Code Loading & Linking

Oleg Šelajev (shelajev@gmail.com)

Abstract

Code linking is a process of mapping identifiers used in the code to an actual implementations for execution. Different kinds and types of linking feature speed of execution, variability trade-offs and greatly influence possibilities of code reloading in the platform. Code loading is the process of referencing, locating and making the virtual machine level code accessible to the caller which is already loaded into the same virtual machine. Code loading mechanisms heavily depend on the representation of units of compilation that machine is using. Linking and loading techniques define code reloading capabilities and influence the existence of an industrial code reloading solutions for the given virtual machine.
Why the problem is important

Code binding, loading and linking are essential concepts for building multi-module applications, libraries and in fact any non-trivial program calls external libraries. Even if the call is to functions defined at the core of the language like to Java Class Library, which forms an essential part of the language, it should be considered an external call and require a compiler or a runtime to make a mapping between an identifier and an exact set of instructions that are be referenced by that identifier. We consider every identifier to be global, so any unit of compilation is able to call another unit of compilation by its identifier. The exact mechanism of distinguishing unit's identifier from all others like: local variables, local function names, etc is not important and is omitted from the discussion.

Once a virtual machine, VM, locates the exact instructions that represent an unit of compilation to load it begins the loading process. The usual activities done during that phase is: verification of the code, verification of the types and making sure the code is safe (in the VM sense), loading the code into the memory and making that unit of compilation available to already existing code. After that the code must be initialized and made ready to use by the caller.

What is specific to this paper and what are expected results

In this survey paper we describe the principles and implementations of several real-life choices about code binding, loading and reloading in several virtual machine platforms. When the solutions are dissected and investigated together, different trade-offs become clear and distinguishing features of every approach are seen clearer.

Theoretical introduction, terms and principles, trade-offs

For the better understanding of the code loading process we need to get back to the roots of the modern programming languages and describe how a program executes in a physical machine. A computer processor has a set of instructions that are implemented in hardware and can be executed sequentially by this hardware. So essentially any program is a specification which instruction should be executed and in which order. A program specified as a sequence of such instructions is referred as a machine code. As the hardware is the only physical entity involved into the computation, it is the only entity which is capable of executing a program. In spite of this any program, written in any representation must be converted to a machine code prior the execution. It is perfectly valid to create and write programs in the machine code directly, however this approach tends to be very cumbersome, as modern programs consist of thousands of machine code instructions. To solve this problem a compiler is used. A compiler is a tool for translation process from a higher, more abstract programming language into a lower level set of instructions, which is called compilation.

Compilation
The existence of compilers allows us to specify programs using high-level programming languages instead of writing millions of machine code instructions by hand. A compiler takes a written program as an input and produces another program, which behaves exactly as the original one, but is written using other set of instructions or another programming language.

Def 1. Compilation – a process of translating a program from one set of instructions into another set of instructions (possibly lower level), such that semantics of the original program is preserved.

Following table shows sequence of stages that compilation consists of and artifacts that compiling accepts and produces.

<table>
<thead>
<tr>
<th>Source program</th>
<th>↓</th>
<th>Source program</th>
<th>↓</th>
<th>Assembly program</th>
<th>↓</th>
<th>Machine instructions</th>
<th>↓</th>
<th>Executable code</th>
</tr>
</thead>
<tbody>
<tr>
<td>preprocessing</td>
<td></td>
<td>compiling</td>
<td></td>
<td>assembling</td>
<td></td>
<td>linking</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1. List of compilation process stages with their input and output descriptions.

Now, there’s a stage there which is called compiling; to avoid ambiguity in this paper term compilation means the whole process of translation into other language, that consists of all 4 stages above unless specified otherwise. The result of a compilation is an executable code, which might be executed by OS or an underlying VM.

Def 2. Unit of compilation is a minimal source program of a certain structure that compiler accepts as a valid input.

Unit of compilation definition uses type of a source program, which is an intuitively straightforward term, which we can explain by an example. Consider Java programming language, to compile file a.java into JVM bytecode, a.java should represent one or more valid classes. For erlang, a unit of compilation is a module, for C – a .c file with int main() function.

