OutOfMemory

How to Deal With Out Of Memory Errors

These errors are rather commonly seen during development phases, and even on production servers. These errors are even more annoying than others, because they do not show any stack trace. The reason for this is that a stack trace would not be of help for these errors. The code that fails with an Out Of Memory will be, in most cases, a "victim" of the problem, and not the problem itself.

Although it is very tempting to blame Tomcat on these errors, the fact is that many of them have their causes in "mistakes" in the webapps. These mistakes usually come from programming patterns and techniques perfectly legal and safe on standalone applications, but that are not correct in a managed environment like a servlet container (that is, Tomcat).

This page will maintain a list of those "well-known mistakes", so anyone experiencing these problems, or wanting to avoid them, could check their webapps and correct them.

The General Rule

The first thing to do is to set the basis for these patterns to be recognized. This way, the developer will be able to find even those mistakes that are not listed in this page, and why not, add them here 🙂

An Out Of Memory can be thrown by several causes:

- A servlet trying to load a several GBytes file into memory will surely kill the server. These kind of errors must be considered a simple bug in our program.
- To compensate for the data your servlet tries to load, you increase the heap size so that there is no room to create the stack size for the threads that need to be created. The memory required by each thread will vary by OS but can be as high as 2M by default and in some OS's (like Debian Sarge) is not reducible with the -Xss parameter. Rule of Thumb, use no more than 1G for heap space in a 32-bit web application.
- Deep recursive algorithms can also lead to Out Of Memory problems. In this case, the only fixes are increasing the thread stack size (`-Xss`), or refactoring the algorithms to reduce the depth, or the local data size per call.
- A webapp that uses lots of libraries with many dependencies, or a server maintaining lots of webapps could exhauste the JVM PermGen space. This space is where the VM stores the classes and methods data. In those cases, the fix is to increase this size. The Sun VM has the flag `-XX:MaxPermSize` that allows to set its size (the default value is 64M)

  > PermGen has been integrated into a new concept called MetaSpace from Java 8 on. The old setting will generate a warning and will be ignored by newer JVMs.

- Hard references to classes can prevent the garbage collector from reclaiming the memory allocated for them when a ClassLoader is discarded. This will occur on JSP recompilations, and webapps reloads. If these operations are common in a webapp having these kinds of problems, it will be a matter of time, until the PermGen space gets full and an Out Of Memory is thrown.

This last case is the one we intend to address here. It is directly related to the fact that the webapp is running on a managed environment, in which code changes can be commited without the application being stopped at all.

Once said this, the patterns to be included here will be those that, although being safe and legal on a standalone application, need to be refactored to make them "compatible" with the servlet container.

Threads

Any threads a web application starts, a web application should stop. ServletContextListener is your friend. Note Tomcat 7 will warn you if you do this and will also provide a (highly dangerous - use at your own risk) option to terminate the threads.

DriverManager

If you load a java.sql.Driver in your own classloader (or servlets), the driver should be removed before undeploying. Each driver is registered in DriverManager which is loaded in system classloader and references the local driver. Note Tomcat will do this for you if you forget.

```
Enumeration<Driver> drivers = DriverManager.getDrivers();
ArrayList<Driver> driversToUnload=new ArrayList<Driver>();
while (drivers.hasMoreElements()) {
        Driver driver = drivers.nextElement();
        if (driver.getClass().getClassLoader().equals(getClass().getClassLoader())) {
                driversToUnload.add(driver);
        }
}
for (Driver driver : driversToUnload) {
    DriverManager.deregisterDriver(driver);
}
```

ThreadLocal

The lifecycle of a ThreadLocal should match that of a request. There is no guarantee that a thread will ever be used to process a request again so if a ThreadLocal is left on the thread at the end of the request there may be no opportunity for the web application to clean it up. Note Tomcat 7 will do this for you.

ContextClassLoader

There are various parts of the Java API that retain a permanent reference to the context class loader. If this happens to be a web application class loader then a memory leak will occur. Tomcat provides workarounds for these where known but there are undoubtedly others.

Logging Frameworks

Most logging frameworks provide a mechanism to release all resources when you have finished with the framework. These should always be used in a container environment.

When all else fails

If you still have a leak then you'll need to debug the root cause. The outline of the process is:

1. You'll need a profiler (I use YourKit), Tomcat and a copy of the app that leaks.
2. Configure Tomcat for use with the profiler. This usually means setting / adding to PATH and CATALINA_OPTS in setenv.(bat|sh)
3. Start Tomcat with the app deployed.
4. Reload the app once.
5. Start up the profiler and connected it to Tomcat.
6. Get a heap dump.
7. Look for instances of WebappClassLoader. If there are more instances than you have apps deployed, you have a leak.
8. If there is a leak, there should be one extra instance of WebappClassLoader.
9. Examine each of the WebappClassLoader objects in turn to find the one where started==false.
10. Trace the GC roots of this object to find out what is holding on to a reference to that object that shouldn't be. That will be the source of the leak.

HTTP sessions

(In response to [1], [2])

Please remember that a JSP page, even one that simply prints out “OK”, will create a session. This is by design and if you do not want it to create a session you need to explicitly indicate that in your JSP. For example:

```
<%@ page session="false" %>
```

This is important in scenarios where you are doing load testing and using custom HTTP clients, because these clients may not be handling sessions correctly and thus end up creating a new session every time they access the page.

One known category of misbehaving clients are web bots. To deal with them you can configure a CrawlerSessionManagerValve.

It is also possible to limit the number of active sessions by setting **maxActiveSessions** attribute on a Manager element, e.g.

```
<Context>
  <Manager maxActiveSessions="500" />
</Context>
```