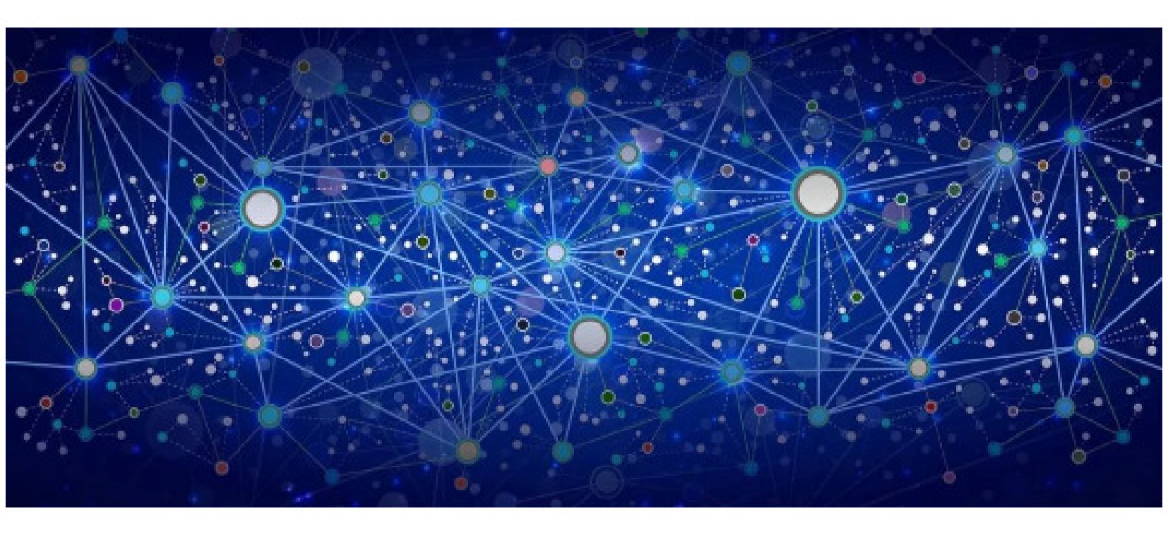
面试官：你知道java类是怎么跑起来的吗？问的我一脸懵

程序员面试 5月7日

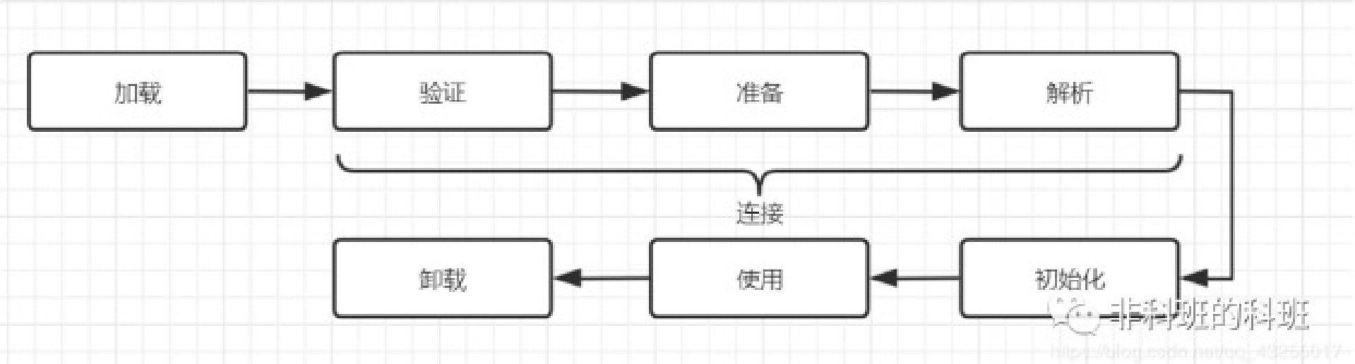
微信扫一扫
关注该公众号

以下文章来源于非科班的科班，作者非科班的科班

非科班的科班
世界上并没有什么救世主，假如有那便是你自己；世界上也没有什么奇迹，假如有那只…



类从加载虚拟机内存中开始到卸载出内存为止，生命周期包括：**加载、验证、准备、解析、初始化、使用、卸载。**



加载、验证、准备、初始化和**卸载**这5个阶段的顺序是**确定的**，类的加载过程必须按照这种顺序进行，而**解析阶段**则不一定，它在某些情况下可能在初始化阶段后在开始，因为java支持**运行时绑定。**

加载阶段

通过一个类的全限定名来获取定义此类的**二进制字节流**（没有指明二进制字节流要从一个**Class文件**中获取，可以从**ZIP包**中读取，从**网络**中获取，**运行时计算**生成等等）

然后，将这个字节流所代表的静态储存结构转化为方法区的运行时数据结构在内存中生成一个代表这个类的 `java.lang.Class` 对象，也就是说，当程序中使用任何类时，系统都会为之建立一个java.lang.Class对象。

