Frugal memory management on the JVM

Galo Navarro
@srvaroa

Guillermo Ontañón
@gontanon

#SchTechTalks
Agenda

- Assuming basic Java / JVM knowledge
- Understanding our app’s workload / footprint
- Tradeoffs to address some of the common pathologies
Memory management on the JVM

Pros:

- Fast allocation / release of memory
- Can optimize on runtime

But if left unchecked may also cause:

- Big latency spikes / jitter
- High memory footprint
### jstat -gc

<table>
<thead>
<tr>
<th>E: Eden</th>
<th>C: capacity</th>
<th>Y: Young GC</th>
<th>T: Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>O: Old gen</td>
<td>U: utilisation</td>
<td>F: Full GG</td>
<td></td>
</tr>
<tr>
<td>M: Metaspace</td>
<td></td>
<td>CCS: Compressed class space</td>
<td></td>
</tr>
<tr>
<td>S: Survivor</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assume generational hypothesis, and expect that most objects will be orphaned soon.
GC logs

-Xloggc:$PATH consider ramdisk / ssd [1]
-XX:+PrintGCDetails detailed GC logging
-XX:+PrintTenuringDistribution ages
-XX:+PrintPromotionFailure
-XX:+PrintGCApplicationStoppedTime pauses, GC but also safepoints ...
-XX:+PrintAdaptiveSizePolicy ergonomic decisions

-XX:+UseGCLogFileRotation
-XX:NumberOfGCLogFiles=$NUM_FILES default 1
-XX:GCLogFileSize=$SIZE[M|K] default 512k
Easy to correlate to service latency metrics

Assume generational hypothesis, and expect most objects to be dereferenced soon

[Object Copy (ms): Min: 2032.6, Avg: 2033.1, Max: 2034.4, Diff: 1.8, Sum: 46762.1]
Different cause

Object Copy (ms): Min: 4781.1, Avg: 4781.7, Max: 4782.4, Diff: 1.3, Sum: 109978.1
GCViewer: gc log visualization

- **Eden**
  - Used
  - See-saw patterns
- **Old Gen**
  - Used
- **Promotions**
  - Evacuation >25G!
  - GCTimes (green)
- **Resizing**
Death by GC shortly after...

inc GC

gc times rise

app pause

again...

resizing
What is in the heap?

jmap -histo $PID
jmap -histo:live $PID -> triggers GC

class JDBCRecord {
    private List<Object> = ..
    private Long timestamp = ..
    private String ..
    private String ..
}

<table>
<thead>
<tr>
<th>num</th>
<th>#instances</th>
<th>#bytes</th>
<th>class name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>456735295</td>
<td>29968183048</td>
<td>C</td>
</tr>
<tr>
<td>2:</td>
<td>141993549</td>
<td>17650832184</td>
<td>[Ljava.lang.Object;</td>
</tr>
<tr>
<td>3:</td>
<td>432874195</td>
<td>13851974240</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>4:</td>
<td>141783960</td>
<td>5671358400</td>
<td>java.util.ArrayList</td>
</tr>
<tr>
<td>5:</td>
<td>220867901</td>
<td>5300829624</td>
<td>java.lang.Long</td>
</tr>
<tr>
<td>6:</td>
<td>3507261</td>
<td>3992725000</td>
<td>I</td>
</tr>
<tr>
<td>7:</td>
<td>90242360</td>
<td>3839836936</td>
<td>[B</td>
</tr>
<tr>
<td>8:</td>
<td>142089373</td>
<td>3410144952</td>
<td>.JDBCRecord</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>num</th>
<th>#instances</th>
<th>#bytes</th>
<th>class name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>309717286</td>
<td>18357971096</td>
<td>C</td>
</tr>
<tr>
<td>2:</td>
<td>103220845</td>
<td>12801064160</td>
<td>[Ljava.lang.Object;</td>
</tr>
<tr>
<td>3:</td>
<td>309717154</td>
<td>9910948928</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>4:</td>
<td>103219696</td>
<td>4128787840</td>
<td>java.util.ArrayList</td>
</tr>
<tr>
<td>5:</td>
<td>103216209</td>
<td>2477189016</td>
<td>.JDBCRecord</td>
</tr>
<tr>
<td>6:</td>
<td>103005153</td>
<td>2472123672</td>
<td>java.lang.Long</td>
</tr>
<tr>
<td>7:</td>
<td>5741</td>
<td>22010304</td>
<td>[B</td>
</tr>
<tr>
<td>8:</td>
<td>211348</td>
<td>5072352</td>
<td>java.lang.Double</td>
</tr>
</tbody>
</table>
Options opt = new OptionsBuilder()
   .include(MyBenchmark.class.getSimpleName())
   .warmupIterations(5)
   .verbosity(VerboseMode.EXTRA)
   .addProfiler(HotspotRuntimeProfiler.class)
   .addProfiler(GCProfiler.class)
   .build();
new Runner(opt).run();
Profiling allocation rate: JMH

