MemoryManager

Memory Management in Apache Harmony

Role of memory management

Memory manager in Harmony VM provides the support of automatic memory management. Basically it provides support for Java application to create object. It also reclaim dead objects when the Java heap's free space is low. This is necessary because Java heap size is finite. Actually the main job of memory manager is dead object reclamation, since the allocation part is rather simpler. That is why the automatic memory management is usually called Garbage Collection in runtime community. Besides freeing memory, garbage collection can also improve the object access locality by rearranging the live objects in the heap. This role is increasingly important along with the evolution of modern platforms.

Collector and mutator

The thread entity that conducts garbage collection in the VM is called Garbage Collector, or simply collector. As a contrast, the Java thread is called mutator, because it reads and writes the heap contents. Collector and mutator contend for heap modifications.

Modular design of Harmony GC

Harmony DRLVM architecture has a good modular GC component design. GC can be built as a shared object (or dynamic linked library) and plugged into the VM, as long as the GC is written by following the defined interface between VM and GC. Here is a GC Developer's Guide giving step by step instructions on writing a simple GC from the scratch.

Existing GC implmenentations

Currently there are three independent garbage collection modules implemented in Harmony DRLVM. They are GCv4, GCv4.1, and GCv5. All of them are stop-the-world garbage collection.

- GCv4 is legacy and no longer maintained, which is a mark compaction collector based on LISP2 compactor algorithm. GCv4 is a sequential non-generational collector.
- GCv4.1 is a copying collector with a compaction fall-back. The compactor is based on the threaded reference algorithm. GCv4.1 is a sequential non-generational collector.
- GCv5 is a fully parallel GC, which can work in both generational and non-generational modes. GCv5 achieves rather good scalability on parallel machines, and has dynamic runtime adaptations for best throughput.

Harmony GCv5 Design Overview

Spaces

GCv5 partitions the heap space into NOS (nursery object space), MOS (mature object space), and LOS (large object space). The boundaries between them are dynamically adjustable by GC automatically according to the space utilization. Normal objects are only allocated in NOS, or LOS if the size is bigger than a threshold. MOS is used to for the survivors in NOS.

In GCv5, all the spaces inherit from a common space class. One the important space type is blocked space, where the space is arranged in fixed-size blocks. Block size is a compilation-time parameter, and is always two's power. Currently, the space is always continuous and the blocked space assumes the blocks are contiguous. In future, GCv5 may remove this assumption.

Collections

There are basically two modes of collections: minor and major. Minor collection copies live objects from NOS to MOS. Major collection compacts NOS and MOS, and sweeps LOS. There are other modes of collections for special situations. When the NOS is inadequate to accommodate MOS survivors during a minor collection, the collection will transition into a major collection. This is called fallback collection. When GC discovers that LOS and MOS are not equally fully utilized, it will trigger an extension collection, which extends either LOS or MOS, and shrink the other one.

Collectors

GCv5 uses by default depth-first copying algorithm in minor collection. It supports also breadth-first copying and allocation-order copying as well. Major collection in GCv5 has two implementations, one is classic LISP2 compactor, the other is 2-pass compactor. Both of them are fully parallelized while preserving the slide-compact property.

Runtime adaptation

- **Major or minor**: Since minor collection is usually much shorter in pause time compared to major collection, we want to have minor collection mostly. Well on the other hand, major collection can usually free more space, which is important for the minor collection to really perform. GCv5 developed automatic adaptation to switch between minor and major collections for best throughput.
- **Gen or non-gen**: GCv5 can work in generational mode, where minor collection uses remember set information, and non-generational mode, where minor collection needs to trace the entire heap for live object marking. Generational mode has advantage when the entire heap traversal is too much time consuming, while its downside is the write barrier overhead. GCv5 developed an innovation that can switch dynamically between gen and non-gen mode. This adaptation is turned off by default, since its performance depends on the workload's behavior.
- **Space size adjustment**: In order to best utilize the free space available at runtime, GCv5 can adjust the spaces' sizes adaptively. Based on the survive ratios and allocation speeds of the spaces, GCv5 tries to reserve only adequate MOS free space for NOS minor collection, and tries to get MOS and LOS equally full when a major colleciton happens.

Parallel load balance

GCv5 developed a couple of load balance mechanisms in past. Now only pool-sharing is kept by default. The other two candidates that might be applied in future are work-stealing and task-pushing. Task-pushing uses the idea of Communicating Sequential Process (CSP) for parallel task assignment among the collectors. Pool-sharing is somehow similar to work-packet mechanism, but pool-sharing is depth-first order, which is believed to have better access locality.

Abstraction

- **Threads and allocator**: In design-wise, GCv5 has an abstraction on collector and mutator concepts, both of which are the subclass of allocator, since they are equal when the mutator allocates in NOS and the collector allocates in MOS.
- **Collection space and GC**: GCv5 also has an abstraction on space. In GCv5, space and a collection algorithm is tied together. E.g., when we say Fspace, we mean the space that is managed by copying algorithm. In this way, a GC is only a combination of multiple spaces, or it is a collaborator of multiple collection algorithms over GC heap. It decouples the collection algorithm from GC construction, hence easing the construction of a new GC based on the existing collection algorithms.

Below is a presentation on Harmony GCv5 design overview and status:

- Xiao-Feng Li, Harmony GCv5 Overview, April 22, 2007.

