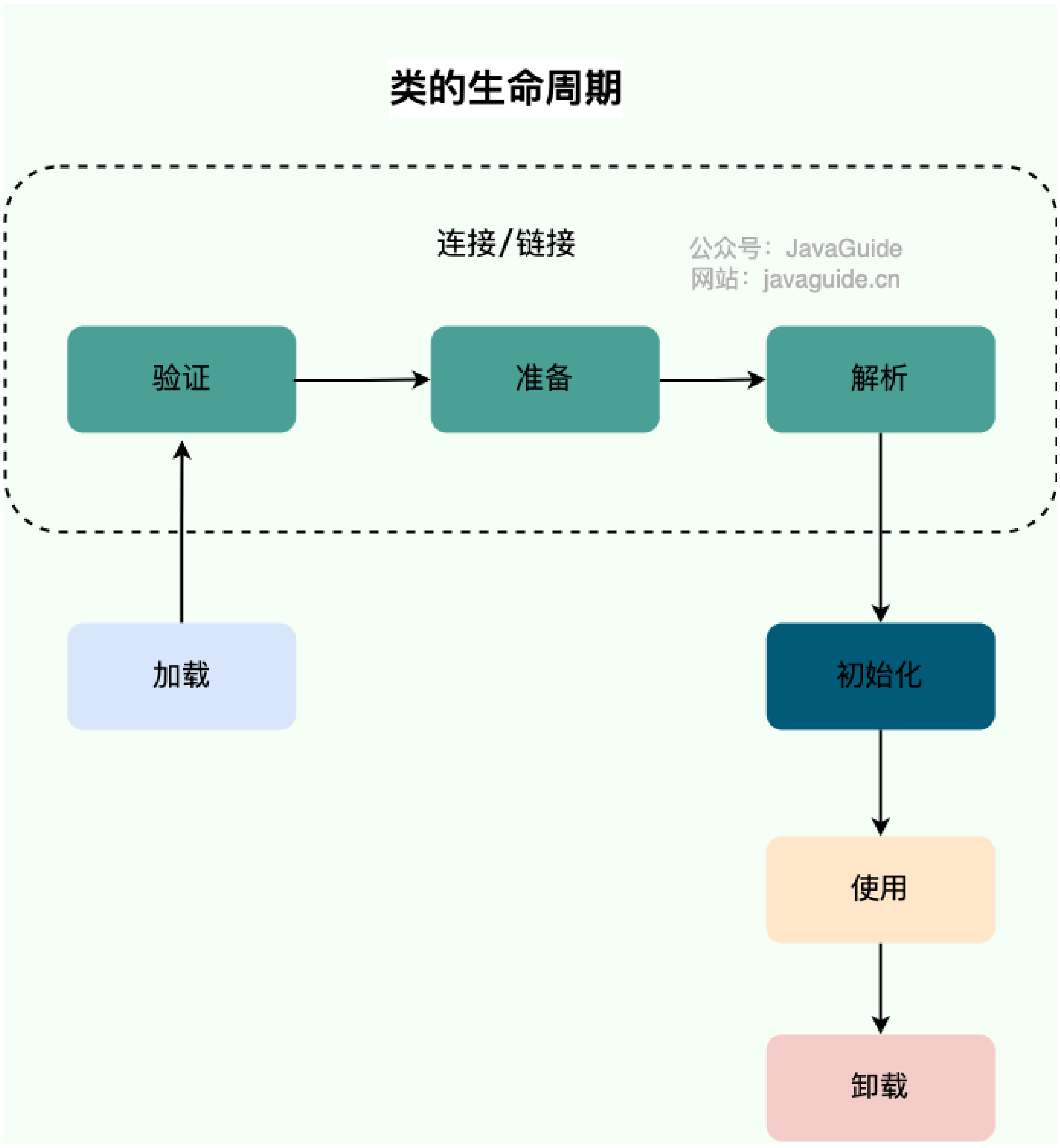
Detailed explanation of the class loading process

Guide Java JVM About 3188 words About 11 minutes

Class lifecycle

The entire lifecycle of a class, from the time it's loaded into the virtual machine's memory to the time it's unloaded, can be summarized into seven phases: loading, verification, preparation, resolution, initialization, using, and unloading. The verification, preparation, and resolution phases are collectively referred to as linking.