Iteration 9: 1464,198 ops/ms

- consumer16: 779,567 ops/ms
- producer16: 684,632 ops/ms
- gc.alloc.rate: 60,088 MB/sec
- gc.alloc.rate.norm: 42,995 B/op
- gc.churn.PS_Eden_Space: 94,241 MB/sec
- gc.churn.PS_Eden_Space.norm: 67,432 B/op
- gc.count: 1,000 counts
- gc.time: 661,000 ms

Useful specially for small-ish sections of the fast path
Best practices
Costs of abstraction

Object headers

http://hg.openjdk.java.net/jdk8/jdk8/hotspot/file/tip/src/share/vm/oops/oop.hpp
http://hg.openjdk.java.net/jdk8/jdk8/hotspot/file/tip/src/share/vm/oops/markOop.hpp

- 64-bit: 12 bytes padded to multiple of 8 → 16 bytes
- 32-bit: 8 bytes padded to multiple of 4 → 12 bytes

References

- Ref = 4 bytes on < 32G heaps
- Ref = 8 bytes on 64-bit JVMs with >32G heaps

Arrays: 1 ref to type, 4 bytes for length, 1 ref per element. Min 8/16 bytes
Boxing

long: 8 bytes → Long: 8 + 16 → 24 bytes (x3)

boolean: 1 bit → Boolean: 1 byte + 12 + 3 bytes padding → 16 bytes (x128)

Real world example: music license cache

20M song catalogue, ~200 countries, 9 types of perms

At 16 bytes per flag = 536 GiB → dataset split in N servers

At 1 bytes per flag = 34 GiB → dataset in 1 server
Boxing

Avoid using primitive types, pack flags in bytes or BitSet:

```java
class Event {
    private long timestamp = ..
    private byte flags = ..
}
```

Scala: value types - very limited: only 1 val & defs, no inheritance, no initialization, sometimes allocates

```scala
class Counter(val underlying: Long) extends AnyVal {
    def inc: Wrapper = new Counter(underlying + 1)
}
```

Java 10: Project Valhalla brings value types “Codes like a class, works like an int!”
while ((line = reader.readLine()) != null) {
    users.add(new User(line));
}

class User {
    private final String name;
    private final Date birth;
    ...
    public User(String s) {
        String[] fields = s.split("::");
        this.name = fields[0];
        this.birth = dateFormat.parse(fields[1]);
        ...
    }
    public String getName() { .. }
    public Date getBirth() { return new Date(birth.getTime) }
    ...
    public String getXXX()
}
Lazy parsing

class User {

    private final String data;
    private volatile Date date = null;

    public Date getBirth() {
        if (date == null) {
            this.date = new Date(findField(1))
        }
        return date;
    }
    ...

    private final String findField(int n) {
        // loop to find field
        ...
    }

    Might make sense.. (or, store offsets but not parse) to delay allocation until it’s really needed

    ● Useful in hashMaps
    ● Think more complex cases (e.g.: network packets)
Intermission
When / where to remove allocations

You’ve already fixed the low hanging fruit (boxing, logs, ...)

