DOWNLOAD PDF MEMORY MANAGEMENT IN THE JAVA HOTSPOT VIRTUAL MACHINE

Chapter 1 : Java Profiler Knowledge Base - JVM Memory Structure

management, thereby shielding the developer from the complexity of explicit memory management. This paper provides a broad overview of memory management in the Java HotSpot virtual machine (JVM) in Sun's J2SE release.

However, garbage collecting events have an impact on performance. In systems which require high availability and responsiveness to data changes, this impact should be minimised. The focus of this article is on the heap area where memory for all class instances and arrays is allocated. During execution of a Java program, threads allocate memory for newly-created objects on the heap, but do not reclaim it when these objects are not required anymore. With time the heap, which has a limited amount of memory space, is filled with unreachable objects: The JVM garbage collector divides the heap into smaller parts called generations. Young Generation and Old or Tenured Generation. Young Generation The Young Generation is an area where all new objects are allocated and aged 2. At the end of the process, only unreachable objects are left in the garbage collected young generation area. That means the area can be reclaimed and used for the allocation of new objects. These areas are used to improve the minor garbage collection process. In this architecture, new objects are created in Eden space. During the first minor GC, surviving objects are moved to one of the Survivor areas, instead of directly to the old generation. The following minor GC will collect garbage from Eden and the initial Survivor with age incremented by 1 and move them to another Survivor space. This step empties the initial Survivor space, which can be then reused. The JVM offers several types of GC with different characteristics, which are suitable for different platforms and architectures: Parallel GC â€” default for server-class machines with more than 2 cores and a large amount of memory. Can be used for minor and major collection. Concurrent Mark and Sweep CMS GC â€” this attempts to minimise the stop-the-world pauses by doing most of the work concurrently with the application threads. This collector drastically changes how the heap is managed. Blue â€” Allocation rate. The rate in which running application allocates new object on the heap. The steeper it is, the more objects are allocated in the same amount of time. Black â€” A GC event. When garbage is collected during a minor or major GC event, memory is freed from unreachable objects and can be allocated again. It represents heap utilisation with live reachable objects. Application with memory leak Fig. Memory utilisation graph for Java application with memory leak. With a large enough heap, garbage collection events might take a long time to be executed.

Chapter 2 : Java Memory Management for Java Virtual Machine (JVM) | Betsol

Java Memory Management for Java Virtual Machine (JVM) on Betsol | Java memory management is an ongoing challenge and a skill that must be mastered to have Java memory management is an ongoing challenge and a skill that must be mastered to have properly tuned applications that function in a scalable manner.

