Summary of JDK monitoring and troubleshooting tools

Guide Java JVM About 3224 words About 11 minutes

JDK command-line tools

These commands are in the bin directory under the JDK installation directory:

- `jps` (JVM Process Status): Similar to the UNIX `ps` command. Used to view information such as the startup class, input parameters, and Java virtual machine parameters of all Java processes;
- `jstat` JVM Statistics Monitoring Tool: used to collect various operating data of the HotSpot virtual machine;
- `jinfo` (Configuration Info for Java): Configuration Info for Java, displays virtual machine configuration information;
- `jmap` (Memory Map for Java) : Generate heap dump snapshot;
- `jhat` JVM Heap Dump Browser: Used to analyze heapdump files. It sets up an HTTP/HTML server so that users can view the analysis results in a browser. Jhat was removed in JDK9.
- `jstack` (Stack Trace for Java): Generates a thread snapshot of the virtual machine at the current moment. The thread snapshot is a collection of method stacks being executed by each thread in the current virtual machine.

`jps` : View all Java processes

`jps` The (JVM Process Status) command is similar to the UNIX `ps` command.

`jps` : Displays the main class name of the virtual machine execution and the local virtual machine unique ID (LVMID) of these processes. `jps -q` : Outputs only the local virtual machine unique ID of the process.

```powershell
C:\Users\SnailClimb>jps
7360 NettyClient2
17396
7972 Launcher
16504 Jps
17340 NettyServer
```

`jps -l` : Output the full name of the main class. If the process executes a Jar package, output the Jar path.

```powershell
C:\Users\SnailClimb>jps -l
7360 firstNettyDemo.NettyClient2
17396
7972 org.jetbrains.jps.cmdline.Launcher
16492 sun.tools.jps.Jps
17340 firstNettyDemo.NettyServer
```

`jps -v` : Output the JVM parameters when the virtual machine process is started.

`jps -m` : Output the parameters passed to the Java process main() function.

`jstat` : Monitor various operating status information of virtual machines

jstat (JVM Statistics Monitoring Tool) is a command-line tool used to monitor various virtual machine (VM) operating status information. It displays class information, memory usage, garbage collection, and JIT compilation statistics for local or remote VM processes (requires RMI support on the remote host). On servers without a GUI and offering only a plain text console, jstat is the preferred tool for locating VM performance issues during runtime.

`jstat` Command usage format:

```powershell
jstat -<option> [-t] [-h<lines>] <vmid> [<interval> [<count>]]
```

For example, `jstat -gc -h3 31736 1000 10` to analyze the GC situation of process ID 31736, print a record every 1000ms, stop after printing 10 times, and print the indicator header after every 3 lines.

Common options are as follows:

- `jstat -class vmid` : Display ClassLoader related information;
- `jstat -compiler vmid` : Displays information related to JIT compilation;
- `jstat -gc vmid` : Displays GC-related heap information;
- `jstat -gccapacity vmid` : Displays the capacity and usage of each generation;
- `jstat -gcnew vmid` : Display new generation information;
- `jstat -gcnewcapcacity vmid` : Displays the size and usage of the new generation;

- `jstat -gcold vmid` : Displays statistics on the behavior of the old generation and the permanent generation. Starting from JDK 1.8, this option only shows the old generation because the permanent generation has been removed.
- `jstat -gcoldcapacity vmid` : Displays the size of the old generation;
- `jstat -gcpermcapacity vmid` : Displays the permanent generation size. Starting from JDK 1.8, this option no longer exists because the permanent generation has been removed.
- `jstat -gcutil vmid` : Display garbage collection information;

In addition, by adding `-t` parameters, you can add a Timestamp column to the output information to show the running time of the program.

`jinfo` : View and adjust virtual machine parameters in real time

`jinfo vmid` : Output all parameters and system properties of the current JVM process (the first part is the system properties, the second part is the JVM parameters).