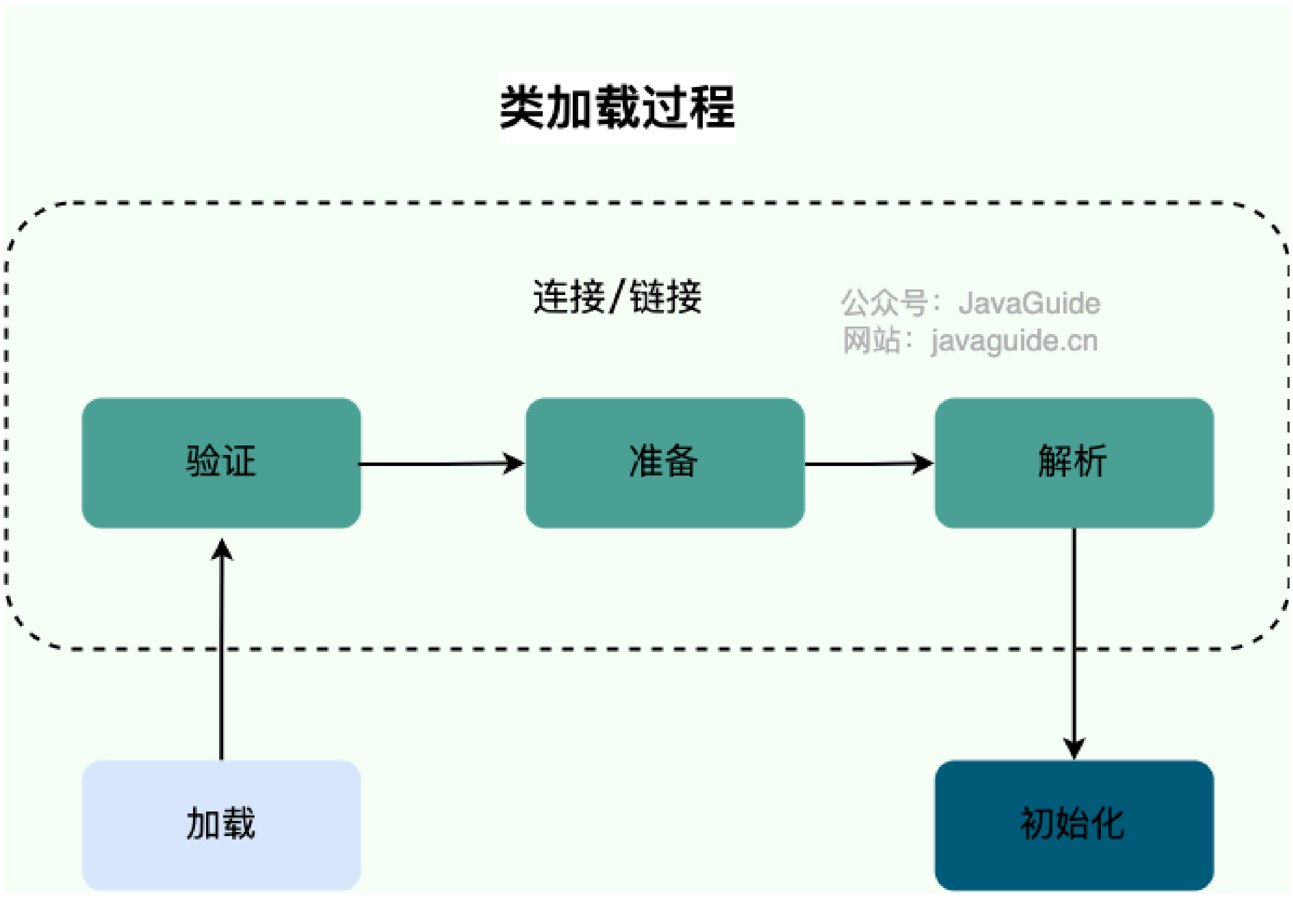
The sequence of these 7 stages is shown in the figure below:



Class loading process

Class files need to be loaded into the virtual machine before they can be run and used. So how does the virtual machine load these Class files?

