

Spring Blog

[All Posts](#)  [Engineering](#)  [Releases](#)  [News and Events](#) 

Spring Boot Memory Performance

It has sometimes been suggested that Spring and Spring Boot are “heavyweight”, perhaps just because they allow apps to punch above their weight, providing a lot of features for not very much user code. In this article we concentrate on memory usage and ask if we can quantify the effect of using Spring? Specifically we would like to know more about the real overhead of using Spring compared to other JVM applications. We start by creating a basic application with Spring Boot, and look at a few different ways to measure it when it is running. Then we look at some comparison points: plain Java apps, apps that use Spring but not Spring Boot, an app that uses Spring Boot but no autoconfiguration, and some Ratpack sample apps.

Vanilla Spring Boot App

As a baseline we build a static app with a few webjars and `spring.resources.enabled=true`. This is perfectly fine for serving nice-looking static content maybe with a REST endpoint or two. The source code for the app we used to test is [in github](#). You can build it with the `mvnw` wrapper script if you have a JDK 1.8 available and on your path (`mvnw package`). It can be launched like this:

```
$ java -Xmx32m -Xss256k -jar target/demo-0.0.1-SNAPSHOT.jar
```

[COPY](#)

Then we add some load, just to warm up the thread pools and force all the code paths to be exercised:

```
$ ab -n 2000 -c 4 http://localhost:8080/
```

[COPY](#)

We can try and limit threads a bit in `application.properties`:

```
server.tomcat.max-threads: 4
```

[COPY](#)

but in the end it doesn't make a lot of difference to the numbers. We conclude from the analysis below that it would save at most a MB with the stack size we are using. All the

Spring Boot webapps we analyse have this same configuration.

We might have to worry about how big the classpath is, in order to estimate what happens to the memory. Despite some claims in the internet that the JVM memory maps all jars on the classpath, we actually don't find any evidence that the size of the classpath has any effect on the running app. For reference, the size of the dependency jars (not including JDK) in the vanilla sample is 18MB:

```
$ jar -tvf target/demo-0.0.1-SNAPSHOT.jar | grep lib/*.jar | awk '{tot+=$1;} END {print tot}'  
18893563
```

COPY

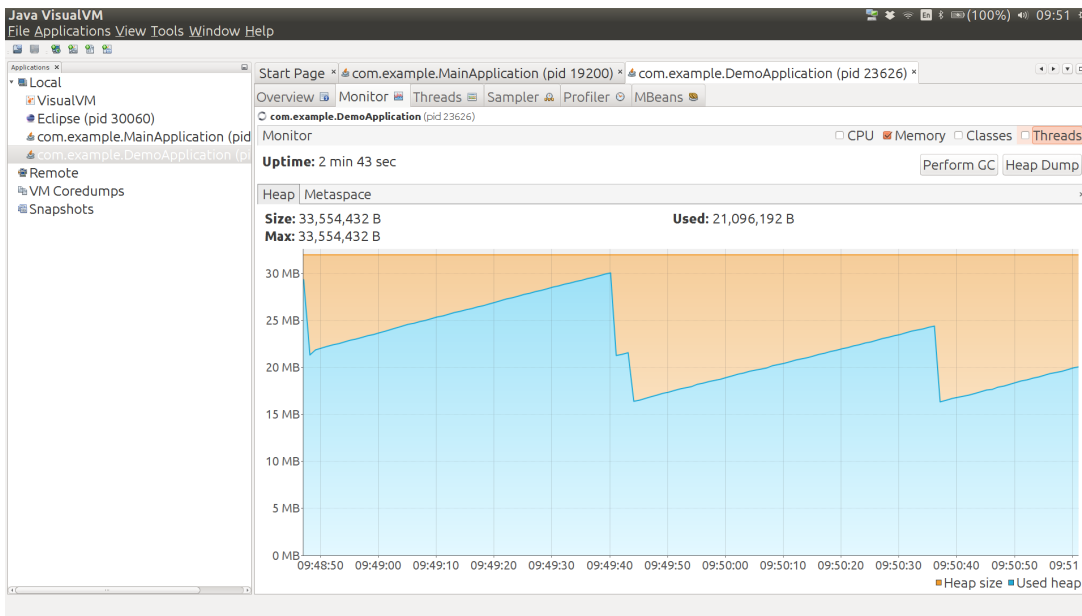
This includes Spring Boot Web and Actuator starters, plus 3 or 4 webjars for static resources and the webjar locator. A completely minimal Spring Boot application including Spring and some logging but no web server would be around 5MB of jars.

JVM Tools

To measure memory usage there are some tools in the JVM. You can get quite a lot of useful information from JConsole or JVisualVM (with the JConsole plugin so you can inspect MBeans).

