Java Memory Management

Märt Bakhoff
Java Fundamentals
01.11.2016
Agenda

- JVM memory
- Reference objects
- Monitoring
- Garbage collectors
  - ParallelGC
  - G1GC
JVM memory

• Heap (user objects)
• Non-heap
  – Stack (per thread: call stack, local variables)
  – Metaspace (class metadata)
  – Direct Byte Buffers
  – Native stuff (JNI, Java internals)
public class Example {

    public static void main(String[] args) {
        String info = new String("luke, i'm your father");
        printSubstring(info, 6);
    }

    private static void printSubstring(String s, int offset) {
        String ss = s.substring(offset);
        System.out.println(ss);
    }
}

Method call | Locals
---|---
printSubstring | s -> 0x1
              | ss -> 0x33
              | offset = 6
main | args -> 0x27
     | info -> 0x1

Address | Value
---|---
0x1 | “luke …”
0x27 | String[0]
0x33 | “i’m your …”
Tuning options

- Ergonomics!
- -Xms512M (initial heap size)
- -Xmx2G (max heap size)
- -Xss2M (max stack size, per thread)
- java [options] classname [args]
- All options at
  https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html
  https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/
PermGen

- Java 8+: class metadata lives in Metaspace
- `-XX:MaxMetaspaceSize=size` (default: unlimited)
- Older versions: classes live in PermGen, PermGen is a special part of the heap
- `OutOfMemoryError: PermGen space`
- `-XX:MaxPermSize=size` (default: limited)
Generating garbage

- Primitives in the stack, objects in the heap
- Using `new` allocates objects in the heap
- When and how are objects “deleted” and heap space freed up?
Garbage collection (GC)

- Sort garbage / live objects
- Reclaims heap space
- Fully automatic, no manual deallocation
  Java GC vs C++ new/delete
- Different GC algorithms exist
GC advantages

• Avoid bugs
  - forgetting to free the memory
  - double freeing memory
  - using already freed memory

• Java specific
  - No direct memory access
  - Can't accidentally overwrite unrelated memory
GC disadvantages

- Consumes resources
- Automatic, no manual control
- Unpredictable stalls
- Harder to understand
How does it work?

• Basic principle
  – Find referenced objects
  – Everything else is garbage

• Reachability (GC roots)
  – Classes loaded by system classloader (static fields!)
  – Stack locals (local variables, parameters)
  – Active threads
  – JNI References
Consumes resources?

- Extra memory + CPU for bookkeeping
- Stop The World pauses all threads
- Some applications need to tune GC: pause duration vs pause frequency
Generational GC

• Most objects die young

• Generations: memory pools holding objects of different ages
  – Young generation: eden, survivors
  – Old/tenured generation

• Young-Old default size ratio 1:2
Young/Old

EDEN → SURVIVOR → OLD GEN

YOUNG GEN (frequent collections)
GC algorithms

- Serial
- **Parallel**
- Concurrent Mark-Sweep
- **Garbage first (G1)**
- IBM, Azul special stuff
Agenda

- JVM memory
- Reference objects
- Monitoring
- Garbage collectors
  - ParallelGC
  - G1GC
Reference objects

• java.lang.ref package docs are useful
• WeakReference<T>
• PhantomReference<T>
• ReferenceQueue<T>
Detour: memory leaks

```java
interface Passenger {
    void trainArrived();
}

class TrainStation {
    private final List<Passenger> passengers = new ...
    public void startWaiting(Passenger passenger) {
        passengers.add(passenger);
    }
    public void leave(Passenger passenger) {
        passengers.remove(passenger);
    }
    public void onTrainArrived() {
        passengers.forEach(Passenger::trainArrived);
    }
}
```
Detour: memory leaks

interface Passenger {
    void trainArrived();
}

class TrainStation {
    private final List<Passenger> passengers = new ...
    public void startWaiting(Passenger passenger) {
        passengers.add(passenger);
    }
    public void leave(Passenger passenger) {
        passengers.remove(passenger);
    }
    public void onTrainArrived() {
        passengers.forEach(Passenger::trainArrived);
    }
}
WeakReference<T>

