JVM虚拟机—类加载机制

|  |  |
| --- | --- |
|  | 🢂 内容概览 |
|  | Why：此文档用来做什么？它存在的意义是什么？为解决什么问题？   |  | | --- | |  |   What：当前包含了那些内容？   |  | | --- | |  |   How：此文档应如何参考？   |  | | --- | |  |   Who：此文档适用于那些人员阅读参考？   |  | | --- | |  |   Summary：摘要   |  | | --- | |  |   Reference：参考文献   |  | | --- | |  | |

目录

[1 什么是类加载机制？ 4](#_Toc500349079)

[2 类加载过程包括哪些阶段？生命周期如何？ 4](#_Toc500349080)

[2.1 加载阶段 4](#_Toc500349081)

[2.1.1 普通类的加载过程 4](#_Toc500349082)

[2.1.2 数组类的加载过程 5](#_Toc500349083)

[2.2 验证阶段 6](#_Toc500349084)

[2.2.1 验证内容包含哪些？ 6](#_Toc500349085)

[2.2.2 如何理解StackMapTable优化？ 6](#_Toc500349086)

[2.3 准备阶段 7](#_Toc500349087)

[2.3.1 如何理解零值？ 7](#_Toc500349088)

[2.4 解析阶段 7](#_Toc500349089)

[2.4.1 解析发生的时机是什么时候？ 7](#_Toc500349090)

[2.4.2 什么是符号引用？什么是直接引用？ 7](#_Toc500349091)

[2.4.3 什么是符号引用缓存？ 7](#_Toc500349092)

[2.4.4 如何理解引用目标？ 8](#_Toc500349093)

[2.4.4.1 引用类的解析； 8](#_Toc500349094)

[2.4.4.2 引用字段的解析； 8](#_Toc500349095)

[2.4.4.3 引用方法的解析； 9](#_Toc500349096)

[2.5 初始化阶段 11](#_Toc500349097)

[2.5.1 如何理解类构造器<clinit>？ 11](#_Toc500349098)

[2.5.2 什么时候进行类的初始化？初始化的条件是什么？ 12](#_Toc500349099)

[2.5.2.1 触发条件 12](#_Toc500349100)

[2.5.2.2 哪些场景不会触发类的初始化？ 12](#_Toc500349101)

[2.5.2.3 创建类的数组时，使用了new关键字，为什么没有初始化类？ 13](#_Toc500349102)

[2.5.2.4 java在创建数组时，为什么要创建[Lmypackage.MyClass类型？ 13](#_Toc500349103)

[2.5.2.5 什么是常量传播优化？为什么调用类的常量不会触发该类的加载？ 13](#_Toc500349104)

[2.5.2.6 接口的初始化和类的初始化有什么区别？为什么会有这个区别？ 13](#_Toc500349105)

[3 什么是类加载器？ 13](#_Toc500349106)

[3.1 类加载器的分类 14](#_Toc500349107)

[3.1.1 启动类加载器（Bootstrap ClassLoader） 14](#_Toc500349108)

[3.1.2 扩展类加载器（Extension ClassLoader） 14](#_Toc500349109)

[3.1.3 应用程序类加载器（Application ClassLoader） 14](#_Toc500349110)

[3.1.4 自定义类加载器（User ClassLoader） 14](#_Toc500349111)

[3.1.4.1 两种方法有什么区别？ 14](#_Toc500349112)

[3.1.5 线程上下文类加载器（Thread Context ClassLoader） 14](#_Toc500349113)

[3.1.5.1 线程上下文类加载器有什么作用？ 14](#_Toc500349114)

[3.2 双亲委派模型 15](#_Toc500349115)

[3.2.1 什么是双亲委派模型？ 15](#_Toc500349116)

[3.2.2 双亲委派模型的调用过程 15](#_Toc500349117)

[3.2.3 为什么要使用双亲委派模型？ 17](#_Toc500349118)

[4 应用案例 17](#_Toc500349119)

[5 相关知识点 17](#_Toc500349120)

[6 问题思考 17](#_Toc500349121)

[7 参考 18](#_Toc500349122)

# 什么是类加载机制？

java程序的从源代码到执行的过程包括编译和运行两个阶段。编译阶段由编译器执行，将源代码（.java）文件编译成字节码文件(class文件)；运行阶段由JVM执行，将字节码文件加载到内存中，变为虚拟机可以直接使用的数据结构，该过程即为类加载机制。

