OpenJDK 17

Using JDK Flight Recorder with OpenJDK
Abstract

OpenJDK 17 is a Red Hat offering on the Red Hat Enterprise Linux platform and Microsoft Windows. The Using JDK Flight Recorder with OpenJDK guide provides an overview of JDK Flight Recorder (JFR) and JDK Mission Control (JMC), and explains how to start the JFR.
# Table of Contents

**MAKING OPEN SOURCE MORE INCLUSIVE** .................................................. 3

**PROVIDING FEEDBACK ON RED HAT DOCUMENTATION** .......................... 4

**CHAPTER 1. INTRODUCTION TO JDK FLIGHT RECORDER** .......................... 5
  1.1. JDK FLIGHT RECORDER (JFR) COMPONENTS .................................. 5
  1.2. DIFFERENCES BETWEEN USING JFR ON JAVA HOTSPOT AND ON MANDREL
      Java HotSpot .................................................. 6
      Mandrel .................................................. 6
  1.3. JDK FLIGHT RECORDER (JFR) STREAMING .................................. 7

**CHAPTER 2. INTRODUCTION TO JDK MISSION CONTROL** ............................ 9
  2.1. JDK MISSION CONTROL (JMC) CONSOLE FUNCTIONALITY .................. 9
  2.2. HISTORICAL ANALYSIS WITH JMC ............................................. 10
  2.3. FLIGHT RECORDING MONITORING WITH THE JMX CONSOLE MBEAN SERVER 11

**CHAPTER 3. BENEFITS OF USING JDK FLIGHT RECORDER** .......................... 12

**CHAPTER 4. STARTING JDK FLIGHT RECORDER** ..................................... 13
  4.1. STARTING JDK FLIGHT RECORDER WHEN JVM STARTS .................... 13
  4.2. STARTING JDK FLIGHT RECORDER ON A RUNNING JVM ................. 13
  4.3. STARTING THE JDK FLIGHT RECORDER ON JVM BY USING THE JDK MISSION CONTROL APPLICATION ........................................... 14
  4.4. DEFINING AND USING THE CUSTOM EVENT API ............................ 15

**CHAPTER 5. CONFIGURATION OPTIONS FOR JDK FLIGHT RECORDER** ............. 18
  5.1. CONFIGURE JDK FLIGHT RECORDER USING THE COMMAND LINE ........ 18
      5.1.1. Start JFR .............................................. 18
      5.1.2. Control behavior of JFR .................................... 19
  5.2. CONFIGURING JDK FLIGHT RECORDER USING DIAGNOSTIC COMMAND (JCMD) ................................................................. 20
      5.2.1. Start JFR .............................................. 21
      5.2.2. Stop JFR .............................................. 21
      5.2.3. Check JFR ............................................ 22
      5.2.4. Dump JFR ............................................ 22
      5.2.5. Configure JFR ........................................ 23
MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright’s message.
PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your feedback on our documentation. To provide feedback, you can highlight the text in a document and add comments.

This section explains how to submit feedback.

Prerequisites

- You are logged in to the Red Hat Customer Portal.
- In the Red Hat Customer Portal, view the document in Multi-page HTML format.

Procedure

To provide your feedback, perform the following steps:

1. Click the Feedback button in the top-right corner of the document to see existing feedback.

   NOTE
   The feedback feature is enabled only in the Multi-page HTML format.

2. Highlight the section of the document where you want to provide feedback.

3. Click the Add Feedback pop-up that appears near the highlighted text.
   A text box appears in the feedback section on the right side of the page.

4. Enter your feedback in the text box and click Submit.
   A documentation issue is created.

5. To view the issue, click the issue tracker link in the feedback view.
CHAPTER 1. INTRODUCTION TO JDK FLIGHT RECORDER

JDK Flight Recorder (JFR) is a low-overhead framework that you can use for monitoring and for profiling Java applications.

You can collect data from events originating within a Java Virtual Machine (JVM) and events originating from application code.

Data is written in memory to a thread-local buffer, transferred to a fixed-size global ring buffer, and then flushed to JFR files (*.jfr) on disk. Other applications can consume these files for analysis. For example, the JDK Mission Control (JMC) tool.

Additional resources

- For more information about JFR, see JEP 328: Flight Recorder in the upstream OpenJDK documentation.

1.1. JDK FLIGHT RECORDER (JFR) COMPONENTS

You can use JFR functionality to observe events that run inside a JVM, and then create recordings from data collected from these observed events.

