类加载的时机

Java类从被加载到虚拟机内存中开始，到卸载出内存，整个生命周期包括：　　加载（Loading）、验证（Verification）、准备(Preparation)、解析(Resolution)、初始化(Initialization)、使用(Using) 和 卸载(Unloading)七个阶段。

加载、验证、准备、初始化和卸载这5个阶段的顺序是确定的，解析和卸载不确定，通常会在一个阶段执行的过程中调用或激活另外一个阶段。

1、类加载时机

虚拟机的具体实现来自由把握。

2、类初始化时机

虚拟机规范指明有且只有五种情况必须立即对类进行初始化（加载、验证、准备之后）：

1）遇到new、getstatic、putstatic或invokestatic这四条字节码指令（注意，new array指令触发的只是数组类型本身的初始化，而不会导致其相关类型的初始化。eg.`private InnerClass[] arrays = new InnerClass[5];`只是声明了arrays变量为一个长度为5类型为InnerClass的数组，每个对象的值都为null。需`private InnerClass[] arrays = {new InnerClass(), new InnerClass(), new InnerClass(), new InnerClass(), new InnerClass()};`）时，如果类没有进行过初始化，则需要先对其进行初始化。生成这四条指令的最常见的Java代码场景是：

- 使用new关键字实例化对象；

- 读取或设置一个类的静态字段（例外：被final修饰，已被编译器放入常量池的静态字段）；

- 调用类的静态方法的时候。

JVM中有4条invoke-*指令：invokevirtual、invokeinterface、invokestatic、invokespecial，其中前两条指令有多态语义而后两条没有。 ：

- invokestatic - 用于调用类（静态）方法
- invokespecial - 用于调用实例方法，特化于super方法调用、private方法调用与构造器调用
- **invokevirtual - 用于调用一般实例方法（包括声明为final但不为private的实例方法）**
- **invokeinterface - 用于调用接口方法**

2）使用java.lang.reflect包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。

3）当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。

4）当虚拟机启动时，用户需要指定一个要执行的主类（包含main()方法的那个类），虚拟机会先初始化这个主类。

5）当使用jdk1.7动态语言支持时，如果一个java.lang.invoke.MethodHandle实例最后的解析结果REF_getstatic,REF_putstatic,REF_invokeStatic的方法句柄，并且这个方法句柄所对应的类没有进行初始化，则需要先出触发其初始化。

这五种场景中的行为称为对一个类进行 主动引用。除此之外，所有引用类的方式，都不会触发初始化，称为 被动引用。被动引用的几种经典场景：

1)、通过子类引用父类的静态字段，不会导致子类初始化。

```
public class SSClass{
    static{
        System.out.println("SSClass");
    }
}

public class SClass extends SSClass{
    static{
        System.out.println("SClass init!");
    }

    public static int value = 123;

    public SClass(){
        System.out.println("init SClass");
    }
}

public class SubClass extends SClass{
    static{
        System.out.println("SubClass init");
    }

    static int a;

    public SubClass(){
        System.out.println("init SubClass");
    }
}

public class NotInitialization{
    public static void main(String[] args){
        System.out.println(SubClass.value);
    }
}/* Output:
        SSClass
        SClass init!
        123
 *///:~
```

对于静态字段，只有直接定义这个字段的类才会被初始化，因此通过其子类来引用父类中定义的静态字段，只会触发父类的初始化而不

会触发子类的初始化。

2)、通过数组定义来引用类，不会触发此类的初始化

```
public class NotInitialization{
    public static void main(String[] args){
        SClass[] sca = new SClass[10];
    }
}
```

运行之后并没有任何输出，说明虚拟机并没有初始化类SClass。但是，这段代码触发了另外一个名为[Lcn.edu.tju.rico.SClass的类的初始化。从类名称我们可以看出，这个类代表了元素类型为SClass的一维数组，它是由虚拟机自动生成的，直接继承于Object的子类，创建动作由字节码指令newarray触发。

3)、常量在编译阶段会存入调用类的常量池中，本质上并没有直接引用到定义常量的类，因此不会触发定义常量的类的初始化

```
public class ConstClass{
    static{
        System.out.println("ConstClass init!");
    }

    public static  final String CONSTANT = "hello world";
}

public class NotInitialization{
    public static void main(String[] args){
        System.out.println(ConstClass.CONSTANT);
    }
}
```

output: “hello world”

虽然在Java源码中引用了ConstClass类中的常量CONSTANT，但是编译阶段将此常量的值“hello world”**存储到了NotInitialization常量池**中，对常量ConstClass.CONSTANT的引用实际都被**转化为NotInitialization类对自身常量池的引用**了。也就是说，实际上NotInitialization的Class文件之中并没有ConstClass类的符号引用入口，这两个类在编译为Class文件之后就不存在关系了。

