EOF-Using EOF-Memory Management

Contents

Overview

Memory Management in Java is a bit of a black art. When combined with EOF, there are certain issues that you need to be aware of. In WebObjects 5.2, there was a major change to the memory management system in EOF that provided support for WeakReferences to EO snapshots rather than only relying on the previous system of reference counting. This provides for vastly superior memory performance under the same conditions with the same app in 5.2 and above, because there are many cases where references will be released by Java's memory system rather than be held onto by your EOEditingContext.

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

General Issues

One important issue to consider prior to taking any drastic measures is whether or not your application really just requires more memory than you are giving it. By default, a Java Virtual Machine gives a Java application 64M of memory. This is not that much memory for an enterprise web application. You can adjust this by setting the "mx" flag on your VM at launch time (-Xmx256M would give you VM 256M of heap to work with instead of the default 64M). Other than just hunting aimlessly in your code, profiling your application is the only way to truly determine your application's memory situation.

The following profiling tools have been verified to work with WebObjects:

- JProfiler (http://www.ej-technologies.com/products/jprofiler/overview.html) has explicit support for WebObjects. It can automatically update your WebObjects Application launch script to insert itself at runtime to allow you to remotely attach to the application and profile it.
- VisualVM is a tool to monitor and troubleshoot Java applications. It runs on Oracle/Sun JDK 6, but is able to monitor applications running on JDK 1.4 and higher. It utilizes various available technologies like jvmstat, JMX, the Serviceability Agent (SA), and the Attach API to get the data and automatically uses the fastest and most lightweight technology to impose minimal overhead on monitored applications.

Raw Rows

The use of Raw Rows can dramatically decrease your memory usage (and increase your application's performance) at the cost of convenience. This uses less memory than normal EO's, allows better support for fetch limits, and you can still promote them to full-blown EO's when need be. Check out the Raw Rows topic for more information.

Qualify your Fetches

If you are not careful when you fetch, you may bring in huge datasets. When you use an EOFetchSpecification, be mindful to specify the most restrictive EOQualifier that still returns the data you need. Otherwise, you will be potentially faulting in much more data than your application will use, and you decrease your overall application performance because of the data that must be transferred from your database.

Relationships from Enumerated Types to other EO's

It is very common to have EO's that represent "Enumerated Types". For instance, you might have a bug tracking system that has a Severity entity (with individual instances of "Crasher", "Minor", etc) that are referenced by all bugs in the system. It is tempting to always create relationship back-references (i. e. Bug=>Severity, Severity=>>Bug). However, be aware that if you create the reference from Severity back to Bug, any time a Bug is added and you properly use the addObjectToBothSidesOfRelationshipWithKey method, EOF will fault in the ENTIRE "bugs" relationship on Severity prior to adding your new bug to the end of the list. The consequences of this can be crippling to your application. As the number of bugs increases, it will take longer and longer to add a new bug. The recommended best practice for this is to only create Bug=>Severity and NOT create the reverse relationship. In the event that you need to perform that fetch, you can manually construct the EOFetchSpecification rather than use the automatic approach.

Note that this is a filed bug against EOF. One approach that EOF could use is to only add the object to Severity's bugs relationship if the bugs relationship has already been faulted into memory. If the array has not yet been faulted, then causing the fault on insert is potentially very expensive. The reasoning for the current behavior is to maintain the ability to access the newly committed Bug via the Severity bugs relationship prior to committing your transaction (i.e. faulting the relationship from the database wouldn't yet return the newly inserted Bug). This is very important to maintaining object graph consistency pre-saveChanges, but does come at a high performance cost.

Dispose()ing your EOEditingContext

Calling dispose() on your EOEditingContext is not strictly required. When the EOEditingContext is no longer referenced, it will be automatically disposed. However, the question is how long it takes the GC to get to your object. If you call dispose() manually, you won't hurt anything, and you immediately decrement the snapshot reference counts. If your EC is falling out of scope or you set it to null, then it's probably a tossup as to who gets to it first - GC finalizing your EC or you manually calling dispose(). It is recommended that you dispose() your EOEditingContext when you are in a long running process in a single method where the EOEditingContext will not go out-of-scope until the end of the method.

Calling System.gc()

Calling System.gc() is usually a bad idea and can cause performance problems in your application. Additionally, System.gc() is not an order, it is a request, so if your application is relying on this command to execute, you should re-evaluate your application's design, because you may run into problems with this.

