time you recreate a

database).

- `destroy`
  This method deletes the database from its environment and from the Freeze catalog. If a transaction is not currently open, the method creates its own transaction in which to perform this task.
- `communicator`
  This method returns the communicator with which the map's connection is associated.

Using Iterators with Freeze Maps in C++

A Freeze map's iterator works like its counterpart in `std::map`. The iterator class supports one convenient (but nonstandard) method:

C++

```
void set(const mapped_type& value)
```

Using this method, a program can replace the value at the iterator's current position.

Sample Freeze Map Program in C++

The program below demonstrates how to use a `StringIntMap` to store `<string, int>` pairs in a database. You will notice that there are no explicit `read` or `write` operations called by the program; instead, simply using the map has the side effect of accessing the database.

C++

```
#include <Freeze/Freeze.h>
#include <Ice/Ice.h>
#include <StringIntMap.h>

int
main(int argc, char* argv[])
{
    // Initialize the Communicator.
    //
    Ice::CommunicatorHolder ich(argc, argv);

    // Create a Freeze database connection.
    //
    Freeze::ConnectionPtr connection =
Freeze::createConnection(ich.communicator(), "db");

    // Instantiate the map.
    //
    StringIntMap map(connection, "simple");

    // Clear the map.
    //
    map.clear();

    Ice::Int i;
    StringIntMap::iterator p;
```

```
// Populate the map.
//
for(i = 0; i < 26; i++)
{
    std::string key(1, 'a' + i);
    map.insert(make_pair(key, i));
}

// Iterate over the map and change the values.
//
for(p = map.begin(); p != map.end(); ++p)
{
    p.set(p->second + 1);
}

// Find and erase the last element.
//
p = map.find("z");
assert(p != map.end());
map.erase(p);

// Clean up.
//
connection->close();
```

```
    return 0;
}
```

Prior to instantiating a Freeze map, the application must connect to a Berkeley DB database environment:

C++

```
Freeze::ConnectionPtr connection =
Freeze::createConnection(ich.communicator(), "db");
```

The second argument is the name of a Berkeley DB database environment; by default, this is also the file system directory in which Berkeley DB creates all database and administrative files. Note that properties with the prefix `Freeze.DbEnv` can modify a number of environment settings, including the file system directory. For the preceding example, you could change the directory to `FreezeDir` by setting the property `Freeze.DbEnv.db.DbHome` to `FreezeDir`.

Next, the code instantiates the `StringIntMap` on the connection. The constructor's second argument supplies the name of the database file, which by default is created if it does not exist:

C++

```
StringIntMap map(connection, "simple");
```

After instantiating the map, we clear it to make sure it is empty in case the program is run more than once:

C++

```
map.clear();
```

Next, we populate the map using a single-character string as the key:

C++

```
for(i = 0; i < 26; i++)
{
    std::string key(1, 'a' + i);
    map.insert(make_pair(key, i));
}
```

Iterating over the map will look familiar to `std::map` users. However, to modify a value at the iterator's current position, we use the nonstandard `set` method:

C++

```
for(p = map.begin(); p != map.end(); ++p)
{
    p.set(p->second + 1);

}
```

Next, the program obtains an iterator positioned at the element with key `z`, and erases it:

C++

```
p = map.find("z");
assert(p != map.end());
map.erase(p);
```

Finally, the program closes the database connection:

C++

```
connection->close();
```

It is not necessary to explicitly close the database connection, but we demonstrate it here for the sake of completeness.

See Also

- Using the Slice Compilers
- slice2freeze Command-Line Options
- Map Concepts

slice2freeze Command-Line Options

The Slice-to-Freeze compiler, `slice2freeze`, generates C++ classes for Freeze maps and Freeze Evictor indices.

`slice2freeze`'s first command-line parameter is the base name for the generated C++ files, and not a Slice file like with the other Slice compilers:

```
slice2freeze [options] BaseName [SliceFile1 SliceFile2 ...]
```

The compiler offers the following command-line options in addition to the standard options:

`--header-ext` *`EXT`*

Changes the file extension for the generated header files from the default h to the extension specified by *`EXT`*.

`--source-ext` *`EXT`*

Changes the file extension for the generated source files from the default `cpp` to the extension specified by *`EXT`*.

`--add-header` *`HDR[,GUARD]`*

This option adds an include directive for the specified header at the beginning of the generated source file (preceding any other include directives). If *`GUARD`* is specified, the include directive is protected by the specified guard.

For example:

```
--add-header precompiled.h,_PRECOMPILED_H__
```

results in the following directives at the beginning of the generated source file:

C++

```
#ifndef __PRECOMPILED_H__
#define __PRECOMPILED_H__
#include <precompiled.h>
#endif
```

As this example demonstrates, the `--add-header` option is useful mainly for integrating the generated code with a compiler's precompiled header mechanism.

This option can be repeated to create include directives for several files.

`--include-dir` *`DIR`*

Modifies `#include` directives in source files to prepend the path name of each header file with the directory *`DIR`*. The discussion of `slice2cpp` provides more information.

`--dict` *`NAME,KEY,VALUE`*`[,sort[,`*`COMPARE`*`]]`

Generate a Freeze map class named *`NAME`* using *`KEY`* as key and *`VALUE`* as value. This option may be specified multiple times to generate several Freeze maps. *`NAME`* may be a scoped C++ name, such as `Demo::Struct1ObjectMap`. *`KEY`* and *`VALUE`* represent Slice types and therefore must use Slice syntax, such as `bool` or `Ice::Identity`. The type identified by *`KEY`* must be a legal dictionary key type. By default, keys are sorted using their binary Ice-encoded representation. Include `sort` to sort with the *`COMPARE`* functor class. If *`COMPARE`* is not specified, the default value is `std::less<`*`KEY`*`>`.

`--dict-index MAP[,`*`MEMBER`*`] [,case-sensitive|case-insensitive][,sort[,`*`COMPARE`*`]]`

Add an index to the Freeze map named *MAP*. If *MEMBER* is specified, the map value type must be a structure or a class, and *MEMBER* must be a member of this structure or class. Otherwise, the entire value is indexed. When the indexed member (or entire value) is a string, the index can be case-sensitive (default) or case-insensitive. An index adds additional member functions to the generated C++ map:

- `iterator findBy`*`MEMBER`*`(`*`MEMBER_TYPE`*`, bool = true);`
- `const_iterator findBy`*`MEMBER`*`(`*`MEMBER_TYPE`*`,bool = true) const;`
- `iterator beginFor`*`MEMBER`*`();`
- `const_iterator begin_For`*`MEMBER`*`() const;`
- `iterator endFor`*`MEMBER`*`();`
- `const_iterator endFor`*`MEMBER`*`() const;`
- `iterator lowerBoundFor`*`MEMBER`*`(`*`MEMBER_TYPE`*`);`
- `const_iterator lowerBoundFor`*`MEMBER`*`(`*`MEMBER_TYPE`*`) const;`
- `iterator upperBoundFor`*`MEMBER`*`(`*`MEMBER_TYPE`*`);`
- `const_iterator upperBoundFor`*`MEMBER`*`(`*`MEMBER_TYPE`*`) const;`
- `std::pair<iterator, iterator> equalRangeFor`*`MEMBER`*`(`*`MEMBER_TYPE`*`);`
- `std::pair<const_iterator, const_iterator> equalRangeFor`*`MEMBER`*`(`*`MEMBER_TYPE`*`) const;`
- `int `*`MEMBER`*`Count(`*`MEMBER_TYPE`*`) const;`

When *MEMBER* is not specified, these functions are `findByValue` (const and non-const), `lowerBoundForValue` (const and non-const), `valueCount`, and so on. When *MEMBER* is specified, its first letter is capitalized in the `findBy` function name. *MEMBER_TYPE* corresponds to an in-parameter of the type of *MEMBER* (or the type of the value when *MEMBER* is not specified). For example, if *MEMBER* is a string, *MEMBER_TYPE* is a `const std::string&`.

By default, keys are sorted using their binary Ice-encoded representation. Include `sort` to sort with the *COMPARE* functor class. If *COMPARE* is not specified, the default value is `std::less<`*`MEMBER_TYPE`*`>`.

`findBy`*`MEMBER`* returns an iterator to the first element in the Freeze map that matches the given index value. It returns `end()` if there is no match. When the second parameter is true (the default), the returned iterator provides only the elements with an exact match (and then skips to `end()`). Otherwise, the returned iterator sets a starting position and then provides all elements until the end of the map, sorted according to the index comparator.

`lowerBoundFor`*`MEMBER`* returns an iterator to the first element in the Freeze map whose index value is not less than the given index value. It returns `end()` if there is no such element. The returned iterator provides all elements until the end of the map, sorted according to the index comparator.

`upperBoundFor`*`MEMBER`* returns an iterator to the first element in the Freeze map whose index value is greater than the given index value. It returns `end()` if there is no such element. The returned iterator provides all elements until the end of the map, sorted according to the index comparator.

`beginFor`*`MEMBER`* returns an iterator to the first element in the map.

`endFor`*`MEMBER`* returns an iterator to the last element in the map.

`equalRangeFor`*`MEMBER`* returns a range (pair of iterators) of all the elements whose index value matches the given index value. This function is similar to `findBy`MEMBER (see above).

`MEMBER``Count` returns the number of elements in the Freeze map whose index value matches the given index value.

Please note that index-derived iterators do not allow you to set new values in the underlying map.

`--index `*`CLASS,TYPE,MEMBER`*` [,case-sensitive|case-insensitive]`

Generate an index class for a Freeze evictor. *CLASS* is the name of the class to be generated. *TYPE* denotes the type of class to be indexed (objects of different classes are not included in this index). *MEMBER* is the name of the data member in *TYPE* to index. When *MEMBER* has type `string`, it is possible to specify whether the index is case-sensitive or not. The default is case-sensitive.

See Also

- Using the Slice Compilers
- slice2cpp Command-Line Options (C++98)
- Map Concepts

Using a Map in Java

This page describes how to generate and use a Freeze map in a Java program.

On this page:

Generating a Simple Map for Java

We can use slice2freezej to generate a simple Freeze map:

```
$ slice2freezej --dict StringIntMap,string,int
```

This command directs the compiler to create a map named `StringIntMap`, with the Slice key type `string` and the Slice value type `int`. The compiler produces one Java source file: `StringIntMap.java`.

The Freeze Map Class in Java

The class generated by `slice2freezej` implements the `Freeze.Map` interface, as shown below:

Java

```
package Freeze;

public interface Map<K, V> extends NavigableMap<K, V>
{
    void fastPut(K key, V value);
    void close();
    int closeAllIterators();
    void destroy();

    public interface EntryIterator<T> extends java.util.Iterator<T>
    {
        void close();
        void destroy(); // an alias for close
    }
}
```

The `Map` interface implements standard Java interfaces and provides nonstandard methods that improve efficiency and support database-oriented features. `Map` defines the following methods:

- `fastPut`
  Inserts a new key-value pair. This method is more efficient than the standard `put` method because it avoids the overhead of reading and decoding the previous value associated with the key (if any).
- `close`
  Closes the database associated with this map along with all open iterators. A map must be closed when it is no longer needed, either by closing the map directly or by closing the Freeze `Connection` object with which this map is associated.

- `closeAllIterators`
  Closes all open iterators and returns the number of iterators that were closed. We discuss iterators in more detail in the next section.
- `destroy`
  Removes the database associated with this map along with any indices.

`Map` inherits much of its functionality from the `Freeze.NavigableMap` interface, which derives from the standard Java interface `java.util.SortedMap` and also supports a subset of the `java.util.NavigableMap` interface:

Java

```
package Freeze;

public interface NavigableMap<K, V> extends java.util.SortedMap<K, V>
{
    java.util.Map.Entry<K, V> firstEntry();
    java.util.Map.Entry<K, V> lastEntry();

    java.util.Map.Entry<K, V> ceilingEntry(K key);
    java.util.Map.Entry<K, V> floorEntry(K key);
    java.util.Map.Entry<K, V> higherEntry(K key);
    java.util.Map.Entry<K, V> lowerEntry(K key);

    K ceilingKey(K key);
    K floorKey(K key);
    K higherKey(K key);
    K lowerKey(K key);

    java.util.Set<K> descendingKeySet();
    NavigableMap<K, V> descendingMap();

    NavigableMap<K, V> headMap(K toKey, boolean inclusive);
    NavigableMap<K, V> tailMap(K fromKey, boolean inclusive);
    NavigableMap<K, V> subMap(K fromKey, boolean fromInclusive,
                              K toKey, boolean toInclusive);

    java.util.Map.Entry<K, V> pollFirstEntry();
    java.util.Map.Entry<K, V> pollLastEntry();

    boolean fastRemove(K key);
}
```

The `NavigableMap` interface provides a number of useful methods:

- `firstEntry`
  `lastEntry`
  Returns the first and last key-value pair, respectively.
- `ceilingEntry`
  Returns the key-value pair associated with the least key greater than or equal to the given key, or null if there is no such key.
- `floorEntry`
  Returns the key-value pair associated with the greatest key less than or equal to the given key, or null if there is no such key.
- `higherEntry`

Returns the key-value pair associated with the least key greater than the given key, or null if there is no such key.

- `lowerEntry`
  Returns the key-value pair associated with the greatest key less than the given key, or null if there is no such key.
- `ceilingKey`
  `floorKey`
  `higherKey`
  `lowerKey`
  These methods have the same semantics as those described above, except they return only the key portion of the matching key-value pair or null if there is no such key.
- `descendingKeySet`
  Returns a set representing a reverse-order view of the keys in this map.
- `descendingMap`
  Returns a reverse-order view of the entries in this map.
- `headMap`
  Returns a view of the portion of this map whose keys are less than (or equal to, if inclusive is true) the given key.
- `tailMap`
  Returns a view of the portion of this map whose keys are greater than (or equal to, if inclusive is true) the given key.
- `subMap`
  Returns a view of the portion of this map whose keys are within the given range.
- `pollFirstEntry`
  `pollLastEntry`
  Removes and returns the first and last key-value pair, respectively.
- `fastRemove`
  Removes an existing key-value pair. As for `fastPut`, this method is a more efficient alternative to the standard `remove` method that returns true if a key-value pair was removed, or false if no match was found.

> You must supply a comparator object when constructing the map in order to use many of these methods.

Note that `NavigableMap` also inherits overloaded methods named `headMap`, `tailMap`, and `subMap` from the `SortedMap` interface. These methods have the same semantics as the ones defined in `NavigableMap` but they omit the boolean arguments (refer to the JDK documentation for complete details). Although these methods are declared as returning a `SortedMap`, the actual type of the returned object is a `NavigableMap` that you can downcast if necessary.

There are some limitations in the sub maps returned by the `headMap`, `tailMap` and `subMap` methods:

- A new entry in the Freeze map cannot be added via a sub map, therefore calling `put` raises `UnsupportedOperationException`.
- An existing entry in the Freeze map cannot be removed via a sub map or iterator for a secondary key.

Now let us examine the contents of the source file created by the example in the previous section:

Java

```
public class StringIntMap extends ...
    // implements Freeze.Map<String, Integer>
{
    public StringIntMap(
        Freeze.Connection connection,
        String dbName,
        boolean createDb,
        java.util.Comparator<String> comparator);

    public StringIntMap(
        Freeze.Connection connection,
        String dbName,
        boolean createDb);

    public StringIntMap(
        Freeze.Connection connection,
        String dbName);
}
```

`StringIntMap` derives from an internal Freeze base class that implements the interface `Freeze.Map<String, Integer>`. The generated class defines several overloaded constructors whose arguments are described below:

- `connection`
  The Freeze connection object.
- `dbName`
  The name of the database in which to store this map's persistent state. Note that a database can only contain the persistent state of one map type. Any attempt to instantiate maps of different types on the same database results in undefined behavior.
- `createDb`
  A flag indicating whether the map should create the database if it does not already exist. If this argument is not specified, the default value is `true`.
- `comparator`
  An object used to compare the map's keys. If this argument is not specified, the default behavior compares the encoded form of the keys.

Why Comparators are Important

The constructor of a Freeze map optionally accepts a comparator object for the primary key and, if any indices are generated, a second object that supplies comparators for each of the index keys. If you do not supply a comparator, Freeze simply compares the encoded form of the keys. This default behavior is acceptable when comparing keys for equality, but using the encoded form cannot work reliably when comparing keys for ordering purposes.

For example, many of the methods in `NavigableMap` perform greater-than or less-than comparisons on keys, including `ceilingEntry`, `headMap`, and `tailMapFor`*`MEMBER`*. All of these methods raise `UnsupportedOperationException` if you failed to supply a corresponding comparator when constructing the map. (The same applies to `NavigableMap` objects created for secondary keys.) In fact, the only `NavigableMap` methods that do *not* require a comparator are `firstEntry`, `lastEntry`, `pollFirstEntry`, `pollLastEntry`, and `fastRemove`.

As you can see, the functionality of a Freeze map is quite limited if no comparators are configured, therefore we recommend using comparators at all times.

Using Iterators with Freeze Maps in Java

You can iterate over a Freeze map just as you can with any container that implements the `java.util.Map` interface. For example, the code below displays the key and value of each element:

Java

```
StringIntMap m = new StringIntMap(...);
java.util.Iterator<java.util.Map.Entry<String, Integer>> i =
m.entrySet().iterator();
while(i.hasNext())
{
    java.util.Map.Entry<String, Integer> e = i.next();
    System.out.println("Key: " + e.getKey());
    System.out.println("Value: " + e.getValue());
}
```

Generally speaking, a program should close an iterator when it is no longer necessary. (An iterator that is garbage collected without being closed emits a warning message.) However, an explicit close was not necessary in the preceding example because Freeze automatically closes a read-only iterator when it reaches the last element (a read-only iterator is one that is opened outside of any transaction). If instead our program had stopped using the iterator prior to reaching the last element, an explicit close would have been necessary:

Java

```
StringIntMap m = new StringIntMap(...);
java.util.Iterator<java.util.Map.Entry<String, Integer>> i =
m.entrySet().iterator();
while(i.hasNext())
{
    java.util.Map.Entry<String, Integer> e = i.next();
    System.out.println("Key: " + e.getKey());
    System.out.println("Value: " + e.getValue());
    if(e.getValue().intValue() == 5)
    {
        break;
    }
}
((Freeze.Map.EntryIterator)i).close();
```

Closing the iterator requires downcasting it to a Freeze-specific interface named `Freeze.Map.EntryIterator`. The definition of this interface was shown in the previous section.