A compiler takes one or more units of code and translates them into a target language. Modern languages often require more than 1 compilation process. If a target platform is a virtual machine, first compilation is done into a VM-specific language, for example JVM bytecode, and then another compiler translates the bytecode into a machine executable instructions. In this case we can speak of multi-tiered compilation.
Def 3. A process of several consecutive compilations, such that following compilation takes the result of previous as an input, which might be followed by an execution of the result of the last compilation is multi-tiered compilation.

\[
\text{Compilation} \Longrightarrow \text{Compilation} \Longrightarrow \ldots \ [\Longrightarrow \text{Execution}]
\]

Linking

The first three stages of a compilation are fairly straightforward to understand and involve only manipulations with the original program's source code or artifacts of the current compilation process. It involves lexical analysis, construction of abstract syntax tree of programming language tokens, transforming this tree according to optimization techniques and producing an equivalent machine code afterwards. However, linking is different in a way that its inputs are specified not only by the current compilation process artifacts, but also by the external environment.

Linking allows the program to use other programs or parts of other programs already existing in the external environment during current compilation process. In a way, linking is referencing sequences of instructions outside of the current compilation.

Def 4. Linking – is a process of creating an executable by an operating system file by combining one or more compiled machine code files.

Linking phase is utterly important, it allows to create programs that consist of more than a single unit of compilation and link or refer to some target artifacts from other ones. During multi-tiered compilation linking may occur multiple times, At each compilation step compiler may choose to verify that provided input is a valid source program which does not refer to missing components which will not be available at the runtime. In general any invocation or access of the code that lies in another unit of compilation must be linked for the final executable. However, if we have a virtual machine to execute the code, we may defer the linking till the last compilation process which must output the machine code and treat every output of the compiled units of code as different source program.

JVM

Java Virtual Machine is an abstract computing machine which can operate on it’s specific instruction set and access hardware memory on runtime. It is a virtual machine essential for the implementation of the Java programming language created by Sun Microsystems in 1995. However it does not depend on the Java language and operates instead on class files. Class files are specifically formatted sequence of bytes, not necessary residing on the filesystem. One way to obtain valid class files is to compile Java source program with a javac program.

JVM can load bytecode instructions from class files and execute them in a platform-specific way to ensure the same semantics of the loaded class files on any hardware and operating system. To do so JVM employs a Java Class Library, which implements several important for the virtual machine concepts and encapsulates all necessary touching points with the underlying operating system, like: threading and scheduling, time API, memory-management, filesystem access, etc.
The following chapters describe how JVM loads, links and initializes classes to use them in the existing process.

Class file format

Class file format is a binary format and is independent from hardware or operating system. It contains instructions for the JVM to compile and run and can be thought of as a source program for JVM. This allows to use existing JVM as a platform and create various programming languages that compile to JVM class files. Class file format describes a class or interface and how to initialize them when loading into JVM.

JVM loads and initializes classes on-demand from a sequence of bytes that correspond to the class file format. The whole process can be divided into three stages. Loading, when the JVM locates and reads the bytes that represent a class with the given name. Loading result is that the class is created in the JVM representation. Linking, when JVM incorporates this class representation with the existing JVM state to be able to initialize the class further. Initialization means that class specific init operation is run, which can be specified by the class itself. After that class is available in the JVM and other classes can access it.

Loading

JVM features a specific mechanism for locating and loading classes, which is based on a hierarchy of classloader object. The Java Class Library classes at the start of virtual machine is done by the bootstrap classloader, then the main class is located and public static void main(String[] args) method is executed.

After that any class can create a custom classloader which will be responsible for the loading of classes. Classloaders are put in tree-like hierarchy where every classloader has a parent, except for the bootstrap classloader whose parent is null.

For an example of classloader hierarchy see appendix A: Glassfish classloaders hierarchy table, it will give some insight which things an application might want to load and why it is a good thing to load them in different classloaders with different logic.

A general consensus is that it is preferred to ask parent classloader to find a class before and only if that fails to locate a class yourself. This allows to follow a hierarchy where parent classloader loads all classes available to it before turning to the own custom logic. However, if the current classloader or any of its parents can find bytes that represent the class by the given name, it will be located.