# 类加载过程包括哪些阶段？生命周期如何？

类加载过程包括如下7个阶段：

1. **加载：**从字节码二进制变为Class对象；
2. **验证：**校验字节码格式是否合法；
3. **准备：**为类变量static修饰变量赋初始零值，分配内存；
4. **解析：**将常量池中的符号引用替换为直接引用；
5. **初始化：**执行类构造器<clinit>，包括：给类变量赋默认值，执行类中的静态代码块；
6. **使用：**在程序方法中使用类；
7. **卸载：**对方法区（元空间）中的Class对象进行GC回收，清除不必要的Class对象；

其中验证、准备、解析3个阶段被合称为连接阶段，即将Class对象与内存关联映射的过程。为了保证类加载的灵活性，java虚拟机规范仅要求加载、验证、准备、初始化、卸载的顺序固定，对于解析在什么阶段进行并没有给出详细约束，解析阶段也可以发生在初始化之后，用于支持运行时绑定（晚绑定、动态绑定）。

注意：此处的生命周期都是针对单个类而言的，出于性能考虑，jvm施行按需加载的策略，只有当类将要被使用时，才会加载。并不会在jvm启动时就加载所有的类。因此类加载的完整过程可能发生在jvm运行的任何时候。

## 加载阶段

加载阶段是整个加载过程的一部分，是指将class二进制流转换为Class对象，存入内存模型中的方法区（元空间）的过程；加载分为2类：普通类的加载和数组类的加载。

普通类的加载是指直接通过类加载加载的类。与之相对应的数组类的加载不是由类加载器加载的。在java中，数组变量也是一种对象，因而具有对象的类型。数组对应的类型，是由虚拟机在运行时自动创建并加载的，其类的全限定名是在数组元素类型的全限定名之前加上[L，比如mypackage.MyClass对应的数组类型为[Lmypackage.MyClass。

### 普通类的加载过程

1. **通过类的全限定名（包名.类名）获取类的二进制字节流。**之所以限定为二进制流，而非文件，是为了提高灵活性，Java在类的数据，既可以来自本地文件，也可以来自网络数据流，甚至可以通过字节码生成工具自动生成。这就极大地丰富了”创造”类对象的手段，比如： jdk提供的动态代理技术在Proxy中，通过ProxyGenerator.generateProxyClass来为特定的接口生成形式为\*$Proxy的代理类二进制字节流，为AOP的实现提供了基础。
2. 将字节流所代表的静态存储结构转化为方法区（jdk1.8为元空间）的运行时数据结构。
3. 在内存中生成一个代表这个类的java.lang.Class对象，作为方法区（jdk1.8为元空间）这个类数据的访问入口；

问：元空间内类数据存放的结构是怎样的，是否有规范可循？

### 数组类的加载过程

1. 如果数组的元素类型是引用类型，则递归加载元素类型，然后在元素类型所属的类加载器的类名空间中标识数组类；即：数组类型和元素类型使用相同的类加载器加载；
2. 如果数组的元素类型是基础类型，则在系统类加载器（AppClassLoader）的类名空间中标识数组类；即：数组类型使用系统类加载器加载；
3. 生成数组类的可见性与元素类型的可见性一致；如果元素类型是基础类型，则数组类型的可见性默认为public；

问1：当数组元素类型为基础类型时，基础类型是由引导类加载器（BootstrapClassLoader）加载的，是否是因为引导类加载器对于数组类型而言不可见，故使用系统类加载器？

问2：类和类加载器是如何关联的?

注意：此处要区分类的加载和初始化2个阶段，当出现如下代码时，虽然不会触发类的初始化，但会触发类的加载；

|  |
| --- |
| /\*\*  \* <h1>被动引用会加载类，但不会进行类初始化</h1>  \* <p>  \* -XX:+TraceClassLoading  \* </p>  \*/  public class NotLoad {  // 不会加载，因为没有初始化ElementClass  static class RefClass {  static {  System.out.println("RefClass has bean initialized.");  }  }  // 会加载，不会初始化  private static class ElementClass {  static {  System.out.println("ElementClass has bean initialized.");  }  private RefClass ref;  }  public static void main(String[] args) {  // 创建数组，元素类型为ElementClass  // 只会加载ElementClass，不会初始化  ElementClass[] elements = new ElementClass[10];  }  } |