该Class对象作为方法区这个类的各种数据的访问入口完成后，虚拟机外部的二进制字节流就按照虚拟机所需格式储存在方法区中。

这里稍微理解一下对象和类的概念，对象是实例化的类。类的信息是存储在方法区中的，对象是存储在Java堆中的。类是对象的模板，对象是类的实例。

类的加载由**类加载器**完成，类加载器通常由JVM提供，这些类加载器也是前面所有程序运行的基础，JVM提供的这些类加载器通常被称为**系统类加载器**。除此之外，开发者可以通过继承ClassLoader基类来创建自己的类加载器。

其实加载阶段用一句话来说就是：把代码数据加载到内存中。这个过程对于我们解答这道问题没有直接的关系，但这是类加载机制的一个过程。

加载阶段，java虚拟机规范中没有进行约束，但初始化阶段，java虚拟机严格规定了如下5种情况必须立即进行初始化（初始化前，必须经过加载、验证、准备阶段）。

1. 创建类的实例，也就是new一个对象
2. 访问某个类或接口的静态变量，或者对该静态变量赋值
3. 调用类的静态方法
4. 反射（Class.forName("com.lyj.load")）
5. 初始化一个类的子类（会首先初始化子类的父类）
6. JVM启动时标明的启动类，即文件名和类名相同的那个类

   除此之外，对于一个 `final` 类型的静态变量，如果该变量的值在编译时就可以确定下来，那么这个变量相当于**宏变量**。Java编译器会在编译时直接把这个变量出现的地方替换成它的值，因此即使程序使用该静态变量，也不会导致该类的初始化。

反之，如果final类型的静态Field的值不能在编译时确定下来，则必须等到运行时才可以确定该变量的值，如果通过该类来访问它的静态变量，则会导致该类被初始化。

验证阶段

确保被加载的**类的正确性**，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。

检验被加载的类是否有正确的内部结构，并和其他类协调一致。Java是相对C++语言是安全的语言，例如它有C++不具有的数组越界的检查。这本身就是对自身安全的一种保护。

验证阶段是Java非常重要的一个阶段，它会直接的保证应用是否会被恶意入侵的一道重要的防线，越是严谨的验证机制越安全。

验证的目的在于确保Class文件的字节流中包含信息符合当前虚拟机要求，不会危害**虚拟机自身安全**。其主要包括四种验证，**文件格式验证**，**元数据验证**，**字节码验证**，**符号引用验证**。

- 文件格式验证：验证字节流是否符合Class文件格式的规范，如：是否以模数0xCAFEBABE开头、主次版本号是否在当前虚拟机处理范围内等等。
- 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。
- 符号引用验证：确保解析动作能正确执行；如：通过符合引用能找到对应的类和方法，符号引用中类、属性、方法的访问性是否能被当前类访问等等。

准备阶段

为类的静态变量分配内存，并将其赋默认值，为类变量分配内存并设置类变量初始值，这些内存都将在方法区中分配。对于该阶段有以下几点需要注意：

- 只对 `static` 修饰的静态变量进行**内存分配**、赋默认值（如0、0L、null、false等）。
- 对 `final` 的静态字面值常量**直接赋初值**（赋初值不是赋默认值，如果不是字面值静态常量，那么会和静态变量一样赋默认值）。

两个关键点，即**内存分配的对象**以及**初始化的类型**。

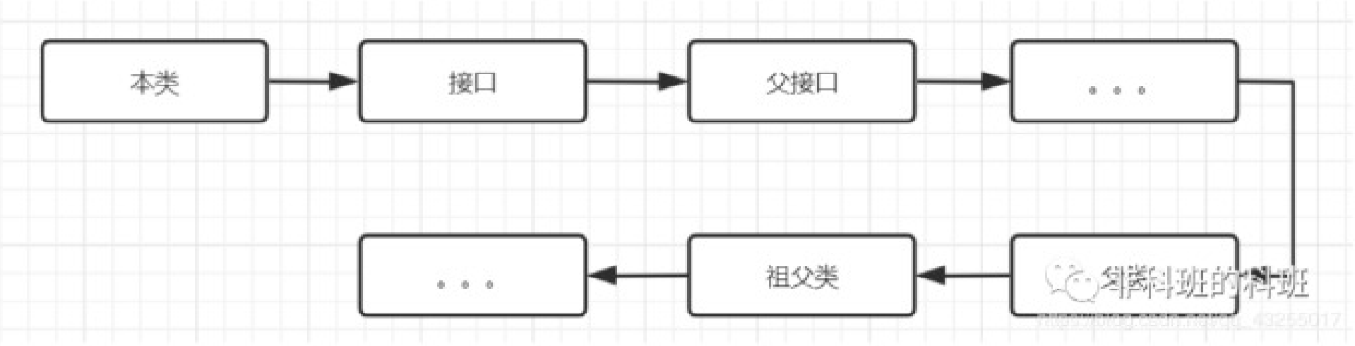
- 内存分配的对象。Java 中的变量有**类变量**和**类成员变量**两种类型，**类变量**指的是被 `static` 修饰的变量，而其他所有类型的变量都属于类成员变量。在**准备阶段**，JVM只会为**类变量**分配内存，而不会为**类成员变量**分配内存。**类成员变量**的内存分配需要等到**初始化阶段**才开始。
- 初始化的类型。在**准备阶段**，JVM 会为类变量分配内存，并为其初始化。但是这里的初始化指的是为变量赋予 Java 语言中该数据类型的**零值**，而不是用户代码里初始化的值。