The system loads Class files in three main steps: **loading -> linking -> initializatio** The linking process can be further divided into three steps: **verification -> prepar** **-> parsing** .



See Java Virtual Machine Specification - 5.3. Creation and Loading for details .

load

The first step in the class loading process mainly completes the following three things:

1. Gets the binary byte stream that defines this class by its full class name.
2. Convert the static storage structure represented by the byte stream into the runtime data structure of the method area.
3. Generate an object representing the class in memory `Class` as the access entry for the data in the method area.

The three points above in the virtual machine specification are not specific and therefore very flexible. For example, "obtaining the binary byte stream defining this class by its fully qualified class name" doesn't specify where or how to obtain it (e.g., from the `ZIP` network , dynamically generated during runtime using dynamic proxy technology, or generated from other files). `JAR` `EAR` `WAR` `JSP`

The loading process is primarily accomplished through **class loaders, which we'll cuss later. There are many different types of class loaders, and when we want to load a class, the specific class loader that loads it is determined by the par-**

ent delegation model (although this can be broken).

> Class loaders and the parent delegation model are also very important knowledge points, which are covered in detail in the article "Detailed Explanation of Class Loaders" . When reading this article, just know that they exist.

Each Java class has a reference to the class that loaded it `ClassLoader` . However, the array class is not `ClassLoader` created by , but is automatically created by the JVM when needed. When the array class `getClassLoader()` is obtained through the method `ClassLoader` , the element type of the array `ClassLoader` is consistent.

The loading phase of a non-array class (the action of obtaining the binary byte stream of the class during the loading phase) is the most controllable phase. We can complete this step and also customize the class loader to control how the byte stream is obtained (override a class loader's `loadClass()` method).

Some actions in the loading phase and the linking phase (such as some bytecode file format verification actions) are performed in an interleaved manner. The linking phase may have already begun before the loading phase is completed.

