EOF-Using EOF-Caching and Freshness

Contents

Overview

There is a large overlap between the content about Memory Management and that of Caching and Freshness. Both should be read to fully understand how caching and memory management should play a part in your application design.

Freshness

One of the most frequently asked, and hard to figure out, questions related to WebObjects is: "My application keeps using old values from the database. How do I ensure it gets fresh data?"

Caveats and Warnings

Until someone else wants to write a chapter on exactly what's going on, and what the Enterprise Objects Framework is doing, that will remain unsaid on this site. But do understand that EOF is attempting to be efficient and not go to the database more often than neccesary — since a db transaction is relatively expensive. Every choice you make in this area is really a trade off between efficiency of your app and 'freshness' of your db values. Depending on the requirements of your app, different choices may be called for.

Some of these methods may also effect 'optimistic locking' behavior, depending on what optimistic locking behavior is neccesary. A chapter on OptimisticLockingTechniques in WO/EOF would be welcome, because effectively using optimistic locking (taking account both of values changed by other app instance, and other sessions in the same instance) can be trickier than one would think.

Note: Some people have suggested that using EOFetchSpecification.setRefreshesRefetchedObjects and EOEditingContext.setDefaultFetchTimestampLag together will result in "unable to decrement snapshot" exceptions. Unfortunately, we really don't know anything more about this at this time.

But here are some ways to ensure 'freshness' of database values:

Recommended

- In your Application constructor, call:

```
EOEditingContext.setDefaultFetchTimestampLag(2);
```

This means that every EOEditingContext, when created, will insist on getting data fresh from the database that's no more than two seconds older than the EOEditingContext itself. If the EOEditingContext was created at 1:23:00, and there are snapshots on record from BEFORE 1:22:58, those snapshots won't be used, instead a new fetch to the db will be done.

The argument '2' up there is the number of seconds prior to the creation of the EOEditingContext, that that EOEditingContext is willing to accept data from. You can use whatever value you want. Thinking about it, you'll realize that setting it to '0' as you might initially want to, your app will potentially make very many more db transactions when under very heavy load, without gaining much in 'freshness' of data, so a non-zero value is probably preferable.

[WO:Oops, depending on which documentation you believe for which version, the argument may be in milliseconds instead of seconds. Beware].

[WO:in WO5.2.3 the default is 3600000 so obviously is milliseconds]

[From the WO 5.3 documentation: public static void setDefaultFetchTimestampLag(long lag)
Sets the default timestamp lag for newly instantiated editing contexts to lag.
The default lag is 3,600,000 milliseconds (one hour). When a new editing context is initialized, it is assigned a fetch timestamp equal to the current time less the default timestamp lag.
Setting the lag to a large number might cause every new editing context to accept very old cached data. Setting the lag to too low a value might degrade performance due to excessive fetching. A negative lag value is treated as 0.]
Just to make this clear: the lag constitutes a fixed time, not a rolling time window. So it ONLY makes sure that any data in your newly created EC is not older than the time of creation minus the lag.

- For some (or all!) EOFetchSpecifications, call setRefreshesRefetchedObjects(true) on it. This will mean that when you actually execute this fetch spec, and it goes and makes a transaction with the db, the values returned by the db will actually be used! Counter-intuitively, with the default value of refreshesRefetchedObjects==false, WO/EOF will sometimes ignore new values returned from the db in favor of older values in cached snapshots.
  Note that combining both methods (setDefaultFetchTimestampLag and setRefreshesRefetchedObjects) is buggy. It is NOT RECOMMENDED. You will see decrementSnapshotCountForGlobalID errors if you do.

Note that if you use setPrefetchingRelationshipKeyPaths on the EOFetchSpecification, the relationships mentioned will be followed and their destinations fetched with this EOFetchSpecification as well, and the setRefreshesRefetchedObjects directive will apply to those destination objects too.

Not Sure Whether to Recommend

- Some people use various methods to invalidate and/or refault their Enterprise Objects to ensure that the next access of those objects will result in fresh data. I am somewhat cautious of these methods, as the documentation is confusing and in some cases actually recommends against using them. Nonetheless, some people say they use them to good effect. I would recommend using other methods if you can. Maybe someone else wants to contribute more information?