The next step is linking, which involves verification and preparation of the class, its superclasses or superinterfaces and resolve any symbolic references in the class.

JVM specification does not dictate the exact process to load and link classes, for example when a recursive loading is necessary. But one general rule is that class must be fully loaded before it is linked and is linked before the initialization.

During verification JVM ensures that the binary representation of the class is correct, in other words it satisfies structural and static constraints as number of methods within the class is under limit, field number, arrays are not longer than can be and so forth. For the full list of constraints one
can refer to JVM specification [1]. Verification might force JVM to load additional classes, however none of them must be linked or initialized.
Preparation involves initializing static or class fields to their respective default values. For example long's default value is 0L, boolean is false, any object type get the default value of null.
Resolution of symbolic references is also quite a straightforward process, JVM bytecode instruction set has a number of instructions that refer to other classes, their methods and fields using runtime constant pool of the current class. Runtime constant pool is like a symbol's table in other languages. Basically, the class operates on the indexes of the symbols in the runtime constant pool, and not on the identifiers thus greatly reducing the size of the class file. In the resolution references phase, verifier can go through the whole table, and initialize loading of classes that are referenced there. Additionally, JVM verifies and resolves method handles, fields, method types, etc.
Initialization of the class involves running static class initializers, which means running custom JVM code, so the class must be almost ready before this. Moreover initialization phase features binding native method implementations to method handles.
After initialization is done, the class is ready to be used in the JVM by any number of threads and the consequent initialization of this class in this classloader must finish trivially, as the class is already defined. However, JVM states that classes are equal when their names are equal and defining classloaders of the classes are the same.

Reloading capabilities
JVM does not states any mechanisms for reloading already defined classes. In fact there is no way to unload a given class from the JVM. However there is one feature that allows JVM to mutate the behavior of the loaded classes.
Method bodies can be swapped on runtime. HotSwap JVM allows to do this in the debug session, however there is no theoretical limitation why this cannot be done during a normal run.
The reason is that every method invocation creates additional stack frame with all the local variables and dynamically links the code that needs to be executed in this method. The current method stack frame is dropped and restarted from the beginning discarding any previous state, making changing method bodies trivial. One can just change bytecode operations that are referenced from the runtime constant pool of the class containing the current method. Loading new method body might trigger resolving other undefined symbols. However it should be done in the usual way of loading classes, so it doesn’t pose any significant interest.
Additionally, there are many other solutions to tackle this problem, both in commercial sector general purpose solutions like Jrebel/LiveRebel for development and production updates and open-source: frameworks featuring reloading of the classes, like Play 2; modified JVM-s like DCEVM and so long.
The conclusion is that JVM is a mature platform, featuring JVM bytecode as source of the program that can be executed. It offers the program various low-level APIs and specifications and ensures that the program does not semantically depend on the operating system or hardware.
The mechanism of loading class files, which are units of code for the JVM, is precisely defined within JVM specification and consists of locating, verifying, linking and initializing the class before program can use it. Reloading capabilities are not defined, JVM specification does not say anything about unloading classes from the memory, however the details of method invocation allows quite straightforward way to change method bodies without changing their signatures.

CLR

All .Net compatible languages, VB.Net, C#, F# and so long, are compiled into an intermediate platform-neutral Common Intermediate Language, CIL, which plays the same role as JVM bytecode. Common Language Runtime, which is a virtual machine component of Microsoft's .Net Framework, can load CIL, compile that into an executable machine code for a specific platform and execute that. CIL is stack-based object-oriented language very similar to JVM bytecode, however there are a couple of differences worth mentioning.

CIL bytecode has polymorphic instructions, for example in JVM bytecode, there are `iadd` and `fadd` instructions to add two ints and two float values respectively, CIL features one `add` instruction that can work on any data it is offered. This reduces code size, makes code generation to recognize types and makes just-in-time compiler, JIT, a bit more complicated. CIL supports structs, which are value types just like primitives as opposed to reference types like objects, closures and coroutines, which are said to come in the upcoming Java 8 release. Also CIL features pointer manipulation. But the last but not least difference is CIL’s assemblies.