`jinfo -flag name vmid` : Outputs the specific value of the parameter with the corresponding name. For example, outputs MaxHeapSize and checks whether the current JVM process has enabled GC log printing (`-XX:PrintGCDetails` : detailed GC log mode, both of which are disabled by default).

```powershell
C:\Users\SnailClimb>jinfo  -flag MaxHeapSize 17340
-XX:MaxHeapSize=2124414976
C:\Users\SnailClimb>jinfo  -flag PrintGC 17340
-XX:-PrintGC
```

Using jinfo, you can dynamically modify JVM parameters without restarting the virtual machine. This is especially useful in online environments. See the following example:

`jinfo -flag [+|-]name vmid` Enable or disable the parameter with the corresponding name.

```powershell
C:\Users\SnailClimb>jinfo  -flag  PrintGC 17340
-XX:-PrintGC

C:\Users\SnailClimb>jinfo  -flag  +PrintGC 17340

C:\Users\SnailClimb>jinfo  -flag  PrintGC 17340
-XX:+PrintGC
```

`jmap` : Generate a heap dump snapshot

`jmap` The Memory Map for Java command is used to generate a heap dump snapshot. To `jmap` obtain a Java heap dump without the Memory Map for Java command, you can use `“-XX:+HeapDumpOnOutOfMemoryError”` the Memory Map for Java command. This command will cause the VM to automatically generate a dump file after an OOM exception occurs. On Linux, you `kill -3` can also obtain a dump file by sending a process exit signal.

`jmap` The function of is not only to obtain the dump file, it can also query the detailed information of the finalizer execution queue, Java heap and permanent generation, such as space usage, which collector is currently used, etc. Like `jinfo` , `jmap` many functions are also limited on the Windows platform.

Example: Output a heap snapshot of a specified application to the desktop. This heap file can then be analyzed using tools such as jhat and Visual VM.

```powershell
C:\Users\SnailClimb>jmap -
dump:format=b,file=C:\Users\SnailClimb\Desktop\heap.hprof 17340
Dumping heap to C:\Users\SnailClimb\Desktop\heap.hprof ...
Heap dump file created
```

`jhat` : Analyze heapdump files

`jhat` Used to analyze heapdump files, it will establish an HTTP/HTML server so that users can view the analysis results on the browser.

```powershell
C:\Users\SnailClimb>jhat C:\Users\SnailClimb\Desktop\heap.hprof
Reading from C:\Users\SnailClimb\Desktop\heap.hprof...
Dump file created Sat May 04 12:30:31 CST 2019
Snapshot read, resolving...
Resolving 131419 objects...
Chasing references, expect 26 dots..........................
Eliminating duplicate references..........................
Snapshot resolved.
Started HTTP server on port 7000
Server is ready.
```

Visit http://localhost:7000/

Note⚠️ : JDK9 removes jhat (JEP 241: Remove the jhat Tool). You can use its replacement Eclipse Memory Analyzer Tool (MAT) and VisualVM, which is also officially recommended.

`jstack` : Generate a thread snapshot of the virtual machine at the current moment

`jstack` The Stack Trace for Java command generates a thread snapshot of the current moment in the virtual machine. A thread snapshot is a collection of the method stacks currently being executed by each thread in the virtual machine.

The purpose of generating thread snapshots is to identify the cause of prolonged thread pauses, such as inter-thread deadlocks, infinite loops, and long waits for external resource requests. By `jstack` viewing the call stack of each thread when a thread pauses, you can determine what the unresponsive thread is doing in the background or what resource it is waiting for.

The following code shows a thread deadlock. We will use `jstack` the command to check for deadlocks, output deadlock information, and find the deadlocked thread.

```java
public class DeadLockDemo {
    private static Object resource1 = new Object();//资源 1
    private static Object resource2 = new Object();//资源 2

    public static void main(String[] args) {
        new Thread(() -> {
            synchronized (resource1) {
                System.out.println(Thread.currentThread() + "get resource1");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread() + "waiting get resource2");
                synchronized (resource2) {
                    System.out.println(Thread.currentThread() + "get resource2");
                }
            }
        }, "线程 1").start();

        new Thread(() -> {
            synchronized (resource2) {
                System.out.println(Thread.currentThread() + "get
resource2");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread() +
"waiting get resource1");
                synchronized (resource1) {
                    System.out.println(Thread.currentThread() +
"get resource1");
                }
            }
        }, "线程 2").start();
    }
}
```

