Java Garbage Collection Study

Mark Volkmann and Brian Gilstrap
Object Computing, Inc.
July 2008

Java GC

- Java objects are eligible for garbage collection (GC), which frees their memory and possibly associated resources, when they are no longer reachable
- Two stages of GC for an object
  - finalization - runs `finalize` method on the object
  - reclamation - reclaims memory used by the object
- In Java 5 & 6 there are four GC algorithms from which to choose
  - but one of those won't be supported in the future, so we'll just consider the three that will live on
  - serial, throughput and concurrent low pause

GC Process

- Basic steps
  - object is determined to be unreachable
  - if object has a `finalize` method
    - object is added to a finalization queue
    - at some point it's `finalize` method is invoked so the object can free associated resources
  - object memory is reclaimed
- Issues with `finalize` methods
  - makes every GC pass do more work
  - if a `finalize` method runs for a long time, it can delay execution of `finalize` methods of other objects
  - may create new strong references to objects that had none, preventing their GC
  - run in a nondeterministic order
  - no guarantee they will be called; app. may exit first

The JVM has a finalizer thread that is used for running finalize methods. Long running `finalize` methods do not prevent a GC pass from completing and do not freeze the application.

These are good reasons to avoid using `finalize` methods in safety-critical code.

Kinds of Object References

- Strong references
  - the normal type

- Other reference types in `java.lang.ref` package
  - `SoftReference`
    - GC'ed any time after there are no strong references to the referent, but is typically retained until memory is low
    - can be used to implement caches of objects that can be recreated if needed
  - `WeakReference`
    - GC'ed any time after there are no strong or soft references to the referent
    - often used for "canonical mappings" where each object has a unique identifier (one-to-one), and in collections of "listeners"
  - `PhantomReference`
    - GC'ed any time after there are no strong, soft or weak references to the referent
    - typically used in conjunction with a `ReferenceQueue` to manage cleanup of native resources associated with the object without using `finalize` methods (more on this later)

For soft and weak references, the `get` method returns `null` when the referent object has been GC'ed.

Also see `java.util.WeakHashMap`.

Alternative to Finalization

- Don't write `finalize` method in a class whose objects have associated native resources that require cleanup
  - call this class `A`
- Instead, do the following for each such class `A`
  - create a new class, `B`, that extends one of the reference types
    - `WeakReference`, `SoftReference` or `PhantomReference`
  - create one or more `ReferenceQueue` objects
  - a `B` constructor that takes an A and passes that, along with a `ReferenceQueue` object, to the superclass constructor
  - create a `B` object for each `A` object
  - iteratively call `remove` on the `ReferenceQueue`
    - free resources associated with returned B object
    - often this is done in a separate thread

When there is an associated `ReferenceQueue`, weak and soft reference are added to it before the referent object has been finalized and reclaimed. Phantom references are added to it after these occur.

GC Metrics

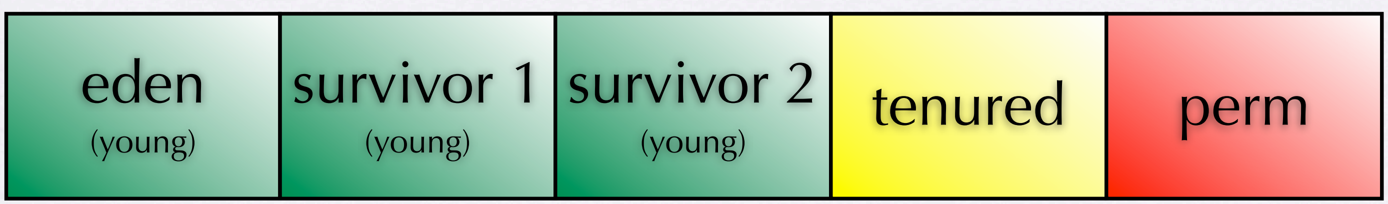
- Different types of applications have different concerns related to GC
- Throughput
  - percentage of the total run time not spent performing GC
- Pauses
  - times when the application code stops running while GC is performed
  - interested in the number of pauses, their average duration and their maximum duration
- Footprint
  - current heap size (amount of memory) being used
- Promptness
  - how quickly memory from objects no longer needed is freed

Generational GC

- All of the GC algorithms used by Java are variations on the concept of generational GC
- Generational GC assumes that
  - the most recently created objects are the ones that are most likely to become unreachable soon
    - for example, objects created in a method and only referenced by local variables that go out of scope when the method exits
  - the longer an object remains reachable, the less likely it is to be eligible for GC soon (or ever)
- Objects are divided into “generations” or “spaces”
  - Java categories these with the names “young”, “tenured” and “perm”
  - objects can move from one space to another during a GC

