Java Profiling with VisualVM:

- VisualVM is a visual tool integrating several command-line JDK tools and lightweight profiling capabilities. Designed for both production and development time use, it further enhances the capability of monitoring and performance analysis for the Java SE platform.

Tracer Probes:

- Monitor Probes -
  - For monitoring CPU & GC activity, heap & permgen usage, number of loaded classes and application threads.
- JVM Probes -
  - For monitoring various virtual machine internals, such as I/O metrics, GC metrics and HotSpot utilization.
- Jvmstat Probes -
  - For visualizing the metrics exported by the monitored JVM as jvmstat counters (including sun.gc.jvmstat, sun.perfdata.jvmstat, sun.threads.jvmstat, etc.).
- Swing Probes -
  - Provide detailed information about AWT and Swing GUI performance in terms of paints count, layout times and events utilization.
- JavaFX Probes -
  - For monitoring the performance of various logical parts of JavaFX applications.
- DTrace Probes -
  - Provides low-level system metrics, including JVM overhead, utilization of each CPU or syscalls numbers. These probes are available only for the Solaris/ OpenSolaris OS.

Java Memory Types

- When a Java application is started, the Java process pre-allocates a given block of memory from the underlying operating system that is dedicated to the Java process.
- Addressable Java memory consists of a heap, permanent generation, and other memory spaces.
- The other memory space includes variables for JNI, Stack space, and the running Java Virtual Machine (JVM).
- A Java application can store variables in either the stack or heap spaces, depending upon the type and scope of the variable being stored.

VisualVM Available plugins:

- MBean Browser -
  - MBeans Browser plugin provides in general the same functionality as MBeans Browser in JConsole JDK tool: shows MBeans of an application, displays values, operations and notifications. In VisualVM, the browser will be further improved to deliver better usability and support for the latest JMX features.
- Visual GC Plugin -
  - Visual GC attaches to an application and collects and graphically displays garbage collection, class loader, and HotSpot compiler performance data.
- Tracer -
  - Framework and GUI for detailed monitoring and analyzing Java applications. Using various types of probes, the Tracer gathers metrics from an application and displays the data in a timeline. The data are displayed both graphically and in a table and can be exported to common formats for further processing in external tools.
- Thread Inspector -
  - Enables analyzing stack trace(s) of one or more selected threads directly without requiring you to take and open full thread dumps. This is extremely useful for quick and easy analyzing of various threading problems.
- BTrace -
  - BTrace is a dynamic tracing tool for Java. With this plugin you can create, deploy and save BTrace tracing scripts directly from the VisualVM.
- VisualVM Extensions -
  - The intent of this module is to add support for additional functionality (such as new JDKs, JVMs, HotSpot versions, etc.) not supported by the VisualVM core modules at the time VisualVM was released. It's always a good idea to get this plugin for a fresh VisualVM installation.
- Security -
  - The GUI for setting the keystore, truststore, protocols and ciphers for SSL/TLS connections in VisualVM. Using the plugin is equivalent to setting appropriate system properties javax.net.ssl.* and javax.rmi.ssl.client.*

Java stack

- The Java stack space consists of local variables, method calls, arguments, reference variables, intermediate computations, and return values, if any, corresponding to the method invoked. Primitive data type variables such as int, long, float and double are also stored in the stack space.
- Every thread, including the main thread and daemon threads, gets own stack space but will share the same heap space.
- The memory-for-stack space does not need to be contiguous and follows a Last In, First out (LIFO) algorithm.

Java permanent generation

- The Java permanent generation space, or permGen, consists of reflective data of the virtual machine such as Class and Method objects. When new Class or Method types are created at runtime, new space is allocated in the permGen space for these types.

Java heap

- The Java heap spaces consist of instances of Objects, instance variables, and instance-level references to Objects.
- There are three types of heaps:
    - Eden Space (young generation):
        - pool from which memory is initially allocated for most objects.
    - Survivor Spaces (young generation):
        - two pools containing objects that have survived GC of Eden space.
    - Tenured Generation (old generation):
        - pool containing objects that have existed for some time in the survivor spaces.

Java Heap Options:
-Xms<n[k|m]> set initial Java heap size
-Xmx<n[k|m]> set maximum Java heap size
-Xss<n[k|m]> set java thread stack size

What is a Memory Leak in Java?

- The standard definition of a memory leak is a scenario that occurs when objects are no longer being used by the application, but the Garbage Collector is unable to remove them from working memory – because they're still being referenced. As a result, the application consumes more and more resources – which eventually leads to a fatal

- Buffer Monitor -
    - Monitors usage of direct buffers created by ByteBuffer.allocateDirect and mapped buffers created by FileChannel.map. Note that the buffers monitoring requires the monitored application to run JDK 7 starting from Build 36.
- Kill Applications -
    - Kills monitored applications that became unresponsive.
- JVM Capabilities -
    - Displays capabilities of monitored application's JVM.
- JConsole plugins container -
    - Provides support for using existing JConsole plugins inside VisualVM.
- Thread Dump Analyzer (TDA) -
    - The Thread Dump Analyzer (TDA) for Java is a small GUI for analyzing Thread Dumps and Heap Information generated by the Sun Java VM. It provides statistics about thread dumps, gives information about locked monitors, waiting threads and much more. http://java.net/ projects/tda
- JTop -
    - A plugin for monitoring per-thread CPU usage and state.
- Top Threads -
    - Plugin for monitoring per-thread CPU usage and state. http://lsd.luminis.nl/top-threads-plugin-for-jconsole/

Fighting Java memory leak in production:

- Memory leaks in Java are typical, and we have some fancy tools to fight them. Jprofiler, VisualVM, and JMC are some of the popular ones for Java.