Output

```plain
Thread[线程 1,5,main]get resource1
Thread[线程 2,5,main]get resource2
Thread[线程 1,5,main]waiting get resource2
Thread[线程 2,5,main]waiting get resource1
```

Thread A acquires the monitor lock on resource1 using synchronized (resource1). `Thread.sleep(1000);` Thread A then sleeps for one second to allow thread B to execute and acquire the monitor lock on resource2. After Thread A and Thread B finish their sleep cycles, they both attempt to acquire each other's resources. As a result, the two threads are stuck waiting for each other, creating a deadlock.

Analysis by `jstack` command:

```powershell
C:\Users\SnailClimb>jps
13792 KotlinCompileDaemon
7360 NettyClient2
17396
7972 Launcher
8932 Launcher
9256 DeadLockDemo
10764 Jps
17340 NettyServer

C:\Users\SnailClimb>jstack 9256
```

Part of the output is as follows:

```powershell
Found one Java-level deadlock:
=============================
"线程 2":
  waiting to lock monitor 0x000000000333e668 (object 0x00000000d5efe1c0, a java.lang.Object),
  which is held by "线程 1"
"线程 1":
  waiting to lock monitor 0x000000000333be88 (object 0x00000000d5efe1d0, a java.lang.Object),
  which is held by "线程 2"

Java stack information for the threads listed above:
===================================================
"线程 2":
        at DeadLockDemo.lambda$main$1(DeadLockDemo.java:31)
        - waiting to lock <0x00000000d5efe1c0> (a java.lang.Object)
        - locked <0x00000000d5efe1d0> (a java.lang.Object)
        at DeadLockDemo$$Lambda$2/1078694789.run(Unknown Source)
        at java.lang.Thread.run(Thread.java:748)
"线程 1":
        at DeadLockDemo.lambda$main$0(DeadLockDemo.java:16)
        - waiting to lock <0x00000000d5efe1d0> (a java.lang.Object)
        - locked <0x00000000d5efe1c0> (a java.lang.Object)
        at DeadLockDemo$$Lambda$1/1324119927.run(Unknown Source)
        at java.lang.Thread.run(Thread.java:748)

Found 1 deadlock.
```

