Java theory and practice: Instrumenting applications with JMX

Instant visibility -- just add beans

Debuggers and profilers can provide insight into an application's behavior, but we usually only break out these tools when there's a serious problem. Building monitoring hooks into an application can make it easier to understand what your programs are doing without breaking out the debugger. Now that Java Management Extensions (JMX) is built into the Java™ SE platform, and the `jconsole` viewer provides a universal monitoring GUI, using JMX to provide a window into your application is easier and more effective than ever.

Share:

Brian Goetz has been a professional software developer for 20 years. He is a senior staff engineer at Sun Microsystems, and he serves on several JCP Expert Groups. Brian's book, Java Concurrency In Practice, was published in May 2006 by Addison-Wesley. See Brian's published and upcoming articles in popular industry publications.

19 September 2006
Also available in Chinese Russian Japanese

How many times have you looked at a running application and wondered "What the heck is it doing, and why is it taking so long?" In these moments, you probably wish you had built more monitoring capabilities into your application. For example, in a server application, you might be interested in viewing the number and types of tasks queued for processing, the tasks currently in progress, throughput statistics over the past minute or hour, average task processing time, and so on. These statistics are easy to gather, but without an unintrusive means of retrieving the data when it is needed, they are not very useful.

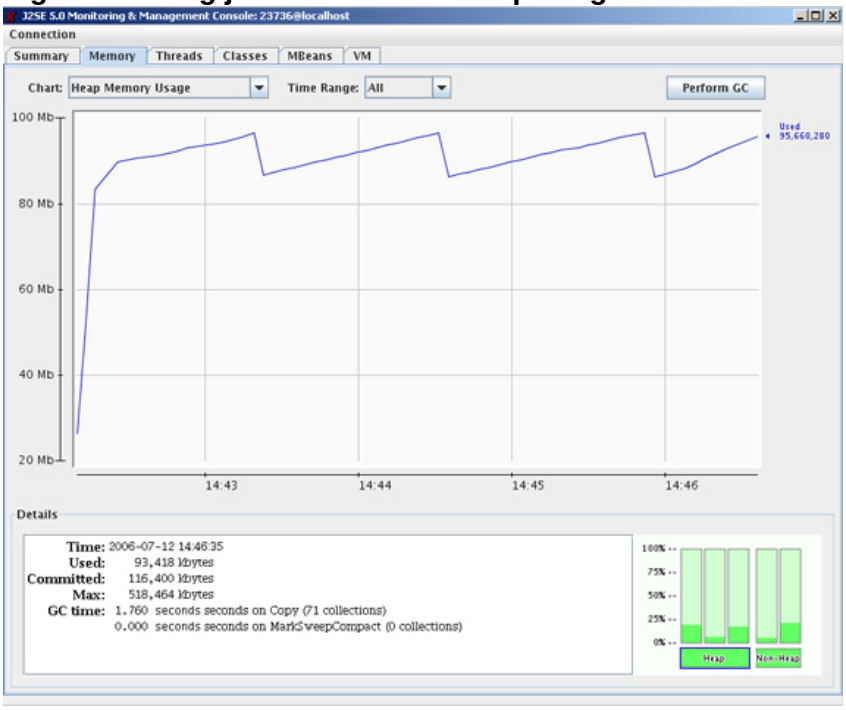
You can export operational data in lots of ways -- you could write periodic statistics snapshots to a log file, create a Swing GUI, use an embedded HTTP server that displays the statistics on a Web page, or publish a Web Service that can be used to query the application status. But in the absence of a monitoring and data publication infrastructure, most application developers do not go to these lengths, resulting in less visibility into the workings of an application than might be desired.

JMX

As of Java 5.0, the class library and JVM provide a comprehensive management and monitoring infrastructure -- JMX. JMX is a standardized means for providing a remotely accessible management interface and is an easy way to add a flexible and powerful management interface to an application. JMX components, called managed beans (MBeans), are JavaBeans that provide accessors and business methods pertaining to the management of an entity. Each managed entity (which could be the entire application or a service within the application) instantiates an MBean and registers it using a human-readable name. A JMX-enabled application relies on an `MBeanServer`, which acts as a container for MBeans, providing remote access, namespace management, and security services. On the client side, the `jconsole` tool can act as a universal JMX client. Taken together, platform support for JMX dramatically reduces the effort required for an application to support an external management interface.

