Logging UndeployMemoryLeak

Disclaimer

> Note: the contents of this page is my personal opinion only. Other commons-logging developers may have different views on this. – Simon Kitching

Overview

Despite Java's garbage collection facilities, it is possible to get memory leaks due to unexpected chains of references.

One issue that has been struck fairly frequently is where people repeatedly undeploy/redeploy components within a container (eg webapps within a servlet engine or j2ee server) and eventually get an out-of-memory situation. Commons Logging can be the cause of this when misused, and as a result a number of people have claimed that commons-logging is at fault.

One commonly-referenced example is this:
http://www.szegedi.org/articles/memleak.html

While I sympathise with anyone who has spent all day looking for a memory leak, I disagree with the conclusions that any of this is the fault of commons-logging.

Read on for more information.

Keep component libs in components

Servlet and J2EE containers have a clear purpose: to provide a set of services to *independent* and *isolated* components. That set of services is defined by the servlet and j2ee specifications. These specifications also define a mechanism for components to provide any libraries they depend on – WEB-INF/lib.

It is therefore a complete mystery to me why people seem so keen to push libraries up out of the components where they belong and into the container's library directories. It's like pushing user code into the operating system kernel. Just don't do it.

A component naturally has references to objects in the container; the fact that the container provides classes (which are objects) that are visible to the component is the most obvious example. This causes no problems for garbage-collecting of the component. Problems come when the links become *bidirec tional*, and classes within the container also have references to classes and objects within the components. The servlet and j2ee frameworks are carefully designed to avoid this where possible; where that is unavoidable the container *knows* about those situations and ensures the necessary references are cleared when a component is undeployed [1] [2]

But if users of the container start tossing libraries that aren't designed to avoid such references into classloaders that are "above" the classloader of the component and the container does not know about these libraries then bidirectional references are very likely to occur and failure to undeploy correctly will result.

In short, if a component's dependent libraries are *all* in the component-specific lib directory then undeploy *will not cause any problems*. No special actions need to be taken by a well-behaved component; when the component's classloader is discarded by the container as a result of an "undeploy" command commons-logging will not be holding any references that prevent garbage collection of that classloader.

[1] Except for the brain-dead design of JDBC where jdbc drivers loaded via a custom classloader apparently get stored in a map within java.sql. DriverManager thereby causing cyclic references to that classloader.

[2] And except for the cache of classes held by the java.beans.Introspector class. In at least some versions of java, introspecting a class loaded by component's classloader causes a strong reference to that class to be put into a global cache. This means that the component's classloader is then prevented from being unloaded. A workaround is to call java.beans.Introspector.flushCaches() when the component is unloaded. Some containers may do this automatically; Apache Tomcat 5.x certainly does. In other cases, a ServletContextListener may need to be registered to force this to be done.

Container libs are responsible for managing themselves

If code at the container level uses commons-logging, then that code is expected to be component-aware. The Apache Jakarta Tomcat servlet container is a fine example. It uses commons-logging internally, and therefore needs to deploy those libs at the container level. However it then correctly takes on the responsibility of managing commons-logging, by ensuring LogFactory.release is called when the component is undeployed.

There are certain libraries that can be considered as "container extensions". One example is the hibernate library. In such cases, when such a library is deployed into the container it is effectively making the container depend on commons-logging. And in that case it should somehow ensure that LogFactory. release is called when the component is undeployed. Extending a container just can't be considered as easy as dropping a jar into the container's classpath (though it would be nice if it *were* that easy).

Using ServletContextListener to call Release as a workaround

If you are determined to abuse the container and commons-logging by putting it into a container-level library directory without arranging for the container to call its release method on component undeploy then there is an ugly workaround.