Keep a reference without preventing GC

```java
private final WeakReference<SomethingBig> weakRef;

public Example(SomethingBig sb) {
    this.weakRef = new WeakReference<>(sb);
}

private void tryPrint() {
    SomethingBig strongRef = weakRef.get();
    System.out.println(strongRef != null ? strongRef : "collected");
}
```
interface Passenger {
    void trainArrived();
}

class TrainStation {
    private final List<Passenger> passengers = new...
    public void startWaiting(Passenger passenger) {
        passengers.add(passenger);
    }
    public void leave(Passenger passenger) {
        passengers.remove(passenger);
    }
    public void onTrainArrived() {
        passengers.forEach(Passenger::trainArrived);
    }
}
Weaker TrainStation

class TrainStation {
    private List<WeakReference<Passenger>> passengers;

    public void startWaiting(Passenger passenger) {
        passengers.add(new WeakReference<>(passenger));
    }

    public void onTrainArrived() {
        for (WeakReference<Passenger> ref : passengers) {
            Passenger passenger = ref.get();
            if (passenger != null)
                passenger.trainArrived();
        }
    }
}
Detour: finalizers

From java.lang.Object JavaDoc

• protected void finalize()
    Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.

• Safety net for file streams, network sockets, JDBC connections, etc.
Detour: finalizers

From “Effective Java” by Joshua Bloch

• Finalizers are unpredictable, often dangerous, and generally unnecessary.

• Not only does the language specification provide no guarantee that finalizers will get executed promptly; it provides no guarantee that they’ll get executed at all.
Detour: finalizers

Trolling the garbage collector:

```java
public class Test {
    static Test t;
    @Override
    public void finalize() {
        t = this; // I refuse to die
    }
}
```
PhantomReference<T>

• Not a reference, but a GC token
• Only usable with a ReferenceQueue
• Enqueued by the garbage collector, only after referent is collected
• get() -> null always!
PhantomReference<T>

Example e = new Example();
ReferenceQueue<Example> queue =
    new ReferenceQueue<>();
PhantomReference<Example> phantom =
    new PhantomReference<>(e, queue);
e = null;

// generate garbage, cause a GC
Reference<?> collected = queue.remove();
if (collected == phantom) {
    // our e has been collected
}
Agenda

- JVM memory
- Reference objects
- Monitoring
- Garbage collectors
  - ParallelGC
  - G1GC
GC logging

- -XX:+PrintGCTimeStamps
- -XX:+PrintGCDetails
- -Xloggc:filename

Output depends heavily on GC algo

Read the fine manual:
plumbr.eu/java-garbage-collection-collection-handbook
“GC Algorithms: Implementations”
ParallelGC minor

[GC (Allocation Failure)
   [PSYoungGen: 2 694 440K -> 1 305 132K (2 796 544K)]
   9 556 775K -> 8 438 926K (11 185 152K), 0.24066 secs
]
[Times: user=1.77 sys=0.01, real=0.24 secs]
ParallelGC full


[Full GC (Ergonomics)
 [PSYoungGen: 1 305 132K -> 0K(2 796 544K)]
 [ParOldGen: 7 133 794K -> 6 597 672K (8 388 608K)]
 8 438 926K -> 6 597 672K (11 185 152K),
 [Metaspace: 6 745K -> 6 745K (1 056 768K)],
0.91588 secs
]

[Times: user=4.49 sys=0.64, real=0.92 secs]
G1 minor

0.134: [GC pause (G1 Evacuation Pause) (young), 0.0144119 secs] ...
  Eden: 24.0M (24.0M) -> 0.0B (13.0M)
  Survivors: 0.0B -> 3072.0K
  Heap: 24.0M (256.0M) -> 21.9M (256.0M)
]
[Times: user=0.04 sys=0.04, real=0.02 secs]
GC overhead

116.356: [Full GC ...  
117.331: [Full GC ...  
118.378: [Full GC ...  
119.316: [Full GC ...

java.lang.OutOfMemoryError: 
   GC overhead limit exceeded  

Frequent+quick minor collections expected
Heap dumps

- JVisualVM
- Eclipse memory analyzer (MAT)
- jmap -heap / -histo / -dump:... <pid>
- -XX:+HeapDumpOnOutOfMemory
  -XX:HeapDumpPath=path
$ jmap -heap 1244

Heap Usage:

PS Young Generation
  capacity = (930.0MB)
  used     = (595.2MB)
  free     = (334.7MB)
  64.0% used