Anecdotes on editingContext.invalidateAllObjects()

It can be tempting to resolve memory problems by calling editingContext.invalidateObject(..) or editingContext.invalidateAllObjects(). Avoid this. You will only hurt yourself in the long run. Invalidating is very aggressive and WILL cause exceptions to be thrown if someone in another EOEditingContext is modifying an EO that you invalidate in your EOEditingContext (the shared snapshost will be destroyed, resulting in an exception on .saveChanges in the other context). It should only be used under very controlled situations where you understand the implications of the method.

Chuck Hill

As for invalidating objects, this is something to be avoided. I admit that I have been driven to desperation a couple of times and used this, but only with regrets and reservations.

Art Isbell

I just have a bias against using invalidateAllObjects() because, for me, it's caused more problems than it's solved.

Mike Schrag

The couple of times I resorted to this, I regretted it later, because it ended up screwing me. The biggest thing is you toss snapshots that people might be in the middle of editing, and that's going to cause really funky problems. It's a really heavy-handed way to go.

NSUndoManager

One of the most common memory problems people run into with WebObjects is the NSUndoManager. WebObjects has a particularly cool feature in that it supports an undo stack for EO transactions. Each time you saveChanges, you push an undoable set of changes onto the stack. The downside of this behavior is that you end up with strong references to old EO states, which can be deadly in a large EOF operation.

There are several fixes for this. One is to simply disable the NSUndoManager on your editing context by calling:

```
editingContext.setUndoManager(null);
```

It has been referenced that setting the undo manager to null may cause issues with deletes under certain circumstances. In testing this with WebObjects 5.3, the situation could not be reproduced, but if you use WebObjects 5.2 or lower, you should be aware that you might run into problems.

The recommended alternative in this case is to disable the registration of undo events by calling:

```
editingContext.undoManager().disableUndoRegistration();
```

Note that setting the undo levels to 0 using setLevelsOfUndo() does not work, as 0 means no limit!
Prior to 5.2, the NSUndoManager had effectively infinite undo levels by default. The javadoc of WO **5.3** still mentions the following:

EOEditingContext's undo support is arbitrarily deep; you can undo an object repeatedly until you restore it to the state it was in when it was first created or fetched into its editing context. Even after saving, you can undo a change. To support this feature, the NSUndoManager can keep a lot of data in memory.

However, as of 5.2, if you are using the session defaultEditingContext, WOSession sets the default undo stack size to 10 if there is no default size specified with the WODEFAULTUNDOSTACKLIMIT parameter.
Lastly you can manually clear your NSUndoManager when you are done with an operation by calling:

```
editingContext.undoManager().removeAllActions();
```

If you use an NSUndoManager, it is recommended that you call this after performing very large saveChanges that involved large amounts of data.

Aaron Rosenzweig

In our experience, the NSUndoManager should be considered an integral part of an EOEditingContext and neither be crippled nor removed. Calling

```
removeAllActions()
```

is safe and can be done in your project's descendent of EOEditingContext after a successful call to

```
super.saveChanges()
```

In WO 5.3 and above, setting the NSUndoManager to null will create the following error if you try to delete and save an EO that has a delete rule of "deny" on an existing relationship:

```
java.lang.RuntimeException: java.lang.IllegalStateException:
Editing context needs an undo manager to recover from delete propagation problems.
Do not set the undo manager of this editing context to null.
```

Additionally, if you simply remove undo registration, you'll find that little undo / redo pieces are still allocated in memory... which defeats the purpose of removing registration. This can be verified with a tool such as Jprofiler, etc. Any tool which allows you to view the java heap and memory allocations.

If what you are really after is processing many EOs in a long running process you should consider making an array of primary keys and materializing them onesy-twosy in a temporary EOEditingContext. Before exiting the loop call

```
ec.dispose()
```

then

```
ec = null
```

Another way to say it, make an EOEditingContext for each pass through the loop then clear it out before the next pass.

WebObjects 5.2+

This is a description of the current behavior in 5.3 that one might be able to gather if one were to, say, decompile and review the entire process - not that i would ever do this or condone it, of course - but if you DID, you would probably gather exactly this info, which is pretty well documented (and seems to behave to-spec) in the 5.2 release notes:

As of 5.2, the way it works is: The snapshots in EODatabase have a reference count. Each editing context that fetches an EO increments the reference count. The EC holds onto that EO via a WeakReference. When the WeakReference is reclaimed, the snapshot reference count can decrease (note CAN, not IMMEDIATELY WILL - the editing context keeps reference queue which is only processed periodically). When the count gets to zero, the database forgets the snapshot. If you have entity caching enabled, then EODatabase ignore reference count (or keeps it at "1" as a minimum) and it will not go away in a read-only scenario. If you modify any entity of that type and saveChanges in your EditingContext, a cached entity's cache will be entirely flushed. (NB: Keep this in mind, because if you are caching a large amount of data that is writable, it will NOT be very smart about updating that cache - It's blown away with every update and then it immediately reloads the entire set of objects for that entity at the next access)

If you have retainsAllRegisteredObjects enabled on your editing context, it will NOT use WeakReferences. Under this circumstance, the EO reference count is only decreased when 1) you dispose the editingcontext or 2) you forget or invalidate the object.

