BEA JRockit JDK Mission Control

Using the Monitoring and Management APIs

2.0
December 2006

Copyright

Restricted Rights Legend

Trademarks and Service Marks

Contents

Using JMAPI

This document provides a short introduction to the BEA JRockit Monitoring and Management APIs (JMAPI) an API that provides a way to monitor and manage the BEA JRockit JVM.

This section includes information on the following subjects:

- Special Note on Monitoring and Management API Support
- Using the Javadoc
- Getting Started
- Using JMAPI to Subscribe to Events
- Using JMAPI to Access the BEA JRockit Profiler
- Using JMAPI to Access Exception Counting
- Accessing JMAPI from Code Running in BEA JRockit Having a Security Manager

Special Note on Monitoring and Management API Support

If you are using BEA JRockit **5.0 R26.0.0 or later, JMAPI is no longer the preferred management API for** BEA JRockit; however, we have provided these procedures to facilitate legacy versions of the JVM (that is, any version built on J2SE 1.4.2 or earlier, such as BEA JRockit 1.4.2 R26.2.0). Instead of JMAPI, the preferred monitoring and management tool for BEA JRockit and other JVMs is the Monitoring and Management of JVMs API specified by JSR-174. This API provides Java applications, system management tools, and RAS-related tools with the ability to monitor the health of the JVM as well as manage certain run-time controls. It supports JVM monitoring and management from within Java applications as well as remote access by system management applications and RAS-related tools. These are standard interfaces available in the Java 5.0 API, under `java.lang.management`. BEA JRockit provides extensions to these APIs, called JLMEXT, that you can obtain at the BEA JRockit Mission Control dev2dev page.

Note: JLMEXT is still a "work in progress" and is subject to change between releases. Any changes will be noted in future BEA JRockit Mission Control Release Notes.

Using the Javadoc

This document is simply an overview of JMAPI.While it provides basic instructions on how to implement this interface and describes some of its capabilities, the best source of documentation are the Javadocs, available at:

- http://edocs.bea.com/jrockit/releases/R27/javadoc/manapi/docs/index.html
- http://edocs.bea.com/jrockit/releases/5026x/javadoc/jlmext/docs/index.html

Getting Started

To implement JMAPI, you first need to fetch a reference to an actual instance of JVM by using the JVMFactory.

- `JVMFactory` provides a static method to fetch an instance of JVM. This is the starting point for working with the API.
- `JVM` provides basic information about the JVM and is also the interface used to access the different information subsystems available. These subsystems are:
  - `ClassLibrary`, which provides a way to monitor and manage the set of currently loaded Classes and ClassLoaders.
  - `CompilationSystem`, which provides a way to monitor and manage the way methods and constructors are compiled.
  - `Machine`, which provides information about the hardware the JVM is running on, like CPUs, network adapters and memory.
  - `MemorySystem`, which provides heap and garbage collection data.
  - `OperatingSystem`, which passes information about the OS the JVM is running on.

  - `ProfilingSystem`, which provides a way to perform lightweight profiling of the JVM, for instance invocation counting.
  - `ThreadSystem`, which provides thread stack dumps, thread snapshots, thread counts and means to access the threads running in BEA JRockit.

To fetch the instance of `JVM`, you need to add code such as the following:

```
com.bea.jvm.JVM myJVM = com.bea.jvm.JVMFactory.getJVM();
```

From the `JVM` instance you can access the different subsystems, such as the memory system. From the memory system you can, among other things, ask for heap size information or access the `GarbageCollector`. Reading the currently used heap size (in bytes) looks like this:

Listing 1 Reading the Current Heap Size

```
com.bea.jvm.JVM myJVM = com.bea.jvm.JVMFactory.getJVM();
long heapSize = myJVM.getMemorySystem().getUsedHeapSize();
```

To check if we are using a parallel garbage collector with a nursery, you might include something similar to the example in Listing 2:

Listing 2 Checking the Garbage Collector Type

```
com.bea.jvm.GarbageCollector myGC =
myJVM.getMemorySystem().getGarbageCollector();
boolean isParallelWithNursery = myGC.isParallel() &&
   myGC.isGenerational();
```

Using JMAPI to Subscribe to Events

You can use JMAPI to subscribe to a number of different events:

- `ClassLoadEvent`, which reports loaded and unloaded classes.
- `CompilationListener`, which reports compiled methods and constructors.

