

Bionet Class Loader

Jun Suzuki, Kevin Nguyen, Mike Le and Tatsuya Suda

{jsuzuki, suda}@ics.uci.edu

{nguyencp, mvle}@uci.edu

<http://netresearch.ics.uci.edu/bionet/>

School of Information and Computer Science
University of California, Irvine

1

Overview

- General overview of Java class loading
- Class loading in the bionet platform
 - Dynamic access to the system class loader
 - A custom class loader in the bionet platform
 - A custom object input stream

General Overview of Java Class Loading

Class Class in Java

- Instances of class `java.lang.Class` represent classes and interfaces in a running Java application.
- `Class` has no public constructors
 - Any applications cannot instantiate it.
 - `Class` is automatically instantiated by JVM/class loader, when a class loading process is done.
- Each instance of `Class` keeps a reference to a class loader that created it.

Class Loading

When you define a class, say `Foo`, and instantiate it (to create its instance),

- i.e. when you execute `new Foo()`;

A local JVM (class loader) tries to

- obtain the class definition of `Foo`.
 - Given the name of a class (i.e. `Foo`), the class loader locates a binary representation of `Foo`'s class definition.
 - Typically, it transforms the class name into a file name, and reads a class file of that name (e.g. `Foo.class`) from the local file system.
 - If this step fails, JVM returns a `ClassNotFoundException`.
- parse the obtained class definition into internal structures in the method area of the JVM.
 - The method area is a logical area of memory in the JVM that is used to store information about loaded classes/interfaces.
- create an instance of `Class`, which is associated with `Foo`.

These 3 activities are collectively called *class loading*.

After successful class loading, an instance of `Foo` is created and returned.

5

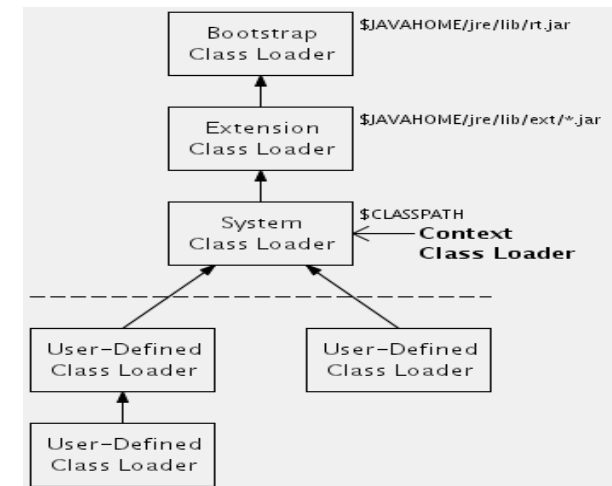
Class Loaders in JVM

- Every JVM has two types of class loaders:
 - Default class loaders
 - Bootstrap class loader
 - used to find and load a set of standard class libraries (i.e. `java.*` classes) that are located in `$JAVA_HOME/jre/lib`
 - » e.g. `$JAVA_HOME/jre/lib/rt.jar`
 - Extension class loader
 - used to find and load a set of extension class libraries (i.e. `javax.*` classes) that are located in `$JAVA_HOME/jre/lib/ext`
 - System class loader (`java.lang.ClassLoader`)
 - used to find and load user-defined classes.
 - Custom class loaders
 - If a user wants to change the behavior of system class loader, the user can extend `java.lang.ClassLoader` as a Java class. The class is called a custom class loader.
 - A custom class loader overloads the methods defined in `java.lang.ClassLoader` to implement its own class loading behavior.
 - Different custom class loaders can specialize different class loading behaviors (policies)

JVM automatically (implicitly) creates the bootstrap, extension, and system class loaders when it starts running.

Every custom class loaders should be explicitly instantiated by applications (developers).

Class Loader Hierarchy



JVM System Class Loader

The system class loader

- transforms a given class name (e.g. `Foo`) into a file name (e.g. `Foo.class`), and
- obtains the class file from
 - directories specified in the environment variable `CLASSPATH`, or
 - JAR files specified in the environment variable `CLASSPATH`.