Heap usage for our vanilla app is a saw tooth, bounded above by the heap size and below by the amount of memory used in a quiescent state. The average weighs in at roughly 25MB for an app under load (and 22MB after a manual GC). JConsole also reports 50MB non-heap usage (which is the same as you get from the `java.lang:type=Memory` MBean). The non-heap usage breaks down as Metaspace: 32MB, Compressed Class Space: 4MB, Code Cache: 13MB (you can get these numbers from the `java.lang:type=MemoryPool,name=*` MBeans). There are 6200 classes and 25 threads, including a few that are added by the monitoring tool that we use to measure them.

Here's a graph of the heap usage from a quiescent app under load, followed by a manual garbage collection (the double nick in the middle) and a new equilibrium with a lower heap usage.



Some tools in the JVM other than JConsole might also be interesting. The first is `jps` which is useful for getting the process id of the app you want to inspect with the other tools:

```
$ jps
4289 Jps
4330 demo-0.0.1-SNAPSHOT.jar
```

COPY

Then we have the `jmap` histogram:

```
$ jmap -histo 4330 | head
```

COPY

num	#instances	#bytes	class name
1:	5241	6885088	[B
2:	21233	1458200	[C
3:	2548	1038112	[I
4:	20970	503280	java.lang.String
5:	6023	459832	[Ljava.lang.Object;
6:	13167	421344	java.util.HashMap\$Node
7:	3386	380320	java.lang.Class

This data is of limited use because you can't trace the "big" objects back to their owners. For that you need a more fully featured profiler, like YourKit. YourKit does the aggregation for you and presents a list (although the details of how it does that are rather unclear).

Classloader statistics might also be revealing, and `jmap` has a way to inspect the classloaders in an app. It needs to run as root:

```
$ sudo ~/Programs/jdk1.8.0/bin/jmap -clstats 4330
Attaching to process ID 4330, please wait...
Debugger attached successfully.
Server compiler detected.
```

COPY

```
JVM version is 25.60-b23
finding class loader instances ..done.
computing per loader stat ..done.
please wait.. computing liveness.....liveness analysis
may be inaccurate ...
class_loader  classes  bytes  parent_loader  alive?  type

<bootstrap>      2123      3609965      null          live      <internal>
0x00000000f4b0d730      1      1476      0x00000000f495c890      dead
sun/reflect/DelegatingClassLoader@0x0000000100009df8
0x00000000f5a26120      1      1483      0x00000000f495c890      dead
sun/reflect/DelegatingClassLoader@0x0000000100009df8
0x00000000f52ba3a8      1      1472      null          dead
sun/reflect/DelegatingClassLoader@0x0000000100009df8
0x00000000f5a30520      1      880      0x00000000f495c890      dead
sun/reflect/DelegatingClassLoader@0x0000000100009df8
0x00000000f495c890      3972      6362902      0x00000000f495c8f0      dead
org/springframework/boot/loader/LaunchedURLClassLoader@0x0000000100060828
0x00000000f5b639b0      1      1473      0x00000000f495c890      dead
sun/reflect/DelegatingClassLoader@0x0000000100009df8
0x00000000f4b80a30      1      1473      0x00000000f495c890      dead
sun/reflect/DelegatingClassLoader@0x0000000100009df8
...

total = 93      6300      10405986      N/A      alive=1, dead=92
N/A
```

There are loads of “dead” entries, but there is also a warning that the liveness information is not accurate. A manual GC doesn’t get rid of them.

Kernel Memory Tools

You would think that a Linux OS would provide plenty of insight into a running process, and it does, but Java processes are notoriously hard to analyse. This [popular SO link](#) talks about some of the problems in general. Lets have a look at some of the tools that are available and see what they tell us about our app.

First up is the good old `ps` (the tool you use to look at processes on the command line). You can get a lot of the same information from `top`. Here’s our application process:

```
$ ps -au
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
dsyer    4330  2.4  2.1 2829092 169948 pts/5    S1   18:03   0:37 java -Xmx32m -
Xss256k -jar target/demo-0.0.1-SNAPSHOT.jar
...
```

COPY

RSS (Resident Set Size) values are in the range 150-190MB according to `ps`. There is a tool called `smem` that is supposed to give a more sanitized view, and to accurately reflect non-shared memory, but the values there (for instance of PSS) aren’t that different. Interestingly the PSS values for a non-JVM process are usually *significantly* lower than RSS, whereas for a JVM they are comparable. The JVM is very jealous of its memory.