The following list details key JFR functionality:

Recordings

- You can manage system recordings. Each recording has a unique configuration. You can start or stop the recording, or save it to disk on demand.

Events

- You can use events or custom events to trace your Java application’s data and metadata, and then save the data and metadata from either event type in a JFR file. You can use various tools, such as Java Mission Control (JMC), jcmd, and so on, to view and analyze information stored in a JFR file. The Java Virtual Machine (JVM) has many pre-existing events that are continuously added. An API is available for users to inject custom events into their applications.

- You can enable or disable any event when recording to minimize overhead by supplying event configurations. These configurations take the form of xml documents and are called JFR profiles (*.jfc). The OpenJDK comes with the following two profiles for the most common set of use cases:
  - **default**: The default profile is a low-overhead configuration that is safe for continuous use in production environments. Typically, overhead is less than 1%.
  - **profile**: The profile profile is a low-overhead configuration that is ideal for profiling. Typically, overhead is less than 2%.

1.2. DIFFERENCES BETWEEN USING JFR ON JAVA HOTSPOT AND ON MANDREL

Java applications can run on the Java HotSpot virtual machine (VM) or run as a native image produced by Mandrel. Both approaches support the use of JDK Flight Recorder (JFR), but JFR performs differently on each implementation. Before you choose how to run your application, you must understand the differences in capabilities between the HotSpot virtual machine and Mandrel-compiled native images. Java HotSpot includes many features that Mandrel might not provide.
IMPORTANT

Using JFR through Mandrel is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using Technology Preview features in production. These features provide early access to upcoming product features, enabling customers to test functionality, and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see https://access.redhat.com/support/offerings/techpreview.

Java HotSpot
The Java HotSpot VM is a Java Virtual Machine (JVM) implementation where you can run your Java applications. Java Hotspot is distributed inside the Java Development Kit (JDK) package and the Java Runtime Environment (JRE) package.

The Java HotSpot VM supports JFR, so you can use JFR to monitor a Java application that is running on the Java HotSpot VM. The following list details key features of the Java Hotspot VM that can enhance your use of JFR:

- Includes over 165 built-in events that detail VM and application information, such as classes, system settings, sockets, threads, and so on.
- Includes numerous optimized implementations that can reduce CPU, disk, and memory requirements for JFR, such as StringPool and concurrent read and write capabilities for buffers.
- Includes an object sampler that can sample objects in a Java heap. Object samplers are useful for detecting memory leaks in a Java heap.
- Provides a capability to stream JFR event data through a Java API.
- Uses StackTrace profiling to attach StackTrace data to any event type, such as built-in events or custom events.

Additionally, you can use JFR to monitor the running performance of the Java HotSpot VM, so that you can identify any performance issues. You could then configure the VM to possibly resolve these performance issues.

Mandrel
The Mandrel GraalVM distribution is a minimal version of GraalVM that focuses on the native image capability. Mandrel only includes GraalVM native image functionality and its GraalVM dependencies. Mandrel does not include any other GraalVM components.

With Mandrel, you can compile a Java application to an ahead-of-time (AOT) compiled native image. Because of this capability, native images produced by Mandrel are ideal for deployments in a container that run on OpenShift. When you use AOT compilation on an application, you can perform build-time optimizations that can reduce application startup times and memory footprint.

Mandrel uses Red Hat OpenJDK through a JVM Compiler Interface and a GNU Compiler Collection (GCC) to produce native images of your application. Unlike GraalVM, Mandrel is not polyglot, so Mandrel only supports the Java programming language.

Mandrel supports JFR functionality, so you can use JFR to monitor a Java application that has been compiled to a native image. The following list details key features of Mandrel:
Includes a memory management component that stores Java objects in a Java heap. The memory management component creates the Java heap when the native image starts. This component pre-initializes an initial segment of the heap with contents that were computed when the native image was generated. Using pre-computed heap data can reduce application startup times and memory footprint. The component can use the garbage collection (GC) functionality to scale the size of the Java heap the native image is running.

- Supports the two types of GC: Serial GC and Epsilon.
- Supports some basic JDK Flight Recorder (JFR) functionality, such as using JFR events that are inside the native image to collect information from a running Java application.
- Uses Red Hat OpenJDK as the base JDK, while Graal VM is based on Labs OpenJDK 17.

