

INSIDE TECHNOLOGY NOTEBOOK

BEYOND JPROBE STATS—LEVERAGING ASPECTJ'S FLEXIBILITY TO MONITOR PROCESSES

By Dan Christopherson, Technical Architect. danch@nvisia.com

N.B. This article assumes a working knowledge of Aspect-Oriented Programming and a rudimentary grasp of the associated terminology. See <http://www.parc.com/research/projects/aspectj/default.html> for background.

The Problem:

Performance tuning an application is often one of the more difficult challenges in a project. There are many reasons for this, but two main problems are that there can be many causes (e.g. database, network, software, third-party libraries, etc.) and that it can be difficult to collect meaningful and useful data about the execution of the code. Third-party products such as JProbe can prove very useful, but there are times when these tools either aren't available or cannot be used for technical reasons (usually deployment environment). This article outlines an approach that can be used when this situation occurs and can provide significant information that will enable tuning of the software appropriately.

The Situation:

During efforts to performance tune a batch process at the State of Wisconsin, two problems occurred: there were issues with JProbe stability on various platforms and a mechanism was needed to monitor the application once it was deployed to the mainframe. While JAMon was already configured to monitor Spring-based services (Thanks to Jim Loverde), more detailed statistics were needed when the internals of various libraries were used.

The Solution:

AspectJ (a very flexible AOP implementation) was used to 'weave' JAMon statistics gathering with almost anything that could be measured. Here is the recipe:

```
import com.jamonapi.Monitor;
import com.jamonapi.MonitorFactory;

public abstract aspect AbstractTrace {
    abstract pointcut methodselect();

    Object around(): methodselect() {
        Monitor mon =
        MonitorFactory.start(thisJoinPoint.toLongString());
        try {
            return proceed();
        } finally {
            mon.stop();
        }
    }
}
```

[This abstract aspect forms the basis for the facility]

Use an 'around' advice that starts and stops a JAMon TimeMonitor, and an abstract pointcut, to be defined later. This aspect needs to be compiled, either using the ajc command line tool, or an ant build script like this:

```
<project name="hibernate-stats" default="rtw-prepare"
    xmlns:aspectj="antlib:org.aspectj" basedir=".">
    <path id="injars">
        <fileset dir="c:\apps\hibernate">
            <include name="hibernate3.jar"/>
        </fileset>
    </path>
    <path id="classpath">
        <fileset dir="c:\apps\hibernate-3.0\lib">
            <include name="*.jar"/>
        </fileset>
        <fileset dir="c:\apps\aspectj1.5\lib">
            <include name="aspectjrt.jar"/>
        </fileset>
        <fileset dir="c:\apps\JAMon-2.0">
            <include name="JAMon.jar"/>
        </fileset>
    </path>
    <path id="rtw-classpath">
        <path refid="classpath"/>
        <path refid="injars"/>
    </path>
    <target name="prepare">
        <mkdir dir="bin"/>
    </target>
    <target name="postprocess">
        <aspectj:ajc
            outjar="hibernate-annotated.jar"
            sourceRoots=""
            inpathRef="injars"
            classpathRef="classpath"
            fork="true"
            maxmem="1024m"
        />
    </target>
    <target name="rtw-prepare" depends="prepare">
        <aspectj:ajc
            sourceRoots="src"
            destDir="bin"
            classpathRef="rtw-classpath"
            fork="true"
            maxmem="1024m"
            outxml="true"
        />
        <jar destfile="trace-aspectj.jar">
            <fileset dir="bin"/>
            <!-- fileset dir=". -->
                <include name="META-INF/**"/>
            </fileset -->
        </jar>
    </target>
```

There are two ways of using this abstract: 'compile time' (actually post compile in this case) and 'load time'. The most flexible is to use load time weaving, which is what the rest of this article will discuss.

The first step in using load-time weaving is to use an aop.xml file to define the pointcut that was left abstract in the aspect definition. While portions of this build script are dependent on the specific environment, the following example shows how certain Hibernate operations were monitored:

```
<aspectj>
<aspects>
<aspect name="AbstractTrace"/>
<concrete-aspect name="DynHibernateTrace"
    extends="AbstractTrace">
<pointcut name="QLList"
    expression="this(org.hibernate.loader.Loader) AND
call(* *(..))"/>
<pointcut name="init_non_lazy"
    expression="withincode(public void
org.hibernate.engine.PersistenceContext.initializeNonLazyCollections(..) AND
call(* *(..))"/>
<pointcut name="do_query"
    expression="withincode(public void
org.hibernate.loader.Loader.doQuery(..))"/>
<pointcut name="methodselect"
    expression="QLList() || init_non_lazy() ||
do_query()"/>
</concrete-aspect>
</aspects>
<weaver options="">
<include within="org.hibernate..*/>
</weaver>
</aspectj>
```

To use at load time, put the AbstractTrace.class file (the aspect becomes a class when compiled) into a directory, with the aop.xml in a subdirectory named META-INF (i.e. structure it like a jar file—It could be made into a jar, but leaving it expanded makes it easier to modify the pointcuts in the aop.xml between test runs).

To use load-time weaving in JDK 1.4 or earlier, use AspectJ's weaving class loader. The easy way to do this is to use the aj(.bat) script (replacing the java executable) to launch the process. One pitfall, aj expects to get a CLASSPATH environment variable and won't work with the -cp command line option to pass the classpath into the java executable. This CLASSPATH should contain the directory of the AbstractTrace.class.

Setting up load time weaving will be more difficult in an application server environment. One approach is to replace the class loader for the entire app server process, but be cautious since the app server is going to set up new class loaders for the specified war/ear. In some cases, it would be simpler to move the classes of interest from the enterprise app into the server's class loader. Another app server technique would be to use compile-time weaving, defining a concrete aspect like this:

```
public aspect HibernateTrace extends AbstractTrace{
    pointcut queryImpl():
        within(org.hibernate.impl.QueryImpl) &&& call(* *(..));

    pointcut sessionList():
        withincode(* org.hibernate.impl.SessionImpl.list(..) &&&
call(* *(..));

    pointcut QLList():
        withincode(*
org.hibernate.hql.ast.QueryTranslatorImpl.list(..) &&& call(* *(..));
/*
    pointcut QLList():
        (withincode(*
org.hibernate.loader.hql.QueryLoader.list(..) &&& call(* *(..)) ||
        (withincode(* org.hibernate.loader.Loader.list(..) &&&
call(* *(..));
*/
    pointcut QLList():
        this(org.hibernate.loader.Loader) &&& call(* *(..);

    pointcut methodselect(): QLList() || QLList();
}
```

The 'postprocess' target in the above build script shows one example of usage. Note that this will produce a new jar file to *replace* the one into which the process weaves.

For more information and for discussion on this article, please refer to the NVISIA wiki; <https://wiki.nvisia.com>.