It is suggested to use the latest utility, jcmd instead of jmap utility for enhanced diagnostics and reduced performance overhead. The â€"heap option can be used to obtain the following Java heap information: Information specific to the GC algorithm, including the name of GC algorithm for example, parallel GC and algorithm-specific details such as number of threads for parallel GC. Heap configuration that might have been specified as command-line options or selected by the VM based on the machine configuration. For each generation area of the heap , the tool prints the total heap capacity, in-use memory, and available free memory. If a generation is organized as a collection of spaces for example, the new generation , then a space specific memory size summary is included. It must be used on the same machine where the JVM is running, and have same effective user and group identifiers that were used to launch the JVM. A heap dump hprof dump can be created using the following command: The tool parses a heap dump in binary format for example, a heap dump produced by jcmd. This utility can help debug unintentional object relation. This term is used to describe an object that is no longer needed but is kept alive due to references through some path from the rootset. This can happen, for example, if an unintentional static reference to an object remains after the object is no longer needed, if an observer or listener fails to unregister itself from its subject when it is no longer needed, or if a thread that refers to an object does not terminate when it should. Unintentional object relation is the Java language equivalent of a memory leak. The All Classes query is displayed by default. This default page displays all of the classes present in the heap, excluding platform classes. This list is sorted by fully qualified class name, and broken out by package. Click the name of a class to go to the Class query. The second variant of this query includes the platform classes. Platform classes include classes whose fully qualified names start with prefixes such as java, sun or javax. On the other hand, the class query displays the information about a class. This includes its superclass, any subclasses, instance data members, and static data members. From this page you can navigate to any of the classes that are referenced, or you can navigate to an instance query. The instance query displays all instances of a given class. In addition, it can report complete heap dumps and states of all the monitors and threads in the JVM. In terms of diagnosing problems, HPROF is useful when analyzing performance, lock contention, memory leaks and other issues. The tool then processes the event data into profiling information. By default, heap profiling information is written out to java. The following command javac â€"J-agentlib: A crucial piece of information in the heap profile is the amount of allocation that occurs in various parts of the program. The output of javac â€"J-agentlib: The following command can be used to get the CPU usage sampling profile results: There are other tools such as VisualVM which provides us the detailed information of memory usages, garbage collections, heap dumps, CPU and memory profiling etc. VisualVM VisualVM is a tool derived from the NetBeans platform and its architecture is modular in design meaning it is easy to extend through the use of plugins. VisualVM allows us to get detailed information about Java applications while they are running on a JVM and it can be in a local or a remote system. Data generated can be retrieved using Java Development Kit JDK tools and all the data and information on multiple Java applications can be viewed quickly for both local and remote running applications. It is also possible to save and capture the data about the JVM software and save data to the local system. VisualVM can do CPU sampling, memory sampling, run garbage collections, analyze heap errors, take snapshots and more. We can also generate thread dumps and memory dumps on the remote machine when connected through JMX Remote port. Once there are applications listed under local or remote section, double click on them to view the details of the application. Overview, Monitor, Threads and Sampler. The main class, the arguments of the command line, JVM arguments, PID, system properties and any saved data such as thread dumps or heap dumps are available in the overview tab. This tab displays the CPU and memory usage of the application. There are four graphs in this view. The X-axis shows the timestamp against the percentage of utilization. The second graph that is on the top right displays the heap space and Perm Gen space or Metaspace. It also shows the maximum size of the heap memory, how much is being used by the application and how much is available for use. When an application is performing a memory intensive job, the used heap represented in blue color on the graph should always be less than the heap size represented in orange color on the graph. When there is an out-of-memory error, heap dump can be obtained by adding the flowing VM parameters: The summary tab displays some basic information such as total classes, total instances, classloaders, GC roots and the environment details in which the application is running. The analysis in the Fig 1. Large objects create a lot of other objects in their constructor or have a lot of fields. We should also analyze code areas that are known to be massively concurrent under production conditions. Under load, these locations will not only allocate more, but they will also increase synchronization within the memory management itself. High memory utilization is the cause for excessive garbage collection. In some cases, hardware restrictions make it impossible to simply increase the heap size of the JVM. In other cases, increasing the heap size does not solve but only delays the problem because the utilization just keeps growing. Following analyses are possible using the heap dumps: Every object that is no longer needed but remains referenced by the application can be considered a memory leak. Practically, we care only about memory leaks that are growing or that occupy a lot of memory. A typical memory leak is one in which a specified object type is created repeatedly but not garbage-collected. To identify this object type, multiple heap dumps are needed which can be compared using trending dumps. In fact, String and char[] will typically have the highest number of instances, but analyzing them would take us nowhere. Even if we were leaking String objects, it would most likely be because they are referenced by an application object, which represents the root cause of the leak. Therefore, concentrating on classes of our application will yield faster results. There are several cases when we want to do a detailed analysis. The trending analysis did not lead us to the memory leak Our application uses too much memory, but has no obvious memory leak and we need to optimize the code We could not do a trending analysis because memory is growing too fast and the JVM is crashing In all the three cases, the root cause is most likely one or more objects that are at the root of a larger object tree. These objects prevent a whole lot of other objects in the tree from being garbage-collected. In case of an out-of-memory error it is likely that a handful of objects prevent a large number of objects from being freed, hence triggering the out-of-memory error. The size of the heap is often a big problem for memory analysis. Generating heap dump requires memory itself. If the heap size is at the limit of what is available or possible bit JVMs cannot allocate more than 3. In addition, a heap dump will suspend the JVM. Manually finding the one object that prevents a whole object tree from being garbage-collected quickly becomes the proverbial needle in a haystack. Fortunately, solutions like Dynatrace is able to identify these objects automatically. To do this we need to use a dominator algorithm that stems from the graph theory. This algorithm should be able to calculate the root of an object tree. In addition to calculating object tree roots, the memory analysis tool calculates how much memory a particular tree holds. This way it can calculate which objects prevent a large amount of memory from being freed â€” In other words, which object dominates memory. This graph displays the total number of classes loaded in the application and the last graph displays the number of threads currently running. With these graphs, we can see if our application is taking too much of CPU or memory. We can also observe the time pass in each state and many other details about the threads. There are filtering options to view only the live threads or finished threads. When we open this tab initially, it contains no information. We will start with CPU sampling. The next sampling is the Memory sampling. The application would be frozen during the sampling until the results are fetched. In both the types of sampling we can save the results for later use to a file. For example, sampling can be taken multiple times at regular intervals and the results can be compared. This can help us improve the application to use less CPU and memory. Finally, it is the task of the developer to examine these areas and improve the code. Java Garbage Collection Tuning Java garbage collection tuning should be last option we use for increasing the throughput of the application and only when we see a drop in performance because of longer GC causing application timeouts. We do not see this error in case of Java 8 and above. If we see a lot of full GC operations, then we should try increasing the old generation memory space. Overall, garbage collection tuning takes a lot of effort and time and there is no hard and fast rule for that. We need to try different options and compare them to find out the best one suitable for our application. Some of the performance solutions are: Also be sure it works the same way across different environments Avoid premature object creation. Creation should be close to the actual place of use. This is a basic concept that we tend to overlook JSPs are usually slower than servlets StringBuilder instead of string concat Use primitives and avoid objects.