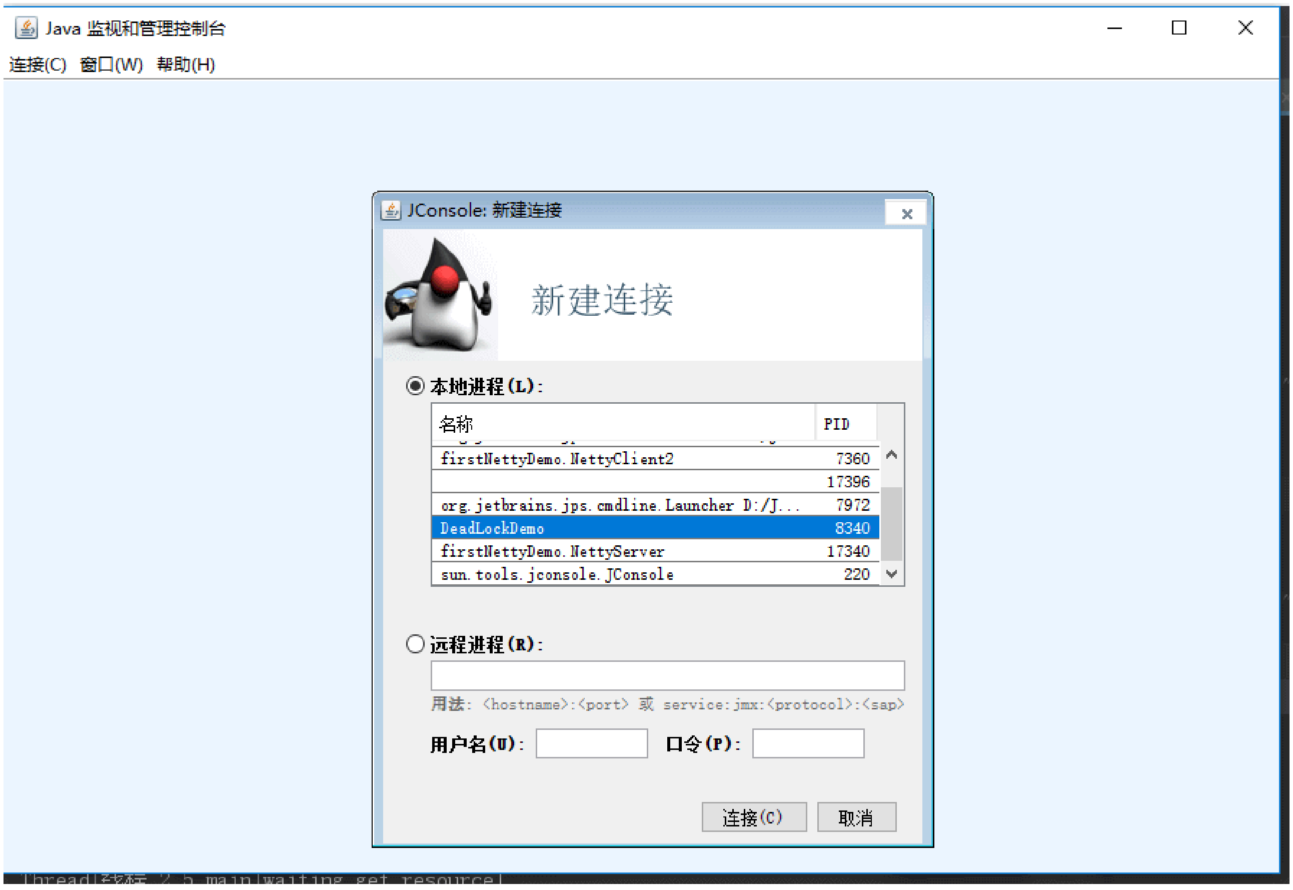
You can see that `jstack` the command has helped us find the specific information of the deadlocked thread.

JDK Visual Analysis Tool

JConsole: Java Monitoring and Management Console

JConsole is a JMX-based visual monitoring and management tool. It easily monitors the memory usage of Java processes on local and remote servers. You can `jconsole` start it by typing a command in the console or by finding it in the bin directory under the JDK directory `jconsole.exe` and double-clicking it.

Connecting to Jconsole



名称	PID
firstNettyDemo.NettyClient2	7360
	17396
org.jetbrains.jps.cmdline.Launcher D:/J...	7972
DeadLockDemo	8340
firstNettyDemo.NettyServer	17340
sun.tools.jconsole.JConsole	220