In addition to providing an `MBeanServer` implementation, Java SE 5.0 also instruments the JVM to provide easy visibility into the state of memory management, class loading, active threads, logging, and platform configuration. Monitoring and management for most of the platform services are turned on by default (the performance impact is minimal), so it is just a matter of connecting to the application with a JMX client. Figure 1 shows the `jconsole` JMX client (part of the JDK) displaying one of the memory management views -- heap usage over time. The Perform GC button illustrates that JMX offers the capability to initiate operations in addition to viewing operating statistics.

Figure 1. Using jconsole to view heap usage



Transports and security

JMX specifies a protocol used to communicate between the `MBeanServer` and the JMX client, which can run over a variety of transports. Built-in transports for local connections, RMI, and SSL are provided, and it is possible to create new transports through the JMX Connector API. Authentication is enforced by the transport; the local transport allows you to connect to JVMs running on the local system under the same user ID, and the remote transports can authenticate with passwords or certificates. The local transport is enabled by default under Java 6; to enable it under Java 5.0, you need to define the system property `com.sun.management.jmxremote` when the JVM is launched. The document "Monitoring and Management using JMX" (see Resources) describes the configuration steps to enable and configure transports.

Instrumenting a Web server

Instrumenting an application to use JMX is easy. Like many other remote invocation frameworks (such as RMI, EJB, and JAX-RPC), JMX is interface-based. To create a managed service, you need to create an MBean interface specifying the management methods. You can then create an MBean that implements that interface, instantiates it, and registers it with the `MBeanServer`.

Listing 1 shows an MBean interface for a network service such as a Web server. It providers getters to retrieve configuration information (such as the port number) and operational information (such as whether the service is started). It also contains getters and setters to view and change configurable parameters such as the current logging level and methods to invoke management operations such as `start()` and `stop()`.

Listing 1. MBean interface for a Web server

```
public interface WebServerMBean {
    public int getPort();

    public String getLogLevel();
    public void setLogLevel(String level);

    public boolean isStarted();
    public void stop();
    public void start();
}
```

Implementing the MBean class is usually fairly straightforward, as the MBean interface is supposed to reflect the properties and management operations of an existing entity or service. For example, the `getLogLevel()` and `setLogLevel()` methods in the MBean will simply forward to the `getLevel()` and `setLevel()` methods on the `Logger` being used by the Web server. JMX imposes some naming restrictions; for instance, MBean interface names must end with `MBean`, and the MBean class for the `FooMBean` interface must be called `Foo`. (You can lift this restriction by using a more advanced JMX feature, dynamic MBeans.) Registering the MBean with the default `MBeanServer` is also easy, and is shown in Listing 2:

Listing 2. Registering an MBean with the built-in JMX implementation

```
public class WebServer implements WebServerMBean { ... }

...

WebServer ws = new WebServer(...);
MBeanServer server = ManagementFactory.getPlatformMBeanServer();
server.registerMBean(ws, new ObjectName("myapp:type=webserver,name=Port 8080"));
```

The `ObjectName` passed to `registerMBean()` identifies the managed entity. Because it is anticipated that a given application may contain many managed entities, the name contains a domain ("myapp" in Listing 2) as well as a number of key-value pairs identifying the managed resource within the domain. The keys "name" and "type" are commonly used, and when present, the name should uniquely identify the managed entity across all MBeans of that type within that domain. Other key-value pairs can be specified as well, and the JMX API includes facilities for wildcard matching of object names.

Having created and registered an MBean, you can immediately point `jconsole` at the application (type `jconsole` at the command line) and see its management attributes and operations in the "MBeans" view. Figure 2 shows the Attributes tab in `jconsole` for the new MBean, and Figure 3 shows the Operations tab. Using reflection, JMX figures out which properties are read-only (`Started`, `Port`) and which are read-write (`LogLevel`), and `jconsole` allows you to modify the read-write properties. If the setter for a read-write property throws an exception (such as `IllegalArgumentException`), JMX reports the exception back to the client.