Chapter 3 : Memory Management in Java: Java Virtual Machine (JVM) Memory Model

Garbage Collection in the Java HotSpot(tm) Virtual Machine - Automatic Memory Management - JavaOne Presentation Keywords memory management garbage collection virtual machine.

In support of this diverse range of deployments, the Java HotSpot virtual machine implementation Java HotSpot VM provides multiple garbage collectors, each designed to satisfy different requirements. This is an important part of meeting the demands of both large and small applications. Java SE selects the most appropriate garbage collector based on the class of the computer on which the application is run. However, this selection may not be optimal for every application. Users, developers, and administrators with strict performance goals or other requirements may need to explicitly select the garbage collector and tune certain parameters to achieve the desired level of performance. This document provides information to help with these tasks. First, general features of a garbage collector and basic tuning options are described in the context of the serial, stop-the-world collector. Then specific features of the other collectors are presented along with factors to consider when selecting a collector. A garbage collector GC is a memory management tool. It achieves automatic memory management through the following operations: Allocating objects to a young generation and promoting aged objects into an old generation. Finding live objects in the old generation through a concurrent parallel marking phase. Recovering free memory by compacting live objects through parallel copying. For some applications, the answer is never. That is, the application can perform well in the presence of garbage collection with pauses of modest frequency and duration. However, this is not the case for a large class of applications, particularly those with large amounts of data multiple gigabytes , many threads, and high transaction rates. This is also true for the Java platform. The graph in Figure , "Comparing Percentage of Time Spent in Garbage Collection" models an ideal system that is perfectly scalable with the exception of garbage collection GC. Figure Comparing Percentage of Time Spent in Garbage Collection Description of "Figure Comparing Percentage of Time Spent in Garbage Collection" This shows that negligible speed issues when developing on small systems may become principal bottlenecks when scaling up to large systems. However, small improvements in reducing such a bottleneck can produce large gains in performance. For a sufficiently large system, it becomes worthwhile to select the right garbage collector and to tune it if necessary. The serial collector is usually adequate for most "small" applications those requiring heaps of up to approximately megabytes MB on modern processors. The other collectors have additional overhead or complexity, which is the price for specialized behavior. If the application does not need the specialized behavior of an alternate collector, use the serial collector. One situation where the serial collector is not expected to be the best choice is a large, heavily threaded application that runs on a machine with a large amount of memory and two or more processors. When applications are run on such server-class machines, the parallel collector is selected by default. See the section Ergonomics. However, the concepts and recommendations presented here apply to all supported platforms, including Linux, Microsoft Windows, the Solaris operating system x64 Platform Edition , and OS X. In addition, the command line options mentioned are available on all supported platforms, although the default values of some options may be different on each platform.