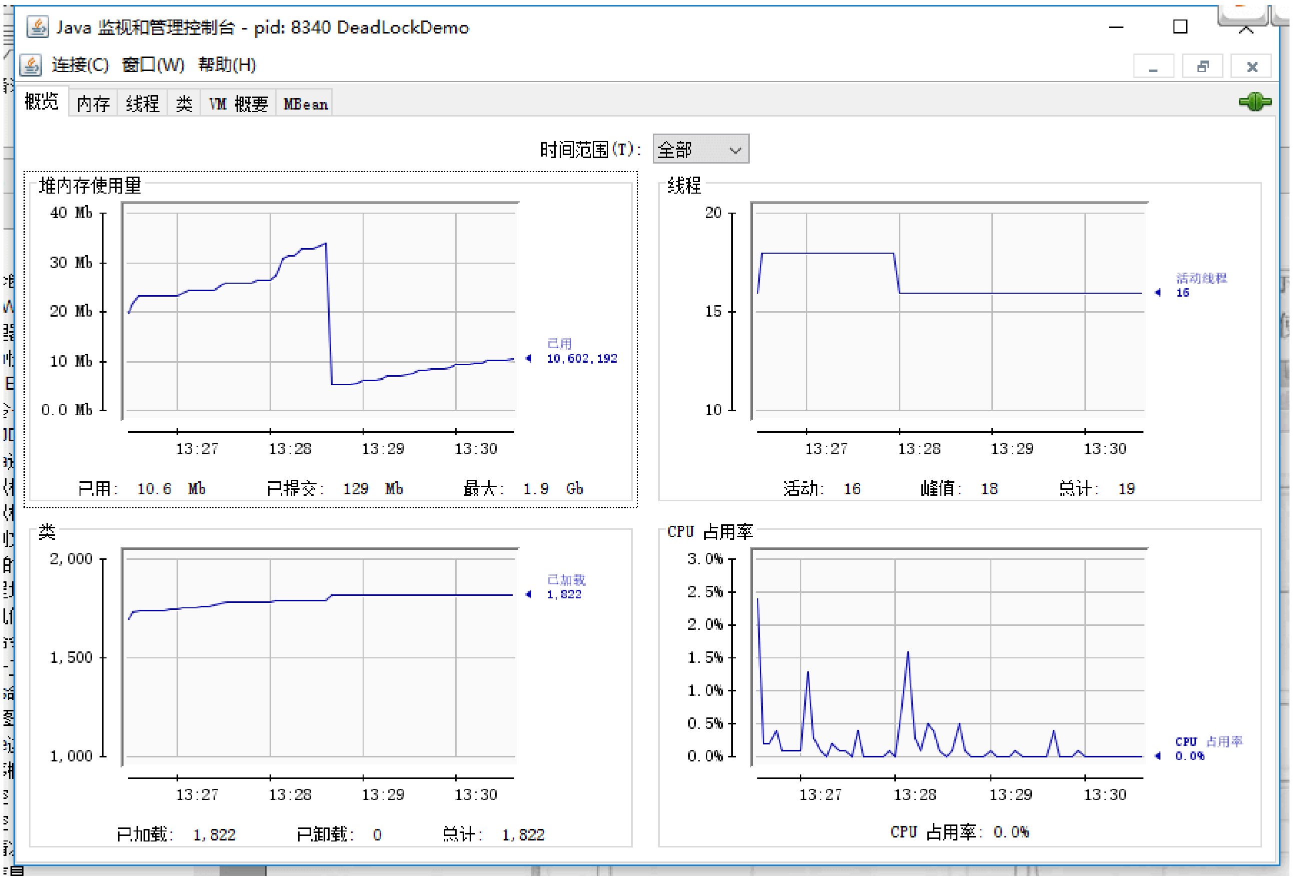
If you need to use JConsole to connect to a remote process, you can add the following parameters when starting the remote Java program:

```properties
-Djava.rmi.server.hostname=外网访问 ip 地址
-Dcom.sun.management.jmxremote.port=60001   //监控的端口号
-Dcom.sun.management.jmxremote.authenticate=false   //关闭认证
-Dcom.sun.management.jmxremote.ssl=false
```

When connecting using JConsole, the remote process address is as follows:

```plain
外网访问 ip 地址:60001
```

