

## Java虚拟机详解（十）-----类加载过程

### 目录

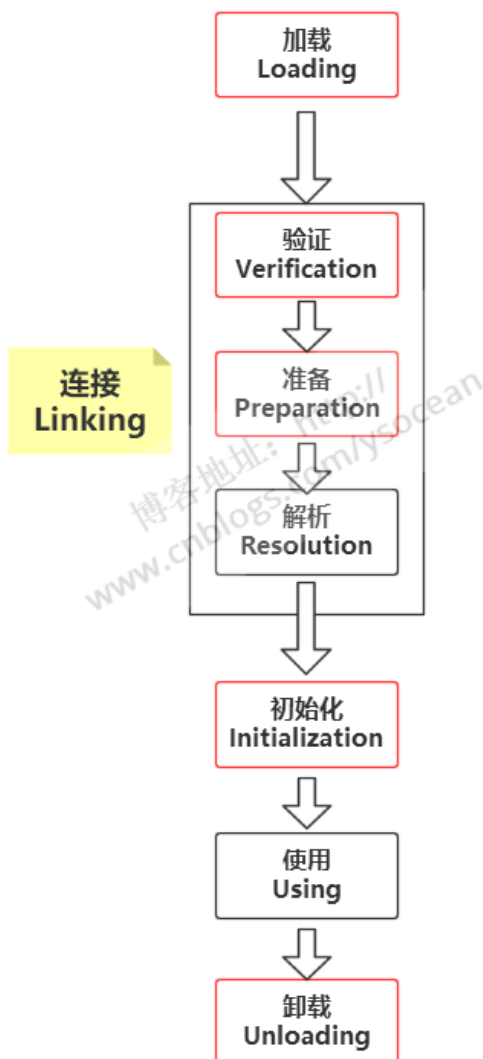
- [1、类的生命周期](#)
- [2、加载](#)
- [3、验证](#)
  - [①、文件格式验证](#)
  - [②、元数据验证](#)
  - [③、字节码验证](#)
  - [④、符号引用验证](#)
- [4、准备](#)
- [5、解析](#)
- [6、初始化](#)

在上一篇文章中，我们详细的介绍了Java类文件结构，那么这些Class文件是如何被加载到内存，由虚拟机来直接使用的呢？这就是本篇博客将要介绍的——类加载过程。

[回到顶部](#)

### 1、类的生命周期

类从被加载到虚拟机内存开始，到卸载出内存为止，其声明周期流程如下：



上图中红色的5个部分（加载、验证、准备、初始化、卸载）顺序是确定的，也就是说，类的加载过程必须按照这种顺序按部就班的开始。这里的“开始”不是按部就班的“进行”或者“完成”，因为这些阶段通常是互相交叉混合的进行的，通常会在一个阶段执行过程中调用另一个阶段。

[回到顶部](#)

## 2. 加载

“加载”阶段是“类加载”生命周期的第一个阶段。在加载阶段，虚拟机要完成下面三件事：

- ①、通过一个类的全限定名来获取定义此类的二进制字节流。
- ②、将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。
- ③、在Java堆中生成一个代表这个类的`java.lang.Class`对象，作为方法区这些数据的访问入口。

PS：类的全限定名可以理解为这个类存放的绝对路径。方法区是JDK1.7以前定义的运行时数据区，而在JDK1.8以后改为元数据区（Metaspace），主要用于存放被Java虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。详情可以参考这边该系列的第二篇文章——[运行时内存结构](#)。

另外，我们看第一点——通过类的权限限定名来获取定义此类的二进制流，这里并没有明确指明要从哪里获取以及怎样获取，也就是说并没有明确规定一定要我们从一个Class文件中获取。基于此，在Java的发展过程中，充满创造力的开发人员在这个舞台上玩出了各种花样：

- 1、从 ZIP 包中读取。这称为后面的 JAR、EAR、WAR 格式的基础。
- 2、从网络中获取。比较典型的应用就是 Applet。
- 3、运行时计算生成。这就是动态代理技术。
- 4、由其它文件生成。比如 JSP 应用。
- 5、从数据库中读取。

加载阶段完成后，虚拟机外部的二进制字节流就按照虚拟机所需的格式存储在方法区中，然后在Java堆中实例化一个 `java.lang.Class` 类的对象，这个对象将作为程序访问方法区中这些类型数据的外部接口。

注意，加载阶段与连接阶段的部分内容（如一部分字节码文件的格式校验）是交叉进行的，加载阶段尚未完成，连接阶段可能已经开始了。

[回到顶部](#)

### 3. 验证

验证是连接阶段的第一步，作用是为了**确保 Class 文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。**

我们说Java语言本身是相对安全，因为编译器的存在，纯粹的Java代码要访问数组边界外的数据、跳转到不存在的代码行之类的，是要被编译器拒绝的。但是前面我们也说过，Class 文件不一定非要由Java源码编译过来，可以使用任何途径，包括你很牛逼，直接用十六进制编辑器来编写 Class 文件。

