Java中类加载

1. 何时进行类加载？

创建类的实例；访问类的静态变量（除常量外）；访问为原静态方法；反射（如Class.forName()）；当初始化一个类时，发现其父类还未初始化，则先进行父类的初始化；虚拟机启动时，定义了main()方法的那个类先初始化。

以上情况称为对一个类进行“主动引用”，除此种情况这外，均不会触发类的初始化，称为“被动引用”。

接口的加载过程与类的加载过程稍有不同。接口中不能使用static{}块。当一个接口在初化时，并不要求其父接口全部完成了初始化，只有真正在使用到父接口时（例如引用接口中定义的常量）才会初始化。

2.被动引用的例子?

子类调用父类的静态变量，子类不会被初始化。只有父类被初始化。对于静态字段，只有直接定义这个字段的类才会被初始化。

通过数组定义来引用类，不会触发类的初始化。

访问类的常量，不会初始化类。

3.类在初始化的时候会为类的静态变量赋值和执行静态代码块。

4.类的加载过程。

1.加载：加载（loading）阶段是“类加载过程的第一个阶段，在这个阶段需要完成三步：

通过一个类的全限定名来获取定义此类的二进制字节流；

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。

在java堆中生成一个这个类的java.lang.Class对象，作为方法区这些数据的访问入口。

加载阶段即可以使用系统提供的类加载器在完成，也可以由用户自定义的类加载器来完成。加载阶段与连接阶段的部分内容（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。

2.验证阶段：验证阶段是连接阶段的第一步，这一阶段的目的是为了确保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。

大致会包含四个阶段：文件格式验证、元数据验证、字节码验证、符号引用验证。

3.准备

准备阶段是为类的静态变量分配内存并将其初始化为默认值，这些内存都将在方法区中进行分配。准备阶段不分配类中的实例变量和内存，实例变量将会在对象在实例化时随着对象一起分配在Java堆 中。

4.解析

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。

5.初始化

类初始化是类加载过程最后一步，前面的类加载过程，除了在加载阶段用户应用程序可以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。

初始化阶段就是执行类构造器方法的过程。构造器方法由编译器自动收集类中的所有变量的赋值动作和静态语句块中的语句合并产生的。

Java中JVM的符号引用和直接引用

在jvm中类加载过程中，在解析阶段会把类的二进制数据中的符号引用转化为直接引用。

1.符号引用：符号引用是用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。例如，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去。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，因此只有使用符号引用来代替。比如org.sinmple.People类引用了org.simple.Language类，在编译时People类并不知道Language类的实际的内存地址，因此只能使用符号org.simple.Language（假设是这个，当然实际是由类似于CONSTANT\_Class\_info的常量来表示的）来表示Language的内存地址。各种虚拟机实现的内存布局可能有所不同，但是它们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、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到内存中了。

**常量在编译阶段会存入调用它的类的常量池中，本质上没有引用到定义该常量的类，因此不会触发定义常量的类的初始化。**

**类的初始化是由虚拟机自己决定的，但是有5种情况下必须马上对类进行初始化动作。**

**Static final常量在编译期就将其结果放入了调用它的类的常量池中。**

**对于一个符号引用进行多次解析请求时是很常见的事情，虚拟机实现可能会对第一次解析的结果进行缓存（在运行时常量池中记录直接引用，并把常量标示为已解析状态），从而避免解析动作的重复进行。**

**Static变量发生在静态解析阶段，也就是在初始化之前，此时已经将字段的符号引用转化为了内存引用，也便将它与对应的类关联在了一起；如果有一个同名字段同时出现在该类的接口和父类中，或同时在自己或父类的接口中出现，编译器可能会拒绝编译。**

**类的初始化：**

**1.<client>()方法由编译器自动收集类中的所有的类变量的赋值动作静态语句块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进行赋值，但是不可以进行访问。**

**2.<client>()方法与实例构造器方法不同，它不需要显示的调用父类构造器，虚心机会保证在子类的<client>()方法执行之前，父类的<client>()方法已经执行完毕。**

**3.<client>方法对于类和接口不是必须的，如果一个类中没有静态代码块，也没有对类变量赋值的动作，那么编译器可以不为这个类生成<client>方法。**

**4.接口中不能静态代码块，但是仍然有类变量的赋值动作，因此类与接口一样会生成<client>()方法。但是接口与类不同的是，执行接口的<client>()方法不需要先执行父类的<client>()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。另外，接口的实现类在初始化时也一样不会先执行接口的<client>()方法。**

**5.虚拟机会保证一个类的<client>()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<client>()方法，其它线程都需要阻塞等待，直到活动线程执行<client>()方法执行完毕。**

**Java中的泛型实际上只是在程序源码中存在，在编译后的字节码文件中，就已经被替换成了原来的原生类型，并且在相应的地方插入了强制转型代码。**

**Byte、Short、Integer、Long、Charcter、Boolean都实现了常量池技术。**