**NOTE**

When you use Mandrel, you must enable JFR recording for the JFR events at native-image build time, so that you can use this API to record events for a Java application compiled to a native image.

**Additional resources**

- For more information about the Java HotSpot VM, see [HotSpot Group](#) in the upstream OpenJDK documentation.
- For more information about the Native Image, see [Native Image](#) in the GraalVM documentation.

### 1.3. JDK FLIGHT RECORDER (JFR) STREAMING

You can use the JFR Streaming API in OpenJDK 17 to continuously monitor your JDK Flight Recorder (JFR) data from a running Java Virtual Machine (JVM). The API maintains a low-overhead configuration by only subscribing to JFR events that were recently stored to a repository on disk.

OpenJDK 17 improves upon the `jdk.jfr.consumer` package in the `jdk.jfr` module of the JFR framework. The package supports the `EventStream` class, and this class includes the following JFR Streaming capabilities:

- Analyzing JFR data that is stored on disk.
- Allocating the amount of event objects that must be assigned to an event.
- Determining if an event object can be re-used for a new JFR event.
- Managing the order of streaming JFR events.
- Setting a memory buffer size limit for the JFR streaming event.
- Subscribing to a JFR event asynchronously.

You can configure the following components of the `jdk.jfr.consumer` package to suit your JFR Streaming needs:

- `RecordingStream` class, which implements the `jdk.jfr.consumer.EventStream` interface and controls the JFR.
- EventStream class, which collects information of an active recording from any in-memory buffers that are full and then transfers the information to disk.

- EventStream::openRepository(Path) method, which starts a JFR streaming event based on data that is stored in a repository, such as jdk.jfr.repository.

- EventStream::openRepository() method, which receives JFR streaming event data from a streaming recording started by an external process, such as the MBean console in Java Mission Control (JMC) that can start recordings.

- EventStream::openFile(Path) method, which starts streaming a .jfr file data.

**NOTE**

A JVM periodically, about once every second, flushes JFR events that are stored in a thread-local buffer to a repository stored on disk. A separate thread analyzes a .jfr file and then pushes any new data to the streaming event. If you want to receive a notification for each time that a streaming event receives data, you can register a handler in the EventStream::onFlush(Runnable) method.
CHAPTER 2. INTRODUCTION TO JDK MISSION CONTROL

JDK Mission Control (JMC) is a collection of tools to read and analyze JDK Flight Recorder (JFR) files. JMC can also perform real-time monitoring of running Java Virtual Machines (JVMs).

JFR is a production time-profiling and diagnostic engine that is built into JVM. JMC scans a JFR file, .jfr, and searches for patterns in the file’s data. JMC attempts to identify performance and functional issues associated with the file. After JMC analyzes the file, JMC displays a list of issues and their locations. You can use such data to resolve a .jfr file’s performance and functional issues.

JMC includes detailed views and graphs that plot JFR events. In the context of JFR analysis, JMC also consists of the following components:

- JMX Console MBean
- Historical analysis by using flight recordings and hprof files (as of JMC 7.1.0)
- HPROF-dump analyzer

JMC is based on the Eclipse platform. You can extend JMC by adding plug-ins that use the Eclipse RCP API and other specific APIs.

2.1. JDK MISSION CONTROL (JMC) CONSOLE FUNCTIONALITY

You can view performance information of your JVM on the JMC console.

The JMC console can display the following performance information for your running JVM:

- CPU
- Exceptions
- File I/O
- Lock instances
- Memory usage
- Method profiling
- MBean objects
- Socket I/O
- Threads and thread dumps

The JMC console includes the following dedicated pages where you can access specific information associated with your Java application:

- Automated Analysis Results
- Environment
- Class loading
- Compilations
Garbage collection
- GC configuration
- Memory
- TLAB allocations
- VM Operations

JMC includes a Stack Trace view by default. This view displays data selections from JFR pages, such as showing a selected class from a Memory page table. You can update your Stack Trace view by adding the Flame View functionality. Flame View displays stack traces in a graphical representation where stack frames are placed on top of each other. You can use a search bar on Flame View to highlight target stack frames.

Additionally, on the JMC console you can use the following functionality to enhance your interaction with a JVM:
- Use triggers to set actions based on JVM activity, such as displaying a message when the JVM completes a specific operation.
- Use diagnostic commands, such as jcmd commands, to run diagnostics on a JVM.