所以，如果虚拟机不检查输入的字节流，将会载入有害的字节流而导致系统崩溃。但是虚拟机规范对于检查哪些方面，何时检查，怎么检查都没有明确的规定，不同的虚拟机实现方式可能都会有所不同，但是大致都会完成下面四个方面的检查。

#### ①、文件格式验证

校验字节流是否符合Class文件格式的规范，并且能够被当前版本的虚拟机处理。

- 一、是否以魔数 `0xCAFEFABE` 开头。
- 二、主、次版本号是否是当前虚拟机处理范围之内。
- 三、常量池的常量中是否有不被支持的常量类型（检查常量tag标志）
- 四、指向常量的各种索引值中是否有指向不存在的常量或不符合类型的常量。
- 五、`CONSTANT_Utf8_info` 型的常量中是否有不符合 UTF8 编码的数据。
- 六、Class 文件中各个部分及文件本身是否有被删除的或附加的其他信息。

以上是一部分校验内容，当然远不止这些。经过这些校验后，字节流才会进入内存的方法区中存储，接下来后面的三个阶段校验都是基于方法区的存储结构进行的。

#### ②、元数据验证

第二个阶段主要是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语言规范要求。

- 一、这个类是否有父类（除了`java.lang.Object` 类之外，所有的类都应当有父类）。
- 二、这个类的父类是否继承了不允许被继承的类（被`final`修饰的类）。
- 三、如果这个类不是抽象类，是否实现了其父类或接口之中要求实现的所有普通方法。
- 四、类中的字段、方法是否与父类产生了矛盾（例如覆盖了父类的`final`字段、或者出现不符合规则的重载）

#### ③、字节码验证

第三个阶段字节码验证是整个验证阶段中最复杂的，主要是进行数据流和控制流分析。该阶段将对类的方法进行分析，保证被校验的方法在运行时不会做出危害虚拟机安全的行为。

- 一、保证任意时刻操作数栈中的数据类型与指令代码序列都能配合工作。例如不会出现在操作数栈中放置了一个 int 类型的数据，使用时却按照 long 类型来加载到本地变量表中。
- 二、保证跳转指令不会跳转到方法体以外的字节码指令中。
- 三、保证方法体中的类型转换是有效的。比如把一个子类对象赋值给父类数据类型，这是安全的。但是把父类对象赋值给子类数据类型，甚至赋值给完全不相干的类型，这就是不合法的。

④、符号引用验证

符号引用验证主要是对类自身以外（常量池中的各种符号引用）的信息进行匹配性的校验，通常需要校验如下内容：

- 一、符号引用中通过字符串描述的全限定名是否能够找到相应的类。
- 二、在指定类中是否存在符合方法的字段描述符及简单名称所描述的方法和字段。
- 三、符号引用中的类、字段和方法的访问性（private、protected、public、default）是否可以被当前类访问。

[回到顶部](#)

4、准备

准备阶段是正式为**类变量**分配内存并设置**类变量**初始值的阶段，这些内存是在方法区中进行分配。

注意：

- 一、上面说的是类变量，也就是被 static 修饰的变量，不包括实例变量。实例变量会在对象实例化时随着对象一起分配在堆中。
- 二、初始值，指的是一些数据类型的默认值。基本的数据类型初始值如下（引用类型的初始值为null）：

表 7-1 基本数据类型的零值

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

比如，定义 public static int value = 123 。那么在准备阶段过后，value 的值是 0 而不是 123,把 value 赋值为123 是在程序被编译后，存放在类的构造器方法之中，是在初始化阶段才会被执行。但是有一种特殊情况，通过final 修饰的属性，比如 定义 public final static int value = 123, 那么在准备阶段过后，value 就被赋值为123了。

[回到顶部](#)

5、解析

解析阶段是虚拟机将常量池中的符号引用替换为直接引用的过程。

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的定位到目标即可。符号引用与虚拟机实现的内存布局无关，引用的目标不一定已经加载到内存中。

直接引用（Direct References）：直接引用可以是直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。直接引用是与虚拟机实现内存布局相关的，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，那么引用的目标必定已经在内存中存在。

解析动作主要针对类或接口、字段、类方法、接口方法四类符号引用，分别对应于常量池的 `CONSTANT_Class_info`、`CONSTANT_Fieldref_info`、`CONSTANT_Methodref_info`、`CONSTANTS_InterfaceMethodref_info` 四种类型常量。

[回到顶部](#)

## 6. 初始化

初始化阶段是类加载阶段的最后一步，前面过程中，除第一个加载阶段可以通过用户自定义类加载器参与之外，其余过程都是完全由虚拟机主导和控制。而到了初始化阶段，则开始真正执行类中定义的Java程序代码（或者说是字节码）。