Chapter 4 : Memory Management - Java 8 Pocket Guide [Book]

1. å¼•è¨€ Javaå¹³å•°ä¸€ä¸ªæœ€å¤§çš„ä¼˜åŠ¿æ˜¯åœ¨äºŽå®ƒçš„è‡ªåŠ¨å†…å˜ç®¡ç†ï¼Œè¿™æ ·å•¯ä»¥ä½¿å¾—Javaçš„å¼€å•‘è€…ä¸•ç”¨è‡ªå·±åŽ»ç¼–å†™ä»£ç •æ•¥è¿›è¡Œå†…å˜çš„ç®¡ç†ï¼Œä»Žè€Œä»Žå¤æ•‚çš„å†…å˜ç®¡ç†çš„å·¥ä½œæ Š½èº«å‡ºæ•¥ä¸“æ³¨äºŽä¸šåŠ¡é€»è¾‘çš„å¼€å•‘ã€‚

It is a repository for live objects, dead objects, and free memory. When an object can no longer be reached from any pointer in the running program, it is considered "garbage" and ready for collection. An acceptable rate for garbage collection is application-specific and should be adjusted after analyzing the actual time and frequency of garbage collections. If you set a large heap size, full garbage collection is slower, but it occurs less frequently. If you set your heap size in accordance with your memory needs, full garbage collection is faster, but occurs more frequently. The goal of tuning your heap size is to minimize the time that your JVM spends doing garbage collection while maximizing the number of clients that WebLogic Server can handle at a given time. To ensure maximum performance during benchmarking, you might set high heap size values to ensure that garbage collection does not occur during the entire run of the benchmark. You might see the following Java error if you are running out of heap space: To configure WebLogic Server to detect automatically when you are running out of heap space and to address low memory conditions in the server, see Automatically Logging Low Memory Conditions and Specifying Heap Size Values. Choosing a Garbage Collection Scheme Depending on which JVM you are using, you can choose from several garbage collection schemes to manage your system memory. For example, some garbage collection schemes are more appropriate for a given type of application. Once you have an understanding of the workload of the application and the different garbage collection algorithms utilized by the JVM, you can optimize the configuration of the garbage collection. Refer to the following links for in-depth discussions of garbage collection options for your JVM: For some pointers about garbage collection from an HP perspective, see "Performance tuning Java: Tuning steps" at http: Using Verbose Garbage Collection to Determine Heap Size The verbose garbage collection option verbosegc enables you to measure exactly how much time and resources are put into garbage collection. To determine the most effective heap size, turn on verbose garbage collection and redirect the output to a log file for diagnostic purposes. The following steps outline this procedure: Monitor the performance of WebLogic Server under maximum load while running your application. Use the -verbosegc option to turn on verbose garbage collection output for your JVM and redirect both the standard error and standard output to a log file. This places thread dump information in the proper context with WebLogic Server informational and error messages, and provides a more useful log for diagnostic purposes. For example, on Windows and Solaris, enter the following: On HPUX, use the following option to redirect stderr stdout to a single file: Because the output includes timestamps for when garbage collection ran, you can infer how often garbage collection occurs. Analyze the following data points: How often is garbage collection taking place? How long is garbage collection taking? Full garbage collection should not take longer than 3 to 5 seconds. What is your average memory footprint? In other words, what does the heap settle back down to after each full garbage collection? If the heap always settles to 85 percent free, you might set the heap size smaller. Review the New generation heap sizes Sun or Nursery size Jrockit. Make sure that the heap size is not larger than the available free RAM on your system. Use as large a heap size as possible without causing your system to "swap" pages to disk. The amount of free RAM on your system depends on your hardware configuration and the memory requirements of running processes on your machine. See your system administrator for help in determining the amount of free RAM on your system. If you find that your system is spending too much time collecting garbage your allocated virtual memory is more than your RAM can handle , lower your heap size. Typically, you should use 80 percent of the available RAM not taken by the operating system or other processes for your JVM. If you find that you have a large amount of available free RAM remaining, run more instances of WebLogic Server on your machine. Remember, the goal of tuning your heap size is to minimize the time that your JVM spends doing garbage collection while maximizing the number of clients that WebLogic Server can handle at a given time. JVM vendors may provide other options to print comprehensive garbage collection reports. This section describes the command line options you use to define the heap sizes values. You must specify Java heap size values each time you start an instance of WebLogic Server. This can be done either from the java command line or by modifying the default values in the sample startup scripts that are provided with the WebLogic distribution for starting WebLogic Server.