verify

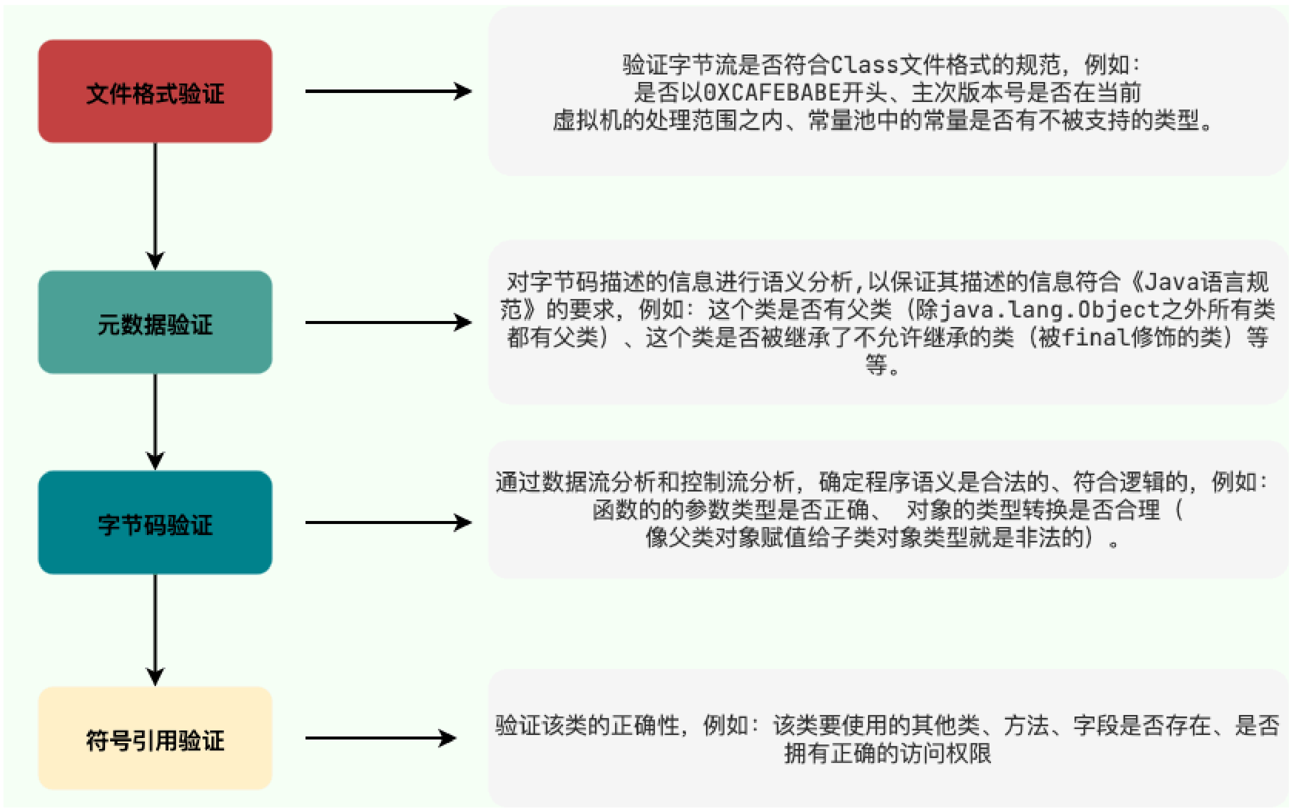
Verification is the first step in the connection phase. The purpose of this phase is to ensure that the information contained in the byte stream of the Class file complies with all the constraints of the Java Virtual Machine Specification and that this information will not endanger the security of the virtual machine itself when it is run as code.

The verification phase consumes relatively more resources than the entire class loading process, but it is necessary to effectively prevent the execution of malicious code. At all times, program security is the top priority.

However, the verification phase isn't mandatory. If all code running in your program (including code written by you, from third-party packages, loaded from external sources, and dynamically generated code) has been repeatedly used and verified, you can consider using `-Xverify:none the ``` parameter to disable most class verification measures during production implementation to shorten VM class loading time. However, it's important to note that ``` -Xverify:none and -noverify ``` are deprecated in JDK 13 and may be removed in future JDK versions.

The verification phase mainly consists of four inspection stages:

1. File format verification (Class file format check)
2. Metadata validation (bytecode semantic checking)
3. Bytecode verification (program semantics checking)
4. Symbol reference verification (class correctness check)



验证字节流是否符合Class文件格式的规范，例如：是否以0XCAFEBABE开头、主次版本号是否在当前虚拟机的处理范围之内、常量池中的常量是否有不被支持的类型。

对字节码描述的信息进行语义分析，以保证其描述的信息符合《Java语言规范》的要求，例如：这个类是否有父类（除java.lang.Object之外所有类都有父类）、这个类是否被继承了不允许继承的类（被final修饰的类）等等。

通过数据流分析和控制流分析，确定程序语义是合法的、符合逻辑的，例如：函数的的参数类型是否正确、 对象的类型转换是否合理（像父类对象赋值给子类对象类型就是非法的）。

验证该类的正确性，例如：该类要使用的其他类、方法、字段是否存在、是否拥有正确的访问权限

The file format verification phase is performed on the class's binary byte stream. Its primary purpose is to ensure that the input byte stream is correctly parsed and stored in the method area, and that the format meets the requirements for describing Java type information. Aside from this phase, the remaining three verification phases are performed on the method area's storage structure and do not directly read or manipulate the byte stream.

