Production-Time Profiling with Oracle Java Mission Control

Low-overhead profiling and diagnostics for Java applications running on the JVM

Marcus Hirt

There’s a new kid on the JDK tooling block: Oracle Java Mission Control, a production-time profiling and diagnostics tool suite. Starting with the release of Java SE Development Kit 7, Update 40 (JDK 7u40), Oracle Java Mission Control is bundled with Oracle’s Java HotSpot VM. This article will explain why it is worthwhile to take a closer look at this technology, as well as provide pointers on how to get started.

History

JRockit Mission Control originated as part of the development effort for JRockit—a proprietary Java Virtual Machine (JVM)—to provide tooling for analyzing runtime performance. The JRockit team needed information about how real production systems were using JRockit. Requests to get customers to lend us their latest top-secret trading applications for in-house evaluation were usually, understandably, met with blank stares calling our sanity into question. Their applications probably wouldn’t have done us much good anyway, because we wanted production data from systems under real loads. Thus, we decided to build a tool (the JRockit Runtime Analyzer, which later evolved into the JRockit Flight Recorder) with low enough overhead that we could convince customers to actually use it to collect production data for us.

Eventually, we accidentally solved some customer problems using the tool, and customers started asking if they could license the tool. Thus, the idea arose to spend some more resources on the tool to create a commercial tool that would pay for its own development costs, and JRockit Mission Control was born.

After Oracle acquired Sun Microsystems, Oracle suddenly had two of the top three most commonly used general-purpose JVMs on the market. One (the HotSpot JVM) was the open source reference JVM, for which many people knew the codebase and on which licensees based their own versions of the JVM. The other (the JRockit JVM), while being a quick and pretty little thing, was proprietary and only a small number of people knew the codebase. Instead of having to support two JVMs, Oracle wanted to pool the available resources to build a best-of-breed JVM. It was decided that the base would be Java HotSpot VM and that the most-useful features in JRockit would be ported over—one of them being JRockit Mission Control.

In JDK 7u40, the functionality available in Java HotSpot VM reached critical mass, and the first version of Oracle Java Mission Control was released. It mainly contains the equivalents of two of the JRockit Mission Control tools: the Java Management Extensions (JMX) Console and the Java Flight Recorder.

One Place
When profiling capabilities or better diagnostic information is needed, the one-stop shop is really the Java Flight Recorder.
Flight Recorder. There is no online heap analyzer yet. There is, however, a set of quite useful (experimental) plugins that extend Oracle Java Mission Control to do heap dump analysis, do targeted analysis for various Oracle products, or simply extend existing Java Mission Control functionality in more-useful ways.

Getting Started with Oracle Java Mission Control
Starting Oracle Java Mission Control is quite easy. Download and install a recent-enough Java SE JDK (7u40 or later), and then simply run `%JDK_HOME%/bin/jmc`. The alien thing that starts is not, as I am sometimes asked, a native application. It’s Java, but it’s built upon Eclipse RCP technology. If you would rather run Oracle Java Mission Control inside the Eclipse IDE, you can install it into your Eclipse from the Oracle Java Mission Control site.

The JMX Console. The console in Oracle Java Mission Control can be thought of as a JConsole on steroids. As shown in Figure 1, it allows you to monitor JMX data in various ways, to take action when attributes attain certain values, and to persist the data and later look at what was recorded. There are various experimental plugins for the console, such as an Oracle Coherence plugin, a plugin for running JConsole plugins, and a plugin for tweeting messages when an action is triggered.

To connect the console to a JVM, simply choose the JVM process you want to connect to in the JVM browser tree and select Start JMX Console. JVM processes will appear automatically in the JVM browser tree if the JVM is started locally or with the Java Discovery Protocol (JDP). If you have a remote JVM without JDP running, just enable the built-in jmxrmi agent as you normally would to be able to connect with JMX clients such as JConsole.

The console is typically used to monitor a small set of critical attributes, such as the CPU load and Java heap usage, that are sampled at a relatively low frequency. The console can be configured to take action when attributes reach an undesirable value, and one of those actions can be to dump Java Flight Recorder data. The console also contains special tabs for looking at thread information, such as deadlocked threads, per-thread allocation information, and per-thread profiling information. That said, the
console is used for monitoring the runtime. When profiling capabilities or better diagnostic information is needed, the one-stop shop is really the Java Flight Recorder. **The Java Flight Recorder.** The Java Flight Recorder can be thought of as the equivalent of an airplane’s flight data recorder for the Java runtime. While it is running, it records information about the JVM and its environment. When something “interesting” happens, the data in the Java Flight Recorder can be dumped, and the information can be analyzed offline to gain an understanding of why things suddenly went from good to “interesting.” Running the Java Flight Recorder has an almost unnoticeable impact on the performance of a Java application running in the JVM. The overhead is usually well below one percent. This is achieved by a high-performance recording engine built directly into the runtime, which collects data that is already being tracked by the runtime or is already being generated by another activity (as opposed to actively having to do additional work to get the data). There are a lot of interesting things that can be said about the recording engine implementation, but because this is an overview article, I’ll move on to how to use it.

