Freeze 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++ mapping or the Java 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

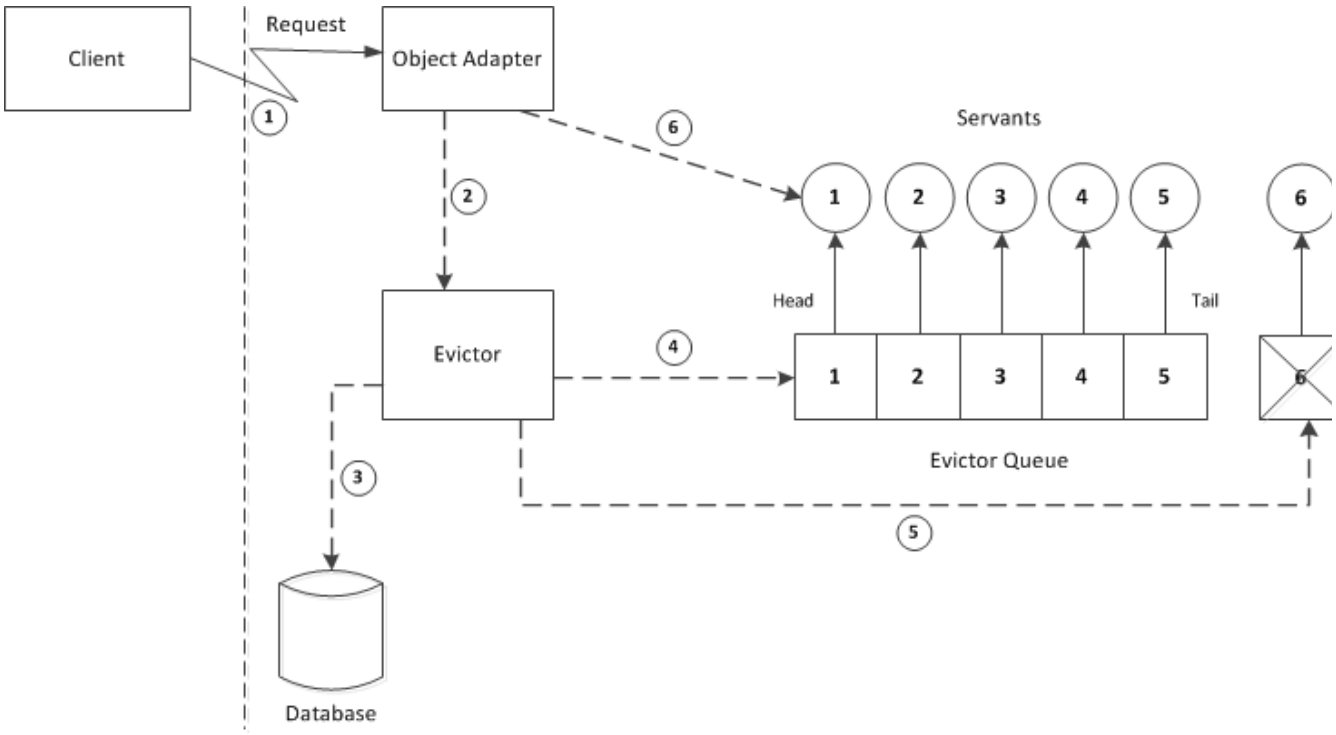
Freeze provides two types of evictors with different storage characteristics. The background save evictor records state changes to the database in a background thread, while the transactional evictor records all state changes immediately within the context of a transaction. You can choose the evictor 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++ Mapping for Classes
- Java Mapping for Classes
- Servant Activation and Deactivation
- Background Save Evictor
- Transactional Evictor
- Dictionaries
- Using a Freeze Evictor in the File System Server