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`: 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.)

See Also

- Transactional Evictor
- Freeze Evictor Concepts