View Java Program Overview



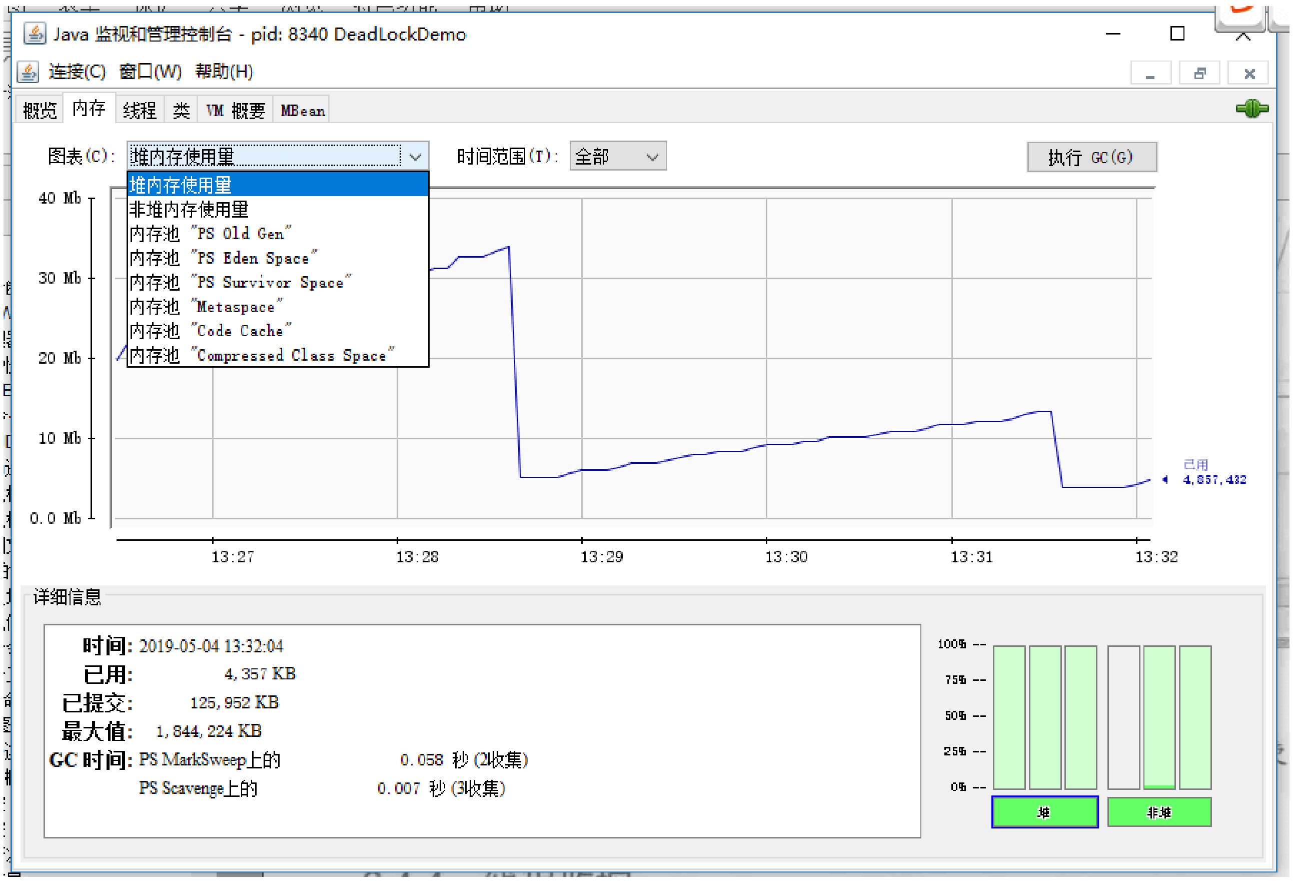
已用: 10.6 Mb 已提交: 129 Mb 最大: 1.9 Gb
活动: 16 峰值: 18 总计: 19
已加载: 1,822 已卸载: 0 总计: 1,822
CPU 占用率: 0.0%

Memory Monitoring

JConsole can display detailed information about current memory usage. This includes not only overall heap and non-heap memory usage, but also detailed information about the usage of Eden and survivor areas, as shown in the following figure.

Click the "Perform GC (G)" button on the right to force the application to perform a F GC.

> - **Minor GC** : refers to the garbage collection action that occurs in the new generation. Minor GC is very frequent and the recovery speed is generally faster.
> - **Old generation GC (Major GC/Full GC)** : refers to the GC that occurs in the old generation. A Major GC is often accompanied by at least one Minor GC (not always). The speed of a Major GC is generally more than 10 times slower than that of a Minor GC.