Figure 2. Attributes tab in jconsole for the MBean

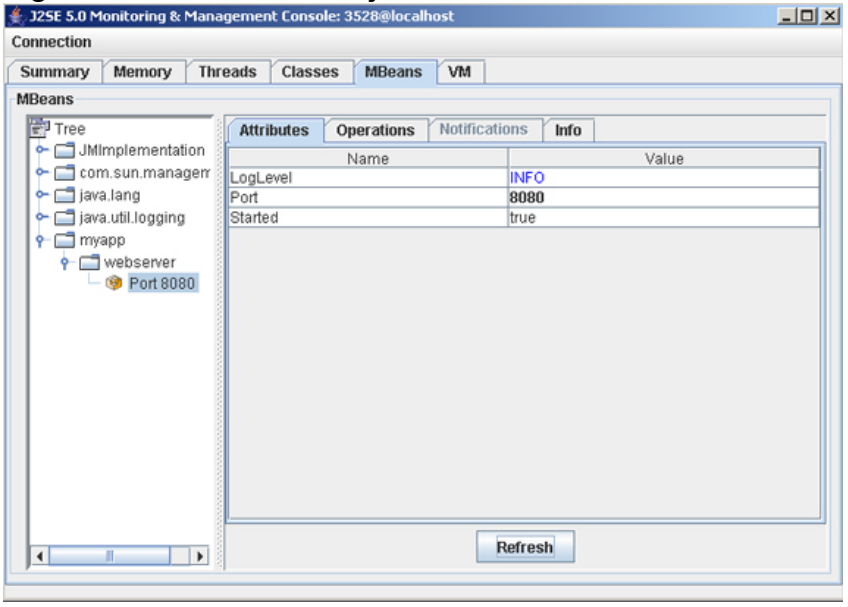
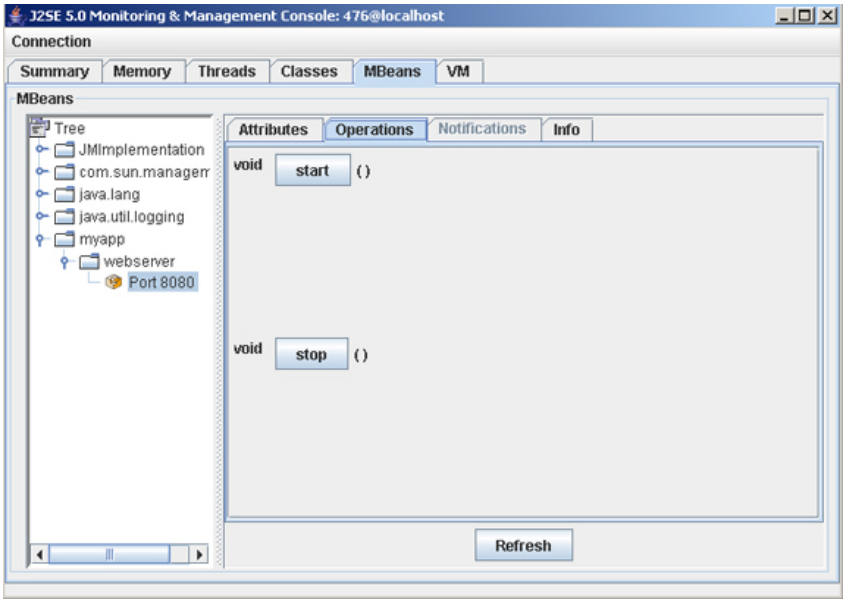


Figure 3. Operations tab in jconsole for the MBean



Data types

Accessors and operations in MBeans can use any primitive type in their signatures, as well as `String`, `Date`, and other standard library classes. You can also use arrays and collections of these permitted types. MBean methods may also use other serializable data types, but doing so can create interoperability issues because the class files must be made available to the JMX client as well. (If you are using the RMI transport, you can use the automatic class downloading features of RMI to deliver required classes to the client.) If you wish to use structured data types in your management interface while avoiding the interoperability problems associated with class availability, you can use the Open MBeans feature of JMX to represent composite or tabular data.

Instrumenting a server application

When creating a management interface, certain parameters and operations suggest themselves as obvious candidates for inclusion, such as configuration parameters, operational statistics, debugging operations (such as changing the logging level or dumping application state to a file), and life cycle operations (start, stop). Retrofitting an application to support access to these attributes and operations is usually quite easy. However, to get the most value out of JMX, you may want to consider at design time what kinds of data would be useful to users and operators at run time.

If you are using JMX to gain insight into what a server application is doing, you need a means of identifying and tracking units of work. If you use the standard `Runnable` and `Callable` interfaces to describe tasks, by making your task classes self-describing (such as by implementing a sensible `toString()` method), you can track tasks as they proceed through their life cycle and provide MBean methods to return lists of waiting, in-process, and finished tasks.