> The method area is a logical area within the JVM's runtime data area, a memory area shared by all threads. When the virtual machine uses a class, it reads and parses the class file to obtain relevant information and then stores it in the method area. The method area stores **information about classes loaded by the virtual machine, fields, methods, constants, static variables, and the code cache compiled by the just-in-time compiler** .
> For a detailed introduction to the method area, it is recommended to read the article Java Memory Area Detailed Explanation .

Symbolic reference verification occurs during the parsing phase of the class loading process. Specifically, it is when the JVM converts symbolic references into direct references (symbolic references and direct references will be introduced in the parsing phase).

The main purpose of symbol reference verification is to ensure that the parsing phase can be executed normally. If the symbol reference verification fails, the JVM will throw an exception, such as:

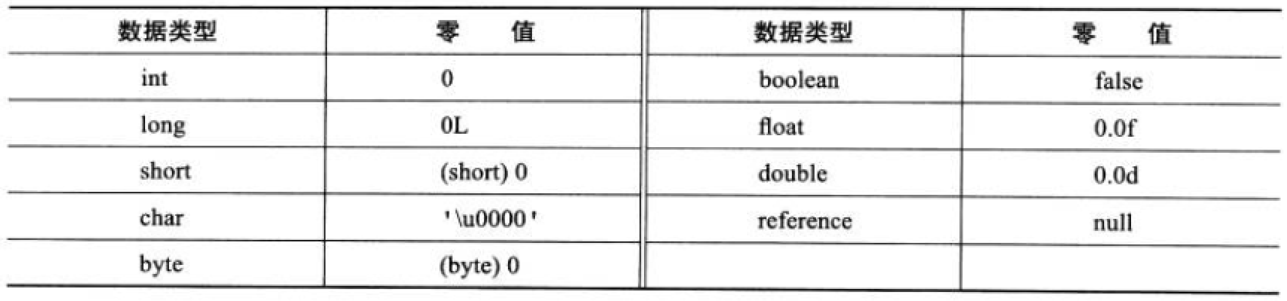
- `java.lang.IllegalAccessError` : This exception is thrown when a class attempts to access or modify a field that it does not have permission to access, or to call a method that it does not have permission to access.
- `java.lang.NoSuchFieldError` : This exception is thrown when a class attempts to access or modify a specified object field and the object no longer contains the field.
- `java.lang.NoSuchMethodError` : This exception is thrown when a class attempts to access a specified method and the method does not exist.
- ...

Prepare

The preparation phase is the stage where memory is allocated for class variables and their initial values are set . This memory is allocated in the method area. There are a few things to note about this phase:

1. At this time, memory allocation only includes class variables (static variables, `static` variables modified by the keyword, and associated only with the class, hence the name class variables), not instance variables. Instance variables are allocated on the Java heap along with the object when it is instantiated.
2. Conceptually, all memory used by class variables should be allocated in **the method area** . However, one thing to note is that before JDK 7, when HotSpot used the permanent generation to implement the method area, the implementation was completely consistent with this logical concept. In JDK 7 and later, HotSpot has moved the string constant pool, static variables, and other objects originally placed in the permanent generation to the heap. At this time, class variables are stored in the Java heap along with the Class object. Related reading: "In-depth Understanding of the Java Virtual Machine (3rd Edition)" Errata #75
3. The initial value set here is usually the data type's default zero value (e.g., 0, 0L, null, false, etc.). For example, if we define it `public static int value=111` , the initial value of the value variable during the preparation phase is 0, not 111 (which is assigned during the initialization phase). Special cases: For example, if the final keyword is added to the value variable `public static final int value=111` , the value during the preparation phase is assigned to 111.

