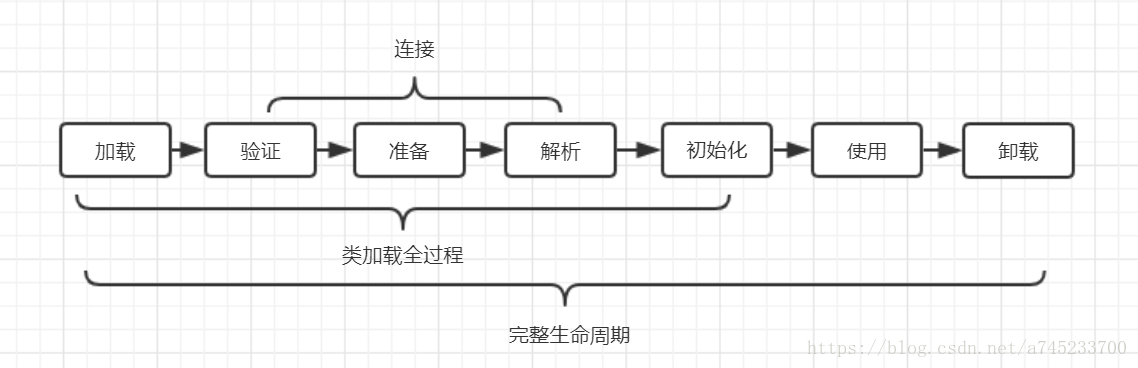
## Java虚拟机：类加载机制与双亲委派模型

### 一、类加载机制：

**.java**文件中的代码在编译后，就会生成JVM能够识别的二进制字节流class文件，class文件中描述的各种信息，都需要加载到虚拟机中才能被运行和使用。

**类加载机制**，就是虚拟机把类的数据从class文件加载到内存，并对数据进行校检，转换解析和初始化，最终形成可以被虚拟机直接使用的Java类型的过程。

类从加载到虚拟机内存开始，到卸载出内存结束，整个生命周期包括七个阶段，如下图（每个过程作用见文章第四部分）：



二、类加载器：

生命周期的第一阶段，即加载阶段需要由类加载器来完成的，类加载器根据一个类的全限定名读取类的二进制字节流到JVM中，然后生成对应的java.lang.Class对象实例，

在虚拟机默认提供了3种类加载器，启动类加载器（Bootstrap ClassLoader）、扩展类加载器（Extension ClassLoader）、应用类加载器（Application ClassLoader），如果有必要还可以加入自己定义的类加载器。

（1）启动类加载器（Bootstrap ClassLoader）：负责加载 在<JAVA\_HOME>\lib目录 和 被-Xbootclasspath参数所指定的路径中的类库

（2）扩展类加载器（Extension ClassLoader）：负责加载 <JAVA\_HOME>\lib\ext目录 和 被java.ext.dirs系统变量所指定的路径中的所有类库

（3）应用程序类加载器（Application ClassLoader）：负责加载用户类路径classPath所指定的类库，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

（4）自定义加载器（CustomClassLoader）：属于应用程序根据自身需要自定义的ClassLoader，如tomcat、jboss都会根据j2ee规范自行实现。

任意一个类在JVM中的唯一性，是由加载它的类加载器和类的全限定名一共同确定的。因此，比较两个类是否“相等”的前提是这两个类是由同一个类加载器加载的，否则，即使两个类来源于同一个Class文件，被同一个虚拟机加载，只要加载他们的类加载器不同，那这两个类就必定不相等。JVM的类加载机制，规定一个类有且只有一个类加载器对它进行加载。而如何保证这个只有一个类加载器对它进行加载呢？则是由双亲委派模型来实现的。

**三、双亲委派模型：**

双亲委派模型要求除了顶层的启动类加载器外，其余类加载器都应该有自己的父类加载器。（类加载器之间的父子关系不是以继承的关系实现，而是使用组合关系来复用父加载器的代码）

**1、双亲委派模型的工作原理：**

如果一个类加载器收到了类加载的请求，他首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层级的类加载器都是如此，因此所有请求最终都会被传到最顶层的启动类加载器中，只有当父类加载器反馈自己无法完成这个加载请求时，子加载器才会尝试自己去加载。

