Geomajas profiling project guide

Geomajas Developers and Geosparc

Geomajas profiling project guide

by Geomajas Developers and Geosparc

1.0.0

Table of Contents

List of Examples

Chapter 1. Introduction

The profiling project provides a some hooks which allow you to log time spent (and invocation count) in pieces of code. It is especially built to be easy to integrate and have a limited impact on the actual execution time (thanks to the LMAX Disruptor).

The profiling information can be split up into groups.

There is JMX support to allow getting the counter information and resetting through JMX (for example using JConsole).

There is a profiling JDBC driver which can be used to profile JDBC access.

Chapter 2. Configuration

1. Dependencies

Make sure your include the plug-in in your project. If you are using Maven, add the following dependency to your pom:

Example 2.1. Plug-in dependency

```
<dependency>
    <groupId>org.geomajas.project</groupId>
    <artifactId>geomajas-project-profiling</artifactId>
    <version>1.0.0</version>
</dependency>
```

2. Base profiling

For starters, you need to define a profiling container. This is done in your Spring configuration file. A container allows your t read, write and reset counters. You can register data in groups and also get the total for all groups in the container. It is normal to have more than one profiling container in your application.

Example 2.2. Define a profiling container.

```
<bean name="profilingTest" class="org.geomajas.project.profiling.service.Profil
    <property name="ringSize" value="128" />
</bean>
```

You can now autowire the profiling container in your code and register execution time for specific invocations.

Example 2.3. Register profiling data.

```
@Autowired
@Qualifier("restProfiling")
private ProfilingContainer profilingContainer;


long start = System.currentTimeMillis();
// do domething
profilingContainer.register("some.grouping", System.currentTimeMillis() - start
```

In this example a qualifier was used to be able to indicate which profiling container needs to be used. If there is only one, this can be omitted.

3. Using AOP to profile services

You can use AOP to register profiling information for the methods in a bean. You can configure both the profiling container to use and the group for the registration.

Example 2.4. Using AOP to profile services.

```
<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyC
    <property name="beanNames" value="service*"/>
    <property name="interceptorNames">
        <list>
            <value>myMethodProfilingInterceptor</value>
        </list>
    </property>
</bean>

<bean name="myMethodProfilingInterceptor" class="org.geomajas.project.profi
    <property name="group" value="test" />
    <property name="profilingContainer" ref="profilingTest" />
</bean>
```

4. Profiling JDBC access

There is also a profiling JDBC driver which registers profiling data for most JDBC related calls (it attempts to not register calls which should be instantaneous, however this can sometimes depend on the JDBC driver, in which case the profiling will register too little).

When this driver is loaded using `Class.forName("org.geomajas.project.profiling.jdbc.ProfilingDriver")` then you can prefix your JDBC connection string with "profiling:" to assure that the profiling driver is used.

You still have to connect the profiling driver to your profiling container though. This is for example be done using a service like this.

Example 2.5. Service which configures the JDBC profiling.

```
public class JdbcProfiling implements ProfilingListener, InitializingBean {

    @Autowired
    @Qualifier("jdbcMethodProfiling")
    private ProfilingContainer jdbcMethodProfilingContainer;

    /**
     * Registreer de profiling listener.
     */
    public void afterPropertiesSet() {
        ProfilingDriver.addListener(this);
    }

    @Override
    public void register(String group, long durationMillis) {
        jdbcMethodProfilingContainer.register(group, durationMillis);
    }
}
```

5. Making your profiling data available through JMX

You can use AOP to indicate which method (or which beans) need to be profiled. For

Example 2.6. Setup an MBean server to access profiling data.

```
<bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFa
    <property name="locateExistingServerIfPossible" value="true"/>
</bean>

<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
        <map>
            <entry key="bean:name=profilingAopTest" value-ref="profilingTes
        </map>
    </property>
    <property name="assembler">
        <bean class="org.springframework.jmx.export.assembler.InterfaceBase
            <property name="managedInterfaces" value="org.geomajas.project.
        </bean>
    </property>
</bean>
```

Chapter 3. How-to

1. How to use JConsole to connect to the MBeans

If you have configured your system to surface the profiling beans as MBeans, you can use JConsole to connect to your application.

To assure that you do not get RMI marshalling exceptions on the profile bean invocations, you may need to add the profiling jar in the classpath when running JConsole.

This can be done by invoking JConsole using a command like this (you may need to fix the path to the jar):

```
jconsole -J-Djava.class.path=$JAVA_HOME/lib/jconsole.jar:\
$JAVA_HOME/lib/tools.jar:\
~/.m2/repository/org/geomajas/project/geomajas-project-profiling/1.0.0/geomajas
```