Chapter 5 : Tuning Java Virtual Machines (JVMs)

Understanding JVM Memory Model, Java Memory Management are very important if you want to understand the working of Java Garbage www.nxgvision.com we will look into memory management in java, different parts of JVM memory and how to monitor and perform garbage collection tuning.

Fragmenting and Compacting Questions In reality, memory is limited and we must use it wisely in our applications. In this article, we will explore how the memory space is, conceptually, divided for overall optimization. Understanding Java Virtual Machine Memory Model and Java Memory Management are important to tune applications for a better response depending upon the situation and to scale applications for serving the entire humanity as the userbase. Following is the overview of the elegant Java Memory Model: Do understand that it is one big chunch of memory which is physically uniform but has been distinctly splitted from a software perspective to ensure efficient usage and organization of memory. We will explore each section in detail in our quest of efficient memory management. Generational Hypothesis Doing a garbage collection entails stopping the application completely. The more objects there are, the longer it takes to collect all the garbage. What if we would have a possibility to deal with less objects? Investigating the possibilities, a group of researchers has observed that most allocations inside applications fall into two categories: Most of the objects become unused quickly The ones that do not usually, survive for a very long time These observations come together in the form of Weak Generational Hypothesis. The key idea is to quickly identify the approximate usage of a particular object and consider objects only of a particular interest. The young generation is divided into Eden space and two survivor spaces S0 and S1. The user does not have much control over the third space PERM which we will explore further in our quest. Young Generation memory space Young generation is the memory space where all the new objects are created. When young generation is filled, a garbage collection is performed. Young Generation is divided into three parts namely Eden Memory space and two Survivor Memory spaces. Keys points about Young Generation memory spaces: Most of the newly created objects are located in the Eden memory space. When Eden space is filled with objects, minor garbage collection is performed and all the survivor objects are moved to one of the survivor spaces. Minor garbage collection checks the survivor objects and move them to the other survivor space. Thus, one of the survivor space is always empty. Objects that are survived after many cycles of garbage collection, are moved to the Old generation memory space. Old Generation memory space Old Generation memory contains the objects that are long lived and survived after many rounds of Minor garbage collection. Usually garbage collection is performed in Old Generation memory when it is full. Garbage Collection in the Old generation memory space is called Major garbage collection and usually takes much longer than Minor garbage collection. Keys points about Old Generation memory spaces: This is where the metadata such as classes were located. It is quite hard to predict how much space all of that would require. Result of these failed predictions took the form of java. Unless the cause of such OutOfMemoryError was an actual memory leak, the way to fix this problem was to simply increase the permgen size similar to the following example setting the maximum allowed permgen size to MB: Available in Java versions less than 8 only Stores metadata of classes Hard to predict memory usage Metaspace As predicting the need for metadata was a complex and inconvenient task, the Permanent Generation was removed in Java 8 in favor of the Metaspace. From this point on, most of the miscellaneous things were moved to regular Java heap. The class definitions are loaded into Metaspace. It is located in the native memory and does not interfere with the regular heap objects. By default, Metaspace size is only limited by the amount of native memory available to the Java process. This saves developers from a situation when adding just one more class to the application results in the java. Keys points about Metaspace memory spaces: Letting the Metaspace to grow uncontrollably can introduce heavy swapping and reach native allocation failures. In case you still wish to protect yourself for such occasions you can limit the growth of Metaspace similar to following, limiting Metaspace size to MB: Method Area Method Area is: A part of space in the Permanent Generation memory space It is used to store class structure runtime constants and static variables and code for methods and constructors. Memory Pool Memory Pools are: Created by JVM memory managers to create a pool of immutable objects. String Pool is an example of this kind of memory pool. Runtime Constant Pool A per-class runtime representation of constant pool in a class It contains class runtime constants and static methods. Runtime constant pool is the part of the method area. Java Stack Memory Java Stack memory is: Used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method. Fragmenting and Compacting Whenever sweeping takes place, the JVM has to make sure the areas filled with unreachable objects can be reused. This can and eventually will lead to memory fragmentation which, similarly to disk fragmentation, leads to two problems: Write operations become more time-consuming as finding the next free block of sufficient size is no longer a trivial operation. When creating new objects, JVM is allocating memory in contiguous blocks. So if fragmentation escalates to a point where no individual free fragment is large enough to accommodate the newly created object, an allocation error occurs. To avoid such problems, the JVM is making sure the fragmenting does not get out of hand. This process relocates all the reachable objects next to each other, eliminating or reducing the fragmentation. Here is an illustration of that: Now, you have a clear idea regarding how management and classification of memory can lead to better memory usage for client and server applications. Questions Where is a newly declared variable allocated memory?