Freeze maps also support the enhanced `for` loop functionality. Here is a simpler way to write our original program:

Java

```
StringIntMap m = new StringIntMap(...);
for(java.util.Map.Entry<String, Integer> e : m.entrySet())
{
    System.out.println("Key: " + e.getKey());
    System.out.println("Value: " + e.getValue());
}
```

As in the first example, Freeze automatically closes the iterator when no more elements are available. Although the enhanced `for` loop is convenient, it is not appropriate for all situations because the loop hides its iterator and therefore prevents the program from accessing the iterator in order to close it. In this case, you can use the traditional `while` loop instead of the `for` loop, or you can invoke `closeAllIterators` on the map as shown below:

Java

```
StringIntMap m = new StringIntMap(...);
for(java.util.Map.Entry<String, Integer> e : m.entrySet())
{
    System.out.println("Key: " + e.getKey());
    System.out.println("Value: " + e.getValue());
    if(e.getValue().intValue() == 5)
    {
        break;
    }
}
int num = m.closeAllIterators();
assert(num <= 1); // The iterator may already be closed.
```

The `closeAllIterators` method returns an integer representing the number of iterators that were actually closed. This value can be useful for diagnostic purposes, such as to assert that a program is correctly closing its iterators.

Generating Indices for Freeze Maps in Java

Using the `--dict-index` option to define an index for a secondary key causes `slice2freezej` to generate the following additional code in a Freeze map:

- A static nested class named `IndexComparators`, which allows you to supply a custom comparator object for each index in the map.
- An overloading of the map constructor that accepts an instance of `IndexComparators`.
- An overloading of the `recreate` method that accepts an instance of `IndexComparators`.
- Searching, counting, and range-searching methods for finding key-value pairs using the secondary key.

We discuss each of these additions in more detail below. In this discussion, *`MEMBER`* refers to the optional argument of the `--dict-index` option, and *`MEMBER_TYPE`* refers to the type of that member. As explained earlier, if *`MEMBER`* is not specified, `slice2freezej` creates an index for the value type of the map. The sample code presented in this section assumes we have generated a Freeze map using the following command:

Java

```
$ slice2freezej --dict StringIntMap,string,int --dict-index StringIntMap
```

By default, index keys are sorted using their binary Ice-encoded representation. This is an efficient sorting scheme but does not necessarily provide a meaningful traversal order for applications. You can choose a different order by providing an instance of the `IndexComparators` class to the map constructor. This class has a public data member holding a comparator (an instance of `java.util.Comparator<`*`MEMBER_TYPE`*`>`) for each index in the map. The class also provides an empty constructor as well as a convenience constructor that allows you to instantiate and initialize the object all at once. The name of each data member is *`MEMBER`*`Comparator`. If *`MEMBER`* is not specified, the `IndexComparators` class has a single data member named `valueComparator`.

> Much of the functionality offered by a map index requires that you provide a custom comparator.

Here is the definition of `IndexComparators` for `StringIntMap`:

Java

```
public class StringIntMap ...
{
    public static class IndexComparators
    {
        public IndexComparators() {}

        public IndexComparators(java.util.Comparator<Integer> valueComp
arator);

        public java.util.Comparator<Integer> valueComparator;
    }

    ...
}
```

To instantiate a Freeze map using your custom comparators, you must use the overloaded constructor that accepts the `IndexComparators` object. For our `StringIntMap`, this constructor has the following definition:

Java

```
public class StringIntMap ...
{
    public StringIntMap(
        Freeze.Connection connection,
        String dbName,
        boolean createDb,
        java.util.Comparator<String> comparator,
        IndexComparators indexComparators);

    ...
}
```

Now we can instantiate our `StringIntMap` as follows:

Java

```
java.util.Comparator<String> myMainKeyComparator = ...;
StringIntMap.IndexComparators indexComparators =
new StringIntMap.IndexComparators();
indexComparators.valueComparator = ...;
StringIntMap m = new StringIntMap(connection, "stringIntMap", true,

myMainKeyComparator, indexComparators);
```

If you later need to change the index configuration of a Freeze map, you can use one of the `recreate` methods to update the database. Here are the definitions from `StringIntMap`:

Java

```
public class StringIntMap ...
{
    public static void recreate(
        Freeze.Connection connection,
        String dbName,
        java.util.Comparator<String> comparator);

    public static void recreate(
        Freeze.Connection connection,
        String dbName,
        java.util.Comparator<String> comparator,
        IndexComparators indexComparators);

    ...
}
```

The first overloading is generated for every map, whereas the second overloading is only generated when the map has at least one index. As its name implies, the `recreate` method creates a new copy of the database. More specifically, the method removes any existing indices, copies every key-value pair to a temporary database, and finally replaces the old database with the new one. As a side-effect, this process also populates any remaining indices. The first overloading of `recreate` is useful when you have regenerated the map to remove the last index and wish to clean up the map's database state.

`slice2freezej` also generates a number of index-specific methods. The names of these methods incorporate the member name (*`MEMBER`*), or use `value` if *`MEMBER`* is not specified. In each method name, the value of *`MEMBER`* is used unchanged if it appears at the beginning of the method's name. Otherwise, if *`MEMBER`* is used elsewhere in the method name, its first letter is capitalized. The index methods are described below:

- `public Freeze.Map.EntryIterator<Map.Entry<K, V>>`
  `findBy`*`MEMBER`*`(`*`MEMBER_TYPE`*` index)`

  `public Freeze.Map.EntryIterator<Map.Entry<K, V>>`
  `findBy`*`MEMBER`*`(`*`MEMBER_TYPE`*` index, boolean onlyDups)`
  Returns an iterator over elements of the Freeze map starting with an element with whose index value matches the given index value. If there is no such element, the returned iterator is empty (`hasNext` always returns false). When the second parameter is true (or is not provided), the returned iterator provides only "duplicate" elements, that is, elements with the very same index value. Otherwise, the iterator sets a starting position in the map, and then provides elements until the end of the map, sorted according to the index comparator. Any attempt to modify the map via this iterator results in an `UnsupportedOperationException`.

- `public int MEMBERCount(MEMBER_TYPE index)`
  Returns the number of elements in the Freeze map whose index value matches the given index value.

- `public NavigableMap<MEMBER_TYPE, Set<Map.Entry<K, V>>>`
  `headMapForMEMBER(MEMBER_TYPE to, boolean inclusive)`

  `public NavigableMap<MEMBER_TYPE, Set<Map.Entry<K, V>>>`
  `headMapForMEMBER(MEMBER_TYPE to)`
  Returns a view of the portion of the Freeze map whose keys are less than (or equal to, if `inclusive` is true) the given key. If `inclusive` is not specified, the method behaves as if `inclusive` is false.

- `public NavigableMap<MEMBER_TYPE, Set<Map.Entry<K, V>>>`
  `tailMapForMEMBER(MEMBER_TYPE from, boolean inclusive)`

- public NavigableMap<*MEMBER_TYPE*, Set<Map.Entry<K, V>>>}}
  `tailMapForMEMBER(MEMBER_TYPE from)`
  Returns a view of the portion of the Freeze map whose keys are greater than (or equal to, if `inclusive` is true) the given key. If `inclusive` is not specified, the method behaves as if `inclusive` is true.

- `public NavigableMap<MEMBER_TYPE, Set<Map.Entry<K, V>>>`
  `subMapForMEMBER(MEMBER_TYPE from, boolean fromInclusive,`
  `MEMBER_TYPE to, boolean toInclusive)`

  `public NavigableMap<MEMBER_TYPE, Set<Map.Entry<K, V>>>`
  `subMapForMEMBER(MEMBER_TYPE from, MEMBER_TYPE to)`
  Returns a view of the portion of the Freeze map whose keys are within the given range. If `fromInclusive` and `toInclusive` are not specified, the method behaves as if `fromInclusive` is true and `toInclusive` is false.

- `public NavigableMap<MEMBER_TYPE, Set<Map.Entry<K, V>>>`
  `mapForMEMBER()`
  Returns a view of the entire Freeze map ordered by the index key.

For the methods returning a `NavigableMap`, the key type is the secondary key type and the value is the set of matching key-value pairs from the Freeze map. (For the sake of readability, we have omitted the `java.util` prefix from `Set` and `Map.Entry`.) In other words, the returned map is a mapping of the secondary key to all of the entries whose value contains the same key. Any attempt to add, remove, or modify an element via a sub map view or an iterator of a sub map view results in an `UnsupportedOperationException`.

Note that iterators returned by the `findBy`*MEMBER* methods, as well as those created for sub map views, may need to be closed explicitly, just like iterators obtained for the main Freeze map.

Here are the definitions of the index methods for `StringIntMap`:

Java

```
public Freeze.Map.EntryIterator<Map.Entry<String, Integer>>
findByValue(Integer index);

public Freeze.Map.EntryIterator<Map.Entry<String, Integer>>
findByValue(Integer index, boolean onlyDups);

public int valueCount(Integer index);

public NavigableMap<Integer, Set<Map.Entry<String, Integer>>>
headMapForValue(Integer to, boolean inclusive);
public NavigableMap<Integer, Set<Map.Entry<String, Integer>>>
headMapForValue(Integer to);

public NavigableMap<Integer, Set<Map.Entry<String, Integer>>>
tailMapForValue(Integer from, boolean inclusive);
public NavigableMap<Integer, Set<Map.Entry<String, Integer>>>
tailMapForValue(Integer from);

public NavigableMap<Integer, Set<Map.Entry<String, Integer>>>
subMapForValue(Integer from, boolean fromInclusive,
               Integer to, boolean toInclusive);
public NavigableMap<Integer, Set<Map.Entry<String, Integer>>>
subMapForValue(Integer from, Integer to);

public NavigableMap<Integer, Set<Map.Entry<String, Integer>>>
mapForValue();
```

Sample Freeze Map Program in Java

The program below demonstrates how to use a `StringIntMap` to store `<string, int>` pairs in a database. You will notice that there are no explicit `read` or `write` operations called by the program; instead, simply using the map has the side effect of accessing the database.

Java

```
public class Client
{
    public static void
    main(String[] args)
    {
        // Initialize the Communicator.
        //
        Ice.Communicator communicator = Ice.Util.initialize(args);

        // Create a Freeze database connection.
        //
        Freeze.Connection connection =
```

```
Freeze.Util.createConnection(communicator, "db");

        // Instantiate the map.
        //
        StringIntMap map = new StringIntMap(connection, "simple", true);

        // Clear the map.
        //
        map.clear();

        int i;

        // Populate the map.
        //
        for(i = 0; i < 26; i++)
        {
            final char[] ch = { (char)('a' + i) };
            map.put(new String(ch), i);
        }

        // Iterate over the map and change the values.
        //
        for(java.util.Map.Entry<String, Integer> e : map.entrySet())
        {
            Integer in = e.getValue();
            e.setValue(in.intValue() + 1);
        }

        // Find and erase the last element.
        //
        boolean b;
        b = map.containsKey("z");
        assert(b);
        b = map.fastRemove("z");
        assert(b);

        // Clean up.
        //
        map.close();
        connection.close();
        communicator.destroy();

        System.exit(0);
```

```
    }
}
```

Prior to instantiating a Freeze map, the application must connect to a Berkeley DB database environment:

Java

```
Freeze.Connection connection =
Freeze.Util.createConnection(communicator, "db");
```

The second argument is the name of a Berkeley DB database environment; by default, this is also the file system directory in which Berkeley DB creates all database and administrative files.

Next, the code instantiates the `StringIntMap` on the connection. The constructor's second argument supplies the name of the database file, and the third argument indicates that the database should be created if it does not exist:

Java

```
StringIntMap map = new StringIntMap(connection, "simple", true);
```

After instantiating the map, we clear it to make sure it is empty in case the program is run more than once:

Java

```
map.clear();
```

We populate the map, using a single-character string as the key. As with `java.util.Map`, the key and value types must be Java objects but the compiler takes care of autoboxing the integer argument:

Java

```
for(i = 0; i < 26; i++)
{
    final char[] ch = { (char)('a' + i) };
    map.put(new String(ch), i);
}
```

Iterating over the map is no different from iterating over any other map that implements the `java.util.Map` interface:

Java

```
for(java.util.Map.Entry<String, Integer> e : map.entrySet())
{
    Integer in = e.getValue();
    e.setValue(in.intValue() + 1);
}
```

Next, the program verifies that an element exists with key `z`, and then removes it using `fastRemove`:

Java

```
b = map.containsKey("z");
assert(b);
b = map.fastRemove("z");
assert(b);
```

Finally, the program closes the map and its connection.

Java

```
map.close();
connection.close();
```

See Also

- Using the Slice Compilers
- slice2freezej Command-Line Options
- Map Concepts

slice2freezej Command-Line Options

The Slice-to-Freeze compiler, `slice2freezej`, creates Java classes for Freeze maps. The compiler offers the following command-line options in addition to the standard options:

`--dict` *`NAME,KEY,VALUE`*

Generate a Freeze map class named *`NAME`* using *`KEY`* as key and *`VALUE`* as value. This option may be specified multiple times to generate several Freeze maps. *`NAME`* may be a scoped Java name, such as `Demo.Struct1ObjectMap`. *`KEY`* and *`VALUE`* represent Slice types and therefore must use Slice syntax, such as `bool` or `Ice::Identity`. The type identified by *`KEY`* must be a legal dictionary key type.

`--dict-index` *`MAP`*`[,`*`MEMBER`*`][,case-sensitive|case-insensitive]`

Add an index to the Freeze map named *`MAP`*. If *`MEMBER`* is specified, the map value type must be a structure or a class, and *`MEMBER`* must be the name of a member of that type. If *`MEMBER`* is not specified, the entire value is indexed. When the indexed member (or entire value) is a string, the index can be case-sensitive (default) or case-insensitive.

`--index` *`CLASS,TYPE,MEMBER`*`[,case-sensitive|case-insensitive]`

Generate an index class for a Freeze evictor. *`CLASS`* is the name of the index class to be generated. *`TYPE`* denotes the type of class to be indexed (objects of different classes are not included in this index). *`MEMBER`* is the name of the data member in *`TYPE`* to index. When *`MEMBER`* has type `string`, it is possible to specify whether the index is case-sensitive or not. The default is case-sensitive.

`--meta` *`META`*

Define the global metadata directive *`META`*. Using this option is equivalent to defining the global metadata *`META`* in each named Slice file, as well as in any file included by a named Slice file.

See Also

- Using the Slice Compilers
- slice2java Command-Line Options
- Map Concepts

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

See Also

- Default Servants
- Evictors

Adding a Map to the C++ File System Server

Here we present a C++ implementation of the file system server.

On this page:

Generating the File System Maps in C++

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

```
$ slice2freeze -I$(ICE_HOME)/slice -I. --ice --dict \
    FilesystemDB::IdentityFileEntryMap,Ice::Identity,\
    FilesystemDB::FileEntry \
    IdentityFileEntryMap FilesystemDB.ice \
    $(ICE_HOME)/slice/Ice/Identity.ice
$ slice2freeze -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 C++

The server's `main` program is very simple:

C++

```
#include <FilesystemI.h>
#include <IdentityFileEntryMap.h>
#include <IdentityDirectoryEntryMap.h>
#include <Freeze/Freeze.h>

using namespace std;
using namespace Filesystem;
using namespace FilesystemDB;

class FilesystemApp : public Ice::Application
{
public:

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

```
    virtual int run(int, char*[])
    {
        shutdownOnInterrupt();

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

        const Freeze::ConnectionPtr connection(Freeze::createConnection
(communicator(), _envName));

        const IdentityFileEntryMap fileDB(connection, FileI::filesDB());
        const IdentityDirectoryEntryMap dirDB(connection, DirectoryI::d
irectoriesDB());

        adapter->addDefaultServant(new FileI(communicator(), _envName),
 "file");
        adapter->addDefaultServant(new DirectoryI(communicator(), _envN
ame), "");

        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 code in detail. First, we are now including `IdentityFileEntry.h` and `IdentityDirectoryEntry.h`. These header files includes all of the other Freeze (and Ice) header files we need.

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`.

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:

C++

```
        const Freeze::ConnectionPtr connection(Freeze::createConnection
(communicator(), _envName));

        const IdentityFileEntryMap fileDB(connection, FileI::filesDB());
        const IdentityDirectoryEntryMap dirDB(connection,
DirectoryI::directoriesDB());

        adapter->addDefaultServant(new FileI(communicator(), _envName),
 "file");
        adapter->addDefaultServant(new DirectoryI(communicator(), _envN
ame), "");
```

`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.

The Servant Class Definitions in C++

The class definition for `FileI` is very simple:

C++

```
namespace Filesystem
{
    class FileI : public File
    {
    public:
        FileI(const Ice::CommunicatorPtr& communicator,
              const std::string& envName);

        // Slice operations...

        static std::string filesDB();

    private:
        void halt(const Freeze::DatabaseException& ex) const;

        const Ice::CommunicatorPtr _communicator;
        const std::string _envName;
    };
}
```

The `FileI` class stores the communicator and the environment name. These members are initialized by the constructor. The `filesDB` static member function 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 `DirectoryI` class looks very much the same, also storing the communicator and environment name. The `directoriesDB` static member function returns the name of the directory map.

C++

```
namespace Filesystem
{
    class DirectoryI : public Directory
    {
    public:
        DirectoryI(const Ice::CommunicatorPtr& communicator,
                   const std::string& envName);

        // Slice operations...

        static std::string directoriesDB();

    private:
        void halt(const Freeze::DatabaseException& ex) const;

        const Ice::CommunicatorPtr _communicator;
        const std::string _envName;
    };
}
```

Implementing `FileI` with a Freeze Map in C++

The `FileI` constructor and the `filesDB` and `halt` member functions have trivial implementations:

C++

```
FileI::FileI(const Ice::CommunicatorPtr& communicator,
             const string& envName) :
    _communicator(communicator), _envName(envName)
{
}

string
FileI::filesDB()
{
    return "files";
}

void
FileI::halt(const Freeze::DatabaseException& ex) const
{
    Ice::Error error(_communicator->getLogger());
    error << "fatal exception: " << ex
<< "\n*** Aborting application ***";

    abort();
}
```

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:

C++

```
string
FileI::someOperation(/* ... */ const Ice::Current& c)
{
    const Freeze::ConnectionPtr connection(Freeze::createConnection(_co
mmunicator, _envName));
    IdentityFileEntryMap fileDB(connection, filesDB());

    for(;;)
    {
       try
       {

            // Operation implementation here...