Object Spaces

- Hold objects of similar ages or generations
  - "young" spaces hold recently created objects
    and can be GC'ed in a "minor" or "major" collection
  - "tenured" space hold objects that
    have survived some number of minor collections
    and can be GC'ed only in a major collection
  - "perm" space hold objects needed by the JVM, such as
    Class & Method objects, their byte code, and interned Strings
    - GC of this space results in classes being "unloaded"
- Size of each space
  - determined by current heap size
    (which can change during runtime)
    and several tuning options



Young Spaces

- Eden space
  - holds objects created after the last GC, except those that belong in the perm space
  - during a minor collection, these objects are either GC'ed or moved to a survivor space
- Survivor spaces
  - these spaces hold young objects that have survived at least one GC
  - during a minor collection, these objects are either GC'ed or moved to the other survivor space
- Minor collections
  - tend to be fast compared to major collections because only a subset of the objects need to be examined
  - typically occur much more frequently than major collections

GC Running Details

- Three approaches

1. Stop the world
   - serial collector does this for minor and major collections
   - throughput collector does this for major collections
   - when a GC pass is started, it runs to completion before the application is allowed to run again

2. Incremental
   none of the Java GC algorithms do this
   - a GC pass can alternate between doing part of the work and letting the application run for a short time, until the GC pass is completed

3. Concurrent
   - throughput collector does this for minor collections
   - concurrent low pause collector does this for minor and major collections
   - a GC pass runs concurrently with the application so the application is only briefly stopped

When Does GC Occur?

- Impacted by heap size
  - from reference #1 (see last slide) ...
  - “If a heap size is small, collection will be fast but the heap will fill up more quickly, thus requiring more frequent collections.”
  - “Conversely, a large heap will take longer to fill up and thus collections will be less frequent, but they take longer.”
- Minor collections
  - occur when a young space approaches being full
- Major collections
  - occur when the tenured space approaches being full

GC Algorithms

> options specified with `-XX:` are turned on with `+` and off with `-`

- Serial: `-XX:+UseSerialGC`
  - uses the same thread as the application for minor and major collections
- Throughput: `-XX:+UseParallelGC`
  - uses multiple threads for minor, but not major, collections to reduce pause times
  - good when multiple processors are available, the app. will have a large number of short-lived objects, and there isn't a pause time constraint
- Concurrent Low Pause: `-XX:+UseConcMarkSweepGC`
  - only works well when objects are created by multiple threads?
  - uses multiple threads for minor and major collections to reduce pause times
  - good when multiple processors are available, the app. will have a large number of long-lived objects, and there is a pause time constraint

Ergonomics

- Sun refers to automatic selection of default options based on hardware and OS characteristics as “ergonomics”
- A “server-class machine” has
  - more than one processor
  - at least 2GB of memory
  - isn’t running Windows on a 32 bit processor

Ergonomics ...

- Server-class machine
  - optimized for overall performance
  - uses throughput collector
  - uses server runtime compiler
  - sets starting heap size = 1/64 of memory up to 1GB
  - sets maximum heap size = 1/4 of memory up to 1GB
- Client-class machine
  - optimized for fast startup and small memory usage
  - targeted at interactive applications
  - uses serial collector
  - uses client runtime compiler
  - starting and maximum heap size defaults?

GC Monitoring

- There are several options that cause the details of GC operations to be output
  - `-verbose:gc`
    - outputs a line of basic information about each collection
  - `-XX:+PrintGCTimeStamps`
    - outputs a timestamp at the beginning of each line
  - `-XX:+PrintGCDetails`
    - implies `-verbose:gc`
    - outputs additional information about each collection
  - `-Xloggc:gc.log`
    - implies `-verbose:gc` and `-XX:+PrintGCTimeStamps`
    - directs GC output to a file instead of stdout
- Specifying the 3rd and 4th option gives all four

Basic GC Tuning

- Recommend approach

See http://java.sun.com/docs/hotspot/gc5.0/ergo5.html, section 4 "Tuning Strategy"

  - set a few goals that are used to adjust the tuning options

1. throughput goal **`-XX:GCTimeRatio=n`**
   - What percentage of the total run time should be spent doing application work as opposed to performing GC?
2. maximum pause time goal **`-XX:MaxGCPauseMillis=n`**
   - What is the maximum number of milliseconds that the application should pause for a single GC?
3. footprint goal
   - if the other goals have been met, reduce the heap size until one of the previous goals can no longer be met, then increase it