Chapter 6 : java - How can I increase the JVM memory? - Stack Overflow

The JVM is a virtual machine that runs Java class files in a portable way. Being a virtual machine means the JVM is an abstraction of an underlying, actual machine--such as the server that your program is running on.

So we assume that there is a scope of leveraging benefits of multiple CPUs or multithreading. All right, enough of theory Need to tune the JVM parameters. OutOfMemoryError can occur due to 3 possible reasons: JavaHeap space low to create new objects. Out of swap space If you use java NIO packages, watch out for this issue. DirectBuffer allocation uses the native heap. The NativeHeap can be increasded by -XX: There are some starting points to diagnose the problem. You may analyze the logs or use a light profiler like JConsole part of JDK to check the memory graph. This is a memory intensive procedure and not meant for production systems. Depending upon your application, these heavy profilers can slow down the app upto 10 times. Java memory leaks or what we like to call unintentionally retained objects , are often caused by saving an object reference in a class level collection and forgetting to remove it at the proper time. The collection might be storing objects, out of which 95 might never be used. So in this case those 95 objects are creating the memory leak, since the GC cannot free them as they are referenced by the collection. A java "memory leak" is more like holding a strong reference to an object though it would never be needed anymore. The fact that you hold a strong reference to an object prevents the GC from deallocating it.. Java "memory leaks" are objects that fall into category 2. Objects that are reachable but not "live" can be considered memory leaks. It does not provide details of the Heap object. E For NativeHeap issues Try to get some Solution: Based on the findings from the diagnosis, you may have to take these actions: Code change - For memory leak issues, it has to be a code change. JVM parameters tuning - You need to find the behavior of your app in terms of the ratio of young to old objects, and then tune the JVM accordingly. We ll talk abt when to tune a parameter as we discuss the relevant params below. A very high value can starve other apps and induce swapping. Hence, Profile the memory requirements to select the right value. NewRatio sets the ratio of the old and new generations in the heap. A NewRatio of 5 sets the ratio of new to old at 1: SurvivorRatio sets the ratio of the survivor space to the eden in the new object area. A SurvivorRatio of 6 sets the ratio of the three spaces to 1: A parallel version of the young generation copying collector is used with the concurrent collector.

Chapter 7 : Memory Management General Rules

The Java HotSpot Virtual Machine is a core component of the Java SE platform. It implements the Java Virtual Machine Specification, and is delivered as a shared library in the Java Runtime Environment.