2.2. HISTORICAL ANALYSIS WITH JMC

JDK Mission Control (JMC) has a suite of pages and tools for displaying flight recording information. You can open flight recording files by selecting the File menu option. Flight recording files automatically open when JMC dumps the flight recording files.

On your JMC console, you can initiate heap dumps in the JVM Browser. After the heap dump operation, the JVM produces a hprof file. You can then use the JOverflow plug-in to see a visual representation of heap dump hprof file. JMC 7.1.0 refactors the JOverflow plug-in and includes the plug-in by default.

The JOverflow plug-in searches for the following memory usage anti-patterns in a hprof file:
- Duplicate strings
- Empty arrays
- Unused collections

You can access JOverflow view on your JMC to view the following information:
- Object selection table, which displays a list of anti-patterns that were flagged by the hprof file analysis.
- Referrer tree table, which displays aggregated reference chains for the hprof file.
- Pie chart and table that shows general information about the hprof file. You can filter information by class on either the pie chart or the table.
- Pie chart that displays objects grouped according to their closest referrer. You can interact with the chart to filter its information.
2.3. FLIGHT RECORDING MONITORING WITH THE JMX CONSOLE MBEAN SERVER

JDK Mission Control (JMC) can use the JMX Console MBean server to monitor flight recording information in real time. You can observe this information on the JVM Browser tab in your JMC console.

You can use the MBean Browser feature on your JMC console to view information for any MBeans related to your JMX Console MBean server. Before you can view MBeans associated with your application, you must expose your application for JMX monitoring and then register the application on the JMX Console MBean server.

On the MBean Browser tab you can complete the following tasks:

- Add attributes to charts.
- Execute numerous MBean operations.
- Modify attribute times.
- View attribute values.
CHAPTER 3. BENEFITS OF USING JDK FLIGHT RECORDER

Some of the key benefits of using JDK Flight Recorder (JFR) are:

- JFR allows recording on a running JVM. It is ideal to use JFR in production environments where it is difficult to restart or rebuild the application.
- JFR allows for the definition of custom events and metrics to monitor.
- JFR is built into the JVM to achieve the minimum performance overhead (around 1%).
- JFR uses coherent data modeling to provide better cross-referencing of events and filtering of data.
- JFR allows for monitoring of third-party applications using APIs.
- JFR helps in reducing the cost of ownership by:
  - Spending less time diagnosing.
  - Aiding in troubleshooting problems.
- JFR reduces operating costs and business interruptions by:
  - Providing faster resolution time.
  - Identifying the performance issues which helps in improving system efficiency.
CHAPTER 4. STARTING JDK FLIGHT RECORDER

4.1. STARTING JDK FLIGHT RECORDER WHEN JVM STARTS

You can start the JDK Flight Recorder (JFR) when a Java process starts. You can modify the behavior of the JFR by adding optional parameters.

Procedure

- Run the `java` command using the `--XX` option.

  ```
  $ java -XX:StartFlightRecording Demo
  ```

  where `Demo` is the name of the Java application.

  The JFR starts with the Java application.

Example

The following command starts a Java process (`Demo`) and initiates an hour-long flight recording which is saved to a file called `demorecording.jfr`:

```
$ java -XX:StartFlightRecording=duration=1h,filename=demorecording.jfr Demo
```

Additional resources

- For a detailed list of JFR options, see [Java tools reference](#).

4.2. STARTING JDK FLIGHT RECORDER ON A RUNNING JVM

You can use the `jcmd` utility to send diagnostic command requests to a running JVM. `jcmd` includes commands for interacting with JFR, with the most basic commands being `start`, `dump`, and `stop`.

To interact with a JVM, `jcmd` requires the process id (pid) of the JVM. You can retrieve this by using the `jcmd -l` command which displays a list of the running JVM process ids, as well as other information such as the main class and command-line arguments that were used to launch the processes.

The `jcmd` utility is located under `$JAVA_HOME/bin`.

Procedure

- Start a flight recording using the following command:

  ```
  $ jcmd <pid> JFR.start <options>
  ```

  For example, the following command starts a recording named `demorecording`, which keeps data from the last four hours, and has size limit of 400 MB:

  ```
  $ jcmd <pid> JFR.start name=demorecording maxage=4h maxsize=400MB
  ```

Additional resources

- For a detailed list of `jcmd` options, see [jcmd Tools Reference](#).
4.3. STARTING THE JDK FLIGHT RECORDER ON JVM BY USING THE JDK MISSION CONTROL APPLICATION

The JDK Mission Control (JMC) application has a Flight Recording Wizard that allows for a streamlined experience of starting and configuring flight recordings.