运行结果如下，可以看到NotLoad类和ElementClass类的加载信息：

|  |
| --- |
| 类加载后未初始化 |

由于静态块是在类的初始化阶段执行，而结果中并未打印静态块中的语句，因而可以断定jvm位对ElementClass类进行初始化；

## 验证阶段

验证阶段的主要目的是为了确保class字节流数据的合法性，防止损害虚拟机自身的安全。因而验证阶段可以看做是出于安全考虑而增加的额外阶段，假定所加载的class字节流可以保证安全，则该阶段可以跳过。通过-Xverify:none参数可以关闭验证。

### 验证内容包含哪些？

1. 文件格式验证：验证字节流是否符合Class文件格式规范；
2. 元数据验证：语义分析，对元数据的数据类型进行校验，保证字节码描述信息符合JAVA语言规范；
3. 字节码验证：通过数据流和控制流分析，确定程序的语义是合法的，主要是对方法体中代码的分析；
4. 符号引用验证：发生在将符号引用转化为直接引用时（与解析阶段重叠），校验符号引用是否能够找到匹配的类；

### 如何理解StackMapTable优化？

该优化仅在jdk<1.7时有效。优化原因是字节码验证复杂度高，对性能消耗较多。为了提高运行时字节码验证的效率，将数据流分析提前到编译阶段完成，并将分析结果存放到字节码文件Code属性表的StackMapTable属性中。从而在校验时，直接读取StackMapTable中的分析结果进行校验即可，缩短了校验时间。但该优化也可能存在风险，即StackMapTable是存放在字节码文件中的，本身也是可以被篡改的。可以通过-XX:-UseSplitVerifier参数关闭StackMapTable优化。

## 准备阶段

准备阶段主要是为类变量（static修饰）分配内存，赋初始零值。此处的零值并非代码中显式为类变量赋予的默认值，而是指数据类型的零值。如果是常量（static final修饰，且字段属性表存在ContantValue属性），则初始值为常量值。ContantValue属性的值是在编译时放入的。

### 如何理解零值？

数据类型的零值，而非代码中给出的默认值。为static变量赋默认值的操作，是在初始化阶段执行类构造器clinit时由putstatic指令完成的。clinit是在编译阶段生成的。不同的数据类型，零值如下：

* 引用类型零值为null；
* 数值类型零值为0；
* boolean值类型零值为false；
* char类型零值为\u0000；

## 解析阶段

将常量池中的符号引用替换为直接引用。符号引用的解析是原子性的，对于同一符号引用的多次解析，要么全部成功，要么全部失败。

### 解析发生的时机是什么时候？

JVM规范并未规定解析阶段发生的具体时间，即虚拟机实现可以根据需要判断在类加载时就进行解析，还是在一个符号引用将要被使用前才去解析。这样做的主要目的是为了提高类整个类加载过程的灵活性。

### 什么是符号引用？什么是直接引用？

符号引用相当于一个占位符，用该占位符来描述代码执行时所引用的目标。目标并不局限于类/接口，它可以是：类/接口、字段、类方法、接口方法、方法类型、方法句柄、调用点限定符。符号引用并不需要考虑实际的内存布局，只要能够唯一标定要引用的目标即可。

直接引用是引用目标内存地址的标识，可以是一个指针、相对偏移量或者一个能够间接定位到目标的句柄。与内存布局强相关，因而不同的虚拟机实例上相同目标的直接引用一般不同。直接引用代表了引用目标在内存中的存在性，如果有直接引用，说明引用目标在内存中一定存在。

### 什么是符号引用缓存？

解析过程中，可能存在对于同一个符号引用进行多次解析请求。为了提高效率，避免重复解析，可以对符号引用进行缓存（在运行时常量池中记录直接引用，并把常量标识为已解析状态）；

### 如何理解引用目标？

解析都是针对方法体或者代码块中的执行的语句来说的。解析的目的是将方法体或者代码块中执行语句的符号引用替换为直接引用。对于成员变量直接赋引用或者用new操作符创建的情况，实际是在类构造器<clinit>和实例构造器<init>中执行的；亦可看做是在方法中的语句。