        }
        catch(const Freeze::DeadlockException&)
        {
            continue;
        }
        catch(const Freeze::DatabaseException& ex)
        {
            halt(ex);
        }
    }
}
```

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:

C++

```
string
FileI::name(const Ice::Current& c)
{
    const Freeze::ConnectionPtr connection(Freeze::createConnection(_co
mmunicator, _envName));
    IdentityFileEntryMap fileDB(connection, filesDB());

    for(;;)
     {
        try
        {
            IdentityFileEntryMap::iterator p = fileDB.find(c.id);
            if(p == fileDB.end())
            {
                throw Ice::ObjectNotExistException(__FILE__, __LINE__);
            }
            return p->second.name;
        }
        catch(const Freeze::DeadlockException&)
        {
            continue;
        }
        catch(const Freeze::DatabaseException& ex)
        {
            halt(ex);
        }
    }
}
```

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`:

C++

```
Lines
FileI::read(const Ice::Current& c)
{
    const Freeze::ConnectionPtr connection(Freeze::createConnection(_co
mmunicator, _envName));
    IdentityFileEntryMap fileDB(connection, filesDB());

    for(;;)
    {
        try
        {
            IdentityFileEntryMap::iterator p = fileDB.find(c.id);
            if(p == fileDB.end())
            {
                throw Ice::ObjectNotExistException(__FILE__, __LINE__);
            }
            return p->second.text;
        }
        catch(const Freeze::DeadlockException&)
        {
            continue;
        }
        catch(const Freeze::DatabaseException& ex)
        {
            halt(ex);
        }
    }
}
```

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

C++

```
void
FileI::write(const Filesystem::Lines& text, const Ice::Current& c)
{
    const Freeze::ConnectionPtr connection(Freeze::createConnection(_co
mmunicator, _envName));
    IdentityFileEntryMap fileDB(connection, filesDB());

    for(;;)
    {
        try
        {
            IdentityFileEntryMap::iterator p = fileDB.find(c.id);
            if(p == fileDB.end())
            {
                throw Ice::ObjectNotExistException(__FILE__, __LINE__);
            }
            FileEntry entry = p->second;
            entry.text = text;
            p.set(entry);
            break;
        }
        catch(const Freeze::DeadlockException&)
        {
            continue;
        }
        catch (const Freeze::DatabaseException& ex)
        {
            halt(ex);
        }
    }
}
```

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:

C++

```
void
FileI::destroy(const Ice::Current& c)
{
    const Freeze::ConnectionPtr connection(Freeze::createConnection(_co
mmunicator, _envName));
    IdentityFileEntryMap fileDB(connection, filesDB());
    IdentityDirectoryEntryMap dirDB(connection,
DirectoryI::directoriesDB());

    for(;;)
    {
        try
        {
            Freeze::TransactionHolder txn(connection);

            IdentityFileEntryMap::iterator p = fileDB.find(c.id);
            if(p == fileDB.end())
            {
                throw Ice::ObjectNotExistException(__FILE__, __LINE__);
            }
            FileEntry entry = p->second;

            IdentityDirectoryEntryMap::iterator pp =
dirDB.find(entry.parent);
            if(pp == dirDB.end())
             {
                halt(Freeze::DatabaseException(
                       __FILE__, __LINE__,
                       "consistency error: file without parent"));
            }

            DirectoryEntry dirEntry = pp->second;
            dirEntry.nodes.erase(entry.name);
            pp.set(dirEntry);

            fileDB.erase(p);
            txn.commit();
            break;
        }
        catch(const Freeze::DeadlockException&)
        {
            continue;
        }
        catch(const Freeze::DatabaseException& ex)
        {
            halt(ex);
        }
    }
}
```

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 `set` on the parent iterator and then removes the file from the file map before committing the transaction.

Implementing `DirectoryI` with a Freeze Map in C++

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:

C++

```
DirectoryI::DirectoryI(const Ice::CommunicatorPtr& communicator,
const string& envName) :
    _communicator(communicator), _envName(envName)
{
    const Freeze::ConnectionPtr connection =
Freeze::createConnection(_communicator, _envName);
    IdentityDirectoryEntryMap dirDB(connection, directoriesDB());

    for(;;)
    {
        try
        {
            Ice::Identity rootId;
            rootId.name = "RootDir";
            IdentityDirectoryEntryMap::const_iterator p =
dirDB.find(rootId);
            if(p == dirDB.end())
            {
                DirectoryEntry d;
                d.name = "/";
                dirDB.put(make_pair(rootId, d));
            }
            break;
        }
        catch(const Freeze::DeadlockException&)
        {
            continue;
        }
        catch(const Freeze::DatabaseException& ex)
        {
            halt(ex);
        }
    }
}
```

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:

C++

```
DirectoryPrx
DirectoryI::createDirectory(const string& name, const Ice::Current& c)
{
    const Freeze::ConnectionPtr connection(Freeze::createConnection(_co
mmunicator, _envName));
    IdentityDirectoryEntryMap directoryDB(connection, directoriesDB());
```

```
    for(;;)
    {
        try
        {
            Freeze::TransactionHolder txn(connection);

            IdentityDirectoryEntryMap::iterator p =
directoryDB.find(c.id);
            if(p == directoryDB.end())
            {
                throw Ice::ObjectNotExistException(__FILE__, __LINE__);
            }

            DirectoryEntry entry = p->second;
            if(name.empty()
|| entry.nodes.find(name) != entry.nodes.end())
            {
                throw NameInUse(name);
            }

            DirectoryEntry d;
            d.name = name;
            d.parent = c.id;

            Ice::Identity id;
            id.name = IceUtil::generateUUID();
            DirectoryPrx proxy = DirectoryPrx::uncheckedCast(c.adapter-
>createProxy(id));

            NodeDesc nd;
            nd.name = name;
            nd.type = DirType;
            nd.proxy = proxy;
            entry.nodes.insert(make_pair(name, nd));

            p.set(entry);
            directoryDB.put(make_pair(id, d));

            txn.commit();

            return proxy;
        }
        catch(const Freeze::DeadlockException&)
        {
            continue;
        }
        catch(const Freeze::DatabaseException& ex)
        {
            halt(ex);
```

```
}
```

```
        }
    }
```

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`:

C++

```
NodeDescSeq
DirectoryI::list(const Ice::Current& c)
{
    const Freeze::ConnectionPtr connection(Freeze::createConnection(_co
mmunicator, _envName));
    IdentityDirectoryEntryMap directoryDB(connection, directoriesDB());

    for(;;)
    {
        try
        {
            IdentityDirectoryEntryMap::iterator p =
directoryDB.find(c.id);
            if(p == directoryDB.end())
            {
                throw Ice::ObjectNotExistException(__FILE__, __LINE__);
            }
            NodeDescSeq result;
            for(StringNodeDescDict::const_iterator q =
p->second.nodes.begin(); q != p->second.nodes.end(); ++q)
            {
                result.push_back(q->second);
            }
            return result;
        }
        catch(const Freeze::DeadlockException&)
        {
            continue;
        }
        catch(const Freeze::DatabaseException& ex)
        {
            halt(ex);
        }
    }
}
```

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

- Maps
- Object Identity and Uniqueness
- The Current Object

Adding a 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.di
rectoriesDB(), true);

            adapter.addDefaultServant(new FileI(communicator(), _envNam
e), "file");
            adapter.addDefaultServant(new DirectoryI(communicator(), _e
nvName), "");

            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

```
    public Server(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(), t
rue);
            IdentityDirectoryEntryMap dirDB =
                new IdentityDirectoryEntryMap(connection, DirectoryI.di
rectoriesDB(), true);

            adapter.addDefaultServant(new FileI(communicator(), _envNam
e), "file");
            adapter.addDefaultServant(new DirectoryI(communicator(), _e
nvName), "");
```

`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` stati c 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.di
rectoriesDB());

            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.r
andomUUID().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

- Maps
- The Current Object
- Object Identity and Uniqueness

Catalogs

In each database environment, Freeze maintains an internal table that contains type information describing all the databases in the environment. This table is an instance of a Freeze map in which the key is a string representing the database name and the value is an instance of `Freeze::CatalogData`:

Slice

```
module Freeze
{
    struct CatalogData
    {
        bool evictor;
        string key;
        string value;
    }
}
```

An entry describes an evictor database if the `evictor` member is true, in which case the `key` and `value` members are empty strings. An entry that describes a Freeze map sets `evictor` to false; the `key` and `value` members contain the Slice types used when the map was defined.

FreezeScript tools such as `transformdb` and `dumpdb` access the catalog to obtain type information when none is supplied by the user. You can also use `dumpdb` to display the catalog of a database environment.

Freeze applications may access the catalog in the same manner as any other Freeze map. For example, the following C++ code displays the contents of a catalog:

C++

```
#include <Freeze/Freeze.h>
...
string envName = ...;
Freeze::ConnectionPtr conn =
Freeze::createConnection(communicator, envName);
Freeze::Catalog catalog(conn, Freeze::catalogName());
for(Freeze::Catalog::const_iterator p = catalog.begin();
p != catalog.end(); ++p)
{
    if(p->second.evictor)
    {
        cout << p->first << ": evictor" << endl;
    }
    else
    {
        cout << p->first << ": map<" << p->second.key
             << ", " << p->second.value << ">" << endl;
    }
}
conn->close();
```

The equivalent Java code is shown below:

Java

```
String envName = ...;
Freeze.Connection conn =
Freeze.Util.createConnection(communicator, envName);
Freeze.Catalog catalog =
new Freeze.Catalog(conn, Freeze.Util.catalogName(), true);
for(java.util.Map.Entry<String, Freeze.CatalogData> e :
catalog.entrySet())
{
    String name = e.getKey();
    Freeze.CatalogData data = e.getValue();
    if(data.evictor)
    {
        System.out.println(name + ": evictor");
    }
    else
    {
        System.out.println(name + ": map<" + data.key + ", " +
data.value + ">");
    }
}
conn.close();
```

See Also

- Maps
- Evictors
- FreezeScript

Creating Backups

When you store important information in a Freeze database environment, you should consider regularly backing up the database environment.

There are two forms of backups: cold backups, where you just copy your database environment directory while no application is using these files (very straightforward), and hot backups, where you backup a database environment while an application is actively reading and writing data.

In order to perform a hot backup on a Freeze environment, you need to configure this Freeze environment with two non-default settings:

- `Freeze.DbEnv.envName.OldLogsAutoDelete=0`
  This instructs Freeze to keep old log files instead of periodically deleting them. This setting is necessary for proper hot backups; it implies that you will need to take care of deleting old files yourself (typically as part of your periodic backup procedure).
- `Freeze.DbEnv.envName.DbPrivate=0`
  By default, Freeze is configured with `DbPrivate` set to 1, which means only one process at a time can safely access the database environment. When performing hot backups, you need to access this database environment concurrently from various Berkeley DB utilities (such as `db_archive` or `db_hotbackup`), so you need to set this property to 0.

The `Freeze/backup` C++ demo shows one way to perform such backups and recovery. Please consult the Berkeley DB documentation for further details.

See Also

- Freeze

FreezeScript

Freeze supplies a valuable set of services for simplifying the use of persistence in Ice applications. However, while Freeze makes it easy for an application to manage its persistent state, there are additional administrative responsibilities that must also be addressed:

- Migration
  As an application evolves, it is not unusual for the types describing its persistent state to evolve as well. When these changes occur, a great deal of time can be saved if existing databases can be migrated to the new format while preserving as much information as possible.
- Inspection
  The ability to examine a database can be helpful during every stage of the application's lifecycle, from development to deployment.

FreezeScript provides tools for performing both of these activities on Freeze map and evictor databases. These databases have a well-defined structure because the key and value of each record consist of the marshaled bytes of their respective Slice types. This design allows the FreezeScript tools to operate on any Freeze database using only the Slice definitions for the database types.

Topics

Migrating a Database

The FreezeScript tool `transformdb` migrates a database created by a Freeze map or evictor. It accomplishes this by comparing the "old" Slice definitions (i.e., the ones that describe the current contents of the database) with the "new" Slice definitions, and making whatever modifications are necessary to ensure that the transformed database is compatible with the new definitions.

This would be difficult to achieve by writing a custom transformation program because that program would require static knowledge of the old and new types, which frequently define many of the same symbols and would therefore prevent the program from being loaded. The `transformdb` tool avoids this issue using an interpretive approach: the Slice definitions are parsed and used to drive the migration of the database records.

The tool supports two modes of operation:

1. Automatic migration – the database is migrated in a single step using only the default set of transformations.
2. Custom migration – you supply a script to augment or override the default transformations.

Topics

- Automatic Database Migration
- Custom Database Migration
- Transformation XML Reference
- Using transformdb

See Also

- Maps
- Evictors

Automatic Database Migration

On this page:

The default transformations performed by `transformdb` preserve as much information as possible. However, there are practical limits to the tool's capabilities, since the only information it has is obtained by performing a comparison of the Slice definitions.

For example, suppose our old definition for a structure is the following:

Slice

```
struct AStruct
{
    int i;
}
```

We want to migrate instances of this struct to the following revised definition:

Slice

```
struct AStruct
{
    int j;
}
```

As the developers, we know that the `int` member has been renamed from `i` to `j`, but to `transformdb` it appears that member `i` was removed and member `j` was added. The default transformation results in exactly that behavior: the value of `i` is lost, and `j` is initialized to a default value. If we need to preserve the value of `i` and transfer it to `j`, then we need to use custom migration.

The changes that occur as a type system evolves can be grouped into three categories:

- Data members
  The data members of class and structure types are added, removed, or renamed. As discussed above, the default transformations initialize new and renamed data members to default values.
- Type names
  Types are added, removed, or renamed. New types do not pose a problem for database migration when used to define a new data member; the member is initialized with default values as usual. On the other hand, if the new type replaces the type of an existing data member, then type compatibility becomes a factor (see the following item).

  Removed types generally do not cause problems either, because any uses of that type must have been removed from the new Slice definitions (e.g., by removing data members of that type). There is one case, however, where removed types become an issue, and that is for polymorphic classes.

  Renamed types are a concern, just like renamed data members, because of the potential for losing information during migration. This is another situation for which custom migration is recommended.
- Type content
  Examples of changes of type content include the key type of a dictionary, the element type of a sequence, or the type of a data member. If the old and new types are not compatible, then the default transformation emits a warning, discards the current value, and reinitializes it with a default value.

Type Compatibility Rules for Automatic Migration

Changes in the type of a value are restricted to certain sets of compatible changes. This section describes the type changes supported by

the default transformations. All incompatible type changes result in a warning indicating that the current value is being discarded and a default value for the new type assigned in its place. Additional flexibility is provided by custom migration.

Boolean

A value of type `bool` can be transformed to and from `string`. The legal string values for a `bool` value are `"true"` and `"false"`.

Integer

The integer types `byte`, `short`, `int`, and `long` can be transformed into each other, but only if the current value is within range of the new type. These integer types can also be transformed into `string`.

Floating Point

The floating-point types `float` and `double` can be transformed into each other, as well as to `string`. No attempt is made to detect a loss of precision during transformation.

String

A `string` value can be transformed into any of the primitive types, as well as into enumeration and proxy types, but only if the value is a legal string representation of the new type. For example, the string value `"Pear"` can be transformed into the enumeration `Fruit`, but only if `Pear` is an enumerator of `Fruit`.

Enum

An enumeration can be transformed into an enumeration with the same type ID, or into a string. Transformation between enumerations is performed symbolically. For example, consider our old type below:

Slice

```
enum Fruit { Apple, Orange, Pear }
```

Suppose the enumerator `Pear` is being transformed into the following new type:

Slice

```
enum Fruit { Apple, Pear }
```

The transformed value in the new enumeration is also `Pear`, despite the fact that `Pear` has changed positions in the new type. However, if the old value had been `Orange`, then the default transformation emits a warning because that enumerator no longer exists, and initializes the new value to `Apple` (the default value).

If an enumerator has been renamed, then custom migration is required to convert enumerators from the old name to the new one.

Sequence

A sequence can be transformed into another sequence type, even if the new sequence type does not have the same type id as the old type, but only if the element types are compatible. For example, `sequence<short>` can be transformed into `sequence<int>`, regardless of the names given to the sequence types.

Dictionary

A dictionary can be transformed into another dictionary type, even if the new dictionary type does not have the same type ID as the old type, but only if the key and value types are compatible. For example, `dictionary<int, string>` can be transformed into `dictionary<long, string>`, regardless of the names given to the dictionary types.

Caution is required when changing the key type of a dictionary, because the default transformation of keys could result in duplication. For example, if the key type changes from `int` to `short`, any `int` value outside the range of `short` results in the key being initialized to a

default value (namely zero). If zero is already used as a key in the dictionary, or another out-of-range key is encountered, then a duplication occurs. The transformation handles key duplication by removing the duplicate element from the transformed dictionary. (Custom migration can be useful in these situations if the default behavior is not acceptable.)

Structure

A `struct` type can only be transformed into another `struct` type with the same type ID. Data members are transformed as appropriate for their types.

Proxy

A proxy value can be transformed into another proxy type, or into `string`. Transformation into another proxy type is done with the same semantics as in a language mapping: if the new type does not match the old type, then the new type must be a base type of the old type (that is, the proxy is widened).

Class

A `class` type can only be transformed into another `class` type with the same type ID. A data member of a `class` type is allowed to be widened to a base type. Data members are transformed as appropriate for their types. See Transforming Objects for more information on transforming classes.

Default Values for Automatic Migration

Data types are initialized with default values, as shown.

Type	Default Value
Boolean	`false`
Numeric	Zero (`0`)
String	Empty string
Enumeration	The first enumerator
Sequence	Empty sequence
Dictionary	Empty dictionary
Struct	Data members are initialized recursively
Proxy	Nil
Class	Nil

Running an Automatic Migration

In order to use automatic transformation, we need to supply the following information to `transformdb`:

- The old and new Slice definitions
- The old and new types for the database key and value
- The database environment directory, the database file name, and the name of a new database environment directory to hold the transformed database

Here is an example of a `transformdb` command:

```
$ transformdb --old old/MyApp.ice --new new/MyApp.ice --key int,string
--value ::Employee db emp.db newdb
```

Briefly, the `--old` and `--new` options specify the old and new Slice definitions, respectively. These options can be specified as many times as necessary in order to load all of the relevant definitions. The `--key` option indicates that the database key is evolving from `int` to `strin`

g. The `--value` option specifies that `::Employee` is used as the database value type in both old and new type definitions, and therefore only needs to be specified once. Finally, we provide the pathname of the database environment directory (`db`), the file name of the database (`emp.db`), and the pathname of the database environment directory for the transformed database (`newdb`).

See Also

- Custom Database Migration
- Type IDs
- Using transformdb

Custom Database Migration

Custom migration is useful when your types have changed in ways that make automatic migration difficult or impossible. It is also convenient to use custom migration when you have complex initialization requirements for new types or new data members, because custom migration enables you to perform many of the same tasks that would otherwise require you to write a throwaway program.

Custom migration operates in conjunction with automatic migration, allowing you to inject your own transformation rules at well-defined intercept points in the automatic migration process. These rules are called *transformation descriptors*, and are written in XML.

On this page:

Simple Example of Custom Migration

We can use a simple example to demonstrate the utility of custom migration. Suppose our application uses a Freeze map whose key type is `string` and whose value type is an enumeration, defined as follows:

Slice

```
enum BigThree { Ford, Chrysler, GeneralMotors }
```

We now wish to rename the enumerator `Chrysler`, as shown in our new definition:

Slice