Procedure

1. Open the JVM Browser.
   
   `$ JAVA_HOME/bin/jmc`

2. Right-click a JVM in JVM Browser view and select **Start Flight Recording**. The Flight Recording Wizard opens.

**Figure 4.1. JMC JFR Wizard**

- **Destination File:**
  
  `/home/alma/flight_recording_1107TheJVMRunningMissionControl_2.jfr`

- **Name:**
  
  `My Recording`

- **Time fixed recording**
  
  **Recording time:** 1 min

- **Continuous recording**
  
  **Maximum size:**
  **Maximum age:**

- **Event settings:**
  
  `Settings for 'My Recording'. last started`

- **Description:**

  These settings were used to start the recording `/home/alma/flight_recording_1107TheJVMRunningMissionControl_1.jfr`.

**Note:** Time fixed recordings will be automatically dumped and opened.

The JDK Flight Recording Wizard has three pages:

- The first page of the wizard contains general settings for the flight recording including:
  
  - Name of the recording
  - Path and filename to which the recording is saved
  - Whether the recording is a fixed-time or continuous recording, which event template will be used
  - Description of the recording

- The second page contains event options for the flight recording. You can configure the level of detail that Garbage Collections, Memory Profiling, and Method Sampling and other events record.

- The third page contains settings for the event details. You can turn events on or off, enable the recording of stack traces, and alter the time threshold required to record an event.
3. Edit the settings for the recording.

4. Click **Finish**.
   The wizard exits and the flight recording starts.

**4.4. DEFINING AND USING THE CUSTOM EVENT API**

The JDK Flight Recorder (JFR) is an event recorder that includes the custom event API. The custom event API, stored in the `jdk.jfr` module, is the software interface that enables your application to communicate with the JFR.

The JFR API includes classes that you can use to manage recordings and create custom events for your Java application, JVM, or operating system.

Before you use the custom event API to monitor an event, you must define a name and metadata for your custom event type.

You can define a JFR base event, such as a **Duration**, **Instant**, **Requestable**, or **Time event**, by extending the `Event` class. Specifically, you can add fields, such as duration values, to the class that matches data types defined by the application payload attributes. After you define an `Event` class, you can create event objects.

This procedure demonstrates how to use a custom event type with JFR and JDK Mission Control (JMC) to analyze the runtime performance of a simple example program.

**Procedure**

1. In your custom event type, in the `Event` class, use the `@name` annotation to name the custom event. This name displays in the JMC graphical user interface (GUI).

   **Example of defining a custom event type name in the `Event` class**

   ```java
   @Name("SampleCustomEvent")
   public class SampleCustomEvent extends Event {...}
   ```

2. Define the metadata for your `Event` class and its attributes, such as name, category, and labels. Labels display event types for a client, such as JMC.

   **NOTE**

   Large recording files might cause performance issues, and this might affect how you would like to interact with the files. Make sure you correctly define the number of event recording annotations you need. Defining unnecessary annotations might increase the size of your recording files.

   **Example of defining annotations for a sample `Event` class**

   ```java
   @Name("SampleCustomEvent")
   @Label("Sample Custom Event")
   @Category("Sample events")
   @Description("Custom Event to demonstrate the Custom Events API")
   @StackTrace(false)
   public class SampleCustomEvent extends Event {
   ```
Details annotations, such as \texttt{@Name}, that define metadata for how the custom event displays on the JMC GUI.

The \texttt{@StackTrace} annotation increases the size of a flight recording. By default, the JFR does not include the stack trace of the location that was created for the event.

The \texttt{@Label} annotations define parameters for each method, such as resource methods for HTTP requests.

The \texttt{@DataAmount} annotation includes an attribute that defines the data amount in bits of bytes. JMC automatically renders the data amount in other units, such as megabytes (MB).

3. Define contextual information in your Event class. This information sets the request handling behavior of your custom event type, so that you configure an event type to collect specific JFR data.