Not Recommended

- I would NOT recommend trying to use the databaseContextShouldUpdateCurrentSnapshot EODatabaseContext delegate method. The documentation implies you can implement this delegate method such that you will always get fresh data---sort of the effect of setRefreshesRefetchecObjects(true) on every single fetch spec without needing to do that on every single fetch spec. But in my own experience, attempting to do this caused various problems for EOF and various cryptic exceptions would occur. So I don't recommend it.

What Various Developers Actually Do

Jonathan Rochkind

In my own highly interactive and collaborative applications, I place a high value on freshness of data, over efficiency of my app. Implementing the setDefaultTimestampLag with a very low value, and calling setRefreshesRefetchedObjects(true) on the majority of fetches has resulted in acceptable and reasonably fresh data in my applications. I do not really think that the efficiency has suffered much either, but I haven't investigated it fully and I am not concerned with maximizing the number of requests-per-time-period that can be dispatched.

Michael Johnston

When an eo absolutely has to be fresh (example 1, choosing an hourly winner in a game that had > 500,000 games played per day at it's peak; example two an asynchronous event processor that receives events both over the wire and from internal threads) I use two step fetch-save with a lock column, with a polling loop with a random, increasing sleep time. This is multi-instance, multi-app-server safe but doesn't ever lock up a thread for more than .8 of a second running against oracle. So far. I tried using database locking, but it resulted in several-second lockups.

Jesse Barnum

I use this code on the detail view of my enterprise objects when I want to make sure that they are up-to-date. I'm not explicitly doing a fetch, I just invalidate the particular object plus it's related objects, and they will automatically be refaulted and fetched when the page is displayed. As shown in the code sample, you must explicitly invalidate related objects. If you can count on your users knowing when information is stale, then a nifty technique is to trigger this code if a user hits reload in their browser.

```
EOEditingContext ec = object.editingContext();
NSMutableArray ids = new NSMutableArray();

// Invalidate the object
ids.addObject(ec.globalIDForObject(object));
// Invalidate a to-one related item
ids.addObject(ec.globalIDForObject((EOEnterpriseObject)object.valueForKey("adreq")));
// Invalidate a to-many relationship
Enumeration en = ((NSArray)object.valueForKey("correspondences")).objectEnumerator();
while(en.hasMoreElements())
  ids.addObject(ec.globalIDForObject((EOEnterpriseObject)en.nextElement()));
ec.invalidateObjectsWithGlobalIDs(ids);
```

Refreshing Many-to-Many Relationships

Chuck Hill

The freshness methods described above are limited in that they only refresh the attributes and the to-one relationships (in other words, the data directly in the row). They do not refresh to-many relationships to show changes in which objects are in the relationship. This means that if objects are added to the relationship by another process (or in another Object Store) your code will not see them as being related. Conversely, if objects are removed from the relationship in another object store they will still appear in the relationship. I don't know why this is, perhaps it is simply a shortcoming of EOF. Refreshing to-many relationships is unfortunately expensive and laborious.

One solution is to invalidate the objects that you need to refresh to-many relationships for. While this works, it can be expensive in terms of processing and have unintended side effects if the invalidated objects had unsaved changes.

There is another option that I've used but which I'm still not 100% convinced has no bad side effects when edits are in progress. This method of refreshing to-many relationships works by setting the snapshot for the to-many relationship to null. It only works for one object and one relationship at a time so refreshing many objects can be a little annoying. The implementation is slightly convoluted as we need to get down to the EODatabase level.

Here is gist of the code which refreshes relationshipName on sourceObject:

```
sourceEditingContext = sourceObject.editingContext();
EOEntity sourceObjectEntity = EOUtilities.entityForObject(sourceEditingContext, sourceObject);
EOModel sourceObjectModel = sourceObjectEntity.model();
EOGlobalID sourceGlobalID = sourceEditingContext.globalIDForObject(sourceObject);
EODatabaseContext dbContext = EODatabaseContext.registeredDatabaseContextForModel(sourceObjectModel,
sourceEditingContext);
EODatabase database = dbContext.database();
database.recordSnapshotForSourceGlobalID(null, sourceGlobalID, relationshipName);
```