When you modify an object in an editing context, the editingcontext keeps a strong reference to the objects until you saveChanges (or revert, reset, etc), at which point the strong references are cleared and the only remaining reference is the weakreference like before.
If you have an undo manager enabled, it will keep a strong reference to the affected EO's as long as the undo is around.

I do wonder if EC's should be using SoftReferences instead of WeakReferences ... Would seem to be more friendly to the users of those EO's.

If you are using WO pre 5.2, then none of the WeakReference stuff applies, and everything is purely done with snapshot reference counting - it should behave like retainsAllRegisteredObjects = true in 5.2.

WebObjects Pre 5.2

Prior to 5.2, all snapshots were simply reference counted by EOEditingContext. If an EO was faulted into your EOEditingContext, it would not go away until the EOEditingContext itself was disposed. As a result, if you have an application deployed on a pre-5.2 WebObjects, you will need to adopt a strategy of periodically disposing your EOEditingContext and creating a new one.

Chuck Hill

In our experience, EOF is not too good at bulk operations like this. If we have a lot to do, and don't need logic from the EO, we wander down to the EOAccess level, and whip up some batch insert statements. You might even want to consider straight JDBC.

OK that said,

- save periodically
- don't retain references to objects you have inserted
- if that does not work, try creating a new EC periodically
- a bit heavy handed, but ec.invalidateAllObjects() should dump out all the snapshots as well.

A couple of useful comments:
http://lists.apple.com/archives/webobjects-dev/2004/Sep/msg00225.html
http://lists.apple.com/archives/webobjects-dev/2004/Sep/msg00228.html

It is always interesting to read the EOF docs, the JavaDocs are for API only while the EOF docs contain much valuable information.
http://developer.apple.com/documentation/WebObjects/Enterprise_Objects/index.html

"EOEditingContexts use weak references to the EOEnterpriseObjects registered with them. ... EOEditingContexts hold all inserted, deleted, or updated objects by strong references. These strong references are cleared by the EOEditingContext methods saveChanges ..." - http://developer.apple.com/documentation/WebObjects/Enterprise_Objects/Managing/chapter_7_section_9.html

Alan Ward

Re: creating a new EC periodically I'd jump straight to this option (based on my experience). I have done a couple of similar tasks and found that the best way to ensure that your app doesn't balloon up is to periodically ditch the EC and create a new one. Seems like it shouldn't be necessary.... but it works.

Ken Anderson

My guess is the memory increase is due to snapshots, which, as I understand it, will never go away, and wouldn't be considered a bug (to me anyway). For dealing with a forever-import problem I had, where I didn't want the strong handed 'new EC' plan, I wrote this method, which keeps the snapshots down:

```
public static void forgetObjectsAndKeypaths(EOEditingContext ec, NSArray eos, NSArray keypaths) {
   Enumeration eoEnum = eos.objectEnumerator();
   while (eoEnum.hasMoreElements()) {
     EOGenericRecord eo = (EOGenericRecord) eoEnum.nextElement();
     Enumeration keypathEnum = keypaths.objectEnumerator();
     while (keypathEnum.hasMoreElements()) {
       String keypath = (String) keypathEnum.nextElement();
       EOFaulting faulting = (EOFaulting) eo.valueForKeyPath(keypath);
       if (faulting != null && !faulting.isFault()) {
         if (eo.isToManyKey(keypath)) {
           NSArray relEos = (NSArray) eo.valueForKeyPath(keypath);
           EOHelper.forgetObjects(ec, relEos);
         } else {
           ec.forgetObject((EOCustomObject) eo.valueForKeyPath(keypath));
         }
       }
     }
     ec.forgetObject(eo);
   }
 }
```

You can use it like this:

```
EOHelper.forgetObjectsAndKeypaths(ec, arrayOfEOsToForget, new NSArray(new String[]{"rel1", "rel2"});
```

Of course, all the EOs in the array need to be the same entity for keypaths to work!