### Creating Flight Recordings

The most important difference between how the recorder worked in JRockit and how it works in Oracle Java Mission Control is that in Java HotSpot VM, two JVM startup flags must be enabled on the JVM for which you want to do flight recordings: `-XX:+UnlockCommercialFeatures` and `-XX:+FlightRecorder`. That was probably the most important line in this article.

There are two different types of recordings, and you can have multiple recordings (of different types) running simultaneously:
- **Timed recordings.** These recordings run for a preconfigured duration. They are automatically stopped when the time is up. If they are initiated from Oracle Java Mission Control, they are automatically downloaded and opened in the Java Flight Recorder user interface when they are done.
- **Continuous recordings.** These recordings have no explicit end time and must be dumped by the end user.

There are three different ways you can do actual recordings, once the parameters are in place:
- **From Oracle Java Mission Control.** This is probably the easiest way. Just point and click.
- **From jcmd.** This is a way to control the Java Flight Recorder from the command line, which is quite useful when you can’t access the machine that is running the JVM of interest from Oracle Java Mission Control and you only have access to a shell.
- **Using command-line flags.** This is handy when you want to always run with a continuous recording or when you want to record the behavior of the JVM right from the very start.

**Figure 2** shows some example recordings.

### Analyzing Flight Recordings

There is a lot of useful information in the flight recordings, and there are a lot of different things the information can be used for, for example:
- **Method profiling.** The Java Flight Recorder will quite happily do method profiling on production systems while causing very low overhead. As a matter of fact, it’s even enabled in the continuous template, so go ahead and use it. It will tell you where the hotspots are in your application. In other words, if you have a CPU-bound problem, the method profiling information will tell you where to optimize to get things to go faster.
- **Garbage collection (GC) profiling.** The GC implementations emit useful events about GC-related activity, such as information that can be used to check on the live set, semireferences, GC pauses (and their individual phases), and so on. This is quite useful for GC tuning, finding out if you’re overusing finalizers, and more.
- **Allocation profiling.** If you notice a lot of garbage collection, but you don’t notice anything strange about the individual GC phases, you might want to reduce the allocation a bit. Allocation profiling will help you see where all that allocation activity is putting a toll on the memory system.
- **Oracle WebLogic Server analysis.** Oracle WebLogic Server produces its own set of events for the Java Flight Recorder. They are quite useful in their own right, but they can also be good for putting all the other recorded information into a context, for example, to see what was really happening during a transaction. This article on the Operative Set feature of the Java Flight Recorder shows some of the capabilities.
- **Latency profiling.** The Java Flight Recorder has many different
events for various thread-stalling activities that can occur, such as blocking on entering a monitor, parking, waiting, and so on. Latency profiling is usually the first place to look if you do not have a CPU-bound problem, but you still have performance issues.

- **OS information.** There is a lot of operating system information, for example, information about CPU load, JVM CPU load, environment variables, and running processes. If you still can’t find what you’re looking for, Oracle Java Mission Control has a DTrace plugin for retrieving everything you ever wanted to know but were too afraid to ask. Note that the overhead for using DTrace, even with very few probes, is usually more than an order of magnitude higher than the overhead for using the Java Flight Recorder, so use with caution. Much more information is available from the event providers built into the JVM, such as class loading and compiler events. One way to learn more about what is available is to take a closer look at the metadata from a recording.

**Conclusion**
As of JDK 7u40, a new tool suite is bundled with the JDK: Oracle Java Mission Control. The main focus of the suite is on production-time profiling and diagnostics. This focus means that the gathered data is quite true to the dynamics of the application being profiled, because the observer effect is kept quite low. In other words, instead of profiling the profiler itself, most of the time is actually spent profiling the application and the runtime.

While the main focus of Oracle Java Mission Control is production systems, it can be quite useful during development, too. It is also free for use during development, per the standard Oracle Binary Code License (BCL).