三. 类加载过程

1、加载（Loading）

虚拟机需要完成以下三件事情：

(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（并没有指明要从一个Class文件中获取，可以从其他渠道，譬如：网络、动态生成、数据库等。 由ClassLoader完成）；

(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；（xf.静态结构应该在编译阶段已形成）

(3). 在内存中(对于HotSpot虚拟就而言就是方法区)生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口；

加载阶段和连接阶段（Linking）的部分内容（如一部分字节码文件格式验证动作）是交叉进行的，加载阶段尚未完成，连接阶段可能已经开始，但这些夹在加载阶段之中进行的动作，仍然属于连接阶段的内容，这两个阶段的开始时间仍然保持着固定的先后顺序。

2、验证（Verification）

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。大致完成四阶段验证：

- 文件格式验证：验证字节流是否符合Class文件格式的规范(例如，是否以魔术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)

- 元数据验证：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(eg：这个类是否有父类，除了java.lang.Object之外)；

- 字节码验证：通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的;

- 符号引用验证：确保解析动作能正确执行。

验证阶段是非常重要的，但不是必须的，它对程序运行期没有影响。如果所引用的类经过反复验证，那么可以考虑采用-Xverifynone参数来关闭大部分的类验证措施，以缩短虚拟机类加载的时间。

3、准备(Preparation)

正式为类变量(static 成员变量)分配内存并设置类变量初始值（零值，这个时候未执行任何java方法。赋值动作将在初始化阶段才会执行。“特殊情况”：当类字段的字段属性是ConstantValue（final）时，会在准备阶段初始化为指定的值。编译器把他们当作值(value)而不是域(field)来对待。XF.常量在编译阶段已存入常量池，并将常量引用替换为具体值，在类的准备阶段进行赋值。），这些变量所使用的内存都将在**方法区**中进行分配。不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在**堆**中。

4、解析(Resolution)

虚拟机将常量池内的符号引用替换为直接引用的过程。解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行。

1.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，只要使用时能够无歧义的定位到目标即可。

eg.在Class文件中它以CONSTANT_Class_info、CONSTANT_Fieldref_info、CONSTANT_Methodref_info等类型的常量出现。

符号引用与虚拟机的内存布局无关，引用的目标并不一定加载到内存中。在Java中，一个java类将会编译成一个class文件。在编译时，java类并不知道所引用的类的实际地址，因此只能使用符号引用来代替。

eg.org.simple.People类引用了org.simple.Language类，在编译时People类并不知道Language类的实际内存地址，因此只能使用符号org.simple.Language （假设是这个，当然实际中是由类似于CONSTANT_Class_info的常量来表示的）来表示Language类的地址。各种虚拟机实现的内存布局可能有所不同，但是它们能接受的符号引用都是一致的，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。

2.直接引用：

（1）直接指向目标的指针（比如，指向“类型”【Class对象】、类变量、类方法的直接引用可能是指向方法区的指针）

（2）相对偏移量（比如，指向实例变量、实例方法的直接引用都是偏移量）

（3）一个能间接定位到目标的句柄

直接引用是和虚拟机的布局相关的，同一个符号引用在不同的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被加载入内存中了。(经历了类的准备阶段)

5、初始化(Initialization)

类加载过程的最后一步。在前面的类加载过程中，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的java程序代码(字节码)。

初始化阶段是执行类构造器<clinit>()方法的过程。<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（类构造器本质），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可以赋值，但是不能访问。**

类构造器<clinit>()与实例构造器<init>()不同，它不需要程序员进行显式调用，虚拟机会保证在子类类构造器<clinit>()执行之前，父类的类构造<clinit>()执行完毕。由于父类的构造器<clinit>()先执行，也就意味着父类中定义的静态语句块/静态变量的初始化要优先于子类的静态语句块/静态变量的初始化执行。特别地，类构造器<clinit>()对于类或者接口来说并不是必需的，**如果一个类中没有静态语句块，也没有对类变量的赋值操作，那么编译器可以不为这个类生产类构造器<clinit>()。**

多线程：虚拟机会保证一个类的类构造器<clinit>()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，那么只会有一个线程去执行这个类的类构造器<clinit>()，其他线程都需要阻塞等待，直到活动线程执行<clinit>()方法完毕。

6、卸载(Unloading)

类卸载满足的条件：

- 该类所有的实例都已经被回收，也就是java堆中不存在该类的任何实例。
- 加载该类的ClassLoader已经被回收。
- 该类对应的java.lang.Class对象没有任何地方被引用，无法在任何地方通过反射访问该类的方法。

创建一个对象：父类的类构造器<clinit>() -> 子类的类构造器<clinit>() -> 父类的成员变量和实例代码块 -> 父类的构造函数 -> 子类的成员变量和实例代码块 -> 子类的构造函数。