Java Platform, Standard Edition

Java Flight Recorder Runtime Guide

Release 5.3

E56381-01

March 2014

Describes the Java Flight Recorder runtime implementation and instructions for using the tool.

ORACLE®

Java Platform, Standard Edition Java Flight Recorder Runtime Guide, Release 5.3

E56381-01

Contents

Preface

This document describes the Java Flight Recorder runtime implementation and instructions for using the tool.

Audience

This document is intended for Java developers and support engineers who need an introduction about the architecture and runtime implementation of Java Flight Recorder. It assumes that the reader has basic knowledge of Java.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

About Java Flight Recorder

Java Flight Recorder (JFR) is a tool for collecting diagnostic and profiling data about a running Java application. It is integrated into the Java Virtual Machine (JVM) and causes almost no performance overhead, so it can be used even in heavily loaded production environments. When default settings are used, both internal testing and customer feedback indicate that performance impact is less than one percent. For some applications, it can be significantly lower. However, for short-running applications (which are not the kind of applications running in production environments), relative startup and warmup times can be larger, which might impact the performance by more than one percent. JFR collects data about the JVM as well as the Java application running on it.

Compared to other similar tools, JFR has the following benefits:

- **Provides better data**: A coherent data model used by JFR makes it easier to cross reference and filter events.
- **Allows for third-party event providers**: A set of APIs allow JFR to monitor third-party applications, including WebLogic Server and other Oracle products.
- **Reduces total cost of ownership**: JFR enables you to spend less time diagnosing and troubleshooting problems, reduces operating costs and business interrupts, provides faster resolution time when problems occur, and improves system efficiency.

JFR is primarily used for:

- **Profiling**

  JFR continuously saves large amounts of data about the running system. This profiling information includes thread samples (which show where the program spends its time), lock profiles, and garbage collection details.
- **Black Box Analysis**

  JFR continuously saves information to a circular buffer. This information can be accessed when an anomaly is detected to find the cause.
- **Support and Debugging**

  Data collected by JFR can be essential when contacting Oracle support to help diagnose issues with your Java application.

Understanding Events

Java Flight Recorder collects data about *events*. Events occur in the JVM or the Java application at a specific point in time. Each event has a name, a time stamp, and an optional *payload*. The payload is the data associated with an event, for example, the

CPU usage, the Java heap size before and after the event, the thread ID of the lock holder, and so on.

Most events also have information about the thread in which the event occurred, the stack trace at the time of the event, and the duration of the event. Using the information available in events, you can reconstruct the runtime details for the JVM and the Java application.

JFR collects information about three types of events:

- A *duration event* takes some time to occur, and is logged when it completes. You can set a threshold for duration events, so that only events lasting longer than the specified period of time are recorded. This is not possible for other types of events.
- An *instant event* occurs instantly, and is logged right away.
- A *sample event* (also called *requestable event*) is logged at a regular interval to provide a sample of system activity. You can configure how often sampling occurs.

JFR monitors the running system at an extremely high level of detail. This produces an enormous amount of data. To keep the overhead as low as possible, limit the type of recorded events to those you actually need. In most cases, very short duration events are of no interest, so limit the recording to events with a duration exceeding a certain meaningful threshold.

Understanding Data Flow

JFR collects data from the JVM (through internal APIs) and from the Java application (through the JFR APIs). This data is stored in small thread-local buffers that are flushed to a global in-memory buffer. Data in the global in-memory buffer is then written to disk. Disk write operations are expensive, so you should try to minimize them by carefully selecting the event data you enable for recording. The format of the binary recording files is very compact and efficient for applications to read and write.

There is no information overlap between the various buffers. A particular chunk of data is available either in memory or on disk, but never in both places. This has the following implications:

- Data not yet flushed to a disk buffer will not be available in the event of a power failure.
- A JVM crash can result in some data being available in the core file (that is, the in-memory buffer) and some in the disk buffer. JFR does not provide the capability to merge such buffers.
- There may be a small delay before data collected by JFR is available to you (for example, when it has to be moved to a different buffer before it can be made visible).
- The data in the recording file may not be in time sequential order as the data is collected in chunks from several thread buffers.

