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 a system 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
- Freeze Evictor Concepts
- Freeze Maps