```
enum BigThree { Ford, FCA, GeneralMotors }
```

According to the rules for default transformations, all occurrences of the `Chrysler` enumerator would be transformed into `Ford`, because `Chrysler` no longer exists in the new definition and therefore the default value `Ford` is used instead.

To remedy this situation, we use the following transformation descriptors:

XML

```
<transformdb>
    <database key="string" value="::BigThree">
        <record>
            <if test="oldvalue == ::Old::Chrysler">
                <set target="newvalue" value="::New::FCA"/>
            </if>
        </record>
    </database>
</transformdb>
```

When executed, these descriptors convert occurrences of `Chrysler` in the old type system into `FCA` in the transformed database's new type system.

Overview of Transformation Descriptors

As we saw in the previous example, FreezeScript transformation descriptors are written in XML.

A transformation descriptor file has a well-defined structure. The top-level descriptor in the file is `<transformdb>`. A `<database>` descriptor must be present within `<transformdb>` to define the key and value types used by the database. Inside `<database>`, the `<record>` descriptor triggers the transformation process.

During transformation, type-specific actions are supported by the `<transform>` and `<init>` descriptors, both of which are children of `<transformdb>`. One `<transform>` descriptor and one `<init>` descriptor may be defined for each type in the new Slice definitions. Each time `transformdb` creates a new instance of a type, it executes the `<init>` descriptor for that type, if one is defined. Similarly, each time `transformdb` transforms an instance of an old type into a new type, the `<transform>` descriptor for the new type is executed.

The `<database>`, `<record>`, `<transform>`, and `<init>` descriptors may contain general-purpose action descriptors such as `<if>`, `<set>`, and `<echo>`. These actions resemble statements in programming languages like C++ and Java, in that they are executed in the order of definition and their effects are cumulative. Actions can make use of the expression language that should look familiar to C++ and Java programmers.

Transformation Flow of Execution

The transformation descriptors are executed as follows:

- `<database>` is executed first. Each child descriptor of `<database>` is executed in the order of definition. If a `<record>` descriptor is present, database transformation occurs at that point. Any child descriptors of `<database>` that follow `<record>` are not executed until transformation completes.
- During transformation of each record, `transformdb` creates instances of the new key and value types, which includes the execution of the `<init>` descriptors for those types. Next, the old key and value are transformed into the new key and value, in the following manner:
    1. Locate the `<transform>` descriptor for the type.
    2. If no descriptor is found, or the descriptor exists and it does not preclude default transformation, then transform the data as in automatic database migration.
    3. If the `<transform>` descriptor exists, execute it.
    4. Finally, execute the child descriptors of `<record>`.

Transformation Descriptor Scopes

The `<database>` descriptor creates a global scope, allowing child descriptors of `<database>` to define symbols that are accessible in any descriptor.

> In order for a global symbol to be available to a `<transform>` or `<init>` descriptor, the symbol must be defined before the `<record>` descriptor is executed.

Furthermore, certain other descriptors create local scopes that exist only for the duration of the descriptor's execution. For example, the `<transform>` descriptor creates a local scope and defines the symbols `old` and `new` to represent a value in its old and new forms. Child descriptors of `<transform>` can also define new symbols in the local scope, as long as those symbols do not clash with an existing symbol in that scope. It is legal to add a new symbol with the same name as a symbol in an outer scope, but the outer symbol will not be accessible during the descriptor's execution.

The global scope is useful in many situations. For example, suppose you want to track the number of times a certain value was encountered during transformation. This can be accomplished as shown below:

XML

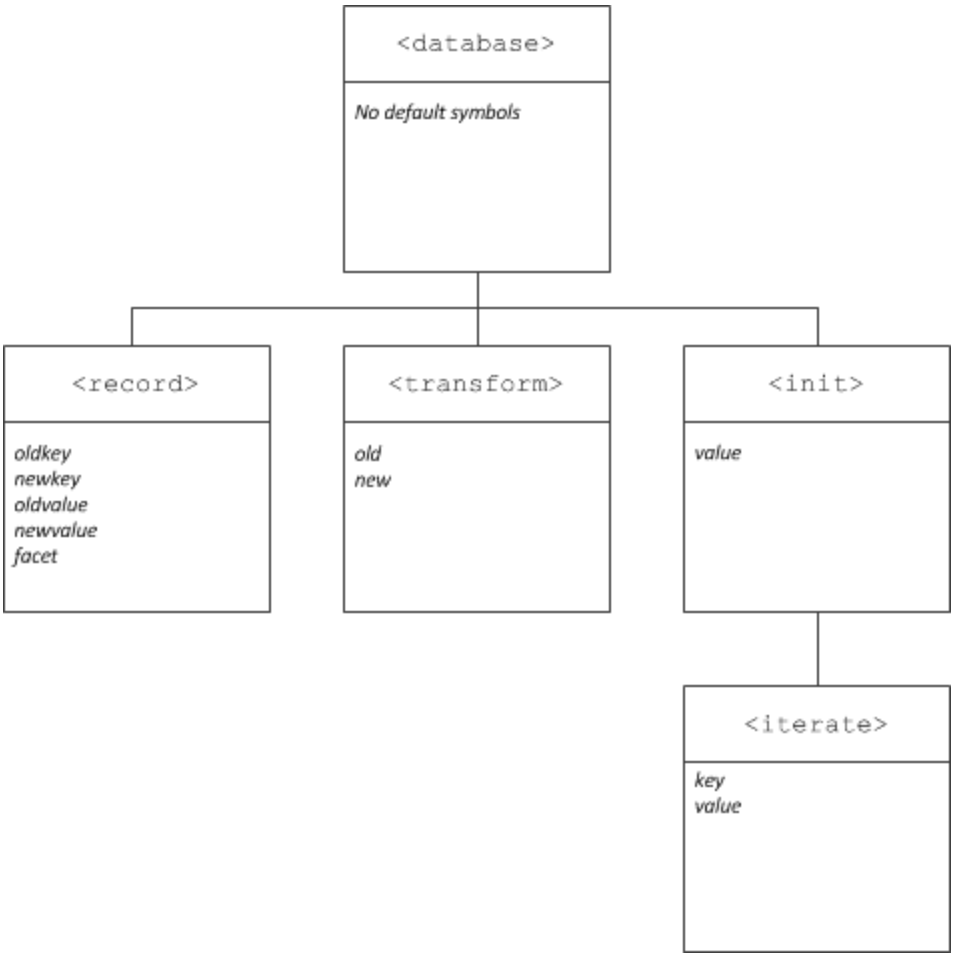
```
<transformdb>
    <database key="string" value="::Ice::Identity">
        <define name="categoryCount" type="int" value="0"/>
        <record/>
        <echo message="categoryCount = " value="categoryCount"/>
    </database>
    <transform type="::Ice::Identity">
        <if test="new.category == 'Accounting'">
            <set target="categoryCount" value="categoryCount + 1"/>
        </if>
    </transform>
</transformdb>
```

In this example, the `<define>` descriptor introduces the symbol `categoryCount` into the global scope, defining it as type `int` with an initial value of zero. Next, the `<record>` descriptor causes transformation to proceed. Each occurrence of the type `Ice::Identity` causes its `<transform>` descriptor to be executed, which examines the `category` member and increases `categoryCount` if necessary. Finally, after transformation completes, the `<echo>` descriptor displays the final value of `categoryCount`.

To reinforce the relationships between descriptors and scopes, consider the following diagram. Several descriptors are shown, including the symbols they define in their local scopes. In this example, the `<iterate>` descriptor has a dictionary target and therefore the default symbol for the element value, `value`, hides the symbol of the same name in the parent `<init>` descriptor's scope.

> This situation can be avoided by assigning a different symbol name to the element value.

In addition to symbols in the `<iterate>` scope, child descriptors of `<iterate>` can also refer to symbols from the `<init>` and `<database>` scopes.



Relationship between descriptors and scopes.

Guidelines for Transformation Descriptors

There are three points at which you can intercept the transformation process: when transforming a record (`<record>`), when transforming an instance of a type (`<transform>`), and when creating an instance of a type (`<init>`).

In general, `<record>` is used when your modifications require access to both the key and value of the record. For example, if the database key is needed as a factor in an equation, or to identify an element in a dictionary, then `<record>` is the only descriptor in which this type of modification is possible. The `<record>` descriptor is also convenient to use when the number of changes to be made is small, and does not warrant the effort of writing separate `<transform>` or `<init>` descriptors.

The `<transform>` descriptor has a more limited scope than `<record>`. It is used when changes must potentially be made to all instances of a type (regardless of the context in which that type is used) and access to the old value is necessary. The `<transform>` descriptor does not have access to the database key and value, therefore decisions can only be made based on the old and new instances of the type in question.

Finally, the `<init>` descriptor is useful when access to the old instance is not required in order to properly initialize a type. In most cases, this activity could also be performed by a `<transform>` descriptor that simply ignored the old instance, so `<init>` may seem redundant. However, there is one situation where `<init>` is required: when it is necessary to initialize an instance of a type that is introduced by the new Slice definitions. Since there are no instances of this type in the current database, a `<transform>` descriptor for that type would never be executed.

See Also

- Automatic Database Migration
- Maps
- Transformation XML Reference

- Descriptor Expression Language

Transformation XML Reference

This page describes the XML elements comprising the FreezeScript transformation descriptors.

On this page:

`<transformdb>` Descriptor Element

The top-level descriptor in a descriptor file. It requires at least one `<database>` descriptor, and supports any number of `<transform>` and `<init>` child descriptors. This descriptor has no attributes.

`<database>` Descriptor Element

The attributes of this descriptor define the old and new key and value types for the database to be transformed, and optionally the name of the database to which these types apply. It supports any number of child descriptors, but at most one `<record>` descriptor. The `<database>` descriptor also creates a global scope for user-defined symbols.

The attributes supported by the `<database>` descriptor are described in the following table:

Name	Description
`name`	Specifies the name of the database defined by this descriptor. (Optional)
`key`	Specifies the Slice types of the old and new keys. If the types are the same, only one needs to be specified. Otherwise, the types are separated by a comma.
`value`	Specifies the Slice types of the old and new values. If the types are the same, only one needs to be specified. Otherwise, the types are separated by a comma.

As an example, consider the following `<database>` descriptor. In this case, the Freeze map to be transformed currently has key type `int` and value type `::Employee`, and is migrating to a key type of `string`:

XML

```
<database key="int,string" value="::Employee">
```

`<record>` Descriptor Element

Commences the transformation. Child descriptors are executed for each record in the database, providing the user with an opportunity to examine the record's old key and value, and optionally modify the new key and value. Default transformations, as well as `<transform>` and `<init>` descriptors, are executed before the child descriptors. The `<record>` descriptor introduces the following symbols into a local scope: `oldkey`, `newkey`, `oldvalue`, `newvalue`, `facet`. These symbols are accessible to child descriptors, but not to `<transform>` or `<init>` descriptors. The `oldkey` and `oldvalue` symbols are read-only. The `facet` symbol is a string indicating the facet name of the object in the current record, and is only relevant for Freeze evictor databases.

Use caution when modifying database keys to ensure that duplicate keys do not occur. If a duplicate database key is encountered, transformation fails immediately.

Note that database transformation only occurs if a `<record>` descriptor is present.

`<transform>` Descriptor Element

Customizes the transformation for all instances of a type in the *new* Slice definitions. The children of this descriptor are executed after the optional default transformation has been performed. Only one `<transform>` descriptor can be specified for a type, but a `<transform>` descriptor is not required for every type. The symbols `old` and `new` are introduced into a local scope and represent the old and new values, respectively. The `old` symbol is read-only. The attributes supported by this descriptor are described in the following table:

Name	Description
`type`	Specifies the Slice type ID for the type's new definition.
`default`	If `false`, no default transformation is performed on values of this type. If not specified, the default value is `true`.
`base`	This attribute determines whether `<transform>` descriptors of base class types are executed. If `true`, the `<transform>` descriptor of the immediate base class is invoked. If no descriptor is found for the immediate base class, the class hierarchy is searched until a descriptor is found. The execution of any base class descriptors occurs after execution of this descriptor's children. If not specified, the default value is `true`.
`rename`	Indicates that a type in the old Slice definitions has been renamed to the new type identified by the `type` attribute. The value of this attribute is the type ID of the old Slice definition. Specifying this attribute relaxes the strict compatibility rules for `enum`, `struct` and `class` types.

Below is an example of a `<transform>` descriptor that initializes a new data member:

XML

```
<transform type="::Product">
    <set target="new.salePrice" value="old.listPrice * old.discount"/>
</transform>
```

For class types, `transformdb` first attempts to locate a `<transform>` descriptor for the object's most-derived type. If no descriptor is found, `transformdb` proceeds up the class hierarchy in an attempt to find a descriptor. The base object type, `Object`, is the root of every class hierarchy and is included in the search for descriptors. It is therefore possible to define a `<transform>` descriptor for type `Object`, which will be invoked for every class instance.

Note that `<transform>` descriptors are executed recursively. For example, consider the following Slice definitions:

Slice

```
struct Inner {
    int sum;
};
struct Outer {
    Inner i;
};
```

When `transformdb` is performing the default transformation on a value of type `Outer`, it recursively performs the default transformation on the `Inner` member, then executes the `<transform>` descriptor for `Inner`, and finally executes the `<transform>` descriptor for `Outer`. However, if default transformation is disabled for `Outer`, then no transformation is performed on the `Inner` member and therefore the `<transform>` descriptor for `Inner` is not executed.

`<init>` Descriptor Element

Defines custom initialization rules for all instances of a type in the new Slice definitions. Child descriptors are executed each time the type is instantiated. The typical use case for this descriptor is for types that have been introduced in the new Slice definitions and whose instances require default values different than what `transformdb` supplies. The symbol `value` is introduced into a local scope to represent the instance. The attributes supported by this descriptor are described in the following table:

Name	Description
`type`	Specifies the Slice type ID of the type's new definition.

Here is a simple example of an `<init>` descriptor:

XML

```
<init type="::Player">
    <set target="value.currency" value="100"/>
</init>
```

Note that, like `<transform>`, `<init>` descriptors are executed recursively. For example, if an `<init>` descriptor is defined for a `struct` type, the `<init>` descriptors of the `struct`'s members are executed before the `struct`'s descriptor.

`<iterate>` Descriptor Element

Iterates over a dictionary or sequence, executing child descriptors for each element. The symbol names selected to represent the element information may conflict with existing symbols in the enclosing scope, in which case those outer symbols are not accessible to child descriptors. The attributes supported by this descriptor are described in the following table:

Name	Description
`target`	The sequence or dictionary.
`index`	The symbol name used for the sequence index. If not specified, the default symbol is `i`.
`element`	The symbol name used for the sequence element. If not specified, the default symbol is `elem`.
`key`	The symbol name used for the dictionary key. If not specified, the default symbol is `key`.
`value`	The symbol name used for the dictionary value. If not specified, the default symbol is `value`.

Shown below is an example of an `<iterate>` descriptor that sets the new data member `reviewSalary` to `true` if the employee's salary is greater than $3000:

XML

```
<iterate target="new.employeeMap" key="id" value="emp">
    <if test="emp.salary > 3000">
        <set target="emp.reviewSalary" value="true"/>
    </if>
</iterate>
```

`<if>` Descriptor Element

Conditionally executes child descriptors. The attributes supported by this descriptor are described in the following table:

Name	Description
`test`	A boolean expression.

Child descriptors are executed if the expression in `test` evaluates to true.

<set> Descriptor Element

Modifies a value. The `value` and `type` attributes are mutually exclusive. If `target` denotes a dictionary element, that element must already exist (i.e., `<set>` cannot be used to add an element to a dictionary). The attributes supported by this descriptor are described in the following table:

Name	Description
`target`	An expression that must select a modifiable value.
`value`	An expression that must evaluate to a value compatible with the target's type.
`type`	If specified, set the target to be an instance of the given Slice class. The value is a type ID from the new Slice definitions. The class must be compatible with the target's type.
`length`	An integer expression representing the desired new length of a sequence. If the new length is less than the current size of the sequence, elements are removed from the end of the sequence. If the new length is greater than the current size, new elements are added to the end of the sequence. If `value` or `type` is also specified, it is used to initialize each new element.
`convert`	If `true`, additional type conversions are supported: between integer and floating point, and between integer and enumeration. Transformation fails immediately if a range error occurs. If not specified, the default value is `false`.

The `<set>` descriptor below modifies a member of a dictionary element:

XML

```
<set target="new.parts['P105J3'].cost"
value="new.parts['P105J3'].cost * 1.05"/>
```

This `<set>` descriptor adds an element to a sequence and initializes its value:

XML

```
<set target="new.partsList" length="new.partsList.length + 1"
value="'P105J3'"/>
```

As another example, the following `<set>` descriptor changes the value of an enumeration:

XML

```
<set target="new.ingredient" value="::New::Apple"/>
```

Notice in this example that the value refers to a symbol in the new Slice definitions.

<add> Descriptor Element

Adds a new element to a sequence or dictionary. It is legal to add an element while traversing the sequence or dictionary using `<iterate>`, however the traversal order after the addition is undefined. The `key` and `index` attributes are mutually exclusive, as are the `value` and `typ`

e attributes. If neither `value` nor `type` is specified, the new element is initialized with a default value. The attributes supported by this descriptor are described in the following table:

Name	Description
`target`	An expression that must select a modifiable sequence or dictionary.
`key`	An expression that must evaluate to a value compatible with the target dictionary's key type.
`index`	An expression that must evaluate to an integer value representing the insertion position. The new element is inserted before `index`. The value must not exceed the length of the target sequence.
`value`	An expression that must evaluate to a value compatible with the target dictionary's value type, or the target sequence's element type.
`type`	If specified, set the target value or element to be an instance of the given Slice class. The value is a type ID from the new Slice definitions. The class must be compatible with the target dictionary's value type, or the target sequence's element type.
`convert`	If `true`, additional type conversions are supported: between integer and floating point, and between integer and enumeration. Transformation fails immediately if a range error occurs. If not specified, the default value is `false`.

Below is an example of an `<add>` descriptor that adds a new dictionary element and then initializes its member:

XML

```
<add target="new.parts" key="'P105J4'"/>
<set target="new.parts['P105J4'].cost" value="3.15"/>
```

`<define>` Descriptor Element

Defines a new symbol in the current scope. The attributes supported by this descriptor are described in the following table:

Name	Description
`name`	The name of the new symbol. An error occurs if the name matches an existing symbol in the current scope.
`type`	The name of the symbol's formal Slice type. For user-defined types, the name should be prefixed with `::Old` or `::New` to indicate the source of the type. The prefix can be omitted for primitive types.
`value`	An expression that must evaluate to a value compatible with the symbol's type.
`convert`	If `true`, additional type conversions are supported: between integer and floating point, and between integer and enumeration. Execution fails immediately if a range error occurs. If not specified, the default value is `false`.

Below are two examples of the `<define>` descriptor. The first example defines the symbol `identity` to have type `Ice::Identity`, and proceeds to initialize its members using `<set>`:

XML