#### 引用类的解析；

1. 如果引用目标不是数组类型，则根据全限定名加载目标类；加载目标类使用当前类的类加载器；
2. 如果引用目标是数组类型，且数组的元素类型引用类型，则先加载数组的元素类型，再创建数组类型对象；
3. 如果上述完成，则验证对引用目类标是否具有访问权限；如果不具有访问权限，则抛出java.lang.IllegalAccessError错误；

#### 引用字段的解析；

1. 先解析字段所属类/接口的符号引用，然后解析字段的符号引用；
2. 字段的直接引用查找顺序：

|  |
| --- |
|  |

#### 引用方法的解析；

1. 先解析方法所属类/接口的符号引用，然后解析方法的符号引用；
2. 类和接口方法符号引用的常量类型定义是分开的，需要分别解析；
3. 类方法的直接引用查找顺序：

|  |
| --- |
|  |

注：类方法和接口方法引用的查找的区别在于，类方法一定要有一个实现了的方法，否则抛出异常；

1. 接口方法的引用查找顺序：

|  |
| --- |
|  |

## 初始化阶段

初始化阶段是执行类构造器<clinit>的阶段。类构造器<clinit>是有编译器生成的，主要用来为类的静态变量设置默认值，执行静态代码块。

### 如何理解类构造器<clinit>？

1. clinit方法是由编译器自动收集类中的所有类变量（静态成员变量）的赋值动作和静态代码块中的语句合并产生的；
2. 编译器的收集顺序是由类变量赋值语句和静态代码块在源文件中出现的顺序决定的；
3. 静态代码块只能访问定义在其前面的类变量，但可以给定义在其后的类变量赋值；
4. clinit方法无需在调用自己之前，调用父类的clinit方法，因为虚拟机能够保证先调用父类的clinit方法，第一个被执行的clinit方法一定是java.lang.Object类；
5. 因为父类的clinit方法先执行，所以父类的静态代码块要先于子类的静态代码块和类变量赋值语句执行；
6. 接口和父接口的clinit方法执行顺序不需要保证，因为接口中没有定义静态块，只可能出现接口变量赋值的情况；而接口变量赋值的情况不需要保证顺序；
7. clinit方法并不是必需的，如果类中没有对类变量的赋值语句，也没有静态代码块，则编译器不会为类生成clinit方法；
8. 虚拟机会保证一个类clinit方法在多线程环境中被正确的加锁、同步；如果多个线程同时去初始化一个类，只有一个线程执行clinit方法，其余线程会被阻塞，直到方法执行完才被唤醒，且唤醒后不会再次执行clinit方法；
9. 对于同一个类加载器，一个类型只会初始化一次，所以一个类的clinit方法只会被执行一次；

### 什么时候进行类的初始化？初始化的条件是什么？

类的初始化分为2中：类的初始化和接口的初始化。类的初始化主要包括：类变量赋值语句、静态代码块初始化两部分。接口的初始化只包括类变量的赋值语句。

#### 触发条件

所有类初始化触发条件的先决条件是：类未被初始化；

1. 代码中遇到new、getstatic/setstatic、invokestatic指令时，执行初始化；4条指令分别对应的操作是创建一个对象，读取/设置类变量，调用类的静态方法。
2. 通过java.lang.reflect包的方法对类进行反射调用；
3. 初始化子类时，如果父类未初始化，则先初始化父类；初始化接口时，与此处有区别，接口初始化不要求先初始化接口的所有父接口；
4. 启动虚拟机时，如果主类（包含main方法的类）未初始化，则先初始化主类；
5. 支持动态语言时，java.lang.invoke.MethodHandle实例解析的结果REF\_getStatic，REF\_putStatic，REF\_invokeStatic的方法句柄，句柄对应的类未初始化，则先初始化；

#### 哪些场景不会触发类的初始化？

1. 通过子类引用父类的静态字段，则只初始化父类，子类不会被初始化；对于静态字段，只有直接定义这个字段的类，在引用时会被初始化。比如下面语句不会触发SubClass类的初始化：

|  |
| --- |
| System.out.println(SubClass.superStaticField)； |

1. 创建数组对象，不会触发数组元素的初始化；newarray指令：定义某个类型的数组时，不会触发该类的初始化；只会触发数组类的初始化。比如如下代码，会触发[Lmypackage.MyClass数组类的初始化；