In some cases, the JVM drops the event order to ensure that it does not crash. Any data that cannot be written fast enough to disk is discarded. When this happens, the recording file will include information on which time period was affected. This information will also be logged to the logging facility of the JVM.

You can configure JFR to not write any data to disk. In this mode, the global buffer acts as a circular buffer and the oldest data is dropped when the buffer is full. This very low-overhead operating mode still collects all the vital data necessary for root-cause problem analysis. Because the most recent data is always available in the global buffer, it can be written to disk on demand whenever operations or surveillance systems detect a problem. However, in this mode, only the last few minutes of data is available, so it only contains the most recent events. If you need to get the full history of operation for a long period of time, use the default mode when events are written to disk regularly.

Java Flight Recorder Architecture

JFR is comprised of the following components:

- *JFR runtime* is the recording engine inside the JVM that produces the recordings. The runtime engine itself is comprised of the following components:
  - The *agent* controls buffers, disk I/O, MBeans, and so on. This component provides a dynamic library written in C and Java code, and also provides a JVM-independent pure Java implementation.
  - The *producers* insert data into the buffers. They can collect events from the JVM and the Java application, and (through a Java API) from third-party applications.
- *Flight Recorder plugin* for Java Mission Control (JMC) enables you to work with JFR from the JMC client, using a graphical user interface (GUI) to start, stop, and configure recordings, as well as view recording files.

Enabling Java Flight Recorder

By default, JFR is disabled in the JVM. To enable JFR, you must launch your Java application with the `-XX:+FlightRecorder` option. Because JFR is a commercial feature, available only in the commercial packages based on Java Platform, Standard Edition (*Oracle Java SE Advanced* and *Oracle Java SE Suite*), you also have to enable commercial features using the `-XX:+UnlockCommercialFeatures` options.

For example, to enable JFR when launching a Java application named `MyApp`, use the following command:

```
java -XX:+UnlockCommercialFeatures -XX:+FlightRecorder MyApp
```

2 Running Java Flight Recorder

This chapter describes how you can run Java Flight Recorder.

You can run multiple recordings concurrently and configure each recording using different settings; in particular, you can configure different recordings to capture different sets of events. However, in order to make the internal logic of Java Flight Recorder as streamlined as possible, the resulting recording always contains the union of all events for all recordings active at that time. This means that if more than one recording is running, you might end up with more information in the recording than you wanted. This can be confusing but has no other negative implications.

The easiest and most intuitive way to use JFR is through the Flight Recorder plugin that is integrated into Java Mission Control. This plugin enables access to JFR functionality through an intuitive GUI. For more information about using the JMC client to control JFR, see the Flight Recorder Plugin section of the Java Mission Control help.

This chapter explains more advanced ways of running and managing JFR recordings and contains the following sections:

- Section 2.1, "Using the Command Line"
- Section 2.2, "Using Diagnostic Command"
- Section 2.3, "Configuring Recordings"
- Section 2.4, "Creating Recordings Automatically"
- Section 2.5, "Security"
- Section 2.6, "Troubleshooting"

2.1 Using the Command Line

You can start and configure a recording from the command line using the `-XX:StartFlightRecording` option of the `java` command, when starting the application. To enable the use of JFR, specify the `-XX:+FlightRecorder` option. Because JFR is a commercial feature, you also have to specify the `-XX:+UnlockCommercialFeatures` option. The following example illustrates how to run the `MyApp` application and immediately start a 60-second recording which will be saved to a file named `myrecording.jfr`:

```
java -XX:+UnlockCommercialFeatures -XX:+FlightRecorder
-XX:StartFlightRecording=duration=60s,filename=myrecording.jfr MyApp
```

To configure JFR, you can use the `-XX:FlightRecorderOptions` option. For more information, see Appendix A.1, "Command-Line Options".