GC has been tuned

Your application processes 1000s of QPS, latency sensitive

You have allocation and GC churn, latency and latency jitter are too high

Goal: less (zero?) allocations in your fast path
Techniques

Most are intrusive, lots of work to retrofit

Before optimizing something:

- JVM optimizes stuff: prove garbage with profiler / microbenchmark
- It’s in your fast path

It’s better to know in advance and design up front
Instant throwaway objects

They are objects that you create on the spot and immediately discard

Essentially: local variables

In C++, you’d declare them on the stack, the JVM does ‘escape analysis’

What do do? Promote to instance members and reuse

IMPORTANT: method becomes thread unsafe unless you use a `ThreadLocal`
Escape analysis

public class A {
    private final int x;
    public A(final int _x) {
        this.x = _x;
    }
    public int getX() { return x; }
}

public void f(int n) {
    A a = new A(i);
    System.out.println(a.getX());
}

Likely JIT’d version

public void f(int n) {
    int _x = x;
    System.out.println(_x);
}

Objects that don’t escape the current method or thread might get stack allocation

Disable with -XX:-DoEscapeAnalysis, to compare behaviour
Collections with per-request churn

Lists → always use array-backed lists, ArrayList

Linked lists, trees → Build your own intrusive implementations

Maps, Sets → Each insertion creates Map.Entry. Move off-process or off-heap

Queues → array backed, bounded (also gives you back pressure)

Primitive type collections → avoid boxing by using specialized implementations
Collections

Primitive types, zero allocations...

Trove → http://trove.starlight-systems.com/ (GPL)
OpenHFT → https://github.com/OpenHFT (Apache)

Off-heap implementations exist (note: have not tried them):

MapDB → www.mapdb.org/
OpenHFT

Off-process

Beware: the client library for your caching system may not be allocation free
Interning

The JVM does this for some objects

i.e. numeric vals. < 128, `String.intern()`

The set of possible values of an immutable object is known / manageable

Objects can be constructed from a key composed of primitive types

Keep them cached in a hash table

Consider making the caches `ThreadLocal`
Reusing objects: object pools

Classes become mutable

Instead of allocating, you take from the pool, then release

Preallocation: optional, but can make life easier for GC

Caveats:

Memory leaks, you are responsible for lifecycle management

If your objects point to other objects, clear them (better avoid)
Reusing objects: “stashes”

A special case of an object pool

Objects that only live during processing a request

A thread processes the request start-to-finish: one stash per thread

You can avoid lifecycle management: reclaim all after processing the request
public void run() {
    MyObject o1 = stash.retrieve()
    doWork(o1);
    MyObject o2 = stash.retrieve()
    doWork(o2, o1);
    ...
    stash.release()
}
Off heap

Native `ByteBuffer` or `MappedByteBuffer`

```java
buffer = ByteBuffer.allocateDirect();
buffer.put(index, value);

sun.misc.Unsafe:

address = Unsafe.allocateMemory();
Unsafe.putInt(address, value);
```
(de)serialization

Purpose: RPC protocols / off-heap storage

Simple: use a zero-allocation serialization library:

- SBE → [github.com/real-logic/simple-binary-encoding/wiki](https://github.com/real-logic/simple-binary-encoding/wiki)
- FlatBuffers → [google.github.io/flatbuffers/](https://google.github.io/flatbuffers/)

They’ll work on `ByteBuffer` you provide:

Can be off-heap based: `ByteBuffer.allocateDirect()`

Pool them or use a larger `ByteBuffer` as a ring buffer
(de)serialization

ByteBuffer data = ...;
...
while (data.hasRemaining()) {
    ByteBuffer bytes = data.slice()
    bytes.limit(RECORD_SIZE)
    data.position(data.position() + RECORD_SIZE)
    users.add(new Record(bytes))
}

- Easy to build an Iterator[Record] over a ByteBuffer
- Single copy of the data (can also be memory-mapped)
- Easier to achieve cache friendliness
We’re hiring!
Q&A

Thanks!