|  |
| --- |
| MyClass[] arr = MyClass[10]; |

1. 常量传播优化：常量在调用时，存储调用类的常量池中，本质上并没有直接引用到定义常量的类，故不会触发定义常量类的初始化；比如如下代码：

|  |
| --- |
| System.out.println(OtherClass.finalStaticField); |

#### 创建类的数组时，使用了new关键字，为什么没有初始化类？

创建类的数组时，并不进行mypackage.MyClass类的初始化，而是进行[Lmypackage.MyClass的初始化；[Lmypackage.MyClass类代表了mypackage.MyClass类的一维数组类型；由newarray指令创建；该类封装了数组的访问方法，包括：clone()和length()方法；数组的创建使用newarray指令而非new指令；当使用newarray指令时，会触发[Lmypackage.MyClass类的创建，这是由虚拟机自动生成的、直接继承java.lang.Object的子类；

#### java在创建数组时，为什么要创建[Lmypackage.MyClass类型？

[Lmypackage.MyClass类记录数组的元数据和访问方法，为了更好的进行数组类型校验和安全访问；数组越界检查封装在xaload和xastore字节码指令中，每次访问或者修改数组都会进行越界检查；如果访问索引越界，则跑出java.lang.ArrayIndexOutOfBoundsException异常；对比：c/c++对于数组的访问直接翻译为数组指针的移动，因而不能进行安全检查；

#### 什么是常量传播优化？为什么调用类的常量不会触发该类的加载？

为了提高性能，在编译阶段会将java类中被final static修饰的常量直接放到调用类自己的常量池中；调用类对常量的引用实际转化成了对自己常量池的引用；因而在调用时，不会加载定义常量的类；

#### 接口的初始化和类的初始化有什么区别？为什么会有这个区别？

**区别：**初始化子接口时，不会要求父接口全部初始化，只有真正用到父接口时才会初始化；但编译器仍然会为接口生成类构造器<clinit>()，用于初始化接口中的成员变量。

**原因：**类中可以定义static块，该块需要在类初始化后执行，且有执行顺序的要求，需要先执行父类中的static块，再执行子类中的static，因此需要先初始化父类；而接口中不允许static块，所以无需初始化父类。

# 什么是类加载器？

虚拟机设计团队把类加载阶段中的"通过一个类的全限定名来获取描述此类的二进制字节流"这个动作放到JVM外部去实现，以便让应用程序自己决定如何去获取所需要的类。实现这个动作的代码模块称为"类加载器"。每一个类加载器都有一个独立的类名称空间，因此类的唯一性需要类加载器和类本身一起确定。类相等的前提是需要在同一个类加载器的前提下判断，不同的类加载器加载相同的类，equals()/isAssignableFrom()/isInstance()/instanceof方法结果都会返回false。类加载器都继承自抽象类：java.lang.ClassLoader。

## 类加载器的分类

### 启动类加载器（Bootstrap ClassLoader）

负责加载放在${JAVA\_HOME}/lib目录中的类，或者由-Xbootclasspath参数所指定的路径中，且是被虚拟机识别的类库；虚拟机按照名称识别类。Bootstrap类加载器无法被java程序直接引用，用户自动义类加载器时，如果需要把加载请求委派给Bootstrap类加载器，则直接返回null即可。

### 扩展类加载器（Extension ClassLoader）

负责加载${JAVA\_HOME}/lib/ext目录中的类，或者被java.ext.dirs变量所指定路径中的类库。扩展类加载器可以直接使用。类型：sun.misc.Launcher.ExtClassLoader。

### 应用程序类加载器（Application ClassLoader）

又称为系统类加载器；负责加载用户类路径${CLASSPATH}上的所指定的类库。通过ClassLoader.getSystemClassLoader()方法可以获得，开发者可以直接使用。如果用户没有自定义类加载器，则默认使用系统类加载器。类型：sun.misc.Launcher.AppClassLoader。

### 自定义类加载器（User ClassLoader）

用户自定义的类加载器；可通过重写loadClass方法或者findClass方法实现。

#### 两种方法有什么区别？

两种实现方式的区别在于重写loadClass可以不遵守双亲委派模型，而重写findClass仍然遵守双亲委派模型。

### 线程上下文类加载器（Thread Context ClassLoader）

为每个线程提供一个上下文类加载器，调用Thread.setContextClassLoader()方法进行设置。如果当前线程没有设置，则会从父线程的类加载器。如果应用全局范围没有设置，则默认使用系统类加载器。

