Java Class Loading In Detail

Felix Becker

Amsterdam.scala

7. März 2017

Contents

Motivation

- Grundlage für Vortrag von Mathias Kub
- ClassLoader-Hölle bei vielen Entwicklern gefürchtet
- Verständnis vom class loading bei komplexen Fehlersituationen notwendig
- Vorbereitung für Folgetalks über Scala Bytecode / JVM Internals + Scala

Task of a class loader?

- Load classes and resources from different sources
- Distinction between data source and application, application only must use the class loader (platform independency)
- Classes are being loaded in the JVM byte code format
- Resources can be any data (e.g. a file from the file system of from a web service call)

Startup life cycle of a Java class (JVM spec §12.1.2)

1. Loading §12.2.1
   - Load the bytecode into the JVM using the class loader
2. Verify §12.3.1
   - Validation of the class structure and data (Opcodes valid, branch instructions check, signature check, ...)
3. Prepare §12.3.2
   - Storage allocation, initialization of static fields (default values, no static initializer)
4. (Resolve §12.3.3)
   - Optional step: symbolic link resolution
5. Initialization §12.4.1
   - Call to static initializers, initialization of static fields

Initialization

Initialization of a class is the first "active" code execution of class code (if static fields / initializers exist). Initialization happens the first time (only once), when:

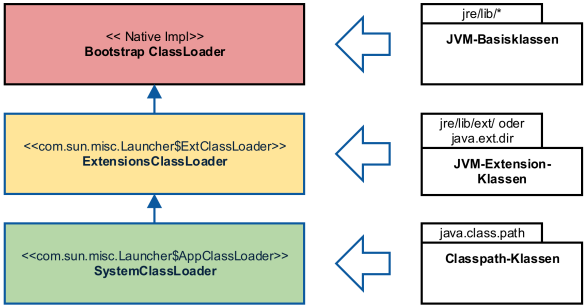
- An instance of the class is being created
- A static method of the class is being invoked
- Static variables are read / written

Initialization of a class forces the initialization of all super classes (does not happen for interface)

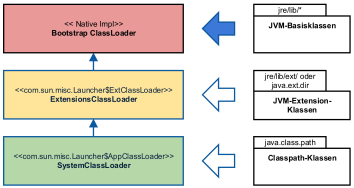
Initialization of a class is thread safe - safe way to create singletons!

The initialization is secured by a jvm internal initialization lock. The jvm guarantees, that the initialization of a class happens only once.

Default class loader JVM 8 (Oracle)

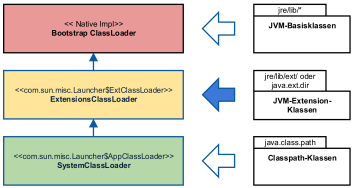


Bootstrap class loader



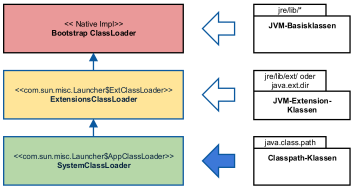
- Loads Java base classes (e.g. java/lang/String)
- Native implementation
- java.lang.ClassLoader: private native Class findBootstrapClass

Extension class loader



- Loads extension classes
- Example: com/sun/nio/zipfs/ZipPath
- Loads classes from jre/lib/ext/ or java.ext.dirs
- com.sun.misc.Launcher$ExtClassLoader

System class loader



- Loads all classes from the java.class.path (java -cp ..)
- java.class.path (com.sun.misc.Launcher$AppClassLoader)

Important functions

- From the applications point of view:
  - Class.forName(String)
  - Class.getClassLoader()
  - ClassLoader.loadClass(String)
- Inside the class loader:
  - findLoadedClass(String)
  - findBootstrapClassOrNull(String)
  - resolveClass
  - defineClass

Live-Demo 1: First steps

ClassLoader.loadClass

```
1  protected Class<?> loadClass(String name, boolean resolve)
2    throws ClassNotFoundException {
3
4    synchronized (getClassLoadingLock(name)) {
5
6      Class<?> c = findLoadedClass(name);
7
8      if (c == null) {
9        try {
10         if (parent != null){
11           c = parent.loadClass(name, false);
12         } else {
13           c = findBootstrapClassOrNull(name);
14         }
15       } catch (ClassNotFoundException e) {}
16
17       if (c == null){  c = findClass(name); }
18     }
19
20     if (resolve){ resolveClass(c); }
21
22     return c;
23   }
24 }
```

- Class loading ist synchronized in loadClass, too
- lookup steps:
  1. findLoadedClass (native)
  2. parent / bootstrap lookup
  3. findClass
- findClass is the function you have to override in your own class loader implementation

ClassLoader.resolveClass

```
1  protected Class<?> loadClass(String name, boolean resolve)
```

Loads the class and binds it with resolveClass() - if resolve is true. - (Java ist auch eine Insel)

The variable resolve is a flag to tell the class loader that classes referenced by this class name should be resolved (that is, any referenced class should be loaded as well). - JavaWorld (The basics of Java class loaders)

As I mentioned previously, loading a class can be done partially (without resolution) or completely (with resolution). When we write our version of loadClass, we may need to call resolveClass, depending on the value of the resolve parameter to loadClass. - IBM Developer Works (Understanding the Java ClassLoader)