There would actually be a nice workaround if Sun had only seen fit to provide a mechanism to attach data to an arbitrary classloader. Suppose that class java.lang.ClassLoader had the following methods:

```
public void putObject(Object key, Object o);
public Object getObject(Object key);
```

In that case, it would be possible for code running at the container level to attach component-specific data to the component's classloader. This would provide a way to avoid those problematic bidirectional references between objects at the container level and those at the component level; it's no problem to have circular reference chains between objects owned by the component classloader; java's garbage collector can handle that fine. But, alas, there is no such functionality. There is simply no way for code at the container level to store data within the container. Well, actually there is - by storing it on static fields of classes loaded by the component's classloader; but you've made that impossible by deploying commons-logging at the container level instead of the component level.

The commons-logging library does provide one last fallback solution; the component being undeployed must tell the commons-logging code at the container level that it is being undeployed so that cleanup can occur. This can be done by adding a section in the webapp's web.xml file to register a ServletContextListener class which calls

```
LogFactory.release(Thread.currentThread().getContextClassLoader());
```

on webapp undeploy. Yes, this is a little ugly. But it's not the fault of commons-logging. By the way, I would not include this class directly in my webapp. I would bundle it as a separate jar (yes, a jar of one class) and deploy it in the WEB-INF/lib directory along with the commons-logging jar file. In future releases of commons-logging this class may in fact be bundled with commons-logging.

But log4j doesn't have this problem

When you deploy log4j in the component's WEB-INF/lib everything works fine. But the same is true for commons-logging.

When you deploy log4j in the container's lib path for a container that is not "log4j-aware" and you don't set up a "Context Repository Selector" for log4j you lose a major feature - logging configuration becomes totally global across all components in the container. Only one logging configuration file is read (from the container's classpath) and every component in the container sees the same logging configuration. This is just not comparable to commons-logging at all. In most cases this is not acceptable behaviour; however commons-logging might provide a feature to behave in this manner if anybody actually thinks this is useful.

When you deploy log4j in the container's lib path and use the "Context Repository Selector" behaviour to get per-component logging configuration AND log4j is also in the component's path you get class cast exceptions (commons-logging currently suffers from the same issue). This is really a separate problem.

When you deploy log4j in the container's lib path and use the "Context Repository Selector" behaviour to get per-component logging configuration and log4j is not in the component's path then you will get memory leaks in exactly the same way as occurs for commons-logging, and for exactly the same reasons.

Can't weak references solve this problem?

Sometimes yes. But sometimes no. If the "commons-optional.jar" included with recent releases of commons-logging is installed alongside commons-logging's jar then LogFactory will use weak references to refer to the component classloaders which automatically avoids this memory leak problem *provided commons-logging is not in the component's classpath*.

Unfortunately due to the lack of facilities to store data on the component's classloader, a commons-logging LogFactory class deployed in the container classpath needs to keep a map of logging configurations keyed by the classloader. But equally unfortunately the values in the map need to be held using strong references as, due to the way logging works, nothing else is holding a reference to the root of the logger tree. If all of the logging adapters are loaded from the container classpath too then the weak-reference solution works and no memory leak occurs on unload despite the abuse of the container model. However if any of the logger objects were loaded from the component's classloader then there is a chain of strong references that leads back to the component's classloader which prevents its garbage collection.

Preventing the loading of logger objects from the component has some ugly implications (for example the configuration file might not be valid if it references a logger only present in the component) so this is not currently available as an option. However the same effect can be had just be ensuring the component doesn't bundle any such classes. And furthermore the ServletContextListener approach also solves this case without hacking commons-logging's behaviour any further.

What about parent-first classloaders?

These are such an abomination I won't even describe the problems these cause. Don't do it. Always use child-first classloaders with components.

What about the class-casting problem

The problem where commons-logging would fail to initialise due to being unable to cast a Log class loaded from a component into a Log class loaded from the container is a separate issue and is not addressed here. And again avoiding abuse of the container model by keeping component libs inside the component will mean this problem never occurs.

Comments by Ceki Gulcu

Simon, I've enjoyed reading your article. Cognizant of the memory leak problem you mention, the version of log4j which is currently CVS HEAD, does "not" include a context selector based on class loaders. The only selector which is included is based on JNDI. Unless I am missing something, ContextJNDISelector does not cause memory leaks. Previous versions of log4j (e.g. 1.2) did not ship with any selectors.