Diagnostics Guide

prev next contents index view as PDF get Adobe Reader

Understanding Memory Management

Memory management is the process of allocating new objects and removing unused objects to make space for those new object allocations. This section presents some basic memory management concepts and explains the basics about object allocation and garbage collection in the Oracle JRockit JVM. The following topics are covered:

- The Heap and the Nursery
- Object Allocation
- Garbage Collection

For information about how to use command line options to tune the memory management system, see Tuning the Memory Management System.

The Heap and the Nursery

Java objects reside in an area called *the heap*. The heap is created when the JVM starts up and may increase or decrease in size while the application runs. When the heap becomes full, *garbage* is *collected.* During the garbage collection objects that are no longer used are cleared, thus making space for new objects.

Note that the JVM uses more memory than just the heap. For example Java methods, thread stacks and native handles are allocated in memory separate from the heap, as well as JVM internal data structures.

The heap is sometimes divided into two areas (or *generations*) called the *nursery* (or *young space*) and the *old space*. The nursery is a part of the heap reserved for allocation of new objects. When the nursery becomes full, garbage is collected by running a special *young collection*, where all objects that have lived long enough in the nursery are *promoted* (moved) to the old space, thus freeing up the nursery for more object allocation. When the old space becomes full garbage is collected there, a process called an *old collection*.

The reasoning behind a nursery is that most objects are temporary and short lived. A young collection is designed to be swift at finding newly allocated objects that are still alive and moving them away from the nursery. Typically, a young collection frees a given amount of memory much faster than an old collection or a garbage collection of a single-generational heap (a heap without a nursery).

In R27.2.0 and later releases, a part of the nursery is reserved as a *keep area*. The keep area contains the most recently allocated objects in the nursery and is not garbage collected until the next young collection. This prevents objects from being promoted just because they were allocated right before a young collection started.

Object Allocation

During object allocation, the JRockit JVM distinguishes between *small* and *large* objects. The limit for when an object is considered large depends on the JVM version, the heap size, the garbage collection strategy and the platform used, but is usually somewhere between 2 and 128 kB. Please see the documentation for `-XXtlaSize` and `-XXlargeObjectLimit` for more information.

Small objects are allocated in *thread local areas* (*TLA*s). The thread local areas are free chunks reserved from the heap and given to a Java thread for exclusive use. The thread can then allocate objects in its TLA without

synchronizing with other threads. When the TLA becomes full, the thread simply requests a new TLA. The TLAs are reserved from the nursery if such exists, otherwise they are reserved anywhere in the heap.

Large objects that don't fit inside a TLA are allocated directly on the heap. When a nursery is used, the large objects are allocated directly in old space. Allocation of large objects requires more synchronization between the Java threads, although the JRockit JVM uses a system of caches of free chunks of different sizes to reduce the need for synchronization and improve the allocation speed.

Garbage Collection

Garbage collection is the process of freeing space in the heap or the nursery for allocation of new objects. This section describes the garbage collection in the JRockit JVM.

The Mark and Sweep Model

The JRockit JVM uses the *mark and sweep* garbage collection model for performing garbage collections of the whole heap. A mark and sweep garbage collection consists of two phases, the *mark phase* and the *sweep phase*.

During the mark phase all objects that are reachable from Java threads, native handles and other root sources are *marked* as alive, as well as the objects that are reachable from these objects and so forth. This process identifies and marks all objects that are still used, and the rest can be considered garbage.

During the sweep phase the heap is traversed to find the gaps between the live objects. These gaps are recorded in a *free list* and are made available for new object allocation.

The JRockit JVM uses two improved versions of the mark and sweep model. One is *mostly concurrent mark and sweep* and the other is *parallel mark and sweep*. You can also mix the two strategies, running for example mostly concurrent mark and parallel sweep.

Mostly Concurrent Mark and Sweep

The *mostly concurrent mark and sweep strategy* (often simply called *concurrent garbage collection*) allows the Java threads to continue running during large portions of the garbage collection. The threads must however be stopped a few times for synchronization.

The mostly concurrent mark phase is divided into four parts:

- *Initial marking*, where the root set of live objects is identified. This is done while the Java threads are paused.
- *Concurrent marking*, where the references from the root set are followed in order to find and mark the rest of the live objects in the heap. This is done while the Java threads are running.
- *Precleaning*, where changes in the heap during the concurrent mark phase are identified and any additional live objects are found and marked. This is done while the Java threads are running.
- *Final marking*, where changes during the precleaning phase are identified and any additional live objects are found and marked. This is done while the Java threads are paused.