```
<define name="identity" type="::New::Ice::Identity"/>
<set target="identity.name" value="steve"/>
<set target="identity.category" value="Admin"/>
```

The second example uses the enumeration we first saw in our discussion of custom database migration to define the symbol `manufacturer` and assign it a default value:

XML

```
<define name="manufacturer" type="::New::BigThree"
value="::New::Daimler"/>
```

<remove> Descriptor Element

Removes an element from a sequence or dictionary. It is legal to remove an element while traversing a sequence or dictionary using <iterate>, however the traversal order after removal is undefined. The attributes supported by this descriptor are described in the following table:

Name	Description
target	An expression that must select a modifiable sequence or dictionary.
key	An expression that must evaluate to a value compatible with the key type of the target dictionary.
index	An expression that must evaluate to an integer value representing the index of the sequence element to be removed.

<fail> Descriptor Element

Causes transformation to fail immediately. If test is specified, transformation fails only if the expression evaluates to true. The attributes supported by this descriptor are described in the following table:

Name	Description
message	A message to display upon transformation failure.
test	A boolean expression.

The following <fail> descriptor terminates the transformation if a range error is detected:

XML

```
<fail message="range error occurred in ticket count!"
test="old.ticketCount > 32767"/>
```

<delete> Descriptor Element

Causes transformation of the current database record to cease, and removes the record from the transformed database. This descriptor has no attributes.

<echo> Descriptor Element

Displays values and informational messages. If no attributes are specified, only a newline is printed. The attributes supported by this descriptor are described in the following table:

Name	Description
message	A message to display.
value	An expression. The value of the expression is displayed in a structured format.

Shown below is an <echo> descriptor that uses both message and value attributes:

XML

```
<if test="old.ticketCount > 32767">
    <echo message="deleting record with invalid ticket count: "
value="old.ticketCount"/>
    <delete/>
</if>
```

See Also

- Custom Database Migration
- Maps
- Evictors
- Automatic Database Migration
- Descriptor Expression Language

Using transformdb

On this page:

Execution Modes for `transformdb`

The tool operates in one of three modes:

- Automatic migration
- Custom migration
- Analysis

The only difference between automatic and custom migration modes is the source of the transformation descriptors: for automatic migration, `transformdb` internally generates and executes a default set of descriptors, whereas for custom migration the user specifies an external file containing the transformation descriptors to be executed.

In analysis mode, `transformdb` creates a file containing the default transformation descriptors it would have used during automatic migration. You would normally review this file and possibly customize it prior to executing the tool again in its custom migration mode.

Using Database Catalogs during Transformation

Freeze maintains schema information in a catalog for each database environment. If necessary, `transformdb` will use the catalog to determine the names of the databases in the environment, and to determine the key and value types of a particular database. There are two advantages to the tool's use of the catalog:

- Allows `transformdb` to operate on all of the databases in a single invocation
- Eliminates the need for you to specify type information for a database.

For example, you can use automatic migration to transform all of the databases at one time, as shown below:

```
$ transformdb [options] old-env new-env
```

Since we omitted the name of a database to be migrated, `transformdb` uses the catalog in the environment `old-env` to discover all of the databases and their types, generates default transformations for each database, and performs the migration. However, we must still ensure that `transformdb` has loaded the old and new Slice types used by *all* of the databases in the environment.

Slice Options for `transformdb`

The tool supports the standard command-line options common to all Slice processors, with the exception of the include directory (`-I`) option. The options specific to `transformdb` are described below:

- `--old` *`SLICE`*
  `--new` *`SLICE`*
  Loads the old or new Slice definitions contained in the file *`SLICE`*. These options may be specified multiple times if several files must

be loaded. However, it is the user's responsibility to ensure that duplicate definitions do not occur (which is possible when two files are loaded that share a common include file). One strategy for avoiding duplicate definitions is to load a single Slice file that contains only `#include` statements for each of the Slice files to be loaded. No duplication is possible in this case if the included files use include guards correctly.

- `--include-old` *`DIR`*
  `--include-new` *`DIR`*
  Adds the directory *`DIR`* to the set of include paths for the old or new Slice definitions.

Type Options for `transformdb`

In invocation modes for which `transformdb` requires that you define the types used by a database, you must specify one of the following options:

- `--key` *`TYPE`*`[,`*`TYPE`*`]`
  `--value` *`TYPE`*`[,`*`TYPE`*`]`
  Specifies the Slice type(s) of the database key and value. If the type does not change, then the type only needs to be specified once. Otherwise, the old type is specified first, followed by a comma and the new type. For example, the option `--key int,string` indicates that the database key is migrating from `int` to `string`. On the other hand, the option `--key int,int` indicates that the key type does not change, and could be given simply as `--key int`. Type changes are restricted to those allowed by the compatibility rules, but custom migration provides additional flexibility.

- `-e`
  Indicates that a Freeze evictor database is being migrated. As a convenience, this option automatically sets the database key and value types to those appropriate for the Freeze evictor, and therefore the `--key` and `--value` options are not necessary. Specifically, the key type of a Freeze evictor database is `Ice::Identity`, and the value type is `Freeze::ObjectRecord`. The latter is defined in the Slice file `Freeze/EvictorStorage.ice`; however, this file does not need to be loaded into your old and new Slice definitions.

General Options for `transformdb`

These options may be specified during analysis or migration, as indicated below:

- `-i`
  Requests that `transformdb` ignore type changes that violate the compatibility rules. If this option is not specified, `transformdb` fails immediately if such a violation occurs. With this option, a warning is displayed but `transformdb` continues the requested action. The `-i` option can be specified in analysis or automatic migration modes.

- `-p`
  During migration, this option requests that `transformdb` purge object instances whose type is no longer found in the new Slice definitions.

- `-c`
  Use catastrophic recovery on the old Berkeley DB database environment prior to migration.

- `-w`
  Suppress duplicate warnings during migration. This option is especially useful to minimize diagnostic messages when `transformdb` would otherwise emit the same warning many times, such as when it detects the same issue in every record of a database.

Database Arguments for `transformdb`

In addition to the options described above, `transformdb` accepts as many as three arguments that specify the names of databases and database environments:

- `dbenv`
  The pathname of the old database environment directory.

- `db`
  The name of an existing database file in `dbenv`. `transformdb` never modifies this database.

- `newdbenv`
  The pathname of the database environment directory to contain the transformed database(s). This directory must exist and must not contain an existing database whose name matches a database being migrated.

Performing an Automatic Migration

You can use `transformdb` to automatically migrate one database or all databases in an environment.

Migrating a Single Database

Use the following command line to migrate one database:

```
$ transformdb [slice-opts] [type-opts] [gen-opts] dbenv db newdbenv
```

If you omit `type-opts`, the tool obtains type information for database `db` from the catalog. For example, consider the following command, which uses automatic migration to transform a database with a key type of `int` and value type of `string` into a database with the same key type and a value type of `long`:

```
$ transformdb --key int --value string,long dbhome data.db newdbhome
```

Note that we did not need to specify the Slice options `--old` or `--new` because our key and value types are primitives. Upon successful completion, the file `newdbhome/data.db` contains our transformed database.

Migrating All Databases

To migrate all databases in the environment, use a command like the one shown below:

```
$ transformdb [slice-opts] [gen-opts] dbenv newdbenv
```

In this invocation mode, you must ensure that `transformdb` has loaded the old and new Slice definitions for all of the types it will encounter among the databases in the environment.

Performing a Migration Analysis

Custom migration is a two-step process: you first write the transformation descriptors, and then execute them to transform a database. To assist you in the process of creating a descriptor file, `transformdb` can generate a default set of transformation descriptors by comparing your old and new Slice definitions. This feature is enabled by specifying the following option:

- `-o` *`FILE`*
  Specifies the descriptor file *`FILE`* to be created during analysis. No migration occurs in this invocation mode.

Generated File

The generated file contains a `<transform>` descriptor for each type that appears in both old and new Slice definitions, and an `<init>` descriptor for types that appear only in the new Slice definitions. In most cases, these descriptors are empty. However, they can contain XML comments describing changes detected by `transformdb` that may require action on your part.

For example, let us revisit the enumeration we defined in our discussion of custom database migration:

Slice

```
enum BigThree { Ford, DaimlerChrysler, GeneralMotors };
```

This enumeration has evolved into the one shown below. In particular, the `DaimlerChrysler` enumerator has been renamed to reflect a

corporate name change:

Slice

```
enum BigThree { Ford, Daimler, GeneralMotors };
```

Next we run `transformdb` in analysis mode:

```
$ transformdb --old old/BigThree.ice --new new/BigThree.ice --key string
\
--value ::BigThree -o transform.xml
```

The generated file `transform.xml` contains the following descriptor for the enumeration `BigThree`:

XML

```
<transform type="::BigThree">
    <!-- NOTICE: enumerator `DaimlerChrysler' has been removed -->
</transform>
```

The comment indicates that enumerator `DaimlerChrysler` is no longer present in the new definition, reminding us that we need to add logic in this `<transform>` descriptor to change all occurrences of `DaimlerChrysler` to `Daimler`.

The descriptor file generated by `transformdb` is well-formed and does not require any manual intervention prior to being executed. However, executing an unmodified descriptor file is simply the equivalent of using automatic migration.

Invocation Modes

The sample command line shown in the previous section specified the key and value types of the database explicitly. This invocation mode has the following general form:

```
$ transformdb [slice-opts] [type-opts] [gen-opts] -o FILE
```

Upon successful completion, the generated file contains a `<database>` descriptor that records the type information supplied by `type-opts`, in addition to the `<transform>` and `<init>` descriptors described earlier.

For your convenience, you can omit `type-opts` and allow `transformdb` to obtain type information from the catalog instead:

```
$ transformdb [slice-opts] [gen-opts] -o FILE dbenv
```

In this case, the generated file contains a `<database>` descriptor for each database in the catalog. Note that in this invocation mode, `transformdb` must assume that the names of the database key and value types have not changed, since the only type information available is the catalog in the old database environment. If the tool is unable to locate a new Slice definition for a database's key or value type, it emits a warning message and generates a placeholder value in the output file that you must modify prior to migration.

Performing a Custom Migration

After preparing a descriptor file, either by writing one completely yourself, or modifying one generated by the analysis mode described in the previous section, you are ready to migrate a database. One additional option is provided for migration:

- `-f` *`FILE`*
  Execute the transformation descriptors in the file *`FILE`*.

To transform one database, use the following command:

```
$ transformdb [slice-opts] [gen-opts] -f FILE dbenv db newdbenv
```

The tool searches the descriptor file for a `<database>` descriptor whose `name` attribute matches `db`. If no match is found, it searches for a `<database>` descriptor that does not have a `name` attribute.

If you want to transform all databases in the environment, you can omit the database name:

```
$ transformdb [slice-opts] [gen-opts] -f FILE dbenv newdbenv
```

In this case, the descriptor file must contain a `<database>` element for each database in the environment.

Continuing our enumeration example from the analysis discussion above, assume we have modified `transform.xml` to convert the `Chrysler` enumerator, and are now ready to execute the transformation:

```
$ transformdb --old old/BigThree.ice --new new/BigThree.ice -f
transform.xml \
dbhome bigthree.db newdbhome
```

`transformdb` Usage Strategies

If it becomes necessary for you to transform a Freeze database, we generally recommend that you attempt to use automatic migration first, unless you already know that custom migration is necessary. Since transformation is a non-destructive process, there is no harm in attempting an automatic migration, and it is a good way to perform a sanity check on your `transformdb` arguments (for example, to ensure that all the necessary Slice files are being loaded), as well as on the database itself. If `transformdb` detects any incompatible type changes, it displays an error message for each incompatible change and terminates without doing any transformation. In this case, you may want to run `transformdb` again with the `-i` option, which ignores incompatible changes and causes transformation to proceed.

Pay careful attention to any warnings that `transformdb` emits, as these may indicate the need for using custom migration. For example, if we had attempted to transform the database containing the `BigThree` enumeration from previous sections using automatic migration, any occurrences of the `Chrysler` enumerator would display the following warning:

```
warning: unable to convert 'Chrysler' to ::BigThree
```

If custom migration appears to be necessary, use analysis to generate a default descriptor file, then review it for `NOTICE` comments and edit as necessary. Liberal use of the `<echo>` descriptor can be beneficial when testing your descriptor file, especially from within the `<record>` descriptor where you can display old and new keys and values.

Transforming Objects

The polymorphic nature of Slice classes can cause problems for database migration. As an example, the Slice parser can ensure that a set of Slice definitions loaded into `transformdb` is complete for all types but classes (and exceptions, but we ignore those because they are not persistent). `transformdb` cannot know that a database may contain instances of a subclass that is derived from one of the loaded classes but whose definition is not loaded. Alternatively, the type of a class instance may have been renamed and cannot be found in the new Slice definitions.

By default, these situations result in immediate transformation failure. However, the `-p` option is a (potentially drastic) way to handle these situations: if a class instance has no equivalent in the new Slice definitions and this option is specified, `transformdb` removes the instance any way it can. If the instance appears in a sequence or dictionary element, that element is removed. Otherwise, the database record

containing the instance is deleted.

Now, the case of a class type being renamed is handled easily enough using custom migration and the `rename` attribute of the `<transform>` descriptor. However, there are legitimate cases where the destructive nature of the `-p` option can be useful. For example, if a class type has been removed and it is simply easier to start with a database that is guaranteed not to contain any instances of that type, then the `-p` option may simplify the broader migration effort.

This is another situation in which running an automatic migration first can help point out the trouble spots in a potential migration. Using the `-p` option, `transformdb` emits a warning about the missing class type and continues, rather than halting at the first occurrence, enabling you to discover whether you have forgotten to load some Slice definitions, or need to rename a type.

Using `transformdb` on an Open Environment

It is possible to use `transformdb` to migrate databases in an environment that is currently open by another process, but if you are not careful you can easily corrupt the environment and cause the other process to fail. To avoid such problems, you must configure both `transformdb` and the other process to set `Freeze.DbEnv.`*`env-name`*`.DbPrivate=0`. This property has a default value of one, therefore you must explicitly set it to zero. Note that `transformdb` makes no changes to the existing database environment, but it requires exclusive access to the new database environment until transformation is complete.

If you run `transformdb` on an open environment but neglect to set `Freeze.DbEnv.`*`env-name`*`.DbPrivate=0`, you can expect `transformdb` to terminate immediately with an error message stating that the database environment is locked. Before running `transformdb` on an open environment, we strongly recommend that you first verify that the other process was also configured with `Freeze.DbEnv.`*`env-name`*`.DbPrivate=0`.

See Also

- Automatic Database Migration
- Custom Database Migration
- Using the Slice Compilers
- Catalogs
- Evictors
- Freeze Property Reference

Inspecting a Database

The FreezeScript tool `dumpdb` is used to examine a Freeze database. Its simplest invocation displays every record of the database, but the tool also supports more selective activities. In fact, `dumpdb` supports a scripted mode that shares many of the same XML descriptors as `transformdb`, enabling sophisticated filtering and reporting.

Topics

Using dumpdb

This page describes dumpdb and provides advice on how to best use it.

On this page:

Overview of Inspection Descriptors

dumpdb can read descriptors from an XML file. A dumpdb descriptor file has a well-defined structure. The top-level descriptor in the file is <dumpdb>. A <database> descriptor must be present within <dumpdb> to define the key and value types used by the database. Inside <database>, the <record> descriptor triggers database traversal. Shown below is an example that demonstrates the structure of a minimal descriptor file:

XML