A lower level tool is `pmap`, where we can look at the memory allocations assigned to a process. Numbers from `pmap` don't seem to make much sense either:

```
$ pmap 4330
0000000000400000      4K r-x-- java
0000000000600000      4K rw--- java
000000000184c000    132K rw--- [ anon ]
00000000fe000000  36736K rw--- [ anon ]
00000001003e0000 1044608K ----- [ anon ]
...
00007ffe2de90000      8K r-x-- [ anon ]
fffffffffff6000000      4K r-x-- [ anon ]
total                3224668K
```

COPY

i.e. over 3GB for a process that we know is only using 80MB. Just counting the '---' entries gives you nearly all the 3GB. At least that's consistent with the VSZ numbers from `ps`, but not very useful for capacity management.

Someone commented that the RSS values were accurate on his machine, which is interesting. They definitely didn't work for me (Ubuntu 14.04 on Lenovo Thinkpad). Also, here's another interesting article on [JVM memory stats in Linux](#).

Scale up Processes

A good test of how much memory is actually being used by a process is to keep launching more of them until the operating system starts to crumple. For example, to launch 40 identical vanilla processes:

```
$ for f in {8080..8119}; do (java -Xmx32m -Xss256k -jar target/demo-0.0.1-SNAPSHOT.jar --server.port=$f 2>&1 > target/$f.log &); done
```

COPY

They are all competing for memory resources so it takes them all a while to start, which is fair enough. Once they all start they serve their home pages quite efficiently (51ms latency over a crappy LAN at 99th percentile). Once they are up and running, stopping and starting one of the processes is relatively quick (a few seconds not a few minutes).

The VSZ numbers from `ps` are off the scale (as expected). The RSS numbers look high too:

```
$ ps -au
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
dsyer    27429   2.4   2.1 2829092 169948 pts/5    S1   18:03   0:37 java -Xmx32m -Xss256k -jar target/demo-0.0.1-SNAPSHOT.jar --server.port=8081
dsyer    27431   3.0   2.2 2829092 180956 pts/5    S1   18:03   0:45 java -Xmx32m -Xss256k -jar target/demo-0.0.1-SNAPSHOT.jar --server.port=8082
...
```

COPY

RSS values are still in the range 150-190MB. If all 40 processes were independently using this much memory that would account for 6.8GB, which would blow my 8GB laptop out of the water. It runs fine, so most of that RSS value is not really independent of other processes.

The Proportional Shared Size (PSS) from `smem` might a better estimate we have of actual memory usage, but in fact it is not that different from the RSS values:

								COPY
\$ smem	PID	User	Command	Swap	USS	PSS	RSS	
...								
27435	dsyer		java -Xmx32m -Xss256k -jar	0	142340	142648	155516	
27449	dsyer		java -Xmx32m -Xss256k -jar	0	142452	142758	155568	
...								
27441	dsyer		java -Xmx32m -Xss256k -jar	0	175156	175479	188796	
27451	dsyer		java -Xmx32m -Xss256k -jar	0	175256	175579	188900	
27463	dsyer		java -Xmx32m -Xss256k -jar	0	179592	179915	193224	

We can hypothesize that maybe the PSS number is still hugely inflated by shared read-only memory (e.g. mapped jar files).

The 40 processes pretty much filled up the available memory on my laptop (3.6GB before the apps started), and some paging was happening, but not much. We can turn that into an estimate of the process size: $3.6\text{GB}/40 = 90\text{MB}$. Not far off the JConsole estimate.

Do Nothing Plain Java App

As a useful comparison point, let's make a really basic Java application that stays alive when we run it so we can measure its memory consumption:

```
public class Main throws Exception {
    public static void main (String[] args) {
        System.in.read();
    }
}
```

Results: heap 6MB, non-heap 14MB (Code Cache 4MB, Compressed Class Space 1MB, Metaspace 9MB), 1500 classes. Hardly any classes loaded so no surprise really.

Do Nothing Spring Boot App

Now suppose we do the same thing but load a Spring application context as well:

```
@SpringBootApplication
public class MainApplication implements ApplicationRunner {
```