`TrackingThreadPool` in Listing 3 illustrates a subclass of `ThreadPoolExecutor` that keeps track of which tasks are in progress, as well as timing statistics for tasks that have been completed. It accomplishes these tasks by overriding the `beforeExecute()` and `afterExecute()` hooks and providing getters to retrieve the collected data.

Listing 3. Thread pool class that gathers task-in-progress and average task time statistics

```
public class TrackingThreadPool extends ThreadPoolExecutor {
    private final Map<Runnable, Boolean> inProgress
        = new ConcurrentHashMap<Runnable,Boolean>();
    private final ThreadLocal<Long> startTime = new ThreadLocal<Long>();
    private long totalTime;
    private int totalTasks;

    public TrackingThreadPool(int corePoolSize, int maximumPoolSize, long keepAliveTime,
       TimeUnit unit, BlockingQueue<Runnable> workQueue) {
        super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue);
    }
```

```
    protected void beforeExecute(Thread t, Runnable r) {
        super.beforeExecute(t, r);
        inProgress.put(r, Boolean.TRUE);
        startTime.set(new Long(System.currentTimeMillis()));
    }

    protected void afterExecute(Runnable r, Throwable t) {
        long time = System.currentTimeMillis() - startTime.get().longValue();
        synchronized (this) {
            totalTime += time;
            ++totalTasks;
        }
        inProgress.remove(r);
        super.afterExecute(r, t);
    }

    public Set<Runnable> getInProgressTasks() {
        return Collections.unmodifiableSet(inProgress.keySet());
    }

    public synchronized int getTotalTasks() {
        return totalTasks;
    }

    public synchronized double getAverageTaskTime() {
        return (totalTasks == 0) ? 0 : totalTime / totalTasks;
    }
}
```

`ThreadPoolStatusMBean` in Listing 4 shows the MBean interface for a `TrackingThreadPool`, providing counts of active tasks, active threads, completed tasks, waiting tasks, and a list of the tasks currently waiting to execute and currently executing. Including the list of waiting and executing tasks in the management interface allows you to see not only how hard the application is working, but what it is working on right now. This feature can give you insight into not only the behavior of your application, but the nature of the data set it is working on as well.

Listing 4. MBean interface for TrackingThreadPool

```
public interface ThreadPoolStatusMBean {
    public int getActiveThreads();
    public int getActiveTasks();
    public int getTotalTasks();
    public int getQueuedTasks();
    public double getAverageTaskTime();
    public String[] getActiveTaskNames();
    public String[] getQueuedTaskNames();
}
```

If your tasks are heavyweight enough, it might even make sense to take it one step further and register an MBean for each task as it is submitted (and unregister it after it is finished). You could then use the management interface to query each task for what it is doing and how long it is taking or request that the task be canceled.

`ThreadPoolStatus` in Listing 5 implements the `ThreadPoolStatusMBean` interface, providing the obvious implementations for each of the accessors. As is typical with MBean implementation classes, each of the operations is trivial to implement, delegating to the underlying managed object. In this example, the JMX code is entirely separate from the code for the managed entity. `TrackingThreadPool` knows nothing about JMX; it provides its own programmatic management interface by providing management methods and accessors for relevant attributes. You have a choice of implementing the management functionality directly in the implementation class (have `TrackingThreadPool` implement a `TrackingThreadPoolMBean` interface) or implementing it separately (as in Listings 4 and 5).

Listing 5. MBean implementation for TrackingThreadpool

```
public class ThreadPoolStatus implements ThreadPoolStatusMBean {
    private final TrackingThreadPool pool;

    public ThreadPoolStatus(TrackingThreadPool pool) {
        this.pool = pool;
    }

    public int getActiveThreads() {
        return pool.getPoolSize();
    }

    public int getActiveTasks() {
        return pool.getActiveCount();
    }

    public int getTotalTasks() {
        return pool.getTotalTasks();
    }

    public int getQueuedTasks() {
        return pool.getQueue().size();
```

```
    }

    public double getAverageTaskTime() {
        return pool.getAverageTaskTime();
    }

    public String[] getActiveTaskNames() {
        return toStringArray(pool.getInProgressTasks());
    }

    public String[] getQueuedTaskNames() {
        return toStringArray(pool.getQueue());
    }

    private String[] toStringArray(Collection<Runnable> collection) {
        ArrayList<String> list = new ArrayList<String>();
        for (Runnable r : collection)
            list.add(r.toString());
        return list.toArray(new String[0]);
    }
}
```

To illustrate how these classes can provide visibility into what the application is working on, consider a Web crawler application that divides its work into two kinds of tasks: fetching remote pages and indexing the pages. Each task is described by either `FetchTask` or `IndexTask`, shown in Listing 6. You can create a `ThreadPoolStatus` MBean to provide the management interface for the thread pool used to process these tasks and register it with JMX.