PS Old Generation
  capacity = (167.0MB)
  used     = (2.9MB)
  free     = (164.0MB)
  1.7% used
### jmap

```
$ jmap -histo 1244

<table>
<thead>
<tr>
<th>num</th>
<th>#instances</th>
<th>#bytes</th>
<th>class name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>250080</td>
<td>217038824</td>
<td>[C</td>
</tr>
<tr>
<td>2:</td>
<td>874</td>
<td>11646608</td>
<td>[I</td>
</tr>
<tr>
<td>3:</td>
<td>250040</td>
<td>6000960</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>4:</td>
<td>577</td>
<td>85968</td>
<td>[Ljava.lang.Object;</td>
</tr>
<tr>
<td>5:</td>
<td>582</td>
<td>66144</td>
<td>java.lang.Class</td>
</tr>
<tr>
<td>6:</td>
<td>22</td>
<td>25312</td>
<td>[B</td>
</tr>
<tr>
<td>7:</td>
<td>109</td>
<td>7848</td>
<td>j.l.r.Field</td>
</tr>
</tbody>
</table>
```
jps + jmap

$ jps -lv
30086 com.intellij.idea.Main ...
1738 sun.tools.jps.Jps ...
1659 org.jetbrains.jps.cmdline.Launcher ...
1660 com.intellij.rt.execution.application.AppMain ...

$ jmap -dump:format=b,file=dump.bin 1660
Dumping heap to /tmp/dump.bin ...
Heap dump file created
Eclipse memory analyzer

Size: 1.4 MB  Classes: 1.7k  Objects: 28.6k  Class Loader: 80  Unreachable Objects Histogram

**Details**

**Biggest Objects by Retained Size**

- 153.3 KB
- 284.6 KB
- 916.2 KB
- Total: 1.4 MB

**Actions**

- **Histogram**: Lists number of instances per class
- **Dominator Tree**: List the biggest objects and what they keep alive.
- **Top Consumers**: Print the most expensive objects grouped by class and by package.
- **Duplicate Classes**: Detect classes loaded by multiple class loaders.

**Reports**

- **Leak Suspects**: Includes leak suspects and a system overview
- **Top Components**: List reports for components bigger than 1 percent of the total heap.

**Step By Step**

- **Component Report**: Analyze objects which belong to a common root package or class loader.
Eclipse memory analyzer
JVisualVM
JVisualVM
JVisualVM

• Bundled with JDK

• Windows:
  C:\Program Files\Java\jdk1.8.x\bin\jvisualvm.exe

• Linux:
  jvisualvm or visualvm
  (apt-get install visualvm)

• Compile & Run: https://goo.gl/L3dhos
5min break

IT'S LIKE THERE'S CONTINUOUS PRESSURE ON ME TO WORK.
Agenda

- JVM memory
- Reference objects
- Monitoring
- Garbage collectors
  - ParallelGC
  - G1GC
ParallelGC

- GC roots
  - static fields
  - stack locals
  - threads
- Young gen: eden, survivor to/from
- Old gen
- Stop The World pauses
EDEN SURVIVOR 1 SURVIVOR 2 OLD

GC ROOTS STACK STATIC FIELDS

RUNNING/ STOP THE WORLD

EDEN SURVIVOR 1 SURVIVOR 2 OLD

YOUNG
[GC (Allocation Failure) ... ]
Find live objects, starting from GC roots (mark)
Move live objects to survivors (compacting)
Mark EDEN as clean
GC ROOTS  
STACK  
STATIC FIELDS  
RUNNING

E  L  1  N  1  S1  S2  OLD
2015-05-26T14:27:41.915-0200: 117.115:
[GC (Allocation Failure) ... ]
Find live objects, starting from GC roots (mark)
Find live objects, starting from GC roots (mark)
Move live objects to survivors (compacting)
Mark EDEN+S1 as clean
S1/S2 compaction
2015-05-26T14:27:43.915-0200: 119.115:
[GC (Allocation Failure) ... ]
Find live objects, starting from GC roots (mark)
Find live objects, starting from GC roots (mark)
Move live objects to survivors (compacting)
Move live objects to survivors or old (compacting)
Old Gen has a “Card Table” (card region ~2M)
Identifies regions that reference Young Gen
Mark EDEN+S2 as clean
GC ROOTS
STACK
STATIC FIELDS
RUNNING

EDEN 2 2 1 S1 S2 OLD
Card Table
2015-05-26T14:27:44.915-0200: 120.115:
[GC (Allocation Failure) ... ]
Find live objects, starting from GC roots (mark)
* Don’t look for live objects in Old Gen
Find live objects, starting from GC roots (mark)
* Don’t look for live objects in Old Gen
* Scan Card Table regions for extra references
Find live objects, starting from GC roots (mark)
* Don’t look for live objects in Old Gen
* Scan Card Table regions for extra references
Find live objects, starting from GC roots (mark)
* Don’t look for live objects in Old Gen
* Scan Card Table regions for extra references
Move live objects to survivors or old (compacting)
Move live objects to survivors or old (compacting)
Mark EDEN+S1 as clean
Recap: ParallelGC