在前面介绍的准备阶段中，类变量已经被赋值过初始值了，而初始化阶段，则根据程序员的编码去初始化变量和资源。

换句话说，**初始化阶段是执行类构造器 `<clinit>()` 方法的过程。**

①、`<clinit>()` 方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块（`static{}`）中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的，静态语句块中只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句块中可以赋值，但是不能访问。

比如如下代码会报错：

```
1 package com.yb.carton.controller;
2
3 /**
4  * Create by YSOcean
5  */
6 public class ClassLoadInitTest {
7
8     |
9     static{
10         i = 0;
11         System.out.println(i);
12     }
13
14     static int i = 1;
15 }
```

Illegal forward reference

但是你把第 14 行代码放到 `static` 静态代码块的上面就不会报错了。或者不改变代码顺序，将第 11 行代码移除，也不会报错。

②、`<clinit>()` 方法与类的构造函数（或者说是实例构造器 `<init>()` 方法）不同，它不需要显示的调用父类构造器，虚拟机会保证在子类的 `<init>()` 方法执行之前，父类的 `<init>()` 方法已经执行完毕。因此虚拟机中第一个被执行的 `<init>()` 方法的类肯定是 `java.lang.Object`。

③、由于父类的 `<clinit>()` 方法先执行，所以父类中定义的静态语句块要优先于子类的变量赋值操作。

④、`<clinit>()` 方法对于接口来说并不是必须的，如果一个类中没有静态语句块，也没有对变量的赋值操作，那么编译器可以不为此类生成 `<clinit>()` 方法。

⑤、接口中不能使用静态语句块，但仍然有变量初始化的赋值操作，因此接口与类一样都会生成 `<clinit>()` 方法。但接口与类不同的是，执行接口中的 `<clinit>()` 方法不需要先执行父接口的 `<clinit>()` 方法。只有当父接口中定义的变量被使用时，父接口才会被初始化。

⑥、接口的实现类在初始化时也一样不会执行接口的 `<clinit>()` 方法。

⑦、虚拟机保证一个类的<clinit>() 方法在多线程环境中被正确的加锁和同步。如果多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的<clinit>() 方法，其他的线程都需要阻塞等待，直到活动线程执行<clinit>() 方法完毕。如果在一个类的<clinit>() 方法中有很耗时的操作，那么可能造成多个进程的阻塞。

比如对于如下代码：

```
package com.yb.carton.controller;

/**
 * Create by YSOcean
 */
public class ClassLoadInitTest {

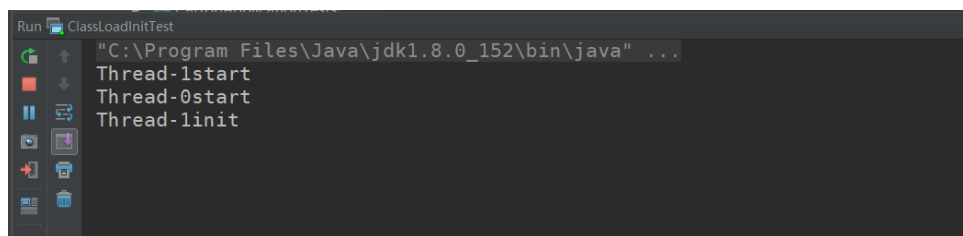
    static class Hello{
        static {
            if(true){
                System.out.println(Thread.currentThread().getName() + "init");
                while(true){}
            }
        }
    }

    public static void main(String[] args) {

        new Thread()->{
            System.out.println(Thread.currentThread().getName()+"start");
            Hello h1 = new Hello();
            System.out.println(Thread.currentThread().getName()+"run over");
        }.start();

        new Thread()->{
            System.out.println(Thread.currentThread().getName()+"start");
            Hello h2 = new Hello();
            System.out.println(Thread.currentThread().getName()+"run over");
        }.start();
    }
}
```

运行结果如下：



线程1抢到了执行<clinit>() 方法，但是该方法是一个死循环，线程2将一直阻塞等待。

知道了类的初始化过程，那么类的初始化何时被触发呢？JVM大概规定了如下几种情况：

①、当虚拟机启动时，初始化用户指定的类。

- ②、当遇到用以新建目标类实例的 new 指令时，初始化 new 指定的目标类。
- ③、当遇到调用静态方法的指令时，初始化该静态方法所在的类。
- ④、当遇到访问静态字段的指令时，初始化该静态字段所在的类。
- ⑤、子类的初始化会触发父类的初始化。
- ⑥、如果一个接口定义了 default 方法，那么直接实现或间接实现该接口的类的初始化，会触发该接口的初始化。
- ⑦、使用反射 API 对某个类进行反射调用时，会初始化这个类。
- ⑧、当初次调用 MethodHandle 实例时，初始化该 MethodHandle 指向的方法所在的类。