The mostly concurrent sweep phase consists of four parts:

- Sweeping of one half of the heap. This is done while the Java threads are running and are allowed to allocate objects in the part of the heap that isn't currently being swept.
- A short pause to switch halves.

- Sweeping of the other half of the heap. This is done while the Java threads are running and are allowed to allocate objects in the part of the heap that was swept first.
- A short pause for synchronization and recording statistics.

Parallel Mark and Sweep

The parallel mark and sweep strategy (also called the *parallel garbage collector*) uses all available CPUs in the system for performing the garbage collection as fast as possible. All Java threads are paused during the entire parallel garbage collection.

Generational Garbage Collection

The nursery, when it exists, is garbage collected with a special garbage collection called a *young collection*. A garbage collection strategy which uses a nursery is called a g*enerational garbage collection strategy*, or simply *generational garbage collection*.

The young collector used in the JRockit JVM identifies and promotes all live objects in the nursery that are outside the keep area to the old space. This work is done in parallel using all available CPUs. The Java threads are paused during the entire young collection.

Dynamic and Static Garbage Collection Modes

By default, the JRockit JVM uses a dynamic garbage collection mode that automatically selects a garbage collection strategy to use, aiming at optimizing the application throughput. You can also choose between two other dynamic garbage collection modes or select the garbage collection strategy statically. The following dynamic modes are available:

- `throughput`, which optimizes the garbage collector for maximum application throughput. This is the default mode.
- `pausetime`, which optimizes the garbage collector for short and even pause times.
- `deterministic`, which optimizes the garbage collector for very short and deterministic pause times. This mode is only available as a part of Oracle JRockit Real Time.

The major static strategies are:

- `singlepar`, which is a single-generational parallel garbage collector (same as `parallel`)
- `genpar`, which is a two-generational parallel garbage collector
- `singlecon`, which is a single-generational mostly concurrent garbage collector
- `gencon`, which is a two-generational mostly concurrent garbage collector

For more information on how to select the best mode or strategy for your application, see Selecting and Tuning a Garbage Collector.

Compaction

Objects that are allocated next to each other will not necessarily become unreachable (“die”) at the same time. This means that the heap may become fragmented after a garbage collection, so that the free spaces in the heap are many but small, making allocation of large objects hard or even impossible. Free spaces that are smaller than the minimum thread local area (TLA) size can not be used at all, and the garbage collector discards them as *dark matter* until a future garbage collection frees enough space next to them to create a space large enough for a TLA.

To reduce fragmentation, the JRockit JVM compacts a part of the heap at every garbage collection (old collection). Compaction moves objects closer together and further down in the heap, thus creating larger free areas near the top of the heap. The size and position of the compaction area as well as the compaction method is selected by advanced heuristics, depending on the garbage collection mode used.

Compaction is performed at the beginning of or during the sweep phase and while all Java threads are paused.

For information on how to tune compaction, see Tuning the Compaction of Memory.

External and Internal Compaction

The JRockit JVM uses two compaction methods called *external compaction* and *internal compaction*. External compaction moves (evacuates) the objects within the compaction area to free positions outside the compaction area and as far down in the heap as possible. Internal compaction moves the objects within the compaction area as far down in the compaction area as possible, thus moving them closer together.

The JVM selects a compaction method depending on the current garbage collection mode and the position of the compaction area. External compaction is typically used near the top of the heap, while internal compaction is used near the bottom where the density of objects is higher.

Sliding Window Schemes

The position of the compaction area changes at each garbage collection, using one or two sliding windows to determine the next position. Each sliding window moves a notch up or down in the heap at each garbage collection, until it reaches the other end of the heap or meets a sliding window that moves in the opposite direction, and starts over again. Thus the whole heap is eventually traversed by compaction over and over again.

Compaction Area Sizing

The size of the compaction area depends on the garbage collection mode used. In throughput mode the compaction area size is static, while all other modes, including the static mode, adjust the compaction area size depending on the compaction area position, aiming at keeping the compaction times equal throughout the run. The compaction time depends on the number of objects moved and the number of references to these objects. Thus the compaction area will be smaller in parts of the heap where the object density is high or where the amount of references to the objects within the area is high. Typically the object density is higher near the bottom of the heap than at the top of the heap, except at the very top where the latest allocated objects are found. Thus the compaction areas are usually smaller near the bottom of the heap than in the top half of the heap.