If a class file (i.e. class definition) is not found in the above places, the system class loader raises a `ClassNotFoundException` exception.

9

Bootstrap and Extension Class Loaders

- The bootstrap and extension class loaders follow similar class loading procedures to the procedure in the system class loader;
 - however, they look up different directories to load different types of classes.
- Bootstrap class loader
 - looks up the directory “lib” under the place where SDK/JRE is installed (`$JAVAHOME/jre/lib`), and
 - loads standard Java API classes that are contained in JAR files.
- Extension class loader
 - looks up the directory “ext” under the place where SDK/JRE is installed (`$JAVAHOME/jre/lib/ext`), and
 - loads the classes contained in JAR files.

Custom Class Loaders

Applications (or users) can define and use their own (i.e. non-default) class loaders,

- each of which has its own policies on
 - where to find to locate class definitions
 - e.g. the place pointed by a URL
 - how to obtain class definitions.
 - e.g. through an HTTP connection

Every custom class loaders extends the system class loader (`java.lang.Classloader`).

Custom class loaders need to be called explicitly by applications (or users)

- `ClassLoader loader = new SampleCustomClassLoader (...);`
`Class class = loader.loadClass(..., true);`

When a class is instantiated regularly, the system class loader is used to load the class.

For instance, `new Foo()`

An Example Custom Class Loader

• An custom class loader

```
– class MyCustomClassLoader
  extends java.lang.ClassLoader{
    .....
    public Class findClass(String name){
        byte[] b = fetchClassDefFromRemoteNode(name);
        return defineClass(name, b, 0, b.length);
    }
    private byte[] fetchClassDefFromRemoteNode(String name) {
        // access a remote node to obtain a binary presentation (byte[])
        // of class definition
    }
  }
```

• Use case

```
– ClassLoader loader = new MyCustomClassLoader (...);
  Object foo = loader.loadClass("Foo", true).newInstance();
```

Default Control Flow in loadClass()

Custom class loaders need to override `findClass()` defined in `java.lang.ClassLoader` so that they implement their own policy to obtain class definitions.

- The responsibility of this method is to
 - accept the name of a class and
 - return an instance of `Class`, which is associated with the class being loaded.

Custom class loaders do not have to override `loadClass()`.

- `java.lang.ClassLoader` has already implemented a default steps (control flow) of class loading.

13

- By default, `loadClass()` calls the following methods in this order
 - `findLoadedClass()`
 - Find out if the class being loaded has already been loaded by this class loader. If it has, then just return the class reference.
 - `getParent().loadClass()`
 - Delegate the parent class loader to load the class.
 - It is not necessary to carry out this delegation to the parent class loader; however, it is less work for this class loader if it's parent can load the class.
 - `findClass()`
 - When the parent class loader cannot load the class, this method is used to find the class definition of the class being loaded.
 - It depends on custom class loaders where to look up for class definitions.
 - Every custom class loader needs to override this method to implement this look up policy.
 - `defineClass()`
 - converts an array of bytes into an instance of class `Class`
 - `resolveClass()`
 - links the class being loaded.

Loading and Linking

Loading

- the process to find the class definition (in the form of byte code (binary code)) of a particular class or interface type.

Linking

- the process to take the found class definition (byte code), verify the byte code, create static fields, and resolve the symbolic references.
- It's the step that comes after loading the class.
- The `resolveClass()` method take care of this process.
 - `java.lang.ClassLoader` provides a default linking

3 Major Steps in Class Linking

- **Verification:**
 - ensures that the binary representation of a class or interface is structurally correct
 - e.g. checks that every instruction has a valid operation code; every function has valid signatures, etc.
- **Preparation:**
 - involves creating static fields for a class or interface, and
 - initializes the static fields to the default values.
- **Resolution:**
 - checks that a symbolic reference is correct, and
 - may replace the reference with a direct reference, which can be more efficiently processed if the reference is used repeatedly.
- **Exceptions raised potentially:**
 - `VerifyError`: class does not obey semantics of java language
 - `AbstractMethodError`: not an abstract class, but has abstract method
 - `OutOfMemoryError`: not enough memory