Zero values of basic data types : (Image from "In-depth Understanding of the Java Virtual Machine" 3rd Edition 7.3.3)

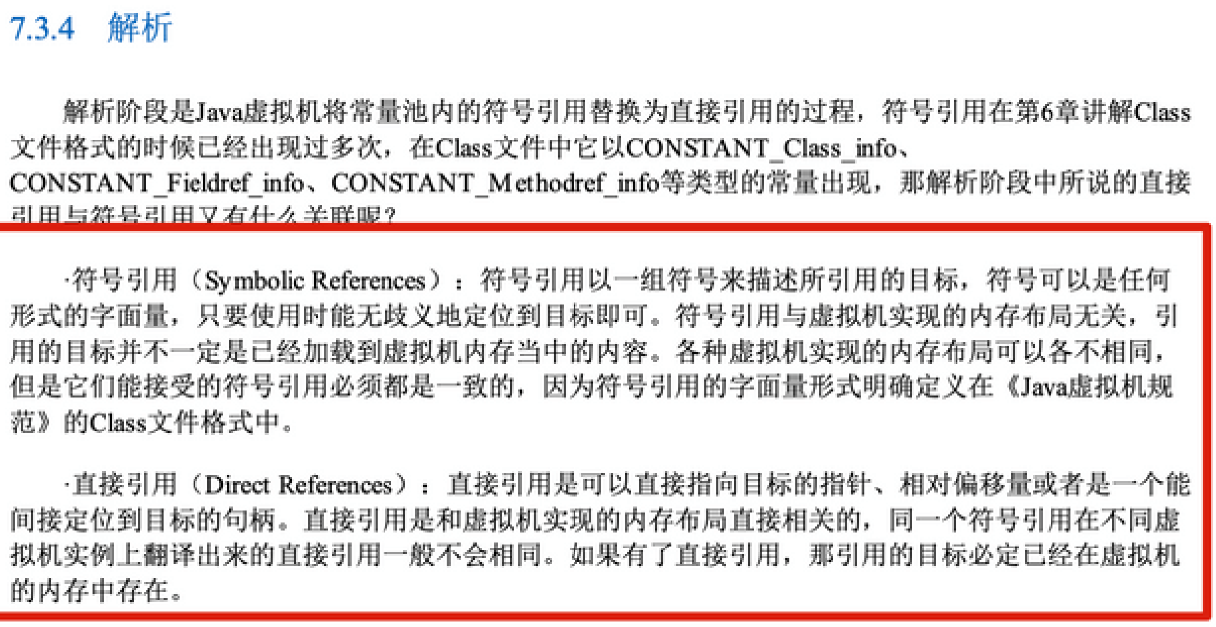


数据类型	零 值	数据类型	零 值
int	0	boolean	false
long	0L	float	0.0f
short	(short) 0	double	0.0d
char	'\u0000'	reference	null
byte	(byte) 0		

Analysis

The resolution phase is the process by which the virtual machine replaces symbolic references in the constant pool with direct references. Resolution is performed on seven types of symbolic references: classes or interfaces, fields, class methods, interface methods, method types, method handles, and call qualifiers.

Section 7.3.4 of the third edition of "In-depth Understanding of the Java Virtual Machine" explains symbolic references and direct references as follows:



7.3.4 解析

解析阶段是Java虚拟机将常量池内的符号引用替换为直接引用的过程，符号引用在第6章讲解Class文件格式的时候已经出现过多次，在Class文件中它以CONSTANT_Class_info、CONSTANT_Fieldref_info、CONSTANT_Methodref_info等类型的常量出现，那解析阶段中所说的直接引用与符号引用又有什么关联呢？

·符号引用（Symbolic References）：符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。符号引用与虚拟机实现的内存布局无关，引用的目标并不一定是已经加载到虚拟机内存当中的内容。各种虚拟机实现的内存布局可以各不相同，但是它们能接受的符号引用必须都是一致的，因为符号引用的字面量形式明确定义在《Java虚拟机规范》的Class文件格式中。

·直接引用（Direct References）：直接引用是可以直接指向目标的指针、相对偏移量或者是一个能间接定位到目标的句柄。直接引用是和虚拟机实现的内存布局直接相关的，同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同。如果有了直接引用，那引用的目标必定已经在虚拟机的内存中存在。

For example, when a program executes a method, the system needs to know exactly where that method is located. The Java Virtual Machine (JVM) maintains a method table for each class, storing all of its methods. To call a class method, simply knowing the offset of the method in the method table allows direct invocation. By resolving the symbolic reference, the target method's location in the class's method table can be directly converted to the method's location within the class, allowing the method to be called.