Anjo Krank

While the method above will get fresh data the next time an EO is created you will have to perform an additional step to see the new data in an existing object:

```
Object o = eo.storedValueForKey(relationshipName);
if(o instanceof EOFaulting) {
  EOFaulting toManyArray = (EOFaulting)o;
  if (!toManyArray.isFault()) {
      EOFaulting tmpToManyArray = (EOFaulting)((EOObjectStoreCoordinator)ec.rootObjectStore())
            .arrayFaultWithSourceGlobalID(gid, relationshipName, ec);
      toManyArray.turnIntoFault(tmpToManyArray.faultHandler());
  }
} else {
      // we should check if the existing object is an array, too
      EOFaulting tmpToManyArray = (EOFaulting)((EOObjectStoreCoordinator)ec.rootObjectStore())
            .arrayFaultWithSourceGlobalID(gid, relationshipName, ec);
      eo.takeStoredValueForKey(tmpToManyArray, relationshipName);
}
```

EOEntity's Cache-In-Memory Setting

When you set Cache In Memory to true for an EOEntity, it tells EOF that you want it to try to always use an in-memory cache to store all instances of this entity. The first time that you fetch an object of the entity marked Cache In Memory, ALL instances of that entity will be fetched into memory. This is very useful for relatively static data (like enumerated type EO's, or other similar types of data that are infrequently modified in your application). Note that this is completely independent of EOF's normal snapshot caching, which will be in use regardless of the value of this setting. This setting is only used to determine if the ENTIRE dataset of your entity should be cached AT ALL TIMES. There are some very important implementation details that you should be aware of when you use this setting.
All behaviors described below MAY change from WO release to WO release and are not intrinsic requirements of the Cache In Memory feature, but are important considerations for anyone using the feature. All of these were verified to be true as of WebObjects 5.3:
The first is that Cache in Memory bypasses snapshot reference counting. Cache In Memory objects WILL NOT be released by EOF. In the end, this should not matter much, because you should not be using this flag on entities that have very large numbers of objects anyway.
On the performance front, be aware that Cache In Memory objects are indexed ONLY by EOGlobalID (primary key). If you are using an EOQualifier to find your objects (as opposed to traversing a to-one or to-many relationship that CONTAINS the object, which uses a primary key lookup), you will be doing a "full table scan" of your objects in memory. This is further reason to only use cache in memory on small cardinality entities. If you cache your 2 million row entity, it may likely be SLOWER to fetch EO's with an EOQualifier than if you just let the database take care of its business if the first place.
Another major performance detail of Cache In Memory is that if you make a change to ANY EO of the cached entity's type, ALL EO's of that type will be flushed from the cache and reloaded on the next access. This further supports the best practice of not using Cache In Memory for mutable EO's. If you have a Person EO set to cache-in-memory and you change the name of one Person, your entire cache will be dumped. The next time you fetch any person, the entire Person table will be reloaded. This can be catastrophic on a large series of changes (you can end up trashing the cache - you save one EO, cache flushes, it then reloads the entire cache, you save the next one, the cache flushes, etc etc).
Lastly, there is a quirk of Cache In Memory EO's where the cache will not be used unless your EOFetchSpecification has setDeep(true) on it. If isDeep is false on your fetch specification, a normal database fetch will occur (negating the usefulness of your cache). This is also true of EOUtilities methods that use EOFetchSpecifications internally. Project Wonder provides alternative implementations of several of the common EOUtilities fetch methods that setDeep(true).

invalidateAllObjects BAD!

First, and most importantly, the concept of invalidation in EOF is about cache coherency with the database, NOT memory management.
Now, invalidation does have some side effects which can have positive influences on Java memory usage, but using it for this purpose is akin to using a 50# sledge hammer to drive those miniature picture hanging nails into the wall.

I strongly recommend you only use invalidation when you as the application programmer have external information which is inaccessible to EOF regarding changes in the state of the database. i.e. You just executed an arbitrary stored procedure which had side effects all over your table. Or you got an RMI message from another process saying it updated a row. Or it's Monday, 3am, and on Monday, 2am, your DBA's cron job always deletes everything in a "temp" table. Or you've found a bug in EOF (alas, it happens), and this is the only workaround.