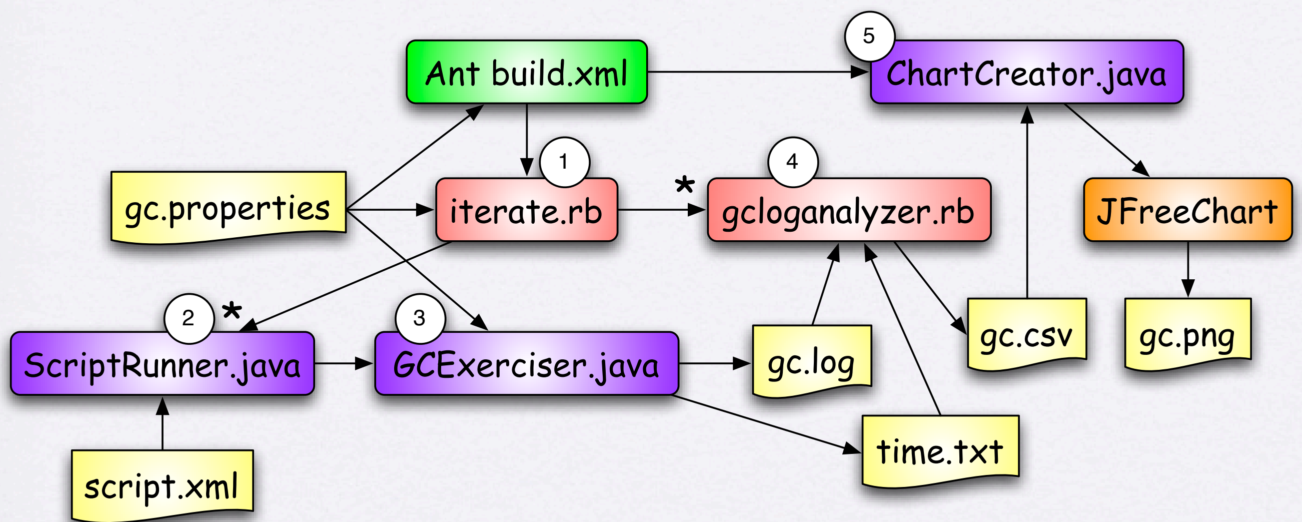
Advanced GC Tuning

- **`-Xms=`*`n`*** (starting) and **`-Xmx=`*`n`*** (maximum) heap size
  - these affect the sizes of the object spaces
- **`-XX:MinHeapFreeRatio=`*`n`***, **`-XX:MaxHeapFreeRatio=`*`n`***
  - bounds on ratio of unused/free space to space occupied by live objects
  - heap space grows and shrinks after each GC to maintain this, limited by the maximum heap size
- **`-XX:NewSize=`*`n`***, **`-XX:MaxNewSize=`*`n`***
  - default and max young size (eden + survivor 1 + survivor 2)
- **`-XX:NewRatio=`*`n`***
  - ratio between young size and tenured size
- **`-XX:SurvivorRatio=`*`n`***
  - ratio between the size of each survivor space and eden
- **`-XX:MaxPermSize=`*`n`***
  - upper bound on perm size
- **`-XX:TargetSurvivorRatio=`*`n`***
  - target percentage of survivor space used after each GC

Even More GC Tuning

- `-XX:+DisableExplicitGC`
  - when on, calls to System.gc() do not result in a GC
  - off by default
- `-XX:+ScavengeBeforeFullGC`
  - when on, perform a minor collection before each major collection
  - on by default
- `-XX:+UseGCOverheadLimit`
  - when on, if 98% or more of the total time is spent performing GC, an OutOfMemoryError is thrown
  - on by default

The Testing Framework



gc.properties

```
# Details about property to be varied.
property.name=gc.pause.max
display.name=Max Pause Goal
start.value=0
end.value=200
step.size=20

processor.bits = 64

# Heap size details.
heap.size.start = 64M
heap.size.max = 1G
```

gc.properties ...

```
# Garbage collection algorithm
# UseSerialGC -> serial collector
# UseParallelGC -> throughput collector
# UseConcMarkSweepGC -> concurrent low pause collector
gc.algorithm.option=UseParallelGC

# Maximum Pause Time Goal
# This is the goal for the maximum number of milliseconds
# that the application will pause for GC.
gc.pause.max = 50

# Throughput Goal
# This is the goal for the ratio between
# the time spent performing GC and the application time.
# The percentage goal for GC will be 1 / (1 + gc.time.ratio).
# A value of 49 gives 2% GC or 98% throughput.
gc.time.ratio = 49
```

gc.properties ...

```
# The number of objects to be created.
object.count = 25

# The size of the data in each object.  1MB
object.size = 1000000

# The number of milliseconds that a reference should be
# held to each object, so it cannot be GCed.
object.time.to.live = 30000

# The number of milliseconds between object creations.
time.between.creations = 30

# The number of milliseconds to run
# after all the objects have been created.
time.to.run.after.creations = 1000
```