Reasons for production memory leaks:

- Unexpected External Party Behaviors:
    - In production, software systems may be integrated with several external third-party systems. Those third-parties can behave in some unexpected ways. Behaviors might cause our side to contribute to the memory leak.
- Longer uptime:
    - Production systems are usually expected to run for longer times without giving any restarts. During these longer runs, it can accumulate tiny bits of memory leak, which will not look significant during the local

OutOfMemoryError.

- We have two types of objects – referenced and unreferenced; the Garbage Collector can remove objects that are unreferenced. Referenced objects won't be collected, even if they're actually not longer used by the application.

Java Heap Leaks:

- An advantageous technique to understand these situations is to make reproducing a memory leak easier by setting a lower size for the Heap. That's why, when starting our application, we can adjust the JVM to suit our memory needs.

Static Field Holding On to the Object Reference.

- First, we need to pay close attention to our usage of static; declaring any collection or heavy object as static ties its lifecycle to the lifecycle of the JVM itself, and makes the entire object graph impossible to collect.

Calling String.intern() on Long String.

- Please remember that interned String objects are stored in PermGen space – if our application is intended to perform a lot of operations on large strings, we might need to increase the size of the permanent generation.

Unclosed Streams.

- We always need to remember to close streams manually, or to make a use of the auto-close feature introduced in Java 8.

Unclosed Connections:

- We need to always close connections in a disciplined manner.

Adding Objects with no hashCode() and equals() into a HashSet:

- It's crucial to provide the hashCode() and equals() implementations.

How to Find Leaking Sources in Your Application?

- Verbose Garbage Collection.
- Do Profiling.
- Review Your Code.

The best and most reliable way to reproduce memory leaks is to simulate the usage patterns of a production environment as close as possible, with the help of a good suite of performance tests.

runs. These kinds of memory leaks are very hard to reproduce in the Testbed, as their accumulation rate is pretty small so that we need to keep the system running with a load for a longer time.

  - Slowly accumulating memory leak objects will climb the ladder slowly and come to the top of the histograms. So, we need continuous monitoring while we are trying to reproduce.
- Unexpected Outages in Production:
  - In a big complicated system, you assume most of the external services are up and running 24/7. But there can be database outages or third-party servers that can go down for a period of time.

How can we hunt for memory leak in production?

- Take the memory dump using the following command: jmap -histo:live <process-id>
- Time gap between the jmap-histos:
- Compare the memory usage of certain local objects

Improvisations to Make the System More Traceable:

- Better to Monitor the Production Servers Time to Time During the Early Stages:
  - If we fail to take these memory-histos, we may have to wait for weeks or months to monitor and understand the reason for the memory leak.
- Introduce a Seconds-Based System Progress Summary Log:
  - These progress log always gives you an indication of the current status of your system. Especially when there is a weird situation happens due to the external entity, it records the number of such failures per second. If there is a connection loss or a third-party system is down, incoming requests would fail due to the dependency on those third-party systems. There is a chance that these failures can lead to a memory leak.
- Enable GC Logs:
  - These GC logs can tell us whether the memory leak has a smooth increment or if it starts suddenly due to some external reason.
- Print Proper Error Logs for Exceptional Cases:
  - These error logs will give us an idea of how production systems are experiencing failures and error situations. To see whether

What is java profiling?

- Java Profiling is the process of monitoring various JVM level parameters such as Method Execution, Thread Execution, Object Creation and Garbage Collection.
- Java Profiling provides you with a finer view of your target application execution and its resource utilization.
- Java Profiling eliminates the need to spend long hours going through the code, and pinpoints the problems associated with your application.
- Java Profiling provides complete and accurate statistical information that helps you trace coding errors in your application.
- Java Profiling helps you diagnose and resolve performance problems, memory leaks and multi-threading problems in your Java or J2EE server applications to ensure the highest level of stability and scalability for your applications. Java Profiling provides you with a root-cause analysis of these problems and helps you resolve them.

1. Early detection and resolution of bottlenecks and performance related issues is a critical step towards improving the stability, efficiency and overall performance of an application.
2. Java Profiling should be performed early and at regular intervals in the software development life-cycle.
3. Java Profiling must be performed before the code is finally executed and deployed to identify critical issues that affect the stability and quality of the software.

- Java Profiling may thus be summarized as measuring statistics of a Java application, specifically in terms of:
    - CPU time being utilized per method
    - Memory being utilized
    - Method call information
    - Objects being created
    - Objects being garbage collected

errors are causing the memory leak, we can compare the number of objects in the memory with the number of such error messages.

Note: Information gathered in this document has been collected from various sources on the Internet.

Sources: https://dzone.com/articles/fighting-java-memory-leak-in-production-systems