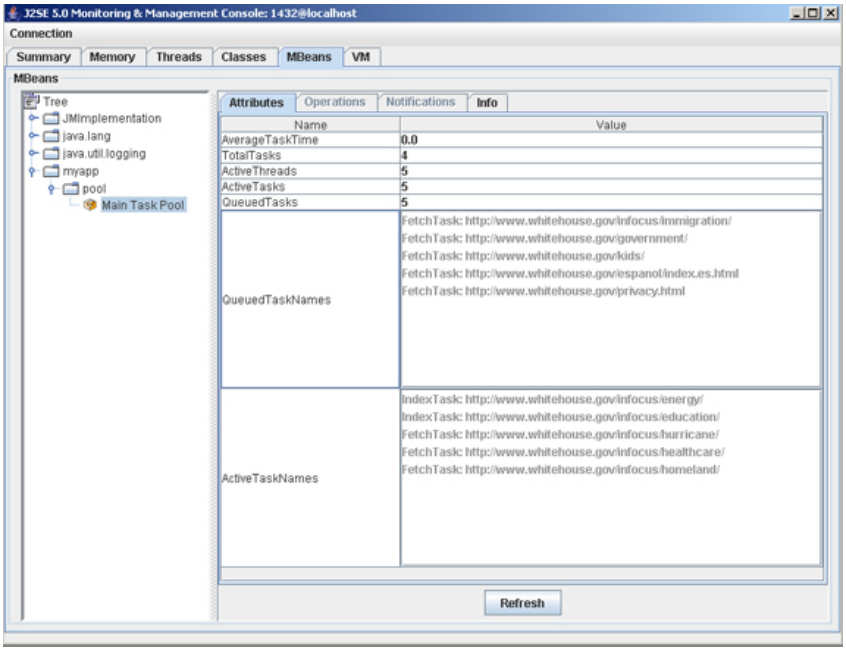
Listing 6. FetchTask class used in Web crawler application

```
public class FetchTask implements Runnable {
    private final String name;

    public FetchTask(String name) {
        this.name = name;
    }

    public String toString() {
        return "FetchTask: " + name;
    }

    public void run() {  /* Fetch remote resource */  }
}
```

As each page is processed by the crawler, new tasks may be queued to fetch pages that are linked from that page, so at any given time there will likely be a mixture of fetching tasks and indexing tasks outstanding. Being able to identify exactly which pages are being processed or waiting to be processed allows you to understand not only your application's performance characteristics, but also the characteristics of the data it is operating on as well.

Figure 4 shows a snapshot of a Web crawler that is processing the whitehouse.gov site. You can see that the home page has already been fetched and indexed, and the crawler is working on fetching and indexing the pages that have been linked directly from it. By hitting the **Refresh** button, you can sample the flow of tasks through the application, which can provide a lot of information about how an application is working without having to introduce extensive logging or run it in the debugger.

Figure 4. Active and queued tasks in a Web crawler application



Summary

The combination of JMX support in the platform and the `jconsole` JMX client offers a painless way to add management and monitoring capabilities to our applications. Even for applications that have no specific management requirements, building in these capabilities allows you to gain insight into how your programs behave and the nature of the data they process -- with very little effort. If your application exports a management interface that allows you to see what it is working on, you can learn more about what it is doing -- and be more confident that it is working as expected -- without resorting to intrusive mechanisms like adding logging code or using debuggers or profilers.

Resources

Learn

"Monitoring and Managing Using JMX" (Sun Microsystems, Inc., 2004): Details the configuration and use of the built-in JMX agent.

JMX Best Practices (Sun Developer Network, 1994-2006): A description of best practices for naming managed objects, selecting JMX features, and choosing data types for managed attributes.

Manage Apache Geronimo with JMX (developerWorks, J. Jeffrey Hanson, August 2006): Learn how the Geronimo application server leverages JMX to facilitate application management.

The Java technology zone: Hundreds of articles about every aspect of Java programming.

Get products and technologies

JMX: Download the JMX specification and documentation.