**Example of defining a simple main class and an event loop**

```java
public class Main {

    private static int requestsSent;

    public static void main(String[] args) {
        // Register the custom event
        FlightRecorder.register(SampleCustomEvent.class);
        // Do some work to generate the events
        while (requestsSent <= 1000) {
            try {
                eventLoopBody();
                Thread.sleep(100);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

    private static void eventLoopBody() {
        // Create and begin the event
        SampleCustomEvent event = new SampleCustomEvent();
```

event.begin();
// Generate some data for the event
Random r = new Random();
int someData = r.nextInt(1000000);
// Set the event fields
event.method = "eventLoopBody";
event.number = someData;
event.size = 4;
// End the event
event.end();
requestsSent++;
}

In the preceding example, the simple main class registers events, and the event loop populates
the event fields and then emits the custom events.

4. Examine an event type in the application of your choice, such as the JMC or the JFR tool.

Figure 4.2. Example of examining an event type in JMC

A JFR recording can include different event types. You can examine each event type in your
application.

Additional resources

- For more information about JMC, see Introduction to JDK Mission Control.
CHAPTER 5. CONFIGURATION OPTIONS FOR JDK FLIGHT RECORDER

You can configure JDK Flight Recorder (JFR) to capture various sets of events using the command line or diagnostic commands.

5.1. CONFIGURE JDK FLIGHT RECORDER USING THE COMMAND LINE

You can configure JDK Flight Recorder (JFR) from the command line using the following options:

5.1.1. Start JFR

Use `-XX:StartFlightRecording` option to start a JFR recording for the Java application. For example:

```
java -XX:StartFlightRecording=delay=5s,disk=false,dumponexit=true,duration=60s,filename=myrecording.jfr <<YOUR_JAVA_APPLICATION>>
```

You can set the following `parameter=value` entries when starting a JFR recording:

- **delay=time**
  
  Use this parameter to specify the delay between the Java application launch time and the start of the recording. Append `s` to specify the time in seconds, `m` for minutes, `h` for hours, or `d` for days. For example, specifying `10m` means 10 minutes. By default, there is no delay, and this parameter is set to 0.

- **disk={true|false}**
  
  Use this parameter to specify whether to write data to disk while recording. By default, this parameter is `true`.

- **dumponexit={true|false}**
  
  Use this parameter to specify if the running recording is dumped when the JVM shuts down. If the parameter is enabled and a file name is not set, the recording is written to a file in the directory where the recording progress has started. The file name is a system-generated name that contains the process ID, recording ID, and current timestamp. For example, `hotspot-pid-47496-id-1-2018_01_25_19_10_41.jfr`. By default, this parameter is `false`.

- **duration=time**
  
  Use this parameter to specify the duration of the recording. Append `s` to specify the time in seconds, `m` for minutes, `h` for hours, or `d` for days. For example, if you specify `duration` as `5h`, it indicates 5 hours. By default, this parameter is set to 0, which means there is no limit set on the recording duration.

- **filename=path**
  
  Use this parameter to specify the path and name of the recording file. The recording is written to this file when stopped. For example:
  ```
  · recording.jfr
  · /home/user/recordings/recording.jfr
  ```

- **name=identifier**
  
  Use this parameter to specify both the name and the identifier of a recording.

- **maxage=time**
Use this parameter to specify the maximum number of days the recording should be available on the disk. This parameter is valid only when the disk parameter is set to true. Append s to specify the time in seconds, m for minutes, h for hours, or d for days. For example, when you specify 30s, it indicates 30 seconds. By default, this parameter is set to 0, which means there is no limit set.

maxsize=size

Use this parameter to specify the maximum size of disk data to keep for the recording. This parameter is valid only when the disk parameter is set to true. The value must not be less than the value for the maxchunksize parameter set with -XX:FlightRecorderOptions. Append m or M to specify the size in megabytes, or g or G to specify the size in gigabytes. By default, the maximum size of disk data isn’t limited, and this parameter is set to 0.

path-to-gc-roots={true|false}

Use this parameter to specify whether to collect the path to garbage collection (GC) roots at the end of a recording. By default, this parameter is set to false. The path to GC roots is useful for finding memory leaks. For OpenJDK 17, you can enable the OldObjectSample event which is a more efficient alternative than using heap dumps. You can also use the OldObjectSample event in production. Collecting memory leak information is time-consuming and incurs extra overhead. You should enable this parameter only when you start recording an application that you suspect has memory leaks. If the JFR profile parameter is set to profile, you can trace the stack from where the object is leaking. It is included in the information collected.

settings=path

Use this parameter to specify the path and name of the event settings file (of type JFC). By default, the default.jfc file is used, which is located in JAVA_HOME/lib/jfr. This default settings file collects a predefined set of information with low overhead, so it has minimal impact on performance and can be used with recordings that run continuously. The second settings file is also provided, profile.jfc, which provides more data than the default configuration, but can have more overhead and impact performance. Use this configuration for short periods of time when more information is needed.

