(http://baeldung.com)

How to Warm Up the JVM

Last modified: April 15, 2018

by baeldung (http://www.baeldung.com/author/baeldung/)

Java (http://www.baeldung.com/category/java/) +

1. Overview

The JVM is one of the oldest yet powerful virtual machines ever built.

In this article, we have a quick look at what it means to warm up a JVM and how to do it.

2. JVM Architecture Basics

Whenever a new JVM process starts, all required classes are loaded into memory by an instance of the ClassLoader (http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/ClassLoader.html). This process takes place in three steps:

1. **Bootstrap Class Loading:** The "*Bootstrap Class Loader*" loads Java code and essential Java classes such as *java.lang.Object* into memory. These loaded classes reside in *JRE\lib\rt.jar*.
2. **Extension Class Loading**: The ExtClassLoader (http://www.docjar.com/docs/api/sun/misc/Launcher$ExtClassLoader.html) is responsible for loading all JAR files located at the *java.ext.dirs* path. In non-Maven or non-Gradle based applications, where a developer adds JARs manually, all those classes are loaded during this phase.
3. **Application Class Loading**: The AppClassLoader (http://www.docjar.com/docs/api/sun/misc/Launcher$AppClassLoader.html) loads all classes located in the application class path.

This initialization process is based on a lazy loading scheme.

3. What Is Warming up the JVM

Once class-loading is complete, all important classes (used at the time of process start) are pushed into the JVM cache (native code) (https://www.ibm.com/support/knowledgecenter/en/SSAW57_8.5.5/com.ibm.websphere.nd.doc/ae/rdyn_tunediskcache.html) – which makes them accessible faster during runtime. Other classes are loaded on a per-request basis.

The first request made to a Java web application is often substantially slower than the average response time during the lifetime of the process. This warm-up period can usually be attributed to lazy class loading and just-in-time compilation.

Keeping this in mind, for low-latency applications, we need to cache all classes beforehand – so that they're available instantly when accessed at runtime.

This process of tuning the JVM is known as warming up.

4. Tiered Compilation

Thanks to the sound architecture of the JVM, frequently used methods are loaded into the native cache during the application life-cycle.

We can make use of this property to force-load critical methods into the cache when an application starts. To that extent, we need to set a VM argument named ***Tiered Compilation***:

```
-XX:CompileThreshold -XX:TieredCompilation
```

Normally, the VM uses the interpreter to collect profiling information on methods that are fed into the compiler. In the tiered scheme, in addition to the interpreter, the client compiler is used to generate compiled versions of methods that collect profiling information about themselves.

Since compiled code is substantially faster than interpreted code, the program executes with better performance during the profiling phase.

Applications running on JBoss and JDK version 7 with this VM argument enabled tend to crash after some time due to a documented bug (https://issues.jboss.org/browse/JBEAP-26). The issue has been fixed in JDK version 8.

Another point to note here is that in order to force load, we've to make sure that all (or most) classes that are to going be executed need to be accessed. It's similar to determining code coverage during unit testing. The more code is covered, the better the performance will be.

The next section demonstrates how this can be implemented.

5. Manual Implementation

We may implement an alternate technique to warm up the JVM. In this case, a simple manual warm-up could include repeating the creation of different classes thousands of times as soon as the application starts.

Firstly, we need to create a dummy class with a normal method:

```java
public class Dummy {
    public void m() {
    }
}
```

Next, we need to create a class that has a static method that will be executed at least 100000 times as soon as application starts and with each execution, it creates a new instance of the aforementioned dummy class we created earlier:

```java
public class ManualClassLoader {
    protected static void load() {
        for (int i = 0; i < 100000; i++) {
            Dummy dummy = new Dummy();
            dummy.m();
        }
    }
}
```

Now, in order to **measure the performance gain**, we need to create a main class. This class contains one static block that contains a direct call to the *ManualClassLoader's load()* method.

Inside the main function, we make a call to the *ManualClassLoader's load()* method once more and capture the system time in nanoseconds just before and after our function call. Finally, we subtract these times to get the actual execution time.

We've to run the application twice; once with the *load()* method call inside the static block and once without this method call:

```java
public class MainApplication {
    static {
        long start = System.nanoTime();
        ManualClassLoader.load();
        long end = System.nanoTime();
        System.out.println("Warm Up time : " + (end - start));
    }
    public static void main(String[] args) {
        long start = System.nanoTime();
        ManualClassLoader.load();
        long end = System.nanoTime();
        System.out.println("Total time taken : " + (end - start));
    }
}
```

Below the results are reproduced in nanoseconds:

With Warm Up	No Warm Up	Difference(%)
1220056	8903640	730
1083797	13609530	1256
1026025	9283837	905
1024047	7234871	706
868782	9146180	1053

As expected, with warm up approach shows much better performance than the normal one.

Of course, this is a **very simplistic benchmark** and only provides some surface-level insight into the impact of this technique. Also, it's important to understand that, with a real-world application, we need to warm up with the typical code paths in the system.

6. Tools

We can also use several tools to warm up the JVM. One of the most well-known tools is the Java Microbenchmark Harness, JMH (http://openjdk.java.net/projects/code-tools/jmh/). It's generally used for micro-benchmarking. Once it is loaded, **it repeatedly hits a code snippet and monitors the warm-up iteration cycle.**

To use it we need to add another dependency to the *pom.xml*:

```
<dependency>
    <groupId>org.openjdk.jmh</groupId>
    <artifactId>jmh-core</artifactId>
    <version>1.19</version>
</dependency>
<dependency>
    <groupId>org.openjdk.jmh</groupId>
    <artifactId>jmh-generator-annprocess</artifactId>
    <version>1.19</version>
</dependency>
```

We can check the latest version of JMH in Central Maven Repository (https://search.maven.org/#search%7Cga%7C1%7Corg.openjdk.jmh).