In summary, the parsing phase is the process in which the virtual machine replaces the symbolic references in the constant pool with direct references, that is, obtaining the pointer or offset of the class, field, or method in memory.

initialization

The initialization phase is `<clinit> ()` the process of executing the initialization method and is the last step of class loading. At this step, the JVM begins to actually execute the Java program code (bytecode) defined in the class.

> Note: `<clinit> ()` The method is automatically generated after compilation.

`<clinit> ()` The virtual machine ensures the safety of calling the method in a multi-threaded environment. Because the `<clinit> ()` method is thread-safe with a lock, class initialization in a multi-threaded environment may cause multiple threads to be blocked, and this blocking may be difficult to detect.

For the initialization phase, the virtual machine strictly regulates that there are only 6 cases in which the class must be initialized (the class will only be initialized if it is actively used):

1. When encountering `new` , `getstatic` , `putstatic` or `invokestatic` these 4 bytecode instructions:
    - `new` : Create an instance object of a class.
    - `getstatic` , `putstatic` : Read or set a type's static field (`final` except for static fields that have been modified and have put the results into the constant pool at compile time).
    - `invokestatic` : Call a static method of a class.
2. When using `java.lang.reflect` the package method to reflectively call a class, such as `Class.forName("...")` , `newInstance()` etc. If the class is not initialized, its initialization needs to be triggered.
3. Initialize a class. If its parent class has not been initialized, the initialization of the parent class will be triggered first.
4. When the virtual machine starts, the user needs to define a main class to be executed (`main` the class that contains the method), and the virtual machine will initialize this class first.
5. `MethodHandle` and `VarHandle` can be regarded as lightweight reflection calling mechanisms. In order to use these two calls, you must first use `findStaticVarHandle` to initialize the class to be called.
6. **"Supplement, from issue 745** " When an interface defines a default method newly added in JDK8 (an interface method modified by the default keyword), if an

implementation class of this interface is initialized, the interface must be initialized before it.

Class Unloading

> The uninstallation section comes from issue#662 and is supplemented by **guang19 .**

Unloading a class means that the Class object of the class is GCed.

Uninstalling a class needs to meet three requirements:

1. All instance objects of this class have been GCed, which means that there are no instance objects of this class in the heap.
2. This class is not referenced anywhere else
3. The class loader instance of this class has been GCed

Therefore, during the JVM lifecycle, classes loaded by the JVM's own class loader will not be unloaded. However, classes loaded by our custom class loader may be unloaded.

Just think about it, the JDK's built-in class loader `BootstrapClassLoader` is `ExtClassLoader` responsible `AppClassLoader` for loading the classes provided by the JDK, so they (class loader instances) will definitely not be recycled. However, our custom class loader instances can be recycled, so classes loaded using our custom loader can be unloaded.

refer to

- Deep Understanding of Java Virtual Machine
- Practical Java Virtual Machine
- Chapter 5. Loading, Linking, and Initializing - Java Virtual Machine Specification: https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-5.html#jvms-5.4

JavaGuide官方公众号
（微信搜索JavaGuide）
1、公众号后台回复“PDF”获取原创PDF面试手册
2、公众号后台回复“学习路线”获取Java学习路线最新版
3、公众号后台回复“开源”获取优质Java开源项目合集
4、公众号后台回复“八股文”获取Java面试真题+面经

Recently Updated2025/6/10 15:53

Contributors: SnailClimb , Ryze-Zhao , Shuang Kou , liwenguang , shuang.kou , Shuo Yu , guide , hening1 , hailong.sha , 553899811 , Joe , vcjmhg , Ma Zhaoming , zdkcomeon , Guide , linxuanstar , Mr.Hope , paigeman , liangyi , gaohang , scclsy , bcdax110 , xiaoxin