Live-Demo 2: resolveClass

ClassLoader.resolveClass

```
1  protected Class<?> loadClass(String name, boolean resolve)
```

Loads the class and binds it with resolveClass() - if resolve is true. - (Java ist auch eine Insel)

The variable resolve is a flag to tell the class loader that classes referenced by this class name should be resolved (that is, any referenced class should be loaded as well). - JavaWorld (The basics of Java class loaders)

As I mentioned previously, loading a class can be done partially (without resolution) or completely (with resolution). When we write our version of loadClass, we may need to call resolveClass, depending on the value of the resolve parameter to loadClass. - IBM Developer Works (Understanding the Java ClassLoader)

Live-Demo 2: resolveClass

```
1  // http://hg.openjdk.java.net/jdk8/jdk8/hotspot/file/tip/src/share/vm/prims/jvm.cpp
2
3  // 732 - 735
4  JVM_ENTRY(void, JVM_ResolveClass(JNIEnv* env, jclass cls))
5    JVMWrapper("JVM_ResolveClass");
6    if (PrintJVMWarnings) warning("JVM_ResolveClass not implemented");
7  JVM_END
```

ClassLoader.getResource

```
1  public URL getResource(String name) {
2
3    URL url;
4
5    if (parent != null) {
6      url = parent.getResource(name);
7    } else {
8      url = getBootstrapResource(name);
9    }
10
11   if (url == null) {
12     url = findResource(name);
13   }
14
15   return url;
16 }
```

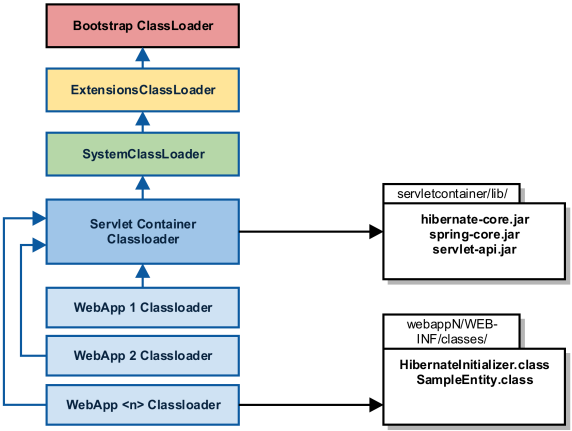
- Lookup the resource at the parent / boot strap classloader first
- findResource is being invoked when the parent / boot strap doesn't provide the resource
- findResource is the function that you have to override in your own class loader implementation

URL-ClassLoader

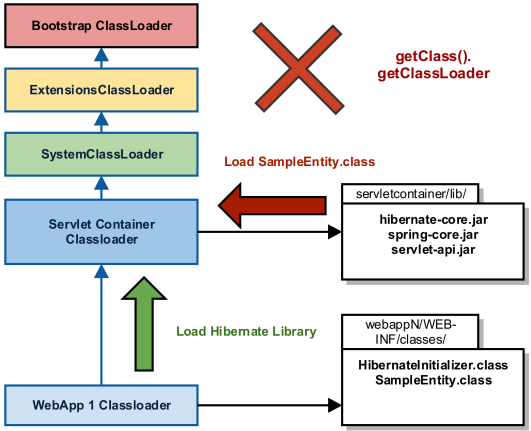
```
1  // Removed SecurityManager Stuff
2  protected Class<?> findClass(final String name) throws ClassNotFoundException {
3    String path = name.replace('.', '/').concat(".class");
4    Resource res = ucp.getResource(path, false);
5    if (res != null) {
6      try {
7        return defineClass(name, res);
8      } catch (IOException e) {
9        throw new ClassNotFoundException(name, e);
10     }
11   } else {
12     throw new ClassNotFoundException(name);
13   }
14 }
```

Live-Demo 3: URL-ClassLoader

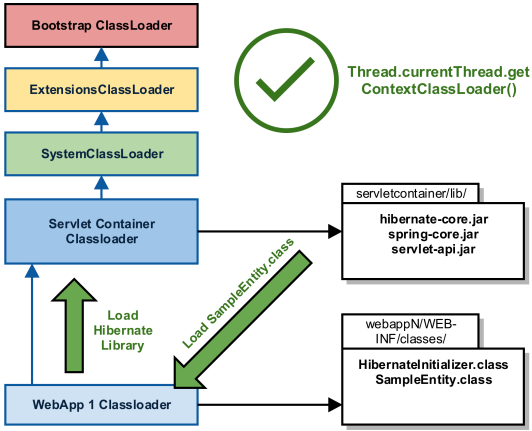
WebApp class loader



WebApp class loader - context class loader



WebApp class loader - context class loader



Live-Demo 4: Loading the same class twice with different class loaders

Synthetic classes

- Synthentic classes are "artificial" classes that doesn't result from your java source code
- Are created by the compiler or at runtime
- Heavy usage in the AOP world - e.g. transaktions proxy classes
- Usage in test frameworks like PowerMock
- Can be created in the class loader by definining native JVM byte code

Live-Demo 5: Creation of synthetic classes

End

Thank you for your attention
Slides and sources at github.com/fbe/classloader-vortrag