NOTE

You can specify values for multiple parameters by separating them with a comma. For example, -XX:StartFlightRecording=disk=false,name=example-recording.

5.1.2. Control behavior of JFR

Use -XX:FlightRecorderOptions option to sets the parameters that control the behavior of JFR. For example:

```
java -XX:FlightRecorderOptions=duration=60s,filename=myrecording.jfr -
XX:FlightRecorderOptions=stackdepth=128,maxchunksize=2M <<YOUR_JAVA_APPLICATION>>
```

You can set the following parameter=value entries to control the behavior of JFR:

```
globalbuffersize=size
```

Use this parameter to specify the total amount of primary memory used for data retention. The default value is based on the value specified for memorysize. You can change the memorysize parameter to alter the size of global buffers.

```
maxchunksize=size
```

Use this parameter to specify the maximum size of the data chunks in a recording. Append m or M to specify the size in megabytes (MB), or g or G to specify the size in gigabytes (GB). By default, the maximum size of data chunks is set to 12 MB. The minimum size allowed is 1 MB.
memorysize=size

Use this parameter to determine how much buffer memory should be used. The parameter sets the globalbuffersize and numglobalbuffers parameters based on the size specified. Append m or M to specify the size in megabytes (MB), or g or G to specify the size in gigabytes (GB). By default, the memory size is set to 10 MB.

numglobalbuffers=number

Use this parameter to specify the number of global buffers used. The default value is based on the size specified in the memorysize parameter. You can change the memorysize parameter to alter the number of global buffers.

old-object-queue-size=number-of-objects

Use this parameter to track the maximum number of old objects. By default, the number of objects is set to 256.

repository=path

Use this parameter to specify the repository for temporary disk storage. By default, it uses system temporary directory.

retransform={true|false}

Use this parameter to specify if event classes should be retransformed using JVMTI. If set to false, instrumentation is added to loaded event classes. By default, this parameter is set to true for enabling class retransformation.

samplethreads={true|false}

Use this parameter to specify whether thread sampling is enabled. Thread sampling only occurs when the sampling event is enabled and this parameter is set to true. By default, this parameter is set to true.

stackdepth=depth

Use this parameter to set the stack depth for stack traces. By default, the stack depth is set to 64 method calls. You can set the maximum stack depth to 2048. Values greater than 64 could create significant overhead and reduce performance.

threadbuffersize=size

Use this parameter to specify the local buffer size for a thread. By default, the local buffer size is set to 8 kilobytes, with a minimum value of 4 kilobytes. Overriding this parameter could reduce performance and is not recommended.

NOTE

You can specify values for multiple parameters by separating them with a comma.

5.2. CONFIGURING JDK FLIGHT RECORDER USING DIAGNOSTIC COMMAND (JCMD)

You can configure JDK Flight Recorder (JFR) using Java diagnostic command. The simplest way to execute a diagnostic command is to use the jcmd tool which is located in the Java installation directory.

To use 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. You can retrieve the JVM or the name of the main class by running jcmd without arguments or by using jps. The jps (Java Process Status) tool lists JVMs on a target system to which it has access permissions.

To see a list of all running Java processes, use the jcmd command without any arguments. To see a complete list of commands available for a running Java application, specify help as the diagnostic command after the process identifier or the name of the main class.
Use the following diagnostic commands for JFR:

5.2.1. Start JFR

Use `JFR.start` diagnostic command to start a flight recording. For example:

```
jcmd <PID> JFR.start delay=10s duration=10m filename=recording.jfr
```

Table 5.1. The following table lists the parameters you can use with this command:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Data type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the recording</td>
<td>String</td>
<td>-</td>
</tr>
<tr>
<td>settings</td>
<td>Server-side template</td>
<td>String</td>
<td>-</td>
</tr>
<tr>
<td>duration</td>
<td>Duration of recording</td>
<td>Time</td>
<td>0s</td>
</tr>
<tr>
<td>filename</td>
<td>Resulting recording file name</td>
<td>String</td>
<td>-</td>
</tr>
<tr>
<td>maxage</td>
<td>Maximum age of buffer data</td>
<td>Time</td>
<td>0s</td>
</tr>
<tr>
<td>maxsize</td>
<td>Maximum size of buffers in bytes</td>
<td>Long</td>
<td>0</td>
</tr>
<tr>
<td>dumponexit</td>
<td>Dump running recording when JVM shuts down</td>
<td>Boolean</td>
<td>-</td>
</tr>
<tr>
<td>path-to-gc-roots</td>
<td>Collect path to garbage collector roots</td>
<td>Boolean</td>
<td>False</td>
</tr>
</tbody>
</table>