Assembling and Loading

Assembly is a compiled code packaged together as an `.exe` or `.dll` package for better versioning, security and the general reasoning about the application classes. An assembly contains one or more modules, which is a unit of compilation for CLR. During the runtime an assembly is getting loaded into `System.AppDomain` module, which will later be accessed to resolve an assembly itself or any types included in the given assembly.

After the assembly is loaded into memory, it can initialize itself using custom code from `Module Initializers`, which are quite like `static initializers` in JVM. However there is no module initializer for every class and the whole bundle is initialized at once. The initialization behavior is consistent, and if you load the assembly to run its main method, it will initialize the module anyway.

Reloading capabilities

Reloading code in the running application clearly was not a goal of CLR designers, very little information is published on the matter. Some basic reloading functionality can be achieved with `hot-swapping` the assembly code in the running virtual machine by setting “Allow hot swapping of assemblies when the server is running” property.
When it is enabled VM will load and initialize new application domain with a fresh copy of assembly modules on every clients managed code call. This might be a satisfying solution in the development environment, when only a single client uses the application. However in the production setting this approach clearly lacks scalability and does not seem as appropriate solution, because initialization of the whole assembly for each client is wasteful in general case. To illustrate this idea, consider a figure in the appendix B, which illustrates the VM with hot-swapping enabled and 3 clients.

The result of our findings is unsatisfactory, the problem of reloading code in the running instance of CLR is not clear enough. There are some basic capabilities of full reinitialization of application bundles, however it is not applicable in the production setting.

**Erlang (BEAM)**

Erlang is a mature functional concurrent programming language and a runtime system, which appeared first in 1986 as a proprietary software and later open-sourced in 1998. The core design values of Erlang include fault-tolerance, hot-swapping code and ability to create real-time applications.

BEAM is the name of virtual machine that executes erlang programs. A unit of compilation for Erlang is a module. A module consists of definitions of attributes and functions declarations. Some attributes are module name, includes, compilation options and which function to run on module load event. Others are custom and include custom program logic. A typical Erlang program consists of several modules, standard Erlang modules, libraries and application ones.

**Compilation**

Module in itself is just a text file. Compilation of a module includes quite usual sequence of steps: lexical processing, preprocessing, parsing into Erlang terms, optimizations and output code generation. Lexical processing is made easy by the module format. As a module is just a sequence of attributes and functions (every one of which ends with a period - '.'), it is very straightforward to parse it into sequence of tokens.

Preprocessing makes use of conditional compilation and expands macros and serves a very similar function as C preprocessor. The output of a preprocessor is a parse trees forest, which is later optimized and used to generate Erlang bytecode. The bytecode can be stored in a file for later use or fed to Erlang VM directly.

**Loading and Reloading**

As Erlang has been designed with code reloading capabilities in mind there is no reason to talk about code loading and reloading separately. Lets fix some terms before we describe the solution that Erlang uses for this. At any moment of time two versions of the same module can be loaded
into the Erlang VM, the old version and the new version. The idea is that the function that is evaluated in the old module should be able to finish evaluation using the old module code, and at the same time all the new function calls should hit the new version. Given the fact that old version exists only to finish existing calls and should not be involved into any computation that starts after the new version becomes available, Erlang poses a requirement that only one instance of old version is available at any moment of time. If the same module is reloaded once more, the first version of the code is dropped and all processes running that version of the code is killed.

When a load_module/2 function is called on a module which is already available in the system the current version of module loaded is marked as a old version, the code is loaded and a new version of the module is created.

Now there is two distinct way how one module can access another module (and itself too): global access and local access. The type of access is determined by the syntax used for a call. A global call explicitly specifies the module name:

\[
\text{ModuleName:FunctionName(Arguments)}
\]

Also the same can be achieved if the ModuleName:FunctionName is assigned to a variable which is called later. Now a local access would be just accessing a function without specifying the module: FunctionName(Arguments). This ability to run old version code creates some caveats for the Erlang programmer. For instance if there is a long-running method call, possibly recursive, which uses local access, and the module is reloaded twice, the process will be, as mentioned previously, killed. The workaround would be to be aware of such possibility and use explicit module name syntax, which ensures running the newest version available.