As others have noted, invalidation forcefully nukes the snapshot in the EODatabase cache. The increases the cache miss rate, reducing the performance of your application. Depending on your pattern of firing faults, 10,000 rows which took only 1 fetch to retrieve initially might take 10,000 fetches to restore into the cache. BAD. Prefetching and batch faulting can ameliorate this a little bit. They're a lot better than naive faulting, actually, but nothing can replace doing the right fetch the first time. Nothing by several orders of magnitude.

Nuking the snapshot has a number of other deleterious consequences. This cache is a shared resource used by all the ECs in your application. When one EC annihilates the resource all the other ECs are depending on ... So, EOF posts notifications about invalidation. Whenever something is invalidated, all the ECs in the application are affected, and must process the notification. In a large web application with many concurrent sessions, this can be a lot of unnecessary chatter. Invalidation and its ensuing notifications are very stressful on concurrently operating threads. Invalidations are propagated throughout the entire application, even if they were initiated from a nested EC.

[More technically precise: throughout the entire EOF stack centered around the EOObjectStoreCoordinator. In 5.1, it is theoretically possible to have several of these stacks, each with their own cache and EOObjectStoreCoordinator. Most EOF notifications are not passed between ECs using different OSCs. In practice, for 5.1, this is the entire application.]

Refaulting is much kinder and gentler. It only affects the EC it is invoked upon, so it won't interfere with ECs for other users. It will decrement the reference count on the snapshot (which may or may not release the snapshot). It will break outbound references within that EC. And if the current snapshot in the cache is "fresh enough" as defined by that EC's fetchTimestamp, firing the fault will use the cache instead of generating another fetch. Refaulting does not post notifications (although if the snapshot is not fresh, firing the fault will cause a fetch which will post)

Okay, so all of that had to do with databases. This thread is really about memory management with EOF in Java, so back to the point.

What does ec.dispose() do?

EOEditingContext does a lot of work. It has relationships to a number of other objects (including registering itself to receive notifications) and it is primarily responsible for maintaining a very large cyclic graph of objects (EOs). dispose() tells the EC that it will never be used again, and it should perform as much clean up work as possible. This decrements snapshot reference counts, unregisters for various things, and breaks references. Depending on your JVM, this may or may not help the GC reclaim memory faster. If you don't invoke dispose(), the EC's finalizer will. Most applications do not need to directly invoke dispose(), but your mileage may vary.

dispose() does not invoke any invalidate method.

For batch style EOF operations, the undo functionality is frequently overlooked as a source of problems (strong references). By default, EOF uses an unlimited undo/redo stack. If you are not going to use undo/redo, then you should seriously consider either setting the EC's undo manager to null, or performing removeAllActions on the EC's undo manager at logical checkpoints (like after saveChanges). The only downside to setting the EC's undo manager to null is that you will not be able to recover from validation exceptions. Depending on your batch operations, that may not be an issue. For a robust application, I would leave the undo manager, but use setLevelsOfUndo() to keep the stack small, and invoke removeAllActions() regularly.

For applications using EOF outside of an WebObjects application, some processing is left until the end of the current event. You may need to manually invoke eventEnded() on the NSDelayedCallbackCenter. Inside a WebObjects application, this is done for you at the end of a request.

In WO 5.1, EOEditingContexts have strong references to all their registered EOs, and EOs have NO reference to their editing context. The implications for GC should be fairly obvious.

invalidate does not clear away the EO objects themselves. It will force the memory for the database cache to go away, but the EOs will still be in memory and retained by their EC ...

Raw Rows

Raw Row results are never cached. If your application is using raw rows for its batch operations and is still having memory problems, then you need to evaluate your code. OptimizeIt or JProbe is definitely worth having. That's true even if you aren't doing raw row work.

Freshness Explorer

At WOWODC 2009 East, Mark Ritchie managed to dig up an old XCode project called Freshness Explorer for his demo. It appears the copy has been updated to work with WOLips and is currently located here (svn site) or it can be downloaded from here. That project has dependency on the MySQL database, which is great if you're using MySQL. If you are not using MySQL, you can use WO:this patch to remove the dependency and use Wonder's memory adaptor instead.