#### 线程上下文类加载器有什么作用？

便于基础类库调用上层服务的类库。比如，涉及SPI（Service Provider Interface，服务提供商接口）接口调用的场景，虚拟机在加载SPI的类库时，使用Thread的ContextClassLoader进行加载，具体厂商可以在加载前通过setContextClassLoader方法指定Thread的ContextClassLoader，用来加载自己的业务实现，从而实现了基础服务调用具体的上层业务实现的功能。

## 双亲委派模型

### 什么是双亲委派模型？

双亲委派模型是一种职责链模式实现，每个类加载器都包含一个parent的类加载器属性，用于存放上一级类加载器的引用，从而组成一个类加载器的调用链。调用链从最顶层开始类加载，每个类加载器都只负责加载符合自身加载条件的类。类加载器按照层次自上而下分别是：Bootstrap类加载器、扩展类加载器、应用程序类加载器、自定义类加载器。其中，Bootstrap类加载器和扩展类加载器分别用来加载JVM运行所需的基本类库和扩展类库；应用程序类加载器和自定义类加载器则用来加载程序运行所需的类库。类加载器中的这种层次关系称为双亲委派模型。如下图：

|  |
| --- |
| D:\share\01_project\mynote\201706_java虚拟机\01 note\职责链模式-双亲委派模型.png双亲委派职责链示意 |

双亲委派模型要求除了顶层的启动类加载器外，其余的类加载器都应当有自己的父类加载器。类加载器之间的父子关系一般不会以继承关系来实现，而是由组合关系来复用父加载器的代码（合成复用原则）。双亲委派模型并非强制约束，允许根据业务需求更改。

### 双亲委派模型的调用过程

虚拟机中所有类的加载，会按照自顶向下的优先级执行加载，父类加载器优先加载，如果失败，再交由子类加载器加载：

1. 类加载器在收到类加载请求时，会先委派给父类进行加载，调用父类的loadClass方法；
2. 如果当前父类加载器没有父类（父类为null），则使用Bootstrap类加载器进行类加载；如果加载失败，则抛出ClassNotFound异常；
3. 子类加载器捕获异常，进行类加载；如果加载失败，则抛出异常，交由下一级子类加载器；

过程如下：

|  |
| --- |
| 类加载过程 |

双亲委派模型的逻辑在ClassLoader类的loadClass方法中实现，代码主要逻辑如下：

|  |
| --- |
| protected Class<?> loadClass(String name, boolean resolve)  throws ClassNotFoundException {  // 1.检查类是否被加载过  Class<?> c = findLoadedClass(name);  if (c == null) {  // 2.使用父类加载器加载  try {  if (parent != null) {  c = parent.loadClass(name, false);  } else {  c = findBootstrapClassOrNull(name);  }  } catch (ClassNotFoundException e) {  // 如果加载失败，则抛出异常  }  // 3.父类加载器加载失败，使用当前类加载器加载  if (c == null) {  c = findClass(name);  }  }  // 4.解析类  if (resolve) {  resolveClass(c);  }  return c;  } |

方法首先检查是否已经加载过。若没有加载，则调用父类加载器的loadClass()方法进行类加载。若父类加载器为null，则默认使用Bootstrap类加载器作为父类加载器。如果父类加载失败，则抛出ClassNotFoundException异常，此时再调用当前类加载器的findClass()方法进行加载。

### 为什么要使用双亲委派模型？

固化类的加载次序，保证类加载层次结构的稳定性，基础类库一定是由层次较高的父类加载器进行加载，从而保证了类的唯一性，避免混乱。

# 应用案例

1. 涉及SPI（Service Provider Interface，服务提供商接口）接口调用的场景，即基础模块调用自定义模块的情况；比如：JDBC、JNDI等；
2. Servlet容器的实现，要求不同的web应用使用不同的类加载器加载，确保隔离性；
3. OSGI实现，为了实现模块的热替换，每个模块（Bundle）包含一个自己的类加载器，当需要替换模块时，将模块连同类加载器一同替换；

# 相关知识点

职责链模式

# 问题思考

# 参考

1. 《深入理解java虚拟机（第2版）》第7章 虚拟机类加载机制；
2. Tomcat9类加载器实现原理：http://tomcat.apache.org/tomcat-9.0-doc/class-loader-howto.html