Erlangs functional paradigm with message passing as a communication medium between threads creates the heaven for code reloading, as the number of state that must be taken care of is minimal. As the module does not store any state, when reload happen the module is ready to fulfill its functions without modifying data structures or any state transformations. Despite being seemingly inefficient solution this model serves very well. As function calls are mostly pure, without side-effects, parallelization of them is very pleasant and comparatively easy problem. Which makes Erlang quite efficient, though several times slower than JVM programs [5].

To conclude, Erlang is a fairly efficient virtual machine, which emphasizes fault-tolerance and allows programmers to substitute code in a running system with easy. There are some minor issues that must be taken into account which may prevent system from running without dropping any execution processes, however Erlang was designed to reload code from the start, so it is much easier that with virtual machines that didn’t have that as a requirement. Erlang achieves it with minimizing application state that must be transformed between versions, functional nature of the language and using message passing instead of shared memory helps a lot.
Conclusion

We have discussed three implementations of virtual machines: JVM, CLR and Erlang's BEAM with the main interest about mechanisms they offer for loading code into VM, and reloading the running code with the new version of classes/modules. It is clear that if runtime code reloading is not a priority at the time VM is designed and created, then it is not easy to add the reloading to the implementations afterwards. JVM and CLR have a very limited functionality for code reloading, while Erlang's system has well-thought implementation of swapping whole modules on-the-fly.

However, there are practical limitations and caveats that should be considered even with Erlang's solution. Although despite the fact that the reloading of code is hard, there are production systems that use official Erlang's approach, there are systems that use workarounds and gimmicks for reloading JVM code with both commercial and open-source tools.

One way to continue this survey is to compare performance of the virtual machines that are discussed above both with code reloading functionality turned on and off to measure performance impact.
References


### Appendix A: Oracle Glassfish 3.1 Servers classloaders hierarchy


<table>
<thead>
<tr>
<th>Class Loader</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bootstrap</td>
<td>The Bootstrap class loader loads the basic runtime classes provided by the JVM software.</td>
</tr>
<tr>
<td>Extension</td>
<td>The Extension class loader loads classes from JAR files present in the system extensions directory, <code>domain-dir/lib/ext</code>. It is parent to the Public API class loader.</td>
</tr>
<tr>
<td>Public API</td>
<td>The Public API class loader makes available all classes specifically exported by the GlassFish Server runtime for use by deployed applications. This includes, but is not limited to, Java EE APIs and other Oracle APIs. It is parent to the Common class loader.</td>
</tr>
<tr>
<td>Common</td>
<td>The Common class loader loads JAR files in the <code>as-install/lib</code> directory, then classes in the <code>domain-dir/lib/classes</code> directory, followed by JAR files in the <code>domain-dir/lib</code> directory. Using <code>domain-dir/lib/classes</code> or <code>domain-dir/lib</code> is recommended whenever possible, and required for custom login modules and realms. It is parent to the Connector class loader.</td>
</tr>
<tr>
<td>Connector</td>
<td>The Connector class loader is a single class loader instance that loads individually deployed connector modules, which are shared across all applications. It is parent to the Applib class loader and the LifeCycleModule class loader.</td>
</tr>
<tr>
<td>LifeCycleModule</td>
<td>The LifeCycleModule class loader is created once per lifecycle module. Each lifecycle module's classpath is used to construct its own class loader. For more information on lifecycle modules.</td>
</tr>
<tr>
<td>Applib</td>
<td>The Applib class loader loads the library classes, specified during deployment, for a specific enabled module or Java EE application; One instance of this class loader is present in each class loader universe; It is parent to the Archive class loader. When multiple deployed applications use the same library, they share the same instance of the library. One library cannot reference classes from another library.</td>
</tr>
<tr>
<td>Archive</td>
<td>The Archive class loader loads classes from the WAR, EAR, and JAR files or directories (for directory deployment) of applications or modules deployed to the GlassFish Server. This class loader also loads any application-specific classes generated by the GlassFish Server runtime, such as stub classes or servlets generated by JSP pages.</td>
</tr>
</tbody>
</table>

### Appendix B: An illustration of .Net VM with hot-swapping enabled and 3 clients.