Class Loading in the Bionet Platform

17

Class Loading in the Bionet Platform

- The bionet platform uses both the system and custom class loaders to load CEs.
 - The system class loader is used to load a newly created CEs.
 - CEs created by developers manually
 - CEs replicated by other CEs
 - CEs reproduced by other CEs
 - The bionet custom class loader is used to load CEs that migrate from other platforms.
 - The bionet platform defines its own class loader that extends the system class loader.
 - Each platform keeps a custom class loader instance at run time.

Dynamic Access to the System Class Loader in the Bionet Platform

The CE factory dynamically access the system class loader to insert (or install) a newly created CE into the bionet environment.

- In the bionet platform, developers create their own CEs (i.e. define their own classes) by extending `CyberEntityImpl`.
 - e.g. class `MyCE` extending `CyberEntityImpl`
- The CE factory accepts the name of a CE (e.g. “MyCE”), load its class definition into JVM, and registers it into the bionet container.
 - `java.lang.ClassLoader ldr = this.getClass().getClassLoader();`
 - `Class ceClass = ldr.loadClass("MyCE");`
 - `CyberEntityImpl ce = (CyberEntityImpl) ceClass.newInstance();`
- See the documentation about CE factory for more details.
 - `edu.uci.ics.bionet.ce.factory.ce.CEFactory`
 - `edu.uci.ics.bionet.ce.factory.ce.CEFactoryImpl`

Motivation to a Custom Class Loader in the Bionet Platform

- In the bionet platform, developers create their own CEs (i.e. defines their own classes) by extending `CyberEntityImpl`.
 - e.g. class `MyCE` extending `CyberEntityImpl`
- When a CE (e.g. an instance of `MyCE`) migrates from a platform to another, a bionet migration service at a destination platform may not have `MyCE`'s class definition.
 - A bionet migration service at a source platform transmits `MyCE`'s class definition during the migration process.
 - A bionet migration service at a destination receives `MyCE`'s class definition and dynamically load the class definition into JVM (i.e. creates a `Class` instance associated with `MyCE`) using a custom class loader.

Bionet Class Loader

A custom class loader defined in the bionet platform is called *bionet class loader*.

- `BionetClassLoader` extending the system class loader (`java.lang.ClassLoader`).

The bionet class loader

- defines `setUpClassDefinition()`
- overrides `findClass()` of the system class loader.

21

- `setUpClassDefinition(byte[] class)`
 - saves (memorizes) the class definition of a class being loaded by the bionet class loader
- `findClass(String className)`
 - returns the class definition of the class being loaded
 - calls `defineClass()` and `resolveClass()`, which are defined by the system class loader.

A use case of the bionet class loader

- `ClassLoader ldr = new BionetClassLoader (...);`
`Class class = ldr.loadClass(..., true);`

Object Input Stream

- After a class loading is completed, a bionet migration service deserializes the state of a CE that migrates from another platform.
- The `ObjectInputStream` class is used for the de-serialization process.
 - When de-serializing an object (CE), it tries to associate the object (CE) with its corresponding `Class` object.
 - through its `resolveClass()` method
 - By default, `resolveClass()` accesses the system class loader to do that.
 - However, the system class loader does not know (has not loaded) the class definition of the object (CE) being de-serialized, because it was loaded by a custom class loader (bionet class loader).
 - Bionet platform provides a custom object input stream in order to revolve this issue.

A Custom Object Input Stream

A custom object input stream is defined for deserializing CEs successfully.

- `BionetObjectInputStream` extending `ObjectInputStream`

It overrides `resolveClass()` and calls the bionet class loader, instead of the system class loader, to associate a CE being de-serialized with its `Class` object.

- The overridden method returns a `Class` object using `forName(className, true, BionetClassLoader)`.
 - `forName()` is a static method of `Class`.
 - returns the `Class` object associated with a class using its string name and class loader.