Please refer to documentation and demos , ask your question in forum , or contact support. As experience has shown, sometimes a sort of uncertainty may arise on the subject of Java Virtual Machine JVM memory structure and other related aspects such as sizes of various kinds of memory, live and dead objects, etc. In this article, we shall try to illuminate these issues to clear up the point. Heap The JVM has a heap that is the runtime data area from which memory for all class instances and arrays are allocated. It is created at the JVM start-up. The heap size may be configured with the following VM options: Heap memory for objects is reclaimed by an automatic memory management system which is known as a garbage collector. It is created at the JVM startup and stores per-class structures such as runtime constant pool, field and method data, and the code for methods and constructors, as well as interned Strings. Unfortunately, the only information JVM provides on non-heap memory is its overall size. No detailed information on non-heap memory content is available. The abnormal growth of non-heap memory size may indicate a potential problem, in this case you may check up the following: If there are class loading issues such as leaked loaders. In this case, the problem may be solved with the help of Class loaders view. If there are strings being massively interned. For detection of such problem, Object allocation recording may be used. If the application indeed needs that much of non-heap memory and the default maximum size of 64 Mb is not enough, you may enlarge the maximum size with the help of -XX: Heap and non-heap memory usage telemetry is shown in the Memory tab: Allocated and Used Memory Allocated and used memory sizes are shown on the graphs for both heap and non-heap. The allocated memory is an overall amount of memory allocated by the JVM, while used memory is the amount of memory which is actually in use. Obviously, the allocated memory cannot be less than the used memory. The exact amount of allocated memory is determined by the JVM internal strategies. Live and Dead Objects Used heap memory consists of live and dead objects. Live objects are accessible by the application and will not be a subject of garbage collection. Dead objects are those which will never be accessible by the application but have not been collected yet by the garbage collector. Such objects occupy the heap memory space until they are eventually collected by the garbage collector. Note that Class list view in memory telemetry shows both live and dead objects. You may observe the decreasing number of objects when garbage collection occurs automatically or it is forced with the help of the corresponding toolbar button. Object Sizes in Memory Snapshots: Shallow and Retained Sizes All individual objects, as well as sets of objects have their shallow and retained sizes. Shallow size of an object is the amount of allocated memory to store the object itself, not taking into account the referenced objects. Shallow size of a regular non-array object depends on the number and types of its fields. Shallow size of an array depends on the array length and the type of its elements objects, primitive types. Shallow size of a set of objects represents the sum of shallow sizes of all objects in the set. Retained size of an object is its shallow size plus the shallow sizes of the objects that are accessible, directly or indirectly, only from this object. In other words, the retained size represents the amount of memory that will be freed by the garbage collector when this object is collected. In general, retained size is an integral measure, which helps to understand the structure clustering of memory and the dependencies between object subgraphs, as well as find potential roots of those subgraphs. Dead objects are shown only with shallow size, as they do not actually retain any other objects.

Chapter 8 : memory management in the java HotSpot Virtual Machineï¼ˆä¸-æ–‡ç¿»è¯‘ï¼‰ - çˆ±ç¨‹åº•ç½‘

The Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide describes the garbage collection methods included in the Java HotSpot Virtual Machine (Java HotSpot VM) and helps you determine which one is the best for.