5.2.2. Stop JFR

Use `JFR.stop` diagnostic command to stop running flight recordings. For example:

```
jcmd <PID> JFR.stop name=output_file
```

Table 5.2. The following table lists the parameters you can use with this command.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Data type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the recording</td>
<td>String</td>
<td>-</td>
</tr>
<tr>
<td>filename</td>
<td>Copy recording data to the file</td>
<td>String</td>
<td>-</td>
</tr>
</tbody>
</table>
5.2.3. Check JFR

Use **JFR.check** command to show information about the recordings which are in progress. For example:

```
jcmd <PID> JFR.check
```

**Table 5.3.** The following table lists the parameters you can use with this command.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Data type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the recording</td>
<td>String</td>
<td>-</td>
</tr>
<tr>
<td>filename</td>
<td>Copy recording data to the file</td>
<td>String</td>
<td>-</td>
</tr>
<tr>
<td>maxage</td>
<td>Maximum duration to dump file</td>
<td>Time</td>
<td>0s</td>
</tr>
<tr>
<td>maxsize</td>
<td>Maximum amount of bytes to dump</td>
<td>Long</td>
<td>0</td>
</tr>
<tr>
<td>begin</td>
<td>Starting time to dump data</td>
<td>String</td>
<td>-</td>
</tr>
<tr>
<td>end</td>
<td>Ending time to dump data</td>
<td>String</td>
<td>-</td>
</tr>
<tr>
<td>path-to-gc-roots</td>
<td>Collect path to garbage collector roots</td>
<td>Boolean</td>
<td>false</td>
</tr>
</tbody>
</table>

5.2.4. Dump JFR

Use **JFR.dump** diagnostic command to copy the content of a flight recording to a file. For example:

```
jcmd <PID> JFR.dump name=output_file filename=output.jfr
```

**Table 5.4.** The following table lists the parameters you can use with this command.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Data type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the recording</td>
<td>String</td>
<td>-</td>
</tr>
<tr>
<td>filename</td>
<td>Copy recording data to the file</td>
<td>String</td>
<td>-</td>
</tr>
<tr>
<td>maxage</td>
<td>Maximum duration to dump file</td>
<td>Time</td>
<td>0s</td>
</tr>
</tbody>
</table>
5.2.5. Configure JFR

Use `JFR.configure` diagnostic command to configure the flight recordings. For example:

```
jcmd <PID> JFR.configure repositorypath=/home/jfr/recordings
```

Table 5.5. The following table lists the parameters you can use with this command.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Data type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>repositorypath</td>
<td>Path to repository</td>
<td>String</td>
<td>-</td>
</tr>
<tr>
<td>dumppath</td>
<td>Path to dump</td>
<td>String</td>
<td>-</td>
</tr>
<tr>
<td>stackdepth</td>
<td>Stack depth</td>
<td>Jlong</td>
<td>64</td>
</tr>
<tr>
<td>globalbuffercount</td>
<td>Number of global buffers</td>
<td>Jlong</td>
<td>32</td>
</tr>
<tr>
<td>globalbuffersize</td>
<td>Size of a global buffer</td>
<td>Jlong</td>
<td>524288</td>
</tr>
<tr>
<td>thread_buffer_size</td>
<td>Size of a thread buffer</td>
<td>Jlong</td>
<td>8192</td>
</tr>
<tr>
<td>memorysize</td>
<td>Overall memory size</td>
<td>Jlong</td>
<td>16777216</td>
</tr>
<tr>
<td>maxchunksize</td>
<td>Size of an individual disk chunk</td>
<td>Jlong</td>
<td>12582912</td>
</tr>
<tr>
<td>Samplethreads</td>
<td>Activate thread sampling</td>
<td>Boolean</td>
<td>true</td>
</tr>
</tbody>
</table>

Revised on 2022-05-09 11:13:37 UTC