2.2 Using Diagnostic Command

You can also control recordings by using Java-specific diagnostic commands. For a more detailed description of Diagnostic Commands, see Appendix A.2, "Diagnostic Command Reference".

The simplest way to execute a diagnostic command is to use the `jcmd` tool (located in the Java installation directory). To issue a command, you have to pass the process identifier of the JVM (or the name of the main class) and the actual command as arguments to `jcmd`. For example, to start a 60-second recording on the running Java process with the identifier 5368 and save it to `myrecording.jfr` in the current directory, use the following:

```
jcmd 5368 JFR.start duration=60s filename=myrecording.jfr
```

To see a list of all running Java processes, run the `jcmd` command without any arguments. To see a complete list of commands available to a runnning Java application, specify `help` as the diagnostic command after the process identificator (or the name of the main class). The commands relevant to Java Flight Recorder are:

- `JFR.start`

  Start a recording.

- `JFR.check`

  Check the status of all recordings running for the specified process, including the recording identification number, file name, duration, and so on.

- `JFR.stop`

  Stop a recording with a specific identification number (by default, recording 1 is stopped).

- `JFR.dump`

  Dump the data collected so far by the recording with a specific identification number (by default, data from recording 1 is dumped).

Note: These commands are available only if the Java application was started with the Java Flight Recorder enabled, that is, using the following options:

```
-XX:+UnlockCommercialFeatures -XX:+FlightRecorder
```

2.3 Configuring Recordings

You can configure an explicit recording in a number of other ways. These techniques work the same regardless of how you start a recording (that is, either by using the command-line approach or by using diagnostic commands).

2.3.1 Setting Maximum Size and Age

You can configure an explicit recording to have a maximum size or age by using the following parameters:

maxsize=*size*

Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, `g` or `G` to indicate gigabytes, or do not specify any suffix to set the size in bytes.

```
maxage=age
```

Append the letter `s` to indicate seconds, `m` to indicate minutes, `h` to indicate hours, or `d` to indicate days.

If both a size limit and an age are specified, the data is deleted when either limit is reached.

2.3.2 Setting the Delay

When scheduling a recording. you might want to add a delay before the recording is actually started; for example, when running from the command line, you might want the application to boot or reach a steady state before starting the recording. To achieve this, use the `delay` parameter:

```
delay=delay
```

Append the letter `s` to indicate seconds, `m` to indicate minutes, `h` to indicate hours, or `d` to indicate days.

2.3.3 Setting Compression

Although the recording file format is very compact, you can compress it further by adding it to a ZIP archive. To enable compression, use the following parameter:

```
compress=true
```

Note that CPU resources are required for the compression, which can negatively impact performance.

2.4 Creating Recordings Automatically

When running with a default recording you can configure Java Flight Recorder to automatically save the current in-memory recording data to a file whenever certain conditions occur. If a disk repository is used, the current information in the disk repository will also be included.

2.4.1 Creating a Recording On Exit

To save the recording data to the specified path every time the JVM exits, start your application with the following option:

```
-XX:FlightRecorderOptions=defaultrecording=true,dumponexit=true,dumponexitpath=pat
h
```

Set *path* to the location where the recording should be saved. If you specify a directory, a file with the date and time as the name is created in that directory. If you specify a file name, that name is used. If you do not specify a path, the recording will be saved in the current directory.

2.4.2 Creating a Recording Using Triggers

You can use the Console in Java Mission Control to set *triggers*. A trigger is a rule that executes an action whenever a condition specified by the rule is true. For example, you can create a rule that triggers a flight recording to commence whenever the heap size exceeds 100 MB. Triggers in Java Mission Control can use any property exposed through a JMX MBean as the input to the rule. They can launch many other actions than just Flight Recorder dumps.

Define triggers on the **Triggers** tab of the JMX Console. For more information on how to create triggers, see the Java Mission Control help.

2.5 Security

Java Flight Recorder is intended only for diagnostic purposes. The recording file can potentially contain confidential information such as Java command-line options and environment variables. Use extreme care when you store or transfer the recording files as you would do with diagnostic core files or heap dumps.