Storage Management Among the facilities that a Java virtual machine has to provide is a storage manager. The storage manager is responsible for the life-cycle of Java objects: A virtual machine can distinguish itself in the qualities of service the storage manager provides. The HotSpot virtual machine provides several storage managers, to meet the needs of different kinds of applications. The two main storage managers are one to provide short pauses to the application, and one to provide high throughput. The klass hierarchy is self-referential, in that it also contains the descriptions of the descriptions of objects, and so on. But we like object oriented programming, so the klass hierarchy provides a mechanism for methods and virtual methods on objects. Perhaps a simple example will suffice. Consider a Java object instance. The storage for that instance consists of the fields declared by the Java programmer. In addition to those, we add a header, the important part of which, for our purposes here, is a pointer to the representation of the Java class that defined this instance. But in addition, it includes a description of the types of the fields of every instance of this Java class, so that the storage manager can find and adjust reference fields, for example, if it moves objects in memory. There are subclasses of klass for each of the types of object managed by the storage manager. Allocation Most Java applications allocate objects frequently, and abandon them almost as frequently. So a performant Java virtual machine should support fast allocation of objects, and be prepared for most objects to become unreachable fairly quickly. If you look in the code you will see code that allocates and initializes objects. Much work has gone into the sizing of TLABs, and adjusting their sizes based on the application thread performance, to balance the number of TLABs needed, the fragmentation lost due to space held in TLABs by inactive threads, etc. Collection Objects eventually become unreferenced, and the storage they occupy can be reclaimed for use by other objects. The basic operation of the collector is to traverse the object graph, finding all the reachable objects and preserving them, while identifying all the objects that are unreachable and recovering their storage. It would be prohibitively expensive to traverse the entire object graph for each collection, so a variety of techniques have been applied to make collections more efficient. But once objects in the old generation we expect them to remain referenced for a while, and so use different algorithms to manage them. The Young Generation The young generation must support fast allocation, but we expect most of those objects to become unreachable fairly quickly. We would like not to have to expend any effort on the unreachable objects, since there are so many of them. So when our young generations need to be collected, we identify the lreachable objects in them and copy them out of the young generation. An advantage of scavenging is that it is fast. A disadvantage is that it requires room of a second copy of all the reachable objects from the scavenged generation. Some Ancillary Data Structures But wait: To do that we take advantage of the second part of the weak generational hypothesis: Recall that objects only get into the old generation under the control of the storage manager. That extra work comes from the fact that the remember set is imprecise, so we need to be able to walk backwards through the old generation looking for the start of objects. In addition, the different collectors have some collector-specific ancillary data structures that will be explained in more detail later. The Old Generation Once objects have been scavenged out of the young generation we expect them to remain reachable for at least a while, and to reference and be referenced by other objects that remain reachable. Copying objects in the old generation can be expensive, both because one has to move the objects, and because one has to update all the references to the object to point to the new location of the object. On the other hand, copying objects means that one can accumulate the recovered space into one large region, from which allocation is faster which speeds up scavenging of the young generation , and allows us to return excess storage to the operating system, which is polite. The Permanent Generation In addition to the objects created by the Java application, there are objects created and used by the HotSpot virtual machine whose storage it is convenient to have allocated and recovered by the storage manager. For example, information about loaded classes is stored in the permanent generation, and recovered when those classes are no longer reachable from the application. Collectors In order to provide a variety of qualities of service: Traversing the object graph is time-consuming, so this collector does almost all the traversal while the application is running. Once the reachable objects are identified, the unreachable space is reclaimed by linking it onto free lists, merging where possible to avoid fragmentation. By avoiding moving reachable objects, we avoid the cost of updating references to those objects. That also avoids the complications that arise if you try to move objects while the application is using them. Toward that end, we have designed a parallel compacting collector. The interesting part of the parallel collector is the way it uses multiple threads to copy objects, but still ends up with one large, unfragmented block of free space when it is done. Other Collectors We have some other collectors in the code base: Those are not as interesting as the short pause collector or the high throughput collector, but they are in there, so you will see them if you look at the code. At first we had a framework into which we could plug generations, each of which would have its own collector. The framework has some inefficiencies due to the generality at allows. When we built the high-throughput collector we decided not to use the framework, but instead designed an interface that a collector would support, with the high-throughput collector as an instance of that interface. A Parallelization Framework In building the parallel collector we built a framework for parallelizing parts of a collection: For example, the young generation collectors run as multiple threads. Notification A final responsibility of the storage manager is to provide notifications to the application of objects that have become unreachable. During the traversal of the object graph, when we find subclasses of java. When the discovery of reachable objects including References is complete, we look at the referents of those References. Depending on the semantics of the Reference type and whether the referent is reachable or not, we may queue the Reference for notification, mark through the referent, or clear the referent field. This part of the storage manager is an interesting dance between the virtual machine and the code in the core libraries for handling References.

Chapter 9 : java - What are www.nxgvision.comtime().totalMemory() and freeMemory()? - Stack Overflow

Page 6 of the the document Memory Management in the Java HotSpotâ„¢ Virtual Machine contains the following paragraphs. Young generation collections occur relatively frequently and are efficient and fast because the young generation space is usually small and likely to contain a lot of objects that are no longer referenced.