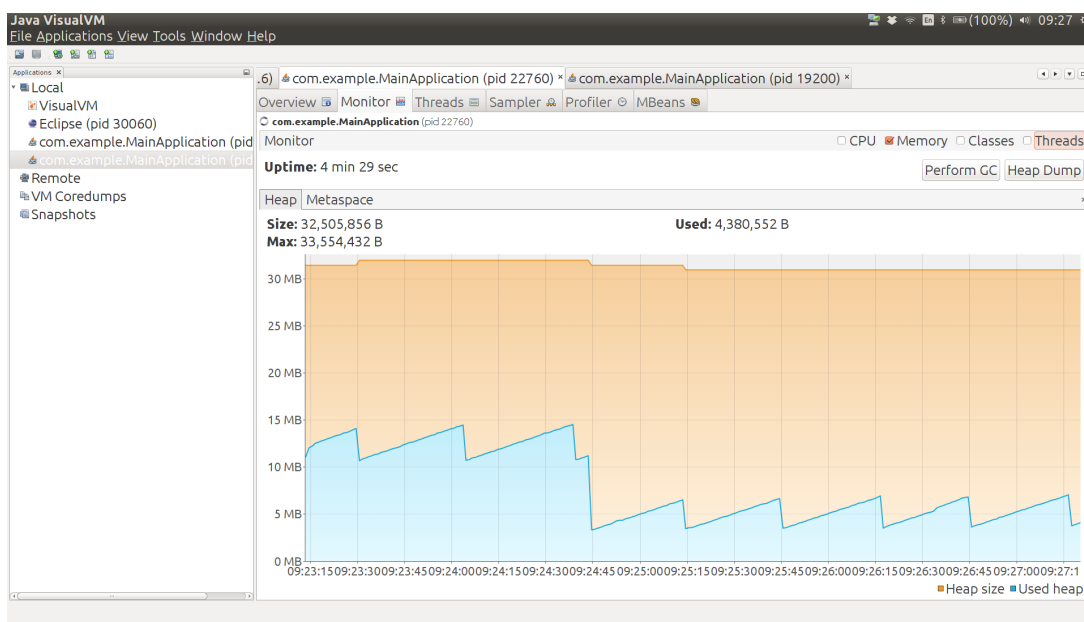
```

@Override
public void run(ApplicationArguments args) throws Exception {
    System.in.read();
}

public static void main(String[] args) throws Exception {
    SpringApplication.run(MainApplication.class, args);
}
}

```

Heap 12MB (but drops to 6MB after a manual GC), non-heap 26MB (Code Cache 7MB, Compressed Class Space 2MB, Metaspace 17MB), 3200 classes. The graph below shows the heap usage from launching the app to the end state. The big drop in the middle is the manual GC, and you can see that after this the app stabilizes at a different saw tooth.



Does Spring Boot itself (as opposed to just Spring) add a lot of overhead to this application? For starters, we can test that by removing the `@SpringBootApplication` annotation. Doing that means that we load a context but don't do any autoconfiguration. The result is: heap 11MB (drops to 5MB after a manual GC), non-heap 22MB (Code Cache 5MB, Compressed Class Space 2MB, Metaspace 15MB), 2700 classes. The Spring Boot autoconfiguration premium, measured this way, is about 1MB heap and 4MB non-heap.

Going a step further, we can create a Spring application context manually without using any Spring Boot code at all. Doing this drops the heap usage to 10MB (drops to 5MB after a manual GC), non-heap to 20MB (Code Cache 5MB, Compressed Class Space 2MB, Metaspace 13MB), 2400 classes. The Spring Boot total premium, measured this way, is less than 2MB heap and about 6MB non-heap memory.

Ratpack Groovy App

A simple Ratpack groovy app can be created using [lazybones](#):

```
$ lazybones create ratpack .  
$ ./gradlew build  
$ unzip build/distributions/ratpack.zip  
$ JAVA_OPTS='-Xmx32m -Xss256k' ./ratpack/bin/ratpack
```

[COPY](#)

```
$ ls -l build/distributions/ratpack/lib/*.jar | awk '{tot+=$5;} END {print tot}'  
16277607
```

[COPY](#)

The used heap is pretty low to start with (13MB), grows to 22MB over time. Metaspace is about 34MB. JConsole reports 43MB non-heap usage. There are 31 threads.

Ratpack Java App

Here's a really basic static app:

```
import ratpack.server.BaseDir;  
import ratpack.server.RatpackServer;  
  
public class DemoApplication {  
  
    public static void main(String[] args) throws Exception {  
        RatpackServer.start(s -> s  
            .serverConfig(c -> c.baseDir(BaseDir.find()))  
            .handlers(chain -> chain  
                .all(ctx -> ctx.render("root handler!"))  
            )  
        );  
    }  
}
```

[COPY](#)

It runs in about 16MB heap, 28MB non-heap as a Spring Boot fat jar. As a regular gradle application it's a bit lighter on heap (the cached jars aren't needed) but uses the same non-heap memory. There are 30 threads. Interestingly there is no object that is bigger than 300KB, whereas our Spring Boot apps with Tomcat generally have 10 or more objects above that level.

Variations on the Vanilla App

Running from exploded jar shaves up to 6MB off the heap (the difference is cached jar data in the launcher). Also makes startup a bit faster: less than 5s compared to as much as 7s when memory is constrained with the fat jar.

A slimmed down version of the app with no static resources or webjars runs at 23MB heap and 41MB non-heap as exploded archive (starts in less than 3s). The non-heap usage

breaks down as Metaspace: 35MB, Compressed Class Space: 4MB, Code Cache: 4MB.

