链滴

JVM 类加载流程

作者：AlanSune

原文链接：https://ld246.com/article/1591531796335

来源网站：链滴

在Java语言里，编译时并不进行链接工作，类型的加载、链接和初始化工作都是在Java虚拟机执行过中进行的。在Java程序启动时，Java虚拟机通过加载指定的类，然后调用该类的main方法而启动。在VM启动过程中，外部class字节码文件会经过一系列的过程转化为JVM中执行的数据。这一系列的过我们称为类加载过程。

类加载整体流程

从类被JVM加载到内存开始，到卸载出内存为止，整个生命周期包括：加载、链接、初始化、使用和载五个过程。其中链接又包括验证、准备和解析三个过程。

在整个生命周期中，加载、验证、准备、初始化和卸载五个过程的启动顺序是确定的，而解析过程在Jva规范中并没有强制规定。这几个过程**启动**顺序确定，但是**执行**顺序并不是依次进行，其中有部分工是交叉进行的，在后面的详情中会进行详细解释。

类加载的时机

Java虚拟机规范并没有对何时进行类加载过程中的第一个步骤**加载**进行强制约束，那类加载的起点如确定？Java虚拟机规范通过对**初始化**阶段进行严格规定，来保证**初始化**的完成，而作为其之前必须启的过程，加载、验证、准备当然也需要在此之前开始。

Java虚拟机规定，**有且只有**以下五种情况时，必须立即对类进行初始化：

- 虚拟机在用户指定包含main方法的主类后启动时，必须先对主类进行初始化
- 当使用 new 关键字对类进行实例化时、读取或者写入类的静态字段时、调用类的静态方法时，必先触发对该类的实例化
- 使用反射对类进行反射调用时，如果该类没有初始化，必须先触发其初始化
- 初始化一个类，而该类父类还未初始化时，需要先对其父类进行初始化
- 在JDK7之后的版本中使用动态语言支持，java.lang.invoke.MethodHandle实例解析的结果是REF_etStatic、REF_putStatic、REF_invokeStatic的方法句柄，而该句柄对应的类还未初始化时，必须先

发其实例化

接口的初始化与类基本相同，唯一不同的是当一个接口初始化时，并不要求其父接口必须初始化，只真正使用父接口时才会初始化。

类加载的过程

下面详细介绍下类加载过程中加载、验证、准备、解析、初始化的具体动作。

加载

在加载阶段，虚拟机需要完成以下3件事情：

- 通过一个类的全限定名来获取此类的class字节码二进制流
- 将这个字节码二进制流中的静态存储结构转化为方法区中的运行时数据结构
- 在内存中生成一个代表该类的java.lang.Class对象，作为方法区中这个类的各种数据的访问入口

对于Class对象，Java虚拟机规范并没有规定是存储在Java堆中，HotSpot虚拟机将其存放在方法区。

验证

验证作为链接过程中的第一步，大致会完成以下4个阶段的检验动作：

- 文件格式验证

该阶段主要在字节流转化为方法区中的运行时数据时，负责检查字节流是否符合Class文件的规范，证其可以正确的被解析并存储于方法区中。后面的检查都是基于方法区的存储结构进行检验，不会再接操作字节流。

- 元数据验证

该阶段负责分析存储于方法区的结构是否符合Java语言规范的要求，如该类是否继承了不允许继承的（被final修饰的类）、是否包含父类等。此阶段进行数据类型的校验，保证符合不存在非法的元数据息。

- 字节码验证

元数据验证保证了字节码中的数据类型符合语言的规范，该阶段则负责分析数据流和控制流，确定方体的合法性，保证被校验的方法在运行时不会危害虚拟机的运行。

- 符号引用验证

最后一个阶段发生在链接的解析阶段。在解析阶段，会将虚拟机中的符号引用转化为直接引用，该阶则负责对各种符号引用进行匹配性校验，保证外部依赖真实存在，并且符合外部依赖类、字段、方法访问性。

准备

准备阶段正式为类的字段变量分配内存，并设置初始值。这些变量存储于方法区中，注意此处的变量为

类变量（被static修饰符修饰），而非实例变量。Java中数据类型的初始值见下表。

数据类型	初始值
boolean	false
byte	(byte) 0
char	\u0000
short	(short) 0
int	0
long	0L
float	0F
double	0D
reference	null

当类字段为常量类型时（即被static final修饰），由于字段的值已经确定，并不会在后面修改，此时直接赋值为指定的值。如下面的变量value，将直接赋值为1。

```
public static final int value = 1;
```

解析

解析阶段将常量池中的符号引用替换为直接引用。在字节码文件中，类、接口、字段、方法等类型都由一组符号来表示，其形式由Java虚拟机规范中的Class文件格式定义。在虚拟机执行特定指令之前需要将符号引用转化为目标的指针、相对偏移量或者句柄，这样可以通过此类直接引用在内存中定位用的具体位置。

初始化

在类的class文件中，包含两个特殊的方法：<clinit>
<init>。这两个方法由编译器自动生成，分别代表类构造器和构造函数。其中构造函数可以由变人员实现，而类构造器则由编译器自动生成。而初始化阶段则负责调用类构造器，来初始化变量和资。

<clinit>方法由编译器自动收集类的赋值动作和静态语句块（static{}块）中的语句合并生成的它有以下特点：

● 编译器收集的顺序由源文件中语句的顺序决定，静态语句块只能访问到在它之前定义的变量，在它后定义的变量，它只能进行赋值操作，但不能访问。

```
public class Test {
    static {
        // 变量赋值编译可以正常通过
        value = 2;
        // 访问变量编译失败
        System.out.println(i);
    }

    static int value = 1;
}
```

● 虚拟机保证在子类的<clinit>方法执行之前，父类的<clinit>
法已经执行完毕。因此父类中的操作对于子类都是可见的。

● 接口的<clinit>方法执行之前，不需要先执行父接口的<clinit>
法，只有父接口中定义的变量被使用时，父接口才会初始化。同时接口的实现类在初始化时也不会执
父接口的<clinit>方法。

● <clinit>方法不是必须的，如果一个类或者接口没有变量赋值和静态语句块，则编译器
以不生成<clinit>方法。

● 虚拟机会保证<clinit>方法在多线程中被正确的加锁、同步。如果多个线
同时去初始化一个类，那么只有一个线程去执行<clinit>
法，其他线程会被阻塞。

总结

正确掌握类加载的过程，可以对平常编程中的各种问题有更深入的了解。下面通过一个违背感觉的实
，来感受下掌握类加载过程的重要性。

```
class SingleTon {
    private static SingleTon singleTon = new SingleTon();
    public static int count1;
    public static int count2 = 0;

    private SingleTon() {
        count1++;
        count2++;
    }

    public static SingleTon getInstance() {
        return singleTon;
    }
}

public class Test {
    public static void main(String[] args) {
        SingleTon singleTon = SingleTon.getInstance();
        System.out.println("count1=" + singleTon.count1);
        System.out.println("count2=" + singleTon.count2);
    }
}
```

上面代码的执行结果为：

```
count1=1
count2=0
```

按照初始化的触发条件，我们可知在main方法调用getInstance时，会进行SingleTon的类加载过程。

在准备阶段，会对变量添加初始值，此时singleTon=null，count1=0，count2=0。

在初始化阶段，先执行new SingleTon()，此时count1=count2=1，然后由于count1无赋值操作，
以count1=1。count2赋值为0，所以结果为count1=1，count2=0。

假如将**private static SingleTon singleTon = new SingleTon();**语句移动到**public static int count

= 0;**之后，则结果为count1=1，count2=1。