- `GarbageCollectionEvent`, which is fired after a garbage collection.

Listing 3 shows how to add an anonymous `ClassLoadListener` that prints out the name of the class that was loaded/unloaded:

Listing 3 Adding and Anonymous `ClassLoadListener`

```
JVM myJVM = JVMFactory.getJVM();
myJVM.getClassLibrary().addClassLoadListener(new
   ClassLoadListener()
   {
      public void onClassLoad(ClassLoadEvent event)
      {
         String prefix = (event.getEventType() ==
            ClassLoadEvent.CLASS_LOADED) ? "Loaded" : "Unloaded";
         System.out.println(prefix + " : " +
            event.getClassObject().getName());
      }
   });
```

Listing 4 shows how to add an anonymous `CompilationListener` that prints out the method/constructor that was compiled and the optimization level used.

Listing 4 Adding an Anonymous `CompilationListener`

```
JVM myJVM = JVMFactory.getJVM();
myJVM.getCompilationSystem().addCompilationListener(
   new CompilationListener()
   {
      public void onMethodCompilation(
         CompilationEvent event)
      {
       String prefix = "Compiled " + (event.hasConstructor() ? " constructor " +
         event.getConstructor().getClass().getName() : "method " +
               event.getMethod().getClass().getName());
         System.out.println(prefix + " : Optimization lvl " +
            event.getOptimizationLevel().getDescription());
   }
});
```

Using JMAPI to Access the BEA JRockit Profiler

The BEA JRockit JVM includes a very efficient, low overhead profiler to get method invocation counts and method timing information.

Listing 6 shows how to call a method in an example class (shown in Listing 5), then print out how many times it has been invoked and the total time spent in that method.

Listing 5 Example Class A

```
public class A
{
   public boolean check(Object obj)
   {
      return this.getClass().isInstance(obj);
   }
}
```

Listing 6 Calling a Method in an Example Class

```
ProfilingSystem profiler =
   JVMFactory.getJVM().getProfilingSystem();
A a = new A();
Method [] methods = A.class.getDeclaredMethods();
profiler.setInvocationCountEnabled(methods[0], true);
profiler.setTimingEnabled(methods[0], true);

for (int i = 0; i < 100000; i++) a.check(a);
System.out.println("Profiling system: check method invoked " +
   myJVM.getProfilingSystem().getInvocationCount(methods[0]) + "
      times");
System.out.println("Time spent in method " +
   myJVM.getProfilingSystem().getTiming(methods[0])
      + " ms");
```

Using JMAPI to Access Exception Counting

JMAPI also provides access to an exception counter that allows you to count how many exceptions of a certain class—and, optionally, all of its subclasses—have been thrown. Listing 7 shows an example of counting `IOExceptions`.

Listing 7 Counting `IOExceptions` with JMAPI

```
profiler.setExceptionCountEnabled(IOException.class,
   true, false);
for (int i = 0; i < 10000; i++)
{
   try
{
throw new IOException();
}
catch (Exception e)
{
   // Deliberately left blank.
}
}
System.out.println("Profiling system: exception counts = "
   + myJVM.getProfilingSystem().
      getExceptionCount(IOException.class));
```

Accessing JMAPI from Code Running in BEA JRockit Having a Security Manager

To access JMAPI from code running in BEA JRockit that has a security manager, the permission `com.bea.jvm.ManagementPermission` “`createInstance`” must first be granted to that code.

For more information on how to grant code permissions, see Permissions in the Java^{TM} 2 Standard Edition Development Kit (JDK).

If the code has not been granted the permission, any attempt to access JMAPI will result in a SecurityException being thrown.

Listing 8 shows a simple policy statement, granting all code the permission to access the JMAPI:

Listing 8 Accessing JMAPI from Code Having a Security Manager

```
grant{
   // Needed to access the JRockit Management API.
   permission com.bea.jvm.ManagementPermission "createInstance";
   };
```