Spring `ReflectionUtils` jumps to near the top of the memory chart in YourKit with Spring 4.2.3 (2nd only to Tomcat `NioEndpoint`). The `ReflectionUtils` should shrink under memory pressure but they don't in practice so Spring 4.2.4 [clears the caches](#) once the context has started, resulting in some memory savings (down to about 20MB heap).

`DefaultListableBeanFactory` drops down to 3rd place and is almost half the size it was with the resource chain (webjars locator) but it won't shrink any further without removing more features.

It turns out that the `NioEndpoint` has a 1MB "oom parachute" that it holds onto until it detects an `OutOfMemoryError`. You can customize it to zero and forgo the parachute to save an extra MB of heap, e.g:

```
@SpringBootApplication
public class SlimApplication implements EmbeddedServletContainerCustomizer {

    @Override
    public void customize(ConfigurableEmbeddedServletContainer container) {
        if (container instanceof TomcatEmbeddedServletContainerFactory) {
            TomcatEmbeddedServletContainerFactory tomcat =
                (TomcatEmbeddedServletContainerFactory) container;
            tomcat.addConnectorCustomizers(connector -> {
                ProtocolHandler handler = connector.getProtocolHandler();
                if (handler instanceof Http11NioProtocol) {
                    Http11NioProtocol http = (Http11NioProtocol) handler;
                    http.getEndpoint().setOomParachute(0);
                }
            });
        }
    }
    ...
}
```

[COPY](#)

Using Jetty instead of Tomcat makes no difference whatsoever to the overall memory or heap, even though the `NioEndpoint` is high on the "Biggest objects" list in YourKit (takes about 1MB), and there is no corresponding blip for Jetty. It also doesn't start up any quicker.

As an example of a "real" Spring Boot app, Zipkin (Java) runs fine with with

`-Xmx32m -Xss256k`, at least for short periods. It settles with a heap of about 24MB and non-heap about 55MB.

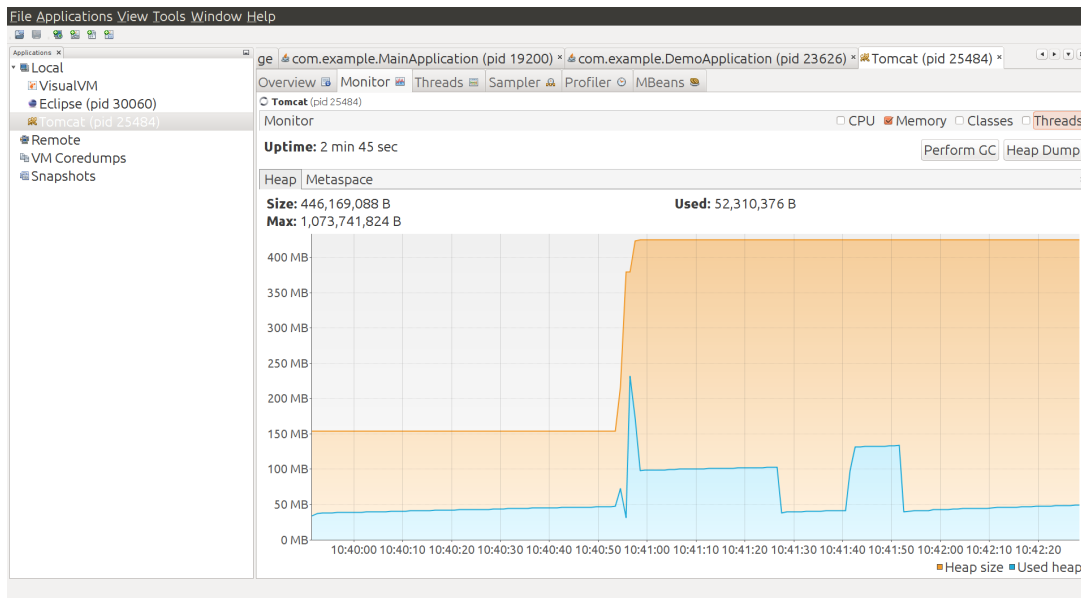
The `spring-cloud-stream` sample sink (with Redis transport) also runs fine with

`-Xmx32m -Xss256k` and similar memory usage profile (i.e. roughly 80MB total). The actuator

endpoints are active but don't do much to the memory profile. Slightly slower startup maybe.

Tomcat Container

Instead of using the emedded container in Spring Boot, what if we deploy a traditional war file to a Tomcat container?