Harmony GCv5 Code Overview

Source tree structure

GCv5 is under Harmomy `working_vm/vm` directory with name `gc_gen`. It has two top level sub-directories: `javasrc` and `src`.

Directory `javasrc` has some Java source code implemented by GCv5, which are used by JIT to inline the fast path of frequent GC operations. Since those GC helpers inlining is only conducted for -server mode of Harmony, this directory can be skipped initially by GC developers, and more explanation will be given later.

Directory `src` has the main body of GCv5 implementation. Basically GCv5 is written in pure C language. The files under `src` directory are C source and header files, although they use .cpp suffices and have not been tested with a C compiler.

The subdirectories under `src` consists of the following:

- `common/`: Common GC routines and definitions;
- Collection algorithms:
  - `trace_forward/`: the copying collection algorithms;
  - `mark_compaction/`: the compacting collection algorithms;
  - `mark_sweep/`: the mark-sweep collection for large objects;
- `thread/`: Threading functionalities, inlcuding mutator and collector;
- `utils/`: Common data structures or routines, not specific to GC;
- `gen/`: Generational collection control;
- `finalizer_weakref/`: GC support for finalizer and weak references;
- `verify/`: Correctness verfication routines for memory management operations;
- `jni/`: Support routines for GC Java helpers in `javasrc` directory;
  I tried my best to make GCv5 modular in both design and source structure. During GCv5 development, the source tree structure has been changed a few times, and I am sure it will keep changing in future to reflect our deeper understandings on a collector kit framework.

Finalizer Subsystem

The processings of finalizer and weak reference are similar and closely related. Here the finalizer subsystem includes also the weak reference support in GC, so I would call them finref subsystem interachangeably with finalizer subsystem.

Finalization processing

Finalizer processing includes following major activities:

- **Remember all the objects that have finalizer**. This is easy to be done at allocation time. I know some GC implementation finds those objects at collection time, which requires scanning of dead objects hence is undesirable in my opinion. To do it at allocation time increases a little bit the allocation path. But since we anyway need certain checkings in fast path allocation, it does not matter to incorporate one more checking for finalizer.
- **Identify the finalizable objects**. Once GC marking phase is finished and all live objects are marked, the collector will go through the remember queue of objects with finalizer, checking if any objects are unreachable. Those unreachable objects in the queue are finalizable objects, and passed to VM for finalization. Before they are handed over to VM, the collector traces through those objects to resurrect all the recursively referenced objects from them.
- **Finalization of the finalizable objects**. VM has a couple of finalizing threads sleeping waiting for new finalization tasks. When GC passes new finalizable objects to VM, the finalizing threads are waken up and start to invoke the finalize() method of those objects. These finalizing threads are native threads associated with Java thread objects, because they will act as Java threads when executing finalizers.

Finalization load balance

One tricky scenario needs special handling. If a mutator keeps producing objects with finalizer, and the finalizers are not able to be executed on time, the heap space will be consumed by dead objects waiting for being finalized. Then the application will casue Out-of-memory exception.

There are two solutions in Harmony for this situation. One is to create more finalizing threads to compete with the mutators for processor resource, and hopefully executing more finalizers than generated by the mutators. The other is to block the guilty mutators until the queue of finalizable objects are shortened by finalizing threads. GCv4.1 adopts the first solution, while GCv5 adopts the second solution.

GCv5 64-bit Support

Harmony GCv5 was originally designed with 32-bit architecture in mind. In first quarter of 2007, it was enhanced to "support" 64-bit platforms. I use the quotation mark because the support is only available in a specifial form, i.e., it only works in compressed reference mode.

Compressed reference

Normally in current available 64-bit machines, people's applications usually run with limited heap size, smaller than 4GB. That means, although the platform gives a potential of 4TB heap space, we use only a portion of it, which can be covered in a 32-bit address range. This observation enlightened the idea of compressed reference, where the runtime uses 32-bit compressed address for object reference representation. The real address is an addition of the 32-bit address value and a heap base address value. And this heap base address' compressed value is zero. In order to distinct this zero value from the NULL reference, we simply avoid to have the zero value by setting the heap base address a few bytes lower than real heap start address.

We encode all the reference fields in 32-bit compressed mode, and we also use 32-bit to encode vtable field in object header. Since the obj_info field is kept 32-bit in both platforms, the total object header overhead remains two 32-bit words (or one 64-bit word).

Object reference

The "compressed reference" is only a form of object reference representation. There is no requirement in JVM specification on the reference representation. To have it 32-bit or 64-bit or whatever is completely JVM internal design issue. It is possible to have hybrid reference representations. The only deciding factor is the cost-efficency in both space and time.

With this in mind, GCv5 defines `REF` type for an object reference. GC has no idea about the layout (or physical meaning) of a `REF` value, except it is an object reference. Anytime when the collector accesses a reference, it always calls ref_to_obj_ptr() to convert the `REF` value to a real address pointer. Conversely, the collector needs to call obj_ptr_to_ref() to encode an address into a reference. The real encoding rule is decided by the implementation of this function. In a 32-bit platform, this function can simply return the same value untouched. In a 64-bit platform, it can compress the pointer.

Currently `REF` is defined as a 32-bit value type. This is not necessarily to be the only option.