图表(C): 堆内存使用量
时间范围(T): 全部

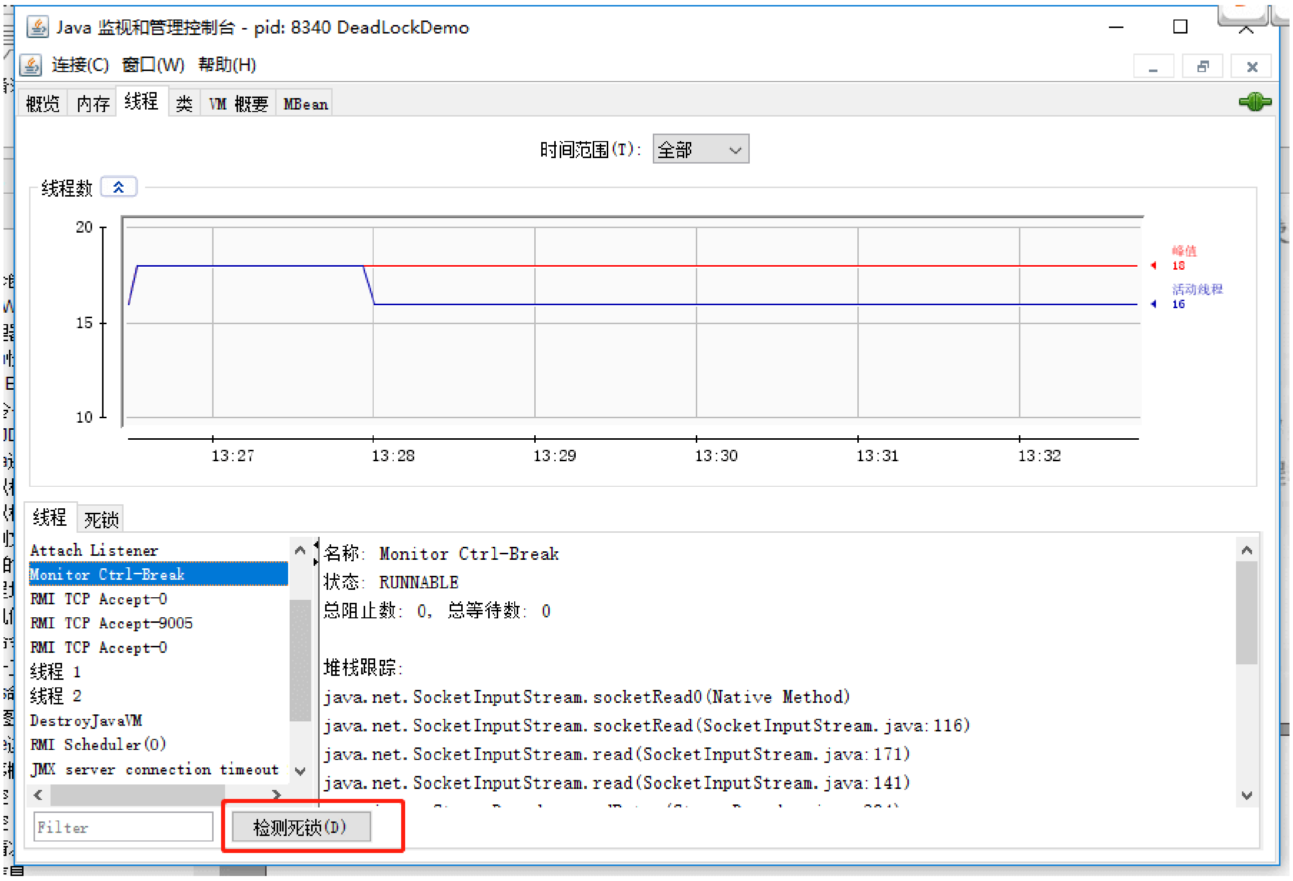
详细信息

时间: 2019-05-04 13:32:04
已用: 4,357 KB
已提交: 125,952 KB
最大值: 1,844,224 KB
GC 时间: PS MarkSweep上的 0.058 秒 (2收集)
PS Scavenge上的 0.007 秒 (3收集)

Thread Monitoring

It is similar to the command we talked about earlier `jstack` , but this one is visual.