The container starts and warms up a bit and uses of order 50MB heap, and 40MB non-heap. Then we deploy a war of the vanilla Spring Boot app, and there's a spike in heap usage, which settles down to about 100MB. We do a manual GC and it drops down to below 50MB, and then add some load and it jumps up to about 140MB. A manual GC drops it back down to below 50MB. So we have no reason to believe this app is really using much if any additional heap compared to the container. It uses some when under load, but it can always reclaim it under GC pressure.

Metaspace, however tells a different story, it goes up from 14MB to 41MB in the single app under load. Total non-heap usage is reported at 59MB in the final state.

Deploy another application

If we add another copy of the same application to the Tomcat container, trough heap consumption goes up a bit (above 50MB) and metaspace is up to 55MB. Under load heap usage jumps to 250MB or so, but always seems to be reclaimable.

Then we add some more apps. With six apps deployed the metaspace is up to 115MB and total non-heap to 161MB. This is consistent with what we saw for a single app: each one costs us about 20MB non-heap memory. Heap usage hits 400MB under load, so this doesn't go up proportionally (however it is being managed from above, so maybe that's

not surprising). The trough of heap usage is up to about 130MB, so the cumulative effect of adding apps on the heap is visible there (about 15MB per app).

When we constrain Tomcat to the same amount of heap that the six apps would have in our vanilla embedded launch (`-Xmx192m`) the heap under load is more or less at its limit (190MB), and the trough after a manual GC is 118MB. Non-heap memory is reported as 154MB. The heap trough and non-heap usage is not identical but consistent with the unconstrained Tomcat instance (which actually had a 1GB heap). Compared to the embedded containers the total memory usage, including the full heap, is a bit smaller because some of the non-heap memory is apparently shared between apps (344MB compared to 492MB). For more realistic apps that require more heap themselves the difference will not be proportionally as big (50MB out of 8GB is negligible). Also any app that manages its own thread pool (not uncommon in real life Spring applications) will incur an additional non-heap memory penalty for the threads it needs.

Rule of Thumb Process Sizes

A very rough estimate for actual memory usage would be the heap size plus 20 times the stack size (for the 20 threads typical in a servlet container), plus a bit, so maybe 40MB per process in our vanilla app. That estimate is a bit low, given the JConsole numbers (50MB plus the heap, or 82MB). We can observe, though, that the non-heap usage in our apps is roughly proportional to the number of classes loaded. Once you correct for the stack size the correlation improves, so a better rule of thumb might be one that is proportional to the number of classes loaded:

```
memory = heap + non-heap
```

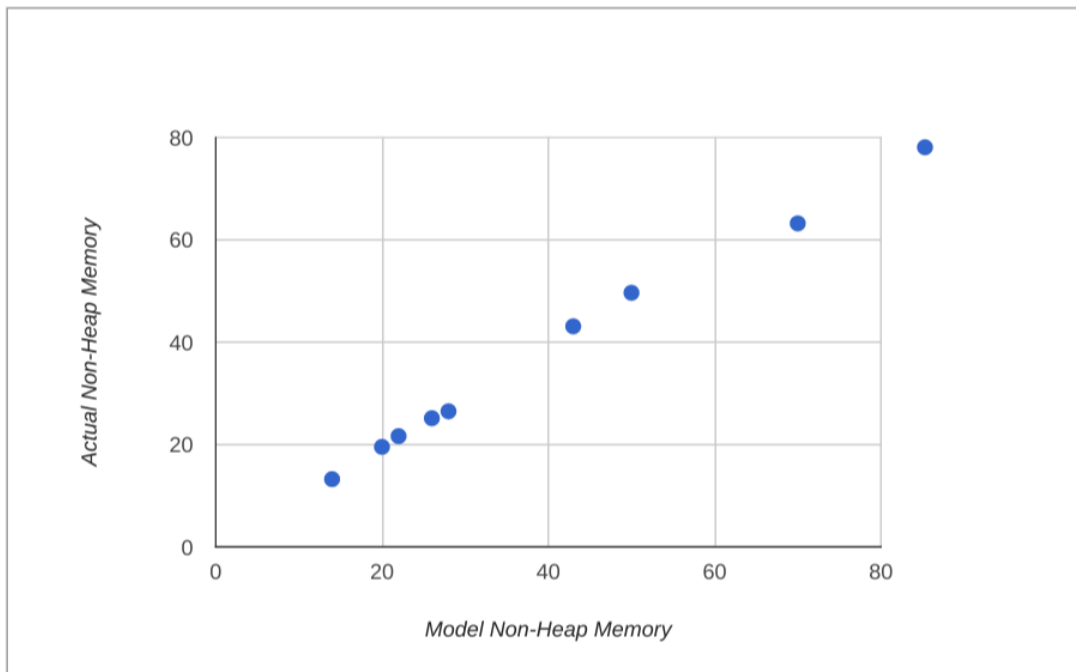
[COPY](#)

```
non-heap = threads x stack + classes x 7/1000
```

The vanilla app loads about 6000 classes and the do nothing Java main loads about 1500. The estimate is accurate for the vanilla app and the do nothing Java app.