因此，加载过程可以看成自底向上检查类是否已经加载，然后自顶向下加载类。

2、双亲委派模型的优点：

（1）使用双亲委派模型来组织类加载器之间的关系，Java类随着它的类加载器一起具备了一种带有优先级的层次关系。

（2）避免类的重复加载，当父类加载器已经加载了该类时，子类加载器就没必要再加载一次。

（3）解决各个类加载器的基础类的统一问题，越基础的类由越上层的加载器进行加载。避免Java核心API中的类被随意替换，规避风险，防止核心API库被随意篡改。

例如类java.lang.Object，它存在在rt.jar中，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的Bootstrap ClassLoader进行加载，因此Object类在程序的各种类加载器环境中都是同一个类。相反，如果没有双亲委派模型而是由各个类加载器自行加载的话，如果用户编写了一个java.lang.Object的同名类并放在ClassPath中，那系统中将会出现多个不同的Object类，程序将混乱。因此，如果开发者尝试编写一个与rt.jar类库中重名的Java类，可以正常编译，但是永远无法被加载运行。

四、类加载生命周期：

1、加载阶段：

这阶段的虚拟机需要完成三件事情：

（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。

（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）在内存中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。

2、验证阶段：

这阶段是为了确保class文件的字节流包含的信息符合当前虚拟机的要求，不会危害虚拟机自身的安全。分为4个校检动作：

（1）文件格式验证：验证字节流是否符合class文件格式的规范，并且能被当前版本的虚拟机处理，通过该阶段后，字节流会进入内存的方法区中进行储存。

（2）元数据验证：对字节码描述的信息进行语言分析，对类的元数据信息进行语义校验，确保其描述的信息符合java语言规范要求。

（3）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。这个阶段对类的方法进行校验分析，保证类的方法在运行时不会做出危害虚拟机安全的事件。

（4）符号引用验证：对类自身以外的信息（常量池中各种符号引用）的信息进行校检，确保解析动作能正常执行（该动作发生在解析阶段中）

3、准备阶段：

正式为类变量分配内存空间并设置数据类型零值。这个阶段进行内存分配的仅包括类变量（static修饰），不包括实例变量，实例变量会在对象实例化时随对象一起分配在java堆。

public static int value= 123 ; //变量value在准备阶段过后的初始值是0，不是123.

public static final int value = 123 ; //特殊情况：会生成ConstantValue属性，在准备阶段初始值是123.

final、static、static final修饰的字段赋值的区别：

（1）static修饰的字段在准备阶段被初始化为0或null等默认值，然后在初始化阶段（触发类构造器<clinit>）才会被赋予代码中设定的值，如果没有设定值，那么它的值就为默认值。

（2）static final修饰的字段在Javac时生成ConstantValue属性，在类加载的准备阶段根据ConstantValue的值为该字段赋值，它没有默认值，必须显式地赋值，否则Javac时会报错。可以理解为在编译期即把结果放入了常量池中。

（3）final修饰的字段在运行时被初始化（可以直接赋值，也可以在实例构造器中<init>()赋值），一旦赋值便不可更改。

4、解析阶段：

将常量池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、和调用限定符7类符号引用。

对同一符号引用进行多次解析请求是很常见的事情，除invokedynamic指令以外，虚拟机可以对第一次解析的结果进行缓存，从而避免解析动作重复进行。invokedynamic对应的引用称为“动态调用限定符”，必须等到程序实际运行到这条指定的时候，解析动作才能进行。因此，当碰到由前面的invokedynamic指令触发过的解析的符号引用时，并不意味着这个解析结果对其他的invokedynamic指令也同样生效。

  （1）符号引用：符号引用是以一组符号来描述所引用的目标，符号可以是任何字面量，只要使用时无歧义定位到目标即可。符号引用与虚拟机的内存布局无关，引用的目标并不一定已经加载到内存中。各种虚拟机实现的内存布局可以不相同，但是他们能接受的符号引用必须都是一致的，符号引用的字面量形式明确地定义在java虚拟机规范的calss文件格式中。

  （2）直接引用：直接引用是可以直接定位到目标的指针、相对偏移量或是一个能间接定位目标的句柄。直接引用是与虚拟机的内存布局相关的，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，那引用的目标必定已经在内存中存在。