Table 2–1 describes security permissions for various methods of using JFR.

Table 2–1 Security Permissions

Method	**Security**
Command line	Anyone with access to the command line of the Java process must be trusted.
Diagnostic commands	Only the owner of the Java process can use `jcmd` to control the process.
Java Mission Control Client	Java Mission Control Client uses JMX to access the JVM.

2.6 Troubleshooting

You can collect a significant amount of diagnostic information from Java Flight Recorder by starting the JVM with one of the following options:

- `-XX:FlightRecorderOptions=loglevel=debug`
- `-XX:FlightRecorderOptions=loglevel=trace`.

A Command Reference

This appendix serves as a basic reference to the commands you can use with Java Flight Recorder. It contains the following sections:

- Section A.1, "Command-Line Options"
- Section A.2, "Diagnostic Command Reference"

A.1 Command-Line Options

When you launch your Java application with the `java` command, you can specify options to enable Java Flight Recorder, configure its settings, and start a recording. The following command-line options are specific to Java Flight Recorder:

- `-XX:+|-FlightRecorder`
- `-XX:FlightRecorderOptions`
- `-XX:StartFlightRecording`

These command-line options are available only in the commercial license of the JDK. To use them, you have to also specify the `-XX:+UnlockCommercialFeatures` option.

Note: You should use `-XX` options only if you have a thorough understanding of your system. If you use these commands improperly, you might affect the stability or performance of your system. `-XX` options are experimental, and they are subject to change at any time.

A.2 Diagnostic Command Reference

This is a description of the diagnostic commands available to control Java Flight Recorder and the parameters available for each command. This information is also available by running the `jcmd` command with the process identifier specified, followed by the `help` parameter and the commad name. For example, to get help information for the `JFR.start` command on a running JVM process with the identifier 5361, run the following:

```
jcmd 5361 help JFR.start
```

To get a full list of diagnostic commands available to the JVM, do not specify the name of the command.

The diagnostic commands associated with Java Flight Recorder are:

- JFR.start
- JFR.check
- JFR.stop
- JFR.dump

A.2.1 JFR.start

The `JFR.start` diagnostic command starts a flight recording. Table A–1 lists the parameters you can use with this command.

Table A–1 JFR.start

Parameter	Description	Type of value	Default
name	Name of recording	String	
settings	Server-side template	String	
defaultrecording	Starts default recording	Boolean	False
delay	Delay start of recording	Time	0s
duration	Duration of recording	Time	0s (means "forever")
filename	Resulting recording filename	String	
compress	GZip compress the resulting recording file	Boolean	False
maxage	Maximum age of buffer data	Time	0s (means "no age limit")
maxsize	Maximum size of buffers in bytes	Long	0 (means "no max size")

A.2.2 JFR.check

The `JFR.check` command shows information about running recordings. Table A–2 lists the parameters you can use with this command.

Table A–2 JFR.check

Parameter	Description	Type of value	Default
name	Recording name	String	
recording	Recording id	Long	1
verbose	Print verbose data	Boolean	False

A.2.3 JFR.stop

The `JFR.stop` diagnostic command stops running flight recordings. Table A–3 lists the parameters you can use with this command.

Table A–3 JFR.stop

Parameter	Description	Type of value	Default
name	Recording name	String	
recording	Recording id	Long	1

Table A–3 (Cont.) JFR.stop

Parameter	Description	Type of value	Default
discard	Discards the recording data	Boolean	
copy_to_file	Copy recording data to file	String	
compress_copy	GZip compress "copy_to_file" destination	Boolean	False

A.2.4 JFR.dump

The `JFR.dump` diagnostic command stops running flight recordings. Table A–4 lists the parameters you can use with this command.

Table A–4 JFR.dump

Parameter	Description	Type of value	Default
name	Recording name	String	
recording	Recording id	Long	1
copy_to_file	Copy recording data to file	String	
compress_copy	GZip compress "copy_to_file" destination	Boolean	False