Adding Spring Cloud Eureka discovery only loads about another 1500 classes, and uses about 40 threads, so it should use a bit more non-heap memory, but not a lot (and indeed it does use about 70MB with 256KB stacks, where the rule of thumb would predict 63MB).

The performance of this model for the apps we measured is shown below:



Summary of Data

Application	Heap (MB)	Non Heap (MB)	Threads	Classes
Vanilla	22	50	25	6200
Plain Java	6	14	11	1500
Spring Boot	6	26	11	3200
No Actuator	5	22	11	2700
Spring Only	5	20	11	2400
Eureka Client	80*	70	40	7600
Ratpack Groovy	22	43	24	5300
Ratpack Java	16	28	22	3000

* Only the Eureka client has a larger heap: all the others are set explicitly to `-Xmx32m`.

Conclusions

The effect Spring Boot on its own has on a Java application is to use a bit more heap and non-heap memory, mostly because of extra classes it has to load. The difference can be quantified as roughly an extra 2MB heap and 12MB non-heap. In a real application that might consume many times as much for actual business purposes this is pretty insignificant. The difference between vanilla Spring and Spring Boot is a few MB total

(neither here nor there really). The Spring Boot team have only just started measuring things in this level of detail so we can probably expect optimizations in the future anyway. When we compare memory usage for apps deployed in a single Tomcat container with the same apps deployed as independent processes, not surprisingly the single container packs the apps more densely in memory. The penalty for a standalone process is mainly related to non-heap usage though, which adds up to maybe 30MB per app when the number of apps is much larger than the number of containers (and less otherwise). We wouldn't expect this to increase as apps use more heap, so in most real apps it is not significant. The benefits of deploying an app as an independent process following the [twelve-factor](#) and Cloud Native principles outweigh the cost of using a bit more memory in our opinion. As a final note, we observe that the native tools in the operating system are not nearly as good as the ones provided by the JVM, when you want to inspect a process and find out about its memory usage.

Comments **Community** **Privacy Policy** **Login** ▾

Favorite 22 **Tweet** **Share** **Sort by Best** ▾

Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS

Name



Arik Cohen • 7 years ago • edited

To hell with the critics. You guys are doing a great job. Thank you for doing it.

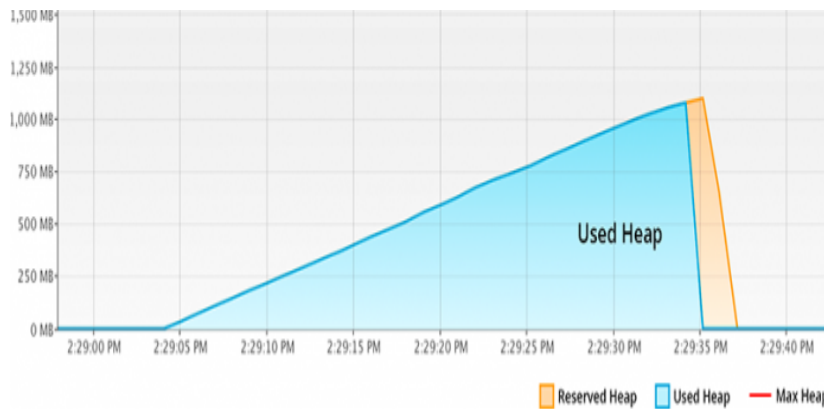
22 ^ | ▾ • Reply • Share ›



sirus • 5 years ago

There is an additional option to make Spring Boot apps more lightweight. If JVM configured properly then memory vertical scaling can work perfectly for Spring Boot apps, optimizing RAM usage according to the current load inside containers. The selected garbage collector is one of the main foundational bricks and its settings can influence the whole project, so use G1 or Shenandoah (experimental) GC to enable the elasticity. More details are in my latest article "Minimize Java Memory Usage with the Right Garbage Collector" <https://www.javacodegeeks.c...>





10 ^ | v • Reply • Share ›



Nicolas Larcipretti • 3 years ago

Isn't it time to redo this study? How much has spring evolved since 2015 in this matter?

4 ^ | v • Reply • Share ›



Dave Syer ➔ Nicolas Larcipretti • 3 years ago • edited

We do a lot of work in this area. Most recently there have been a lot of changes in Spring and Boot to improve startup time (<https://spring.io/blog/2018...>). Many of those changes were implemented by limiting memory usage on startup, so the effect is to be able to start in smaller containers, but not there's not actually much difference to the long-running quiescent values reported here. The other thing that happened since 2015 is the whole reactive stack. Using Webflux can definitely reduce off-heap memory usage (fewer threads), but that only works for the right class of application (non-blocking io).