```
<dumpdb>
    <database key="string" value="::Employee">
        <record>
            <echo message="Key: " value="key"/>
            <echo message="Value: " value="value"/>
        </record>
    </database>
</dumpdb>
```

During traversal, type-specific actions are supported by the <dump> descriptor, which is a child of <dumpdb>. One <dump> descriptor may be defined for each type in the Slice definitions. Each time dumpdb encounters an instance of a type, the <dump> descriptor for that type is executed.

The <database>, <record>, and <dump> descriptors may contain general-purpose action descriptors such as <if> and <echo>. These actions resemble statements in programming languages like C++ and Java, in that they are executed in the order of definition and their effects are cumulative. Actions can make use of the FreezeScript expression language.

Although dumpdb descriptors are not allowed to modify the database, they can still define local symbols for scripting purposes. Once a symbol is defined by the <define> descriptor, other descriptors such as <set>, <add>, and <remove> can be used to manipulate the symbol's value.

Inspection Flow of Execution

The descriptors are executed as follows:

- <database> is executed first. Each child descriptor of <database> is executed in the order of definition. If a <record> descriptor is present, database traversal occurs at that point. Any child descriptors of <database> that follow <record> are not executed until traversal completes.
- For each record, dumpdb interprets the key and value, invoking <dump> descriptors for each type it encounters. For example, if the value type of the database is a struct, then dumpdb first attempts to invoke a <dump> descriptor for the structure type, and then recursively interprets the structure's members in the same fashion.

Inspection Descriptor Scopes

The <database> descriptor creates a global scope, allowing child descriptors of <database> to define symbols that are accessible in any descriptor.

> In order for a global symbol to be available to a <dump> descriptor, the symbol must be defined before the <record> descriptor is executed.

Furthermore, certain other descriptors create local scopes that exist only for the duration of the descriptor's execution. For example, the <dump> descriptor creates a local scope and defines the symbol value to represent a value of the specified type. Child descriptors of <dump> can also define new symbols in the local scope, as long as those symbols do not clash with an existing symbol in that scope. It is legal to add a new symbol with the same name as a symbol in an outer scope, but the outer symbol will not be accessible during the descriptor's execution.

The global scope is useful in many situations. For example, suppose you want to track the number of times a certain value was encountered during database traversal. This can be accomplished as shown below:

XML

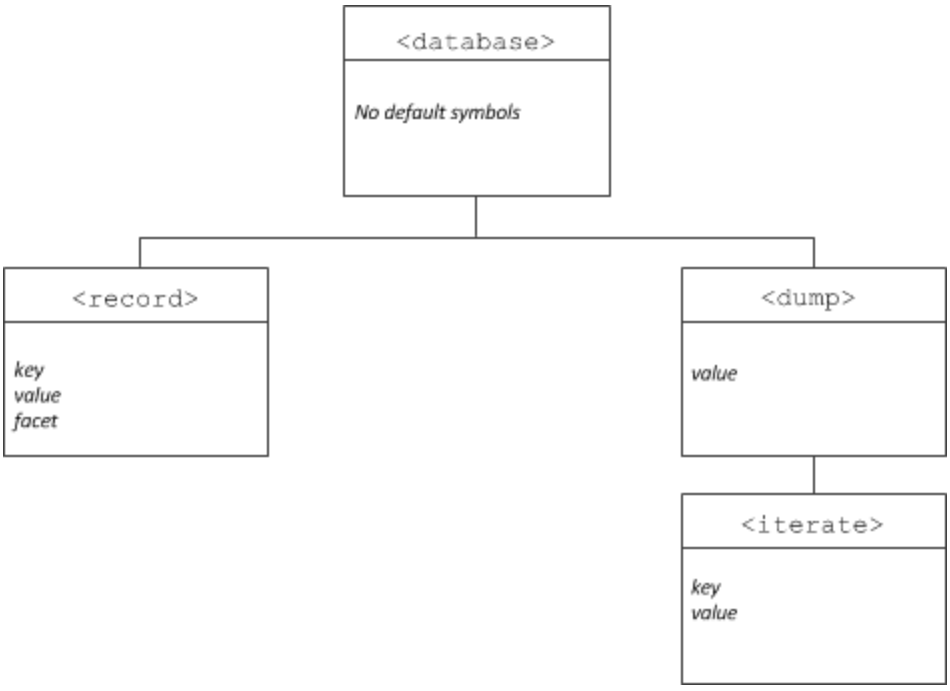
```
<dumpdb>
    <database key="string" value="::Ice::Identity">
        <define name="categoryCount" type="int" value="0"/>
        <record/>
        <echo message="categoryCount = " value="categoryCount"/>
    </database>
    <dump type="::Ice::Identity">
        <if test="value.category == `Accounting'">
            <set target="categoryCount" value="categoryCount + 1"/>
        </if>
    </dump>
</dumpdb>
```

In this example, the <define> descriptor introduces the symbol categoryCount into the global scope, defining it as type int with an initial value of zero. Next, the <record> descriptor causes traversal to proceed. Each occurrence of the type Ice::Identity causes its <dump> descriptor to be executed, which examines the category member and increases categoryCount if necessary. Finally, after traversal completes, the <echo> descriptor displays the final value of categoryCount.

To reinforce the relationships between descriptors and scopes, consider the diagram in the figure below. Several descriptors are shown, including the symbols they define in their local scopes. In this example, the <iterate> descriptor has a dictionary target and therefore the default symbol for the element value, value, hides the symbol of the same name in the parent <dump> descriptor's scope.

> This situation can be avoided by assigning a different symbol name to the element value.

In addition to symbols in the <iterate> scope, child descriptors of <iterate> can also refer to symbols from the <dump> and <database> scopes.



Relationship between descriptors and scopes.

Command Line Options for dumpdb

The tool supports the standard command-line options common to all Slice processors listed. The options specific to `dumpdb` are described below:

- `--load` *`SLICE`*
  Loads the Slice definitions contained in the file *`SLICE`*. This option may be specified multiple times if several files must be loaded. However, it is the user's responsibility to ensure that duplicate definitions do not occur (which is possible when two files are loaded that share a common include file). One strategy for avoiding duplicate definitions is to load a single Slice file that contains only `#include` statements for each of the Slice files to be loaded. No duplication is possible in this case if the included files use include guards correctly.
- `--key` *`TYPE`*
  `--value` *`TYPE`*
  Specifies the Slice type of the database key and value. If these options are not specified, and the `-e` option is not used, `dumpdb` obtains type information from the Freeze catalog.
- `-e`
  Indicates that a Freeze evictor database is being examined. As a convenience, this option automatically sets the database key and value types to those appropriate for the Freeze evictor, and therefore the `--key` and `--value` options are not necessary. Specifically, the key type of a Freeze evictor database is `Ice::Identity`, and the value type is `Freeze::ObjectRecord`. The latter is defined in the Slice file `Freeze/EvictorStorage.ice`, however this file does not need to be explicitly loaded. If this option is not specified, and the `--key` and `--value` options are not used, `dumpdb` obtains type information from the Freeze catalog.
- `-o` *`FILE`*
  Create a file named *`FILE`* containing sample descriptors for the loaded Slice definitions. If type information is not specified, `dumpdb` obtains it from the Freeze catalog. If the `--select` option is used, its expression is included in the sample descriptors. Database traversal does not occur when the `-o` option is used.
- `-f` *`FILE`*
  Execute the descriptors in the file named *`FILE`*. The file's `<database>` descriptor specifies the key and value types; therefore it is not necessary to supply type information.
- `--select` *`EXPR`*
  Only display those records for which the expression *`EXPR`* is true. The expression can refer to the symbols `key` and `value`.

- `-c, --catalog`
  Display information about the databases in an environment, or about a particular database. This option presents the type information contained in the Freeze catalog.

Database Arguments for dumpdb

If `dumpdb` is invoked to examine a database, it requires two arguments:

- `dbenv`
  The pathname of the database environment directory.
- `db`
  The name of the database file. `dumpdb` opens this database as read-only, and traversal occurs within a transaction.

To display catalog information using the `-c` option, the database environment directory `dbenv` is required. If the database file argument `db` is omitted, `dumpdb` displays information about every database in the catalog.

dumpdb Use Cases

The command line options support several modes of operation:

- Dump an entire database.
- Dump selected records of a database.
- Emit a sample descriptor file.
- Execute a descriptor file.
- Examine the catalog.

These use cases are described in the following sections.

Dump an Entire Database

The simplest way to examine a database with `dumpdb` is to dump its entire contents. You must specify the database key and value types, load the necessary Slice definitions, and supply the names of the database environment directory and database file. For example, this command dumps a Freeze map database whose key type is `string` and value type is `Employee`:

```
$ dumpdb --key string --value ::Employee --load Employee.ice db emp.db
```

As a convenience, you may omit the key and value types, in which case `dumpdb` obtains them from the catalog:

```
$ dumpdb --load Employee.ice db emp.db
```

Dump Selected Records

If only certain records are of interest to you, the `--select` option provides a convenient way to filter the output of `dumpdb` using an expression. In the following example, we select employees from the accounting department:

```
$ dumpdb --load Employee.ice --select "value.dept == 'Accounting'" db
emp.db
```

In cases where the database records contain polymorphic class instances, you must be careful to specify an expression that can be successfully evaluated against all records. For example, `dumpdb` fails immediately if the expression refers to a data member that does not exist in the class instance. The safest way to write an expression in this case is to check the type of the class instance before referring to any of its data members.

In the example below, we assume that a Freeze evictor database contains instances of various classes in a class hierarchy, and we are only interested in instances of `Manager` whose employee count is greater than 10:

```
$ dumpdb -e --load Employee.ice \
--select "value.servant.ice_id == '::Manager' and
value.servant.group.length > 10" \
db emp.db
```

Alternatively, if `Manager` has derived classes, then the expression can be written in a different way so that instances of `Manager` and any of its derived classes are considered:

```
$ dumpdb -e --load Employee.ice \
--select "value.servant.ice_isA('::Manager') and
value.servant.group.length > 10" \
db emp.db
```

Creating a Sample Descriptor File

If you require more sophisticated filtering or scripting capabilities, then you must use a descriptor file. The easiest way to get started with a descriptor file is to generate a template using `dumpdb`:

```
$ dumpdb --key string --value ::Employee --load Employee.ice -o dump.xml
```

The output file `dump.xml` is complete and can be executed immediately if desired, but typically the file is used as a starting point for further customization. Again, you may omit the key and value types by specifying the database instead:

```
$ dumpdb --load Employee.ice -o dump.xml db emp.db
```

If the `--select` option is specified, its expression is included in the generated `<record>` descriptor as the value of the `test` attribute in an `<if>` descriptor.

`dumpdb` terminates immediately after generating the output file.

Executing a Descriptor File

Use the `-f` option when you are ready to execute a descriptor file. For example, we can execute the descriptor we generated in the previous section using this command:

```
$ dumpdb -f dump.xml --load Employee.ice db emp.db
```

Examine the Catalog

The `-c` option displays the contents of the database environment's catalog:

```
$ dumpdb -c db
```

The output indicates whether each database in the environment is associated with an evictor or a map. For maps, the output includes the key and value types.

If you specify the name of a database, `dumpdb` only displays the type information for that database:

```
$ dumpdb -c db emp.db
```

Using dumpdb on an Open Environment

It is possible to use `dumpdb` to migrate databases in an environment that is currently open by another process, but if you are not careful you can easily corrupt the environment and cause the other process to fail. To avoid such problems, you must configure both `dumpdb` and the other process to set `Freeze.DbEnv.`*`env-name`*`.DbPrivate=0`. This property has a default value of one, therefore you must explicitly set it to zero.

If you run `dumpdb` on an open environment but neglect to set `Freeze.DbEnv.`*`env-name`*`.DbPrivate=0`, you can expect `dumpdb` to terminate immediately with an error message stating that the database environment is locked. Before running `dumpdb` on an open environment, we strongly recommend that you first verify that the other process was also configured with `Freeze.DbEnv.`*`env-name`*`.DbPrivate=0`.

See Also

- Using the Slice Compilers
- Catalogs
- Evictors
- Descriptor Expression Language
- Freeze Property Reference

Inspection XML Reference

This page describes the XML elements comprising the FreezeScript inspection descriptors.

On this page:

<dumpdb> Descriptor Element

The top-level descriptor in a descriptor file. It requires one child descriptor, `<database>`, and supports any number of `<dump>` descriptors. This descriptor has no attributes.

<database> Descriptor Element

The attributes of this descriptor define the key and value types of the database. It supports any number of child descriptors, but at most one `<record>` descriptor. The `<database>` descriptor also creates a global scope for user-defined symbols.

The attributes supported by the `<database>` descriptor are described in the following table:

Name	Description
`key`	Specifies the Slice type of the database key.
`value`	Specifies the Slice type of the database value.

As an example, consider the following `<database>` descriptor. In this case, the Freeze map to be examined has key type `int` and value type `::Employee`:

XML

```
<database key="int" value="::Employee">
```

<record> Descriptor Element

Commences the database traversal. Child descriptors are executed for each record in the database, but after any `<dump>` descriptors are executed. The `<record>` descriptor introduces the read-only symbols `key`, `value` and `facet` into a local scope. These symbols are accessible to child descriptors, but not to `<dump>` descriptors. The `facet` symbol is a string indicating the facet name of the object in the current record, and is only relevant for Freeze evictor databases.

Note that database traversal only occurs if a `<record>` descriptor is present.

<dump> Descriptor Element

Executed for all instances of a Slice type. Only one `<dump>` descriptor can be specified for a type, but a `<dump>` descriptor is not required for every type. The read-only symbol `value` is introduced into a local scope. The attributes supported by this descriptor are described in the following table:

Name	Description
`type`	Specifies the Slice type ID.
`base`	If `type` denotes a Slice class, this attribute determines whether the <dump> descriptor of the base class is invoked. If `true`, the base class descriptor is invoked after executing the child descriptors. If not specified, the default value is `true`.
`contents`	For `class` and `struct` types, this attribute determines whether descriptors are executed for members of the value. For `sequence` and `dictionary` types, this attribute determines whether descriptors are executed for elements. If not specified, the default value is `true`.

Below is an example of a <dump> descriptor that searches for certain products:

XML

```
<dump type="::Product">
    <if test="value.description.find('scanner') != -1">
        <echo message="Scanner SKU: " value="value.SKU"/>
    </if>
</dump>
```

For class types, `dumpdb` first attempts to locate a <dump> descriptor for the object's most-derived type. If no descriptor is found, `dumpdb` proceeds up the class hierarchy in an attempt to find a descriptor. The base object type, `Object`, is the root of every class hierarchy and is included in the search for descriptors. It is therefore possible to define a <dump> descriptor for type `Object`, which will be invoked for every class instance.

Note that <dump> descriptors are executed recursively. For example, consider the following Slice definitions:

Slice

```
struct Inner
{
    int sum;
}
struct Outer
{
    Inner i;
}
```

When `dumpdb` is interpreting a value of type `Outer`, it executes the <dump> descriptor for `Outer`, then recursively executes the <dump> descriptor for the `Inner` member, but only if the `contents` attribute of the `Outer` descriptor has the value `true`.

<iterate> Descriptor Element

Iterates over a dictionary or sequence, executing child descriptors for each element. The symbol names selected to represent the element information may conflict with existing symbols in the enclosing scope, in which case those outer symbols are not accessible to child descriptors. The attributes supported by this descriptor are described in the following table:

Name	Description
`target`	The sequence or dictionary.
`index`	The symbol name used for the sequence index. If not specified, the default symbol is `i`.
`element`	The symbol name used for the sequence element. If not specified, the default symbol is `elem`.

`key`	The symbol name used for the dictionary key. If not specified, the default symbol is `key`.
`value`	The symbol name used for the dictionary value. If not specified, the default symbol is `value`.

Shown below is an example of an `<iterate>` descriptor that displays the name of an employee if the employee's salary is greater than $3000.

XML

```
<iterate target="value.employeeMap" key="id" value="emp">
    <if test="emp.salary > 3000">
        <echo message="Employee: " value="emp.name"/>
    </if>
</iterate>
```

`<if>` Descriptor Element

Conditionally executes child descriptors. The attributes supported by this descriptor are described in the following table:

Name	Description
`test`	A boolean expression.

Child descriptors are executed if the expression in `test` evaluates to true.

`<set>` Descriptor Element

Modifies a value. The `value` and `type` attributes are mutually exclusive. If `target` denotes a dictionary element, that element must already exist (i.e., `<set>` cannot be used to add an element to a dictionary). The attributes supported by this descriptor are described in the following table:

Name	Description
`target`	An expression that must select a modifiable value.
`value`	An expression that must evaluate to a value compatible with the target's type.
`type`	The Slice type ID of a class to be instantiated. The class must be compatible with the target's type.
`length`	An integer expression representing the desired new length of a sequence. If the new length is less than the current size of the sequence, elements are removed from the end of the sequence. If the new length is greater than the current size, new elements are added to the end of the sequence. If `value` or `type` is also specified, it is used to initialize each new element.
`convert`	If `true`, additional type conversions are supported: between integer and floating point, and between integer and enumeration. Transformation fails immediately if a range error occurs. If not specified, the default value is `false`.

The `<set>` descriptor below modifies a member of a dictionary element:

XML

```
<set target="new.parts['P105J3'].cost"
value="new.parts['P105J3'].cost * 1.05"/>
```

This `<set>` descriptor adds an element to a sequence and initializes its value:

XML

```
<set target="new.partsList" length="new.partsList.length + 1"
value="'P105J3'"/>
```

<add> Descriptor Element

Adds a new element to a sequence or dictionary. It is legal to add an element while traversing the sequence or dictionary using `<iterate>`, however the traversal order after the addition is undefined. The `key` and `index` attributes are mutually exclusive, as are the `value` and `type` attributes. If neither `value` nor `type` is specified, the new element is initialized with a default value. The attributes supported by this descriptor are described in the following table:

Name	Description
`target`	An expression that must select a modifiable sequence or dictionary.
`key`	An expression that must evaluate to a value compatible with the target dictionary's key type.
`index`	An expression that must evaluate to an integer value representing the insertion position. The new element is inserted before `index`. The value must not exceed the length of the target sequence.
`value`	An expression that must evaluate to a value compatible with the target dictionary's value type, or the target sequence's element type.
`type`	The Slice type ID of a class to be instantiated. The class must be compatible with the target dictionary's value type, or the target sequence's element type.
`convert`	If `true`, additional type conversions are supported: between integer and floating point, and between integer and enumeration. Transformation fails immediately if a range error occurs. If not specified, the default value is `false`.

Below is an example of an `<add>` descriptor that adds a new dictionary element and then initializes its member:

XML

```
<add target="new.parts" key="'P105J4'"/>
<set target="new.parts['P105J4'].cost" value="3.15"/>
```

<define> Descriptor Element

Defines a new symbol in the current scope. The attributes supported by this descriptor are described in the following table:

Name	Description
`name`	The name of the new symbol. An error occurs if the name matches an existing symbol in the current scope.
`type`	The name of the symbol's formal Slice type.
`value`	An expression that must evaluate to a value compatible with the symbol's type.
`convert`	If `true`, additional type conversions are supported: between integer and floating point, and between integer and enumeration. Execution fails immediately if a range error occurs. If not specified, the default value is `false`.

Below are two examples of the `<define>` descriptor. The first example defines the symbol `identity` to have type `Ice::Identity`, and proceeds to initialize its members using `<set>`:

XML

```
<define name="identity" type="::Ice::Identity"/>
<set target="identity.name" value="steve"/>
<set target="identity.category" value="Admin"/>
```

The second example uses the enumeration we first saw in our discussion of custom database migration to define the symbol `manufacturer` and assign it a default value:

XML

```
<define name="manufacturer" type="::BigThree"
value="::DaimlerChrysler"/>
```

`<remove>` Descriptor Element

Removes an element from a sequence or dictionary. It is legal to remove an element while traversing a sequence or dictionary using `<iterate>`, however the traversal order after removal is undefined. The attributes supported by this descriptor are described in the following table:

Name	Description
`target`	An expression that must select a modifiable sequence or dictionary.
`key`	An expression that must evaluate to a value compatible with the key type of the target dictionary.
`index`	An expression that must evaluate to an integer value representing the index of the sequence element to be removed.

`<fail>` Descriptor Element

Causes transformation to fail immediately. If `test` is specified, transformation fails only if the expression evaluates to `true`. The attributes supported by this descriptor are described in the following table:

Name	Description
`message`	A message to display upon transformation failure.
`test`	A boolean expression.

The following `<fail>` descriptor terminates the transformation if a range error is detected:

XML

```
<fail message="range error occurred in ticket count!"
test="value.ticketCount > 32767"/>
```

`<echo>` Descriptor Element

Displays values and informational messages. If no attributes are specified, only a newline is printed. The attributes supported by this descriptor are described in the following table:

Name	Description
`message`	A message to display.
`value`	An expression. The value of the expression is displayed in a structured format.

Shown below is an `<echo>` descriptor that uses both `message` and `value` attributes:

XML

```
<if test="value.ticketCount > 32767">
    <echo message="range error occurred in ticket count: "
value="value.ticketCount"/>
</if>
```

See Also

- Maps
- Evictors
- Versioning
- Custom Database Migration
- Descriptor Expression Language

Descriptor Expression Language

An expression language is provided for use in FreezeScript descriptors.

On this page:

Operators in FreezeScript

The language supports the usual complement of operators: `and`, `or`, `not`, {+}, `-`, `/`, `*`, %, <, >, ==, `!=`, <=, >=, `(`, `)`. Note that the < character must be escaped as `<` in order to comply with XML syntax restrictions.

Literals in FreezeScript

Literal values can be specified for integer, floating point, boolean, and string. The expression language supports the same syntax for literal values as that of Slice, with one exception: string literals must be enclosed in single quotes.

Symbols in FreezeScript

Certain descriptors introduce symbols that can be used in expressions. These symbols must comply with the naming rules for Slice identifiers (i.e., a leading letter followed by zero or more alphanumeric characters). Data members are accessed using dotted notation, such as `value.memberA.memberB`.

Expressions can refer to Slice constants and enumerators using scoped names. In a `transformdb` descriptor, there are two sets of Slice definitions, therefore the expression must indicate which set of definitions it is accessing by prefixing the scoped name with `::Old` or `::New`. For example, the expression `old.fruitMember == ::Old::Pear` evaluates to `true` if the data member `fruitMember` has the enumerated value `Pear`. In `dumpdb`, only one set of Slice definitions is present and therefore the constant or enumerator can be identified without any special prefix.

The `nil` Keyword in FreezeScript

The keyword `nil` represents a nil value of type `Object`. This keyword can be used in expressions to test for a nil object value, and can also be used to set an object value to nil.

Accessing Elements in FreezeScript

Dictionary and sequence elements are accessed using array notation, such as `userMap['joe']` or `stringSeq[5]`. An error occurs if an expression attempts to access a dictionary or sequence element that does not exist. For dictionaries, the recommended practice is to check for the presence of a key before accessing it:

XML