None of these properties are used if ScriptRunner and script.xml are used!

script.xml

- Here's an example of a script file
  - `object` elements create an object of a given size and lifetime
  - `work` elements simulate doing work between object creations
  - note the support for loops, including nested loops

```
<script>
  <object size="1M" life="30"/>
  <work time="200"/>
  <loop times="3">
    <object size="2M" life="50"/>
    <work time="500"/>
    <loop times="2">
      <object size="3M" life="50"/>
      <work time="500"/>
    </loop>
  </loop>
</script>
```

iterate.rb

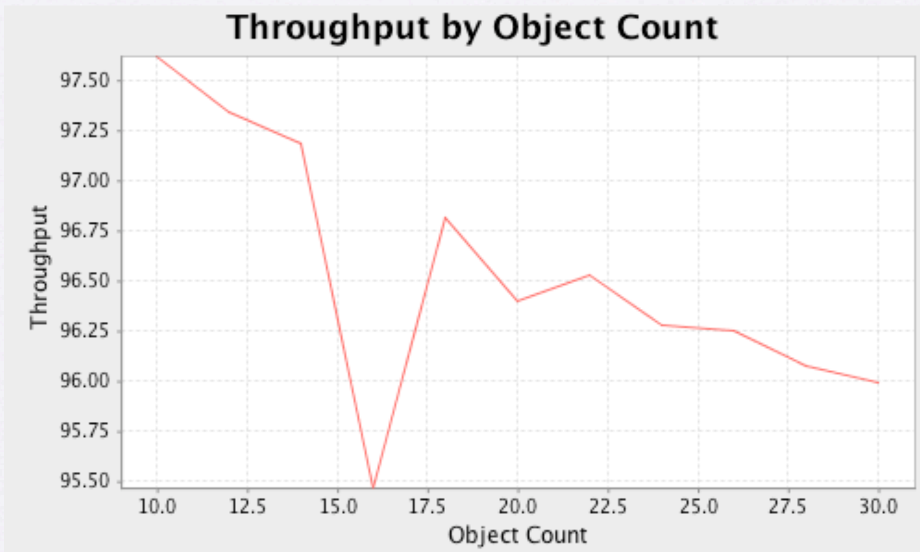
1. Obtains properties in `gc.properties`
2. Iterates `property.name` from `start.value` to `end.value` in steps of `step.size` and passes the value to the `run` method
3. The `run` method
    1. runs `ScriptRunner.java` which reads `script.xml` and processes the steps in it by invoking methods of `GCExcercizer.java` to produce `gc.log` and `time.txt`
    2. runs `gcloganalyzer.rb` which adds a line to `gc.csv`

GCExerciser.java

1. Obtains properties in `gc.properties` and properties specified on the "java" command line to override them
   - for iterating through a range of property values
2. Creates `object.count` objects that each have a size of `object.size` and are scheduled to live for `object.time.to.live` milliseconds
3. Each object is placed into a TreeSet that is sorted based on the time at which the object should be removed from the TreeSet
   - makes the object eligible for GC
4. After each object is created, the TreeSet is repeatedly searched for objects that are ready to be removed until `time.between.creations` milliseconds have elapsed
5. After all the objects have been created, the TreeSet is repeatedly searched for object that are ready to be removed until `time.to.run.after.creations` milliseconds have elapsed
6. Write the total run time to `time.txt`

ChartCreator.java

- Uses the open-source library JFreeChart to create a line graph showing the throughput for various values of a property that affects GC
- Example



Further Work

- Possibly consider the following modifications to the GC test framework
  - have the objects refer to a random number of other objects
  - have each object know about the objects that refer to it so it can ask them to release their references
  - use ScheduledExecutorService to initiate an object releasing itself
  - run multiple times with the same options and average the results
  - make each run much longer ... perhaps 10 minutes

References

1. "Memory Management in the Java HotSpot Virtual Machine"
   - http://java.sun.com/javase/technologies/hotspot/gc/memorymanagement_whitepaper.pdf
2. "Tuning Garbage Collection with the 5.0 Java Virtual Machine"
   - http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html
3. "Ergonomics in the 5.0 Java Virtual Machine"
   - http://java.sun.com/docs/hotspot/gc5.0/ergo5.html
4. "Garbage Collection in the Java HotSpot Virtual Machine"
   - http://www.devx.com/Java/Article/21977/