5、初始化：

初始化阶段才真正执行类中定义的java代码。执行类构造器<clinit>()方法的过程，并按程序的设置去初始类变量和其他资源。

5.1、类的主动引用：

在初始化阶段，有且只有5种场景必须立即对类进行“初始化”，称为主动引用：

    （1）遇到new、getstatic、putstatic、invokestatic这4条指定时。对应的场景是：使用new关键字实例化对象的时候，读取或设置一个类的静态字段（被final修饰、已经在编译期把结果放入常量池的静态字段除外），以及调用一个类的静态方法的时候。

    （2）使用java.lang.reflect包的方法对类进行发射调用的时候。

    （3）当初始化一个类的时候，如果发现其父类还没进行初始化，则必须对父类进行初始化。（与接口的区别：接口在初始化时，并不要求其父接口全部都完成了初始化，只有在真正使用到父接口的时候，才会初始化）

    （4）当虚拟机启动时，用户指定的要执行的主类（包含main方法的类）。

    （5）java.lang.invoke.MethodHandle实例最后的解析结果REF\_getStatic、REF\_putStatic、REF\_invokeStatic的方法句柄，并且这个句柄所对应的类没有进行过初始化，则需要触发其初始化。

5.2、类的被动引用：

除了主动引用，其他引用类的方式都不会触发初始化，称为被动引用：

    （1）对于静态字段，只有直接定义这个字段的类才会被初始化，通过其子类来引用父类中定义的静态字段，只会触发其父类的初始化而不会触发子类的初始化。

//父类

public class SuperClass {

//静态变量value

public static int value =123456;

//静态块，父类初始化时会调用

static{

System.out.println("父类初始化！");

}

}

//子类

public class SubClass extends SuperClass{

//静态块，子类初始化时会调用

static{

System.out.println("子类初始化！");

}

}

//主类、测试类

public class InitTest {

public static void main(String[] args){

System.out.println(SubClass.value); //输出结果：父类初始化！ 123456

}

}

    （2）通过数组定义来引用类，不会触发此类的初始化。

//父类

public class SuperClass {

//静态变量value

public static int value = 123456;

//静态块，父类初始化时会调用

static{

System.out.println("父类初始化！");

}

}

//主类、测试类

public class InitTest {

public static void main(String[] args){

SuperClass[] test = new SuperClass[10]; //输出结果：没有任何输出结果

}

}

    （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。

//定义常量的类

public class ConstClass {

static{

System.out.println("常量类初始化！");

}

public static final String HELLOWORLD = "hello world!";

}

//主类、测试类

public class InitTest {

public static void main(String[] args){

System.out.println(ConstClass.HELLOWORLD); //输出结果：hello world

}

}

5.3、<clinit>()方法的特点：

   （1）<clinit>()方法是有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中的语句合并产生的，收集的顺序是由语句在源文件中出现的顺序决定的。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，但是不可以访问。

public class Test {

static{

i=0; //给变量赋值可以正常编译通过

System.out.println(i); //编译器会提示非法向前引用

}

static int i=1;

}

    （2）<clinit>()方法与实例构造器<init>()方法不同，它不需要显示调用父类构造器，虚拟机会保证子类的<clinit>()方法执行之前，父类的<clinit>()方法已经执行完毕，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，虚拟机中第一个被执行的<clinit>()方法的类是java.lang.Object。

    （3）<clinit>()方法对于类或接口并不是必需的，如果一个类中没有静态语句块，也就没有对变量的赋值操作，那么编译器可以不为这个类生成<clinit>()方法。

    （4）接口中不能使用静态语句块，仍然有变量初始化操作，因此仍然会生成<clinit>()方法，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。只有父接口中定义的变量被使用是，才需要初始化父接口，同时，接口实现类在初始化时也不会执行接口的<clinit>()方法。

    （5）<clinit>()方法在多线程环境中被正确的加锁、同步，多个线程同时去初始化一个类，只会有一个线程执行<clinit>()方法，其他线程则需要阻塞等待，直到活动线程执行<clinit>()方法完毕，活动线程执行完毕后，其他线程唤醒后被不会再次进入<clinit>()方法，因为同一个类加载器下，一个类型只会被初始化一次。

————————————————

版权声明：本文为CSDN博主「张维鹏」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/a745233700/article/details/90232862