- Allocate to eden
- Copy live objects to survivors or old
- Clear entire eden + cleared survivor space
- Promote repeat-survivors to Old gen
- Use Card Tables to avoid scanning Old gen
- Use Old objects as GC roots in minor collection
- Full GC when everything is full
ParallelGC insights (1)

- Reference scanning expensive
- size(Old) > size(Young)
- Card Table -> avoid most scanning
ParallelGC insights (2)

- Objects die young
- Copy only live objects
- Don’t touch Old Gen until Full GC
G1

- Generational: young (eden, survivor), old
- Aims for short Stop The World pauses
- Thousands of non-contiguous regions
- Concurrent marking
STACK

STATIC FIELDS

GC ROOTS

FREE
EDEN
SURVIVOR
OLD

RUNNING
STACK
STATIC FIELDS
GC ROOTS
FREE
EDEN
SURVIVOR
OLD
RUNNING
GC ROOTS

STACK

STATIC FIELDS

FREE
EDEN
SURVIVOR
OLD

RUNNING
[GC pause (G1 Evacuation Pause) (young) ..]
Collection Set: all young
GC ROOTS

STACK

STATIC FIELDS

FREE
EDEN
SURVIVOR
OLD

STOP THE WORLD

Mark objects reachable from roots
STOP THE WORLD

Move to new survivor region
Free evacuated regions
MG}

STACK

STATIC FIELDS

FREE
EDEN
SURVIVOR
OLD

GC ROOTS

RUNNING
[GC pause (G1 Evacuation Pause) (young) ..]
Collection Set: all young
Mark objects reachable from roots
Move to new survivor region / old
STOP THE WORLD

Move to new survivor region / old
STOP THE WORLD

Move to new survivor region / old
STOP THE WORLD

Free evacuated regions
Update RSets (Per region Card Table on steroids)
old->old, old->young
GC ROOTS

STACK

STATIC FIELDS

FREE

EDEN

SURVIVOR

OLD

RUNNING
Update RSet: old->young
STACK

STATIC FIELDS

GC ROOTS

FREE

EDEN

SURVIVOR

OLD

RUNNING
STOP THE WORLD

[GC pause (G1 Evacuation Pause) (young) ..]
Collection Set: all young
Mark objects reachable from roots
Mark objects reachable from roots and RSets
Move to new survivor region / old
STOP THE WORLD

Move to new survivor region / old
STOP THE WORLD

Move to new survivor region / old
UPDATE RSet: old->young
Free evacuated regions
Meanwhile..

- Concurrent marking
- Start from GC roots
- Find all live objects
- Sort regions by “liveness”
After concurrent marking

• Stop The World

• Scrub RSets

• Collection set: all young
  + Old with least live objects

• 1.269: [GC pause (mixed) ... ]

• Amount of old regions selected ~ pause time
Recap: G1

- Use concurrent marking
- Collection set: all young + hand picked old (most garbage)
- Find live objects from GC roots + RSets
- Compact to new survivors / old regions
- Free entire evacuated regions
G1 insights

- Avoid reference scanning with RSets
- Avoid long pauses with mixed collection: never clean entire Old Gen
- Only collect Old regions with most garbage
  - -> don’t touch live Old objects
  - -> more time to become garbage
  - Less live objects -> less copying
Homework (1)

• Use PhantomReferences to write a finalize() replacement
• User can register several cleanup tasks for any object
• PostCollectionTaskRunner starts a thread that runs cleanup tasks in the background

interface PostCollectionTaskRunner {
  void register(Object o, Runnable task);
  void shutdown() throws Exception;
}
Homework (2)

• Run with ParallelGC (Java8 default). Enable detailed GC logging. Cause a Full GC.

• Submit the GC log + following comments:
  - for one minor collection: time since last collection and bytes freed for young gen
  - for one Full GC: bytes freed for young gen, old gen, total heap size

• Use max heap size 64M

• Also submit code for triggering Full GC
Homework (3)

- Phantoms expensive -> use 1 per managed object
- Don’t keep stuff for dead objects
- Reasonably efficient code: no Thread#sleep, crazy list iterations, etc.
- shutdown() -> stop thread, clear data
- Deadline: 07 Nov 23:59 local time
Read more..

- https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/
- https://plumbr.eu/handbook/garbage-collection-algorithms-impl-
fications
- https://vimeo.com/181948157
- https://stackoverflow.com/q/19154607