There is a "Detect Deadlock (D)" button at the bottom. Clicking this button will automatically find the deadlocked threads and their detailed information for you.



```
Attach Listener
Monitor Ctrl-Break
RMI TCP Accept-0
RMI TCP Accept-9005
RMI TCP Accept-0
线程 1
线程 2
DestroyJavaVM
RMI Scheduler(0)
JMX server connection timeout
```

```
名称: Monitor Ctrl-Break
状态: RUNNABLE
总阻止数: 0, 总等待数: 0

堆栈跟踪:
java.net.SocketInputStream.socketRead0(Native Method)
java.net.SocketInputStream.socketRead(SocketInputStream.java:116)
java.net.SocketInputStream.read(SocketInputStream.java:171)
java.net.SocketInputStream.read(SocketInputStream.java:141)
```

Visual VM: All-in-one troubleshooting tool

VisualVM provides detailed information about Java applications running on the Java Virtual Machine (JVM). Within VisualVM's graphical user interface, you can quickly and easily view information about multiple Java applications. Visit the Visual VM official website: https://visualvm.github.io/ . Visual VM documentation: https://visualvm.github.io/documentation.html .

The following passage is excerpted from "In-depth Understanding of Java Virtual Machine".

> VisualVM (All-in-One Java Troubleshooting Tool) is the most powerful runtime monitoring and troubleshooting tool released with the JDK to date. The official description of VisualVM as "All-in-One" indicates that, in addition to runtime monitoring and troubleshooting, it also offers a wide range of other features, such as performance analysis (profiling). VisualVM's performance analysis capabilities are comparable to those of professional, paid profiling tools like JProfiler and YourKit. Furthermore, VisualVM offers a significant advantage: it does not require the mon

> program to run as a dedicated agent, resulting in minimal impact on the actual application performance, making it suitable for direct use in production environments. This advantage is unmatched by tools like JProfiler and YourKit.

VisualVM is developed based on the NetBeans platform, so it has the feature of plug-in extension from the beginning. With plug-in extension support, VisualVM can do the following:

- Displays virtual machine processes and their configuration and environment information (jps, jinfo).
- Monitor the application's CPU, GC, heap, method area, and thread information (jstat, jstack).
- dump and analyze heap dump snapshots (jmap, jhat).
- Method-level program performance analysis to find the methods that are called the most and have the longest running time.
- Offline program snapshot: Collects program runtime configuration, thread dump, memory dump and other information to create a snapshot, which can be sent to developers for bug feedback.
- Endless possibilities with other plugins...

I won't go into detail about how to use VisualVM here. If you want to learn more, you can read:

- https://visualvm.github.io/documentation.html
- https://www.ibm.com/developerworks/cn/java/j-lo-visualvm/index.html

MAT: Memory Analyzer Tool

MAT (Memory Analyzer Tool) is a fast, convenient, and powerful offline JVM heap memory analysis tool. It displays runtime heap dump snapshots (also available for analysis during normal operation) recorded during JVM exceptions, helping to locate memory leaks or optimize high-memory consumption logic.

When encountering OOM and GC problems, I usually choose to use MAT to analyze the dump file. This is also the most common application scenario of this tool.

For a detailed introduction to MAT, I recommend the following two articles, which are very well written:

- In-depth explanation and practice of JVM memory analysis tool MAT - Getting Sta

- In-depth explanation and practice of JVM memory analysis tool MAT - Advanced Edition

JavaGuide官方公众号
(微信搜索JavaGuide)
1、公众号后台回复"PDF"获取原创PDF面试手册
2、公众号后台回复"学习路线"获取Java学习路线最新版
3、公众号后台回复"开源"获取优质Java开源项目合集
4、公众号后台回复"八股文"获取Java面试真题+面经
JavaGuide

Recently Updated2024/4/19 14:23

Contributors: SnailClimb , JKSAGE , Shuang Kou ,何正海, shuang.kou , guide , anaer , Erzbir , Guide , Mr.Hope , paigeman

Copyright © 2025 Guide