^ | v • Reply • Share ›



karthik muthyam • 3 years ago

let us assume application have 10+ spring boot micro service .which is the best way of deploying in production environment bellow two options?.

1, using embedded server per service run through the java - jar xyz.jar?

2, using the external application server like (Jboss or tomcat)?

2 ^ | v • Reply • Share ›



Surya b • 4 years ago

Good Information! About analyzing GC logs, <http://gceasy.io/> is also good. It is free web tool. Just upload your application's GC Logs & review the beautiful reports instantly.

1 ^ | v • Reply • Share ›



AlexH • 7 years ago • edited



"The effect Spring Boot on its own has on a Java application is to use a bit more heap and non-heap memory, mostly because of extra classes it has to load. The difference can be quantified as roughly an extra 2MB heap and 12MB non-heap. In a real application that might consume many times as much for actual business purposes this is pretty insignificant."

This is also my impression with several services I've done with Spring Boot. The rest is much up to Java platform itself. So keep an eye on this guys and continue with great Spring Boot!

1 ^ | v • Reply • Share ›



jiujiu • 6 years ago

Running several microservices on my laptop (a mac with 16GB) I have no memory issues. Nevertheless the services are taking 10x times to start on a cluster managed by Marathon/Mesos where we put constraints on memory usage. It is strange I do not know how to tune the memory of my microservice

1 ^ | v 3 • Reply • Share ›



yhjho • 3 years ago

is it possible to deploy an basic spring-boot application with less than 256Mb memory?

^ | v • Reply • Share ›



Ole Ersoy • 5 years ago

Hear, Hear!

^ | v • Reply • Share ›



Subramanya Mujaadar • 6 years ago



Below is the heap graph for my Spring Boot application.

^ | v • Reply • Share ›



Archimedes Trajano → Subramanya Mujaadar

• 5 years ago

Having it have a saw pattern like that would indicate that GC is kicking in quite often.

2 ^ | v • Reply • Share ›



Mariano Eloy Fernández Osca • 6 years ago

Just out of curiosity, does anybody know what's causing that saw tooth in memory profile? I've seen it myself even when the Spring Boot app is idle.

^ | v • Reply • Share ›



J. Carlos Muro → Mariano Eloy Fernández Osca

• 5 years ago

Hi Mariano. Even though the application is idle from the user's perspective, internally there is much stuff going on, not necessarily due to Spring Boot. For instance, providing data to Jvisualvm (as it seems those charts are screenshots from jvisualvm) is causing activity in the JVM that is creating some objects.

What you see in the chart are the collections of such short lived objects (I believe these are minor GC's), by the looks every ~20 minutes.

I have attached jvisualvm to a running instance of spring boot and can see exactly the same pattern with the same frequency and memory plant.

^ | v • Reply • Share ›



Enric Sala • 7 years ago • edited

This post was very interesting, thanks!

Lately I had been wondering about the memory performance of Spring Boot because of some problems I had on Pivotal Web Services.

Basically, I was unable to start a little Spring Boot app on a 128mb instance. It only started when using 256mb but the instance restarted several times a day with very light load (50 rpts / day) because it ran out of memory. The only way for it to run smoothly was to use 512mb instances. I still don't know where the memory went, since the application was a REST API with 10 CRUD-like endpoints, a little logic, and 3 Spring Data Mongo repositories.

^ | v • Reply • Share ›



jambay → Enric Sala • 7 years ago

in your case, it sounds like tweaking the memory settings the java buildpack sets for you may be necessary. you can do this with env vars. see <https://github.com/cloudfou...>

^ | v • Reply • Share ›



Enric Sala → jambay • 7 years ago

I will take a look at those settings, thanks!

^ | v • Reply • Share ›



Dave Syer → Enric Sala
• 7 years ago • edited

To expand a little in what James said: Cloud Foundry (and other similar platforms) impose limits on *container* size, which doesn't directly correlate with anything in the JVM. So it has to guess the JVM settings, and you can easily find yourself with an app that busts the container limit even though it wasn't using a lot of heap, or didn't need a lot of heap (Spring Cloud apps are very hungry for non-heap memory for instance, if they use load balancers and circuit breakers, because they need a lot of threads). The only solution is to tweak the JVM settings manually. It takes a bit of trial and error but you should be able to test it locally first quite easily.

^ | v • Reply • Share ›



Enric Sala → Dave Syer • 7 years ago

Thanks Dave, I was assuming the container did this for me. I feel that I need to get a good read on those JVM settings