Alternatively, we can use JMH's maven plugin to generate a sample project:

```
mvn archetype:generate \
    -DinteractiveMode=false \
    -DarchetypeGroupId=org.openjdk.jmh \
    -DarchetypeArtifactId=jmh-java-benchmark-archetype \
    -DgroupId=com.baeldung \
    -DartifactId=test \
    -Dversion=1.0
```

Next, let's create a *main* method:

```
public static void main(String[] args)
  throws RunnerException, IOException {
    Main.main(args);
}
```

Now, we need to create a method and annotate it with JMH's *@Benchmark* annotation:

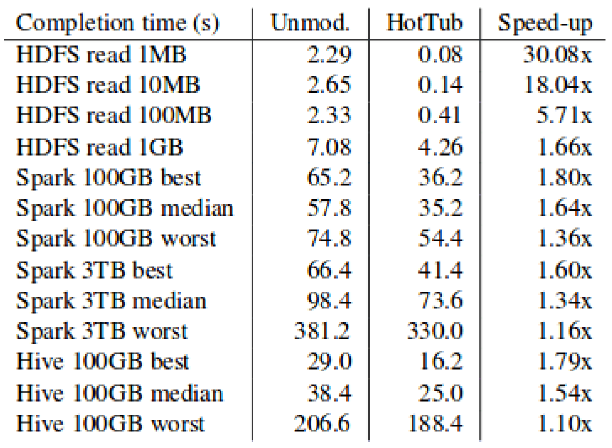
```
@Benchmark
public void init() {
    //code snippet
}
```

Inside this *init* method, we need to write code that needs to be executed repeatedly in order to warm up.

7. Performance Benchmark

In the last 20 years, most contributions to Java were related to the GC (Garbage Collector) and JIT (Just In Time Compiler). Almost all of the performance benchmarks found online are done on a JVM already running for some time. However,

However, *Beihang University* (http://www.eecg.toronto.edu/~yuan/papers/osdi16-hottub.pdf) has published a benchmark report taking into account JVM warm-up time. They used Hadoop and Spark based systems to process massive data:



Completion time (s)	Unmod.	HotTub	Speed-up
HDFS read 1MB	2.29	0.08	30.08x
HDFS read 10MB	2.65	0.14	18.04x
HDFS read 100MB	2.33	0.41	5.71x
HDFS read 1GB	7.08	4.26	1.66x
Spark 100GB best	65.2	36.2	1.80x
Spark 100GB median	57.8	35.2	1.64x
Spark 100GB worst	74.8	54.4	1.36x
Spark 3TB best	66.4	41.4	1.60x
Spark 3TB median	98.4	73.6	1.34x
Spark 3TB worst	381.2	330.0	1.16x
Hive 100GB best	29.0	16.2	1.79x
Hive 100GB median	38.4	25.0	1.54x
Hive 100GB worst	206.6	188.4	1.10x

(http://www.baeldung.com/wp-content/uploads/2017/06/jvm.png)
Here HotTub designates the environment in which the JVM was warmed up.

As you can see, the speed-up can be significant, especially for relatively small read operations – which is why this data is interesting to consider.

8. Conclusion

In this quick article, we showed how the JVM loads classes when an application starts and how we can warm up the JVM in order gain a performance boost.

This book (https://www.safaribooksonline.com/library/view/java-performance-the/9781449363512/) goes over more information and guidelines on the topic if you want to continue.

And, like always, the full source code is available over on GitHub (https://github.com/eugenp/tutorials/tree/master/libraries/src/main/java/com/baeldung/jmh).

(http://www.baeldung.com/wp-content/uploads/2016/05/baeldung-rest-post-footer-main-1.2.0.jpg)

(http://www.baeldung.com/wp-content/uploads/2016/05/baeldung-rest-post-footer-icn-1.0.0.png)

Learning to "Build your API
with Spring"?

Enter your Email Address

>> Get the eBook

CATEGORIES

SPRING (HTTP://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTP://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTP://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTP://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTP://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTP://WWW.BAELDUNG.COM/CATEGORY/JACKSON/)
HTTPCLIENT (HTTP://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTP://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (HTTP://WWW.BAELDUNG.COM/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (HTTP://WWW.BAELDUNG.COM/JACKSON)
HTTPCLIENT 4 TUTORIAL (HTTP://WWW.BAELDUNG.COM/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (HTTP://WWW.BAELDUNG.COM/REST-WITH-SPRING-SERIES/)
SPRING PERSISTENCE TUTORIAL (HTTP://WWW.BAELDUNG.COM/PERSISTENCE-WITH-SPRING-SERIES/)
SECURITY WITH SPRING (HTTP://WWW.BAELDUNG.COM/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (HTTP://WWW.BAELDUNG.COM/ABOUT/)
THE COURSES (HTTP://COURSES.BAELDUNG.COM)
CONSULTING WORK (HTTP://WWW.BAELDUNG.COM/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
THE FULL ARCHIVE (HTTP://WWW.BAELDUNG.COM/FULL_ARCHIVE)
WRITE FOR BAELDUNG (HTTP://WWW.BAELDUNG.COM/CONTRIBUTION-GUIDELINES)
CONTACT (HTTP://WWW.BAELDUNG.COM/CONTACT)