```
<if test="userMap.containsKey('joe') and userMap['joe'].active">
```

(This example shows that you can also call functions in FreezeScript.)

Similarly, expressions involving sequences should check the length of the sequence:

XML

```
<if test="stringSeq.length > 5 and stringSeq[5] == 'fruit'">
```

The `length` member is an implicit data member.

Reserved Keywords in FreezeScript

The following keywords are reserved: `and`, `or`, `not`, `true`, `false`, `nil`.

Implicit Data Members in FreezeScript

Certain Slice types support implicit data members:

- Dictionary and sequence instances have a member `length` representing the number of elements.
- Object instances have a member `ice_id` denoting the actual type of the object.

Calling Functions in FreezeScript

The expression language supports two forms of function invocation: member functions and global functions. A member function is invoked on a particular data value, whereas global functions are not bound to a data value. For instance, here is an expression that invokes the `find` member function of a `string` value:

```
old.stringValue.find('theSubstring') != -1
```

And here is an example that invokes the global function `stringToIdentity`:

```
stringToIdentity(old.stringValue)
```

If a function takes multiple arguments, the arguments must be separated with commas.

String Member Functions

The `string` data type supports the following member functions:

- `int find(string match[, int start])`
  Returns the index of the substring, or `-1` if not found. A starting position can optionally be supplied.
- `string replace(int start, int len, string str)`
  Replaces a given portion of the string with a new substring, and returns the modified string.
- `string substr(int start[, int len])`
  Returns a substring beginning at the given start position. If the optional length argument is supplied, the substring contains at most `l` en characters, otherwise the substring contains the remainder of the string.

Dictionary Member Functions

The `dictionary` data type supports the following member function:

- `bool containsKey(key)`
  Returns `true` if the dictionary contains an element with the given key, or `false` otherwise. The `key` argument must have a value

that is compatible with the dictionary's key type.

Object Member Functions

Object instances support the following member function:

- `bool ice_isA(string id)`
  Returns `true` if the object implements the given interface type, or `false` otherwise. This function cannot be invoked on a nil object.

Global Functions

The following global functions are provided:

- `string generateUUID()`
  Returns a new UUID.
- `string identityToString(Ice::Identity id)`
  Converts an identity into its string representation.
- `string lowercase(string str)`
  Returns a new string converted to lowercase.
- `string proxyToString(Ice::ObjectPrx prx)`
  Returns the string representation of the given proxy.
- `Ice::Identity stringToIdentity(string str)`
  Converts a string into an `Ice::Identity`.
- `Ice::ObjectPrx stringToProxy(string str)`
  Converts a string into a proxy.
- `string typeOf(val)`
  Returns the formal Slice type of the argument.

See Also

- Constants and Literals

Freeze Property Reference

On this page:

Freeze.DbEnv.*env-name*.CheckpointPeriod

Synopsis

`Freeze.DbEnv.`*`env-name`*`.CheckpointPeriod=`*`num`*

Description

Every Berkeley DB environment created by Freeze has an associated thread that checkpoints this environment every *`num`* seconds. If *`num`* is less than `0`, no checkpointing is performed. The default is 120 seconds.

Freeze.DbEnv.*env-name*.DbHome

Synopsis

`Freeze.DbEnv.`*`env-name`*`.DbHome=`*`db-home`*

Description

Defines the home directory of this Freeze database environment. The default directory is *`env-name`*.

Freeze.DbEnv.*env-name*.DbPrivate

Synopsis

`Freeze.DbEnv.`*`env-name`*`.DbPrivate=`*`num`*

Description

If *`num`* is set to a value larger than zero, Freeze instructs Berkeley DB to use process-private memory instead of shared memory. The default value is `1`. Set this property to `0` in order to run a FreezeScript utility, or a Berkeley DB utility such as `db_archive`, on a running environment.

Freeze.DbEnv.*env-name*.DbRecoverFatal

Synopsis

`Freeze.DbEnv.`*`env-name`*`.DbRecoverFatal=`*`num`*

Description

If *`num`* is set to a value larger than zero, fatal recovery is performed when the environment is opened. The default value is `0`.

Freeze.DbEnv.*env-name*.EncodingVersion

Synopsis

`Freeze.DbEnv.`*`env-name`*`.EncodingVersion=`*`encoding`*

Description

Defines the encoding used to decode keys and to encode keys and values. The default value is the value of `Ice.Default.EncodingVersion`.

Freeze.DbEnv.*env-name*.LockFile

Synopsis

`Freeze.DbEnv.`*`env-name`*`.LockFile=`*`num`*

Description

If *`num`* is set to a value larger than zero, Freeze creates a lock file in the database environment to prevent other processes from opening the environment. The default value is `1`.

Note that applications should not normally disable the lock file because simultaneous access to the same environment by multiple processes can lead to data corruption.

FreezeScript utilities automatically disable the lock file when `Freeze.DbEnv.env-name.DbPrivate` is set to zero.

Freeze.DbEnv.*env-name*.OldLogsAutoDelete

`Freeze.DbEnv.env-name.OldLogsAutoDelete=num`

If *num* is set to a value larger than zero, old transactional logs no longer in use are deleted after each periodic checkpoint (see `Freeze.DbEnv.env-name.CheckpointPeriod`). The default value is `1`.

Freeze.DbEnv.*env-name*.PeriodicCheckpointMinSize

Synopsis

`Freeze.DbEnv.env-name.PeriodicCheckpointMinSize=num`

Description

num is the minimum size in kilobytes for the periodic checkpoint (see `Freeze.DbEnv.env-name.CheckpointPeriod`). This value is passed to Berkeley DB's `checkpoint` function. The default is `0` (which means no minimum).

Freeze.Evictor.*env-name.filename*.MaxTxSize

Synopsis

`Freeze.Evictor.env-name.filename.MaxTxSize=num`

Description

Freeze can use a background thread to save updates to the database. Transactions are used to save many facets together. *num* defines the maximum number of facets saved per transaction. The default is `10 * SaveSizeTrigger` (see `Freeze.Evictor.env-name.filename.SaveSizeTrigger`); if this value is negative, the actual value is set to `100`.

Freeze.Evictor.*env-name.filename.name*.BtreeMinKey

Synopsis

`Freeze.Evictor.env-name.filename.name.BtreeMinKey=num`

Description

name represents a database name or an index name. This property sets the B-tree minkey of the corresponding Berkeley DB database. *num* is ignored if it is less than 2. Please refer to the Berkeley DB documentation for details.

Freeze.Evictor.*env-name.filename.name*.Checksum

Synopsis

{{Freeze.Evictor.*env-name*.*filename*.Checksum=*num*

Description

If *num* is greater than 0, checksums on the corresponding Berkeley DB database(s) are enabled. Please refer to the Berkeley DB documentation for details.

Freeze.Evictor.*env-name.filename*.PageSize

Synopsis

`Freeze.Evictor.`*`env-name`*`.`*`filename`*`.PageSize=`*`num`*

Description

If *num* is greater than 0, it sets the page size of the corresponding Berkeley DB database(s). Please refer to the Berkeley DB documentation f or details.

Freeze.Evictor.*env-name.filename*.PopulateEmptyIndices

Synopsis

`Freeze.Evictor.`*`env-name`*`.`*`filename`*`.PopulateEmptyIndices=`*`num`*

Description

When *num* is not `0` and you create an evictor with one or more empty indexes, the `createBackgroundSaveEvictor` or `createTransac tionalEvictor` call will populate these indexes by iterating over all the corresponding facets. This is particularly useful after transforming a Freeze evictor with FreezeScript, since FreezeScript does not transform indexes; however this can significantly slow down the creation of the evictor if you have an empty index because none of the facets currently in the database match the type of this index. The default value for this property is `0`.

Freeze.Evictor.*env-name.filename*.RollbackOnUserException

Synopsis

`Freeze.Evictor.`*`env-name`*`.`*`filename`*`.RollbackOnUserException=`*`num`*

Description

If *num* is set to a value larger than zero, a transactional evictor rolls back its transaction if the outcome of the dispatch is a user exception. If *num* is 0 (the default), the transactional evictor commits the transaction.

Freeze.Evictor.*env-name.filename*.SavePeriod

Synopsis

`Freeze.Evictor.`*`env-name.filename`*`.SavePeriod=`*`num`*

Description

Freeze can use a background thread to save updates to the database. After *`num`* milliseconds without saving, if any facet is created, modified, or destroyed, this background thread wakes up to save these facets. When num is `0`, there is no periodic saving. The default is `60 000`.

Freeze.Evictor.*env-name.filename*.SaveSizeTrigger

Synopsis

`Freeze.Evictor.`*`env-name.filename`*`.SaveSizeTrigger=`*`num`*

Description

Freeze can use a background thread to save updates to the database. When *`num`* is `0` or positive, as soon as *`num`* or more facets have been created, modified, or destroyed, this background thread wakes up to save them. When *`num`* is negative, there is no size trigger. The default is `10`.

Freeze.Evictor.*env-name.filename*.StreamTimeout

Synopsis

`Freeze.Evictor.`*`env-name.filename`*`.StreamTimeout=`*`num`*

Description

When the saving thread saves an object, it needs to lock this object in order to get a consistent copy of the object's state. If the lock cannot be acquired within *`num`* seconds, a fatal error is generated. If a fatal error callback was registered by the application, this callback is called; otherwise the program is terminated immediately. When *`num`* is `0` or negative, there is no timeout. The default value is `0`.

Freeze.Map.*name*.BtreeMinKey

Synopsis

`Freeze.Map.`*`name`*`.BtreeMinKey=`*`num`*

Description

`name` may represent a database name or an index name. This property sets the B-tree minkey of the corresponding Berkeley DB database. *`num`* is ignored if it is less than 2. Please refer to the Berkeley DB documentation for details.

Freeze.Map.*name*.Checksum

Synopsis

`Freeze.Map.`*`name`*`.Checksum=`*`num`*

Description

`name` may represent a database name or an index name. If *`num`* is greater than 0, checksums for the corresponding Berkeley DB database are enabled. Please refer to the Berkeley DB documentation for details.

Freeze.Map.*name*.PageSize

Synopsis

`Freeze.Map.`*`name`*`.PageSize=`*`num`*

Description

`name` may represent a database name or an index name. If *`num`* is greater than 0, it sets the page size of the corresponding Berkeley DB database. Please refer to the Berkeley DB documentation for details.

Freeze.Trace.DbEnv

Synopsis

`Freeze.Trace.DbEnv=`*`num`*

Description

The Freeze database environment activity trace level:

0	No database environment activity trace (default).
1	Trace database open and close.
2	Also trace checkpoints and the removal of old log files.

Freeze.Trace.Evictor

Synopsis

`Freeze.Trace.Evictor=`*`num`*

Description

The Freeze evictor activity trace level:

0	No evictor activity trace (default).
1	Trace Ice object and facet creation and destruction, facet streaming time, facet saving time, object eviction (every 50 objects) and evictor deactivation.
2	Also trace object lookups, and all object evictions.
3	Also trace object retrieval from the database.

Freeze.Trace.Map

Synopsis

`Freeze.Trace.Map=`*`num`*

Description

The Freeze map activity trace level:

0	No map activity trace (default).
1	Trace database open and close.
2	Also trace iterator and transaction operations, and reference counting of the underlying database.

Freeze.Trace.Transaction

Synopsis

`Freeze.Trace.Transaction=`*`num`*

Description

The Freeze transaction activity trace level:

0	No transaction activity trace (default).
1	Trace transaction IDs and commit and rollback activity.

Freeze.Warn.Deadlocks

Synopsis

`Freeze.Warn.Deadlocks=`*`num`*

Description

If *`num`* is set to a value larger than zero, Freeze logs a warning message when a deadlock occur. The default value is `0`.

Freeze.Warn.Rollback

Synopsis

`Freeze.Warn.Deadlocks=`*`num`*

Description

If *`num`* is set to a value larger than zero, Freeze logs a warning message when it rolls back a transaction that goes out of scope together with its associated connection. The default value is `1`. (C++ only)

Release Notes

The release notes provide information about a Freeze release, including descriptions of significant new features and changes, instructions for upgrading from an earlier release, and important platform-specific details.

Topics

Supported Platforms for Freeze 3.7.0

Freeze 3.7.0 is supported on the platform, compiler, and environment combinations shown below. Other platforms and compilers might work as well but have not been tested. Please contact us if you need support for a platform or compiler that is not on this list.

On this page:

Freeze for C++

Run-Time Platform	Compiler	Run-Time Architecture	Development Platform
Windows 10	Visual Studio 2013, Visual Studio 2015	x86, x64	Same as Run-Time
Red Hat Enterprise Linux 7 Amazon Linux 2017.03 SuSE Linux Enterprise Server 12 Ubuntu 16.04 (Xenial Xerus)	GCC (default version)	x86_64, x86 x86_64 x86_64 amd64	Same as Run-Time
macOS 10.12 (Sierra)	Xcode 8	x86_64	Same as Run-Time

Freeze for Java

Platform	Environment
All Freeze for C++ platforms	JDK 1.8

What's New in Freeze 3.7?

Freeze is a transactional object-oriented database management system that stores Ice types in Berkeley DB databases. Freeze used to be included in Ice, and two Ice services (IceGrid and IceStorm) relied on Freeze to store their data in Berkeley DB databases.

As of version 3.7, Freeze is an independent component, with its own GitHub repository and manual. Freeze 3.7 depends on Ice 3.7, but is not part of Ice 3.7. IceGrid and IceStorm in Ice 3.7 no longer use Freeze or Berkeley DB: they store their data in LMDB databases instead.

Freeze is now **deprecated.** Freeze 3.7 is provided primarily for backwards compatibility with Ice 3.6 and prior releases: if you use Ice 3.6 or older, we encourage you to upgrade to Ice 3.7 and Freeze 3.7 in case you were using the Freeze component. We do not recommend that you create new Ice-based applications with Freeze, as Freeze 3.7 will be the last Freeze release.

We deprecated Freeze for the following reasons:

- Berkeley DB open-source license compatibility with GPLv2
  Freeze, like Ice, is licensed under GPLv2, which is compatible with the open-source license for Berkeley DB until Berkeley DB 5.x. As of version 6.0, Berkeley DB's open-source license is AGPLv3, which is not compatible with GPLv2. This means Freeze can only rely on Berkeley DB 5.x, and Berkeley DB 5.x is quickly becoming obsolete.

- Operations on Slice classes
  Freeze offers two storage mechanism, Freeze evictors and Freeze maps, and Freeze evictors rely heavily on the ability to define operations on Slice classes. Other than for Freeze evictors, this feature (defining operations on classes) has limited use and adds complexity to the Ice programming model and language mappings. Deprecating Freeze allowed us to deprecate operations on classes and related features in Ice.

- Persistent storage is not a core feature for Ice
  Ice is all about helping you create networked applications, not store data. If you need persistent storage for your Ice application, you can use any number of storage mechanisms. Ice is completely database-agnostic.

Using the Windows Binary Distribution

This page provides important information for users of the Ice binary distributions on Windows platforms.

On this page:

Overview of the Freeze Binary Distribution for Windows

The Freeze 3.7 binary distribution for Windows consists of two NuGet packages: `zeroc.freeze.v120` and `zeroc.freeze.v140`.

These NuGet packages depend on the corresponding `zeroc.ice` packages and provide a C++ SDK to develop applications with Freeze and Visual Studio 2013 (`zeroc.freeze.v120`) or Visual Studio 2015 (`zeroc.freeze.v140`). These NuGet packages also contain the FreezeScript tools. You install these NuGet packages just like the `zeroc.ice` NuGet packages, as described in the Ice Release Notes.

> The Freeze NuGet packages include the `slice2freezej` compiler, but don't include the Freeze for Java JAR file. If you want to develop with Freeze in Java, you need to build it from source.

NuGet Package Details

The following table shows the Freeze C++ NuGet package layout:

Folder	Description
`build\native\include`	C++ header files
`build\native\lib\<Platform>\<Configuration>`	C++ import libraries
`build\native\bin\<Platform>\<Configuration>`	C++ DLLs, FreezeScript tools, Berkeley DB tools
`tools`	slice2freeze and slice2freezej compilers
`build\native`	Visual Studio property and target files

Installing the NuGet package imports the property and target files from the `build\native` folder into the project. The property file defines the following properties:

Name	Value	Description
`FreezeHome`	`$(MSBuildThisFileDirectory)..\..`	Full path to the package root folder
`FreezeToolsPath`	`$(FreezeHome)\tools`	Full path to the folder of the Slice compilers and FreezeScript tools

The targets file configures the C++ Additional Include Directories and Additional Library Directories to locate C++ headers and import libraries in the package's include and lib folders.

Using the Sample Programs

The freeze repository includes sample programs for Freeze. Simply clone this repository:

```
git clone -b 3.7 https://github.com/zeroc-ice/freeze.git
cd freeze/demos
```

Using the Linux Binary Distributions

This page provides important information for users of the Freeze binary distributions on Linux platforms.

On this page:

Overview of the Binary Distributions for Linux

DEB Packages

ZeroC provides the following DEB packages for Ubuntu:

Package	Description
`libzeroc-freeze3.7`	C++ run-time library
`libzeroc-freeze-dev`	C++ header files and `libFreeze.so`
`zeroc-freeze-compilers`	Slice compilers: `slice2freeze` and `slice2freezej`
`zeroc-freeze-utils`	FreezeScript utilities: `dumpdb` and `transformdb`

RPM Packages

ZeroC provides the following RPMs for Red Hat Enterprise Linux, SUSE Linux Enterprise Server, and Amazon Linux:

RPM	Description
`libfreeze3.7-c++`	C++ run-time library
`libfreeze-c++-devel`	C++ header files and `libFreeze.so`
`freeze-compilers`	Slice compilers: `slice2freeze` and `slice2freezej`
`freeze-utils`	FreezeScript utilities: `dumpdb` and `transformdb`

ZeroC also supplies RPMs for the following third-party packages:

RPM	Description
`db53`	Berkeley DB 5.3.28 C and C++ run time libraries
`db53-devel`	C++ development files for Berkeley DB 5.3.28
`db53-java`	Berkeley DB 5.3.28 Java run time
`db53-utils`	Berkeley DB 5.3.28 command-line utilities

The `db53` packages are only necessary on SUSE Linux Enterprise Server and Amazon Linux; Berkeley DB 5.3 is already available in the standard Red Hat Enterprise Linux 7 repository as `libdb`.

The `db53-devel` RPM is only necessary for building Freeze from source.

Installing Freeze on Linux

The Freeze packages are in the same repositories as the Ice packages. Follow the instructions in the Ice Release Notes to add the Ice package repository to your system and install Ice packages. Then install the Freeze packages you need, for example:

```
sudo yum install libfreeze-c++-devel
```

> The Freeze packages include the `slice2freezej` compiler, but don't include the Freeze for Java JAR file. If you want to develop with Freeze in Java, you need to build it from source.

Using the Sample Programs

The freeze repository includes sample programs for Freeze. Simply clone this repository:

```
git clone -b 3.7 https://github.com/zeroc-ice/freeze.git
cd freeze/demos
```

Using the macOS Binary Distribution

This page provides important information for users of the Freeze binary distribution for macOS.

On this page:

Installing Freeze

Using Homebrew, you can install the Freeze distribution for macOS with this command:

```
brew install zeroc-ice/tap/freeze
```

> The freeze formula includes the `slice2freezej` compiler, but doesn't include the Freeze for Java JAR file. If you want to develop with Freeze in Java, you need to build it from source.

Using the Sample Programs

The freeze repository includes sample programs for Freeze. Simply clone this repository:

```
git clone -b 3.7 https://github.com/zeroc-ice/freeze.git
cd freeze/demos
```

Slice API Reference

Modules

Freeze — Freeze provides automatic persistence for Ice servants.

Freeze Slice API

Freeze

Overview

module Freeze

Freeze provides automatic persistence for Ice servants.

Interface Index

BackgroundSaveEvictor — A background-save evictor is an evictor that saves updates asynchronously in a background thread.
Connection — A connection to a database (database environment with Berkeley DB).
Evictor — An automatic Ice object persistence manager, based on the evictor pattern.
EvictorIterator — An iterator for the objects managed by the evictor.
ServantInitializer — A servant initializer provides the application with an opportunity to perform custom servant initialization.
Transaction — A transaction.
TransactionalEvictor — A transactional evictor is an evictor that performs every single read-write operation within its own transaction.

Exception Index

DatabaseException — A Freeze database exception.
DeadlockException — A Freeze database deadlock exception.
EvictorDeactivatedException — This exception is raised if the evictor has been deactivated.
IndexNotFoundException — Exception raised when Freeze fails to locate an index.
InvalidPositionException — This Freeze Iterator is not on a valid position, for example this position has been erased.
NoSuchElementException — This exception is raised if there are no further elements in the iteration.
NotFoundException — A Freeze database exception, indicating that a database record could not be found.
TransactionAlreadyInProgressException — An attempt was made to start a transaction while a previous transaction has not yet been committed or rolled back.

Structure Index

CatalogData — The catalog keeps information about Freeze Maps and Freeze evictors in a Berkeley Db environment.
ObjectRecord — ObjectRecord is the value-type for the persistent maps maintained by evictors when using Ice encoding version is 1.0.
Statistics — Evictors maintain statistics about each object, when using Ice encoding version 1.0.

Sequence Index

Key — A database key, represented as a sequence of bytes.
Value — A database value, represented as a sequence of bytes.

Sequences

sequence<byte> Key

A database key, represented as a sequence of bytes.

sequence<byte> Value

A database value, represented as a sequence of bytes.

Freeze-BackgroundSaveEvictor

Freeze::BackgroundSaveEvictor

Overview

local interface BackgroundSaveEvictor extends Freeze::Evictor

A background-save evictor is an evictor that saves updates asynchronously in a background thread.

Operation Index

keep — Lock this object in the evictor cache.
keepFacet — Like keep, but with a facet.
release — Release a lock acquired by keep.
releaseFacet — Like release, but with a facet.

Operations

void keep(Ice::Identity id)

Lock this object in the evictor cache. This lock can be released by release or `remove`. release releases only one lock, while `remove` release s all the locks.

Parameters

`id` — The identity of the Ice object.

Exceptions

Ice::NotRegisteredException — Raised if this identity was not registered with the evictor.
Freeze::DatabaseException — Raised if a database failure occurred.

See Also

- keepFacet
- release
- `remove`

void keepFacet(Ice::Identity id, string facet)

Like keep, but with a facet. Calling `keep(id)` is equivalent to calling keepFacet with an empty facet.

Parameters

`id` — The identity of the Ice object.
`facet` — The facet. An empty facet means the default facet.

Exceptions

Ice::NotRegisteredException — Raised if this identity was not registered with the evictor.
Freeze::DatabaseException — Raised if a database failure occurred.

See Also

- keep
- releaseFacet
- `removeFacet`

void release(Ice::Identity id)

Release a lock acquired by keep. Once all the locks on an object have been released, the object is again subject to the normal eviction strategy.

Parameters

`id` — The identity of the Ice object.

Exceptions

Ice::NotRegisteredException — Raised if this object was not locked with keep or keepFacet.

See Also

- keepFacet
- release

void releaseFacet(Ice::Identity id, string facet)

Like release, but with a facet. Calling `release(id)` is equivalent to calling releaseFacet with an empty facet.

Parameters

`id` — The identity of the Ice object.
`facet` — The facet. An empty facet means the default facet.

Exceptions

Ice::NotRegisteredException — Raised if this object was not locked with keep or keepFacet.

See Also

- keep
- releaseFacet

Freeze-CatalogData

Freeze::CatalogData

Overview

struct CatalogData

The catalog keeps information about Freeze Maps and Freeze evictors in a Berkeley Db environment. It is used by FreezeScript.

Data Member Index

Data Members

bool evictor;

True if this entry describes an evictor database, false if it describes a map database.

string key;

The Slice type for the database key.

string value;

The Slice type for the database value.

Freeze-Connection

Freeze::Connection

Overview

local interface Connection

A connection to a database (database environment with Berkeley DB). If you want to use a connection concurrently in multiple threads, you need to serialize access to this connection.

Used By

- Freeze::Transaction::getConnection

Operation Index

beginTransaction — Create a new transaction.
currentTransaction — Returns the transaction associated with this connection.
removeMapIndex — Remove an old unused Freeze Map index @throws IndexNotFoundException Raised if this index does not exist
close — Closes this connection.
getCommunicator — Returns the communicator associated with this connection.
getEncoding — Returns the encoding version used to encode the data.
getName — The name of the connected system (for example, the Berkeley DB environment).

Operations

Freeze::Transaction beginTransaction()

Create a new transaction. Only one transaction at a time can be associated with a connection.

Return Value

The new transaction.

Exceptions

Freeze::TransactionAlreadyInProgressException — Raised if a transaction is already associated with this connection.

Freeze::Transaction currentTransaction()

Returns the transaction associated with this connection.

Return Value

The current transaction if there is one, null otherwise.

void removeMapIndex(string mapName, string indexName)

Remove an old unused Freeze Map index

Exceptions

Freeze::IndexNotFoundException — Raised if this index does not exist

void close()

Closes this connection. If there is an associated transaction, it is rolled back.

Ice::Communicator getCommunicator()

Returns the communicator associated with this connection.

Ice::EncodingVersion getEncoding()

Returns the encoding version used to encode the data.

string getName()

The name of the connected system (for example, the Berkeley DB environment).

Freeze-DatabaseException

Freeze::DatabaseException

Overview

local exception DatabaseException

A Freeze database exception.

Derived Exceptions

- Freeze::DeadlockException
- Freeze::NotFoundException

See Also

- Freeze::Evictor
- Freeze::Connection

Data Member Index

message — A message describing the reason for the exception.

Data Members

string message;

A message describing the reason for the exception.

Freeze-DeadlockException

Freeze::DeadlockException

Overview

local exception DeadlockException extends Freeze::DatabaseException

A Freeze database deadlock exception. Applications can react to this exception by aborting and trying the transaction again.

Data Member Index

tx — The transaction in which the deadlock occurred.

Data Members

Freeze::Transaction tx;

The transaction in which the deadlock occurred.

Freeze-Evictor

Freeze::Evictor

Overview

local interface Evictor extends Ice::ServantLocator

An automatic Ice object persistence manager, based on the evictor pattern. The evictor is a servant locator implementation that stores the persistent state of its objects in a database. Any number of objects can be registered with an evictor, but only a configurable number of servants are active at a time. These active servants reside in a queue; the least recently used servant in the queue is the first to be evicted when a new servant is activated.

Derived Classes and Interfaces

- Freeze::BackgroundSaveEvictor
- Freeze::TransactionalEvictor

See Also

- Freeze::ServantInitializer

Operation Index

setSize — Set the size of the evictor's servant queue.
getSize — Get the size of the evictor's servant queue.
add — Add a servant to this evictor.
addFacet — Like add, but with a facet.
remove — Permanently destroy an Ice object.
removeFacet — Like remove, but with a facet.
hasObject — Returns true if the given identity is managed by the evictor with the default facet.
hasFacet — Like hasObject, but with a facet.
getIterator — Get an iterator for the identities managed by the evictor.

Operations

void setSize(int sz)

Set the size of the evictor's servant queue. This is the maximum number of servants the evictor keeps active. Requests to set the queue size to a value smaller than zero are ignored.

Parameters

`sz` — The size of the servant queue. If the evictor currently holds more than `sz` servants in its queue, it evicts enough servants to match the new size. Note that this operation can block if the new queue size is smaller than the current number of servants that are servicing requests. In this case, the operation waits until a sufficient number of servants complete their requests.

Exceptions

Freeze::EvictorDeactivatedException — Raised if a the evictor has been deactivated.

See Also

- getSize

int getSize()

Get the size of the evictor's servant queue.

Return Value

The size of the servant queue.

Exceptions

Freeze::EvictorDeactivatedException — Raised if a the evictor has been deactivated.

See Also

- setSize

Object* add(Object servant, **Ice::Identity** id)

Add a servant to this evictor. The state of the servant passed to this operation will be saved in the evictor's persistent store.

Parameters

`servant` — The servant to add.
`id` — The identity of the Ice object that is implemented by the servant.

Return Value

A proxy that matches the given identity and this evictor's object adapter.

Exceptions

Ice::AlreadyRegisteredException — Raised if the evictor already has an object with this identity.
Freeze::DatabaseException — Raised if a database failure occurred.
Freeze::EvictorDeactivatedException — Raised if the evictor has been deactivated.

See Also

- addFacet
- remove
- removeFacet

Object* addFacet(Object servant, **Ice::Identity** id, string facet)

Like add, but with a facet. Calling `add(servant, id)` is equivalent to calling addFacet with an empty facet.

Parameters

`servant` — The servant to add.
`id` — The identity of the Ice object that is implemented by the servant.
`facet` — The facet. An empty facet means the default facet.

Return Value

A proxy that matches the given identity and this evictor's object adapter.

Exceptions

Ice::AlreadyRegisteredException — Raised if the evictor already has an object with this identity.
Freeze::DatabaseException — Raised if a database failure occurred.
Freeze::EvictorDeactivatedException — Raised if the evictor has been deactivated.

See Also

- add
- remove
- removeFacet

Object remove(Ice::Identity id)

Permanently destroy an Ice object.

Parameters

`id` — The identity of the Ice object.

Return Value

The removed servant.

Exceptions

Ice::NotRegisteredException — Raised if this identity was not registered with the evictor.
Freeze::DatabaseException — Raised if a database failure occurred.
Freeze::EvictorDeactivatedException — Raised if the evictor has been deactivated.

See Also

- add
- removeFacet

Object removeFacet(Ice::Identity id, string facet)

Like remove, but with a facet. Calling `remove(id)` is equivalent to calling removeFacet with an empty facet.

Parameters

`id` — The identity of the Ice object.
`facet` — The facet. An empty facet means the default facet.

Return Value

The removed servant.

Exceptions

Ice::NotRegisteredException — Raised if this identity was not registered with the evictor.
Freeze::DatabaseException — Raised if a database failure occurred.
Freeze::EvictorDeactivatedException — Raised if the evictor has been deactivated.

See Also

- remove
- addFacet

bool hasObject(Ice::Identity id)

Returns true if the given identity is managed by the evictor with the default facet.

Return Value

true if the identity is managed by the evictor, false otherwise.

Exceptions

Freeze::DatabaseException — Raised if a database failure occurred.
Freeze::EvictorDeactivatedException — Raised if a the evictor has been deactivated.

bool hasFacet(Ice::Identity id, string facet)

Like hasObject, but with a facet. Calling `hasObject(id)` is equivalent to calling hasFacet with an empty facet.

Return Value

true if the identity is managed by the evictor for the given facet, false otherwise.

Exceptions

Freeze::DatabaseException — Raised if a database failure occurred.
Freeze::EvictorDeactivatedException — Raised if a the evictor has been deactivated.

Freeze::EvictorIterator getIterator(string facet, int batchSize)

Get an iterator for the identities managed by the evictor.

Parameters

`facet` — The facet. An empty facet means the default facet.
`batchSize` — Internally, the Iterator retrieves the identities in batches of size batchSize. Selecting a small batchSize can have an adverse effect on performance.

Return Value

A new iterator.

Exceptions

Freeze::EvictorDeactivatedException — Raised if a the evictor has been deactivated.

Freeze-EvictorDeactivatedException

Freeze::EvictorDeactivatedException

Overview

local exception EvictorDeactivatedException

This exception is raised if the evictor has been deactivated.

Freeze-EvictorIterator

Freeze::EvictorIterator

Overview

local interface EvictorIterator

An iterator for the objects managed by the evictor. Note that an EvictorIterator is not thread-safe: the application needs to serialize access to a given EvictorIterator, for example by using it in just one thread.

Used By

- Freeze::Evictor::getIterator

See Also

- Freeze::Evictor

Operation Index

hasNext — Determines if the iteration has more elements.
next — Obtains the next identity in the iteration.

Operations

bool hasNext()

Determines if the iteration has more elements.

Return Value

True if the iterator has more elements, false otherwise.

Exceptions

Freeze::DatabaseException — Raised if a database failure occurs while retrieving a batch of objects.

Ice::Identity next()

Obtains the next identity in the iteration.

Return Value

The next identity in the iteration.

Exceptions

Freeze::NoSuchElementException — Raised if there is no further elements in the iteration.
Freeze::DatabaseException — Raised if a database failure occurs while retrieving a batch of objects.

Freeze-IndexNotFoundException

Freeze::IndexNotFoundException

Overview

local exception IndexNotFoundException

Exception raised when Freeze fails to locate an index.

Data Member Index

mapName — The name of the map in which the index could not be found.
indexName — The name of the index.

Data Members

string mapName;

The name of the map in which the index could not be found.

string indexName;

The name of the index.

Freeze-InvalidPositionException

Freeze::InvalidPositionException

Overview

local exception InvalidPositionException

This Freeze Iterator is not on a valid position, for example this position has been erased.

Freeze-NoSuchElementException

Freeze::NoSuchElementException

Overview

local exception NoSuchElementException

This exception is raised if there are no further elements in the iteration.

Freeze-NotFoundException

Freeze::NotFoundException

Overview

local exception NotFoundException extends Freeze::DatabaseException

A Freeze database exception, indicating that a database record could not be found.

Freeze-ObjectRecord

Freeze::ObjectRecord

Overview

struct ObjectRecord

ObjectRecord is the value-type for the persistent maps maintained by evictors when using Ice encoding version is 1.0.

Data Member Index

servant — The servant implementing the object.
stats — The statistics for the object.

Data Members

Object servant;

The servant implementing the object.

Freeze::Statistics stats;

The statistics for the object.

Freeze-ServantInitializer

Freeze::ServantInitializer

Overview

local interface ServantInitializer

A servant initializer provides the application with an opportunity to perform custom servant initialization.

See Also

- Freeze::Evictor

Operation Index

initialize — Called whenever the evictor creates a new servant.

Operations

void initialize(**Ice::ObjectAdapter** adapter, **Ice::Identity** identity, string facet, Object servant)

Called whenever the evictor creates a new servant. This operation allows application code to perform custom servant initialization after the servant has been created by the evictor and its persistent state has been restored.

Parameters

`adapter` — The object adapter in which the evictor is installed.
`identity` — The identity of the Ice object for which the servant was created.
`facet` — The facet. An empty facet means the default facet.
`servant` — The servant to initialize.

See Also

- Ice::Identity

Freeze-Statistics

Freeze::Statistics

Overview

struct Statistics

Evictors maintain statistics about each object, when using Ice encoding version 1.0.

Used By

- Freeze::ObjectRecord::stats

Data Member Index

creationTime — The time the object was created, in milliseconds since Jan 1, 1970 0:00.
lastSaveTime — The time the object was last saved, in milliseconds relative to `creationTime`.
avgSaveTime — The average time between saves, in milliseconds.

Data Members

long creationTime;

The time the object was created, in milliseconds since Jan 1, 1970 0:00.

long lastSaveTime;

The time the object was last saved, in milliseconds relative to `creationTime`.

long avgSaveTime;

The average time between saves, in milliseconds.

Freeze-Transaction

Freeze::Transaction

Overview

local interface Transaction

A transaction. If you want to use a transaction concurrently in multiple threads, you need to serialize access to this transaction.

Used By

- Freeze::Connection::beginTransaction
- Freeze::Connection::currentTransaction
- Freeze::DeadlockException::tx
- Freeze::TransactionalEvictor::getCurrentTransaction
- Freeze::TransactionalEvictor::setCurrentTransaction

Operation Index

commit — Commit this transaction.
rollback — Roll back this transaction.
getConnection — Get the connection associated with this Transaction

Operations

void commit()

Commit this transaction.

Exceptions

Freeze::DatabaseException — Raised if a database failure occurred.

void rollback()

Roll back this transaction.

Exceptions

Freeze::DatabaseException — Raised if a database failure occurred.

Freeze::Connection getConnection()

Get the connection associated with this Transaction

Freeze-TransactionalEvictor

Freeze::TransactionalEvictor

Overview

local interface TransactionalEvictor extends Freeze::Evictor

A transactional evictor is an evictor that performs every single read-write operation within its own transaction.

Operation Index

Operations

Freeze::Transaction getCurrentTransaction()

Get the transaction associated with the calling thread.

Return Value

The transaction associated with the calling thread.

void setCurrentTransaction(Freeze::Transaction tx)

Associate a transaction with the calling thread.

Parameters

`tx` — The transaction to associate with the calling thread.

Freeze-TransactionAlreadyInProgressException

Freeze::TransactionAlreadyInProgressException

Overview

local exception TransactionAlreadyInProgressException

An attempt was made to start a transaction while a previous transaction has not yet been committed or rolled back.