解析阶段

解析阶段JVM 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 类引用进行解析。这个阶段的主要任务是将其在常量池中的**符号引用**替换成直接其在内存中的**直接引用。**

类或接口的解析过程判断所要转化成的直接引用是对数组类型，还是对普通的对象类型的引用，从而进行不同的解析。

字段解析过程是对字段进行解析时，会先在本类中查找是否包含有简单名称和字段描述符都与目标相匹配的字段，如果有，则查找结束；如果没有，则会按照继承关系从上往下递归搜索该类所实现的各个接口和它们的父接口，还没有，则按照继承关系从上往下递归搜索其父类，直至查找结束，查找流程如下图所示：



将类的二进制数据中的符号引用替换成直接引用。说明一下：符号引用：符号引用是以一组符号来描述所引用的目标，符号可以是任何的字面形式的字面量，只要不会出现冲突能够定位到就行。

布局和内存无关。直接引用：是指向目标的指针，偏移量或者能够直接定位的句柄。该引用是和内存中的布局有关的，并且一定加载进来的。

初始化阶段

初始化是为类的静态变量赋予正确的初始值，**准备阶段**和**初始化阶段**看似有点矛盾，其实是不矛盾的。

如果类中有语句：`private static int a = 10`，它的执行过程是这样的，首先字节码文件被加载到内存后，先进行链接的验证这一步骤，验证通过后准备阶段，给a分配内存，因为变量a是static的。

所以，此时a等于int类型的默认初始值0，即a=0,然后到解析（后面在说），到初始化这一步骤时，才把a的真正的值10赋给a,此时a=10。

赋初值两种方式：

1. 定义静态变量时指定初始值。如 private static String x="123";
2. 在静态代码块里为静态变量赋值。如 static{ x="123"; }

初始化阶段，才真正开始执行类中的Java程序代码。即**初始化阶段**是执行类构造器 `clinit` 方法的过程。

在编译生成class文件时，编译器会产生两个方法加于class文件中，一个是类的初始化方法 `clinit`，另一个是实例的初始化方法 `init`。

1. `clinit` 指的是**类构造器**，主要作用是在类加载过程中的**初始化阶段**进行执行，执行内容包括**静态变量初始化**和**静态块**的执行。
2. `init` 指的是**实例构造器**，主要作用是在**类实例化过程**中执行，执行内容包括**成员变量初始化**和**代码块**的执行。

使用阶段

当 JVM 完成初始化阶段之后，JVM 便开始从入口方法开始执行用户的程序代码。这个阶段也只是了解一下就可以。

卸载阶段

最后卸载阶段，执行了System.exit()方法，程序正常执行结束，程序在执行过程中遇到了异常或错误而异常终止， 由于操作系统出现错误而导致Java虚拟机进程终止

类加载器

类加载器负责加载所有的类，其为所有被载入内存中的类生成一个java.lang.Class实例对象。一旦一个类被加载入JVM中，同一个类就不会被再次载入了。正如一个对象有一个唯一的标识一样，一个载入JVM的类也有一个唯一的标识。

在Java中，一个类用其**全限定类名**（包括包名和类名）作为标识；但在JVM中，一个类用其全限定类名和其类加载器作为其唯一标识。

JVM预定义有三种类加载器，当一个 JVM启动的时候，Java开始使用如下三种类加载器：

1. **引导类加载器**：这个类加载器负责将 \lib目录下的类库加载到虚拟机内存中，用来加载java的核心库，此类加载器并不继承于java.lang.ClassLoader，不能被java程序直接调用，代码是使用C++编写的。是虚拟机自身的一部分。
2. **扩展类加载器**：这个类加载器负责加载 \lib\ext目录下的类库，用来加载java的扩展库，开发者可以直接使用这个类加载器。
3. **应用程序类加载器**：这个类加载器负责加载用户类路径(CLASSPATH)下的类库，一般我们编写的java类都是由这个类加载器加载，这个类加载器是CLassLoader中的getSystemClassLoader()方法的返回值，所以也称为系统类加载器。一般情况下这就是系统默认的类加载器。

类加载器加载 `Class` 经过如下步骤：

1. 检测此Class是否载入过，即在缓冲区中是否有此Class，如果有直接进入第8步，否则进入第2步。
2. 如果没有父类加载器，则要么Parent是根类加载器，要么本身就是根类加载器，则跳到第4步，如果父类加载器存在，则进入第3步。
3. 请求使用父类加载器去载入目标类，如果载入成功则跳至第8步，否则接着执行第5步。
4. 请求使用根类加载器去载入目标类，如果载入成功则跳至第8步，否则跳至第7步。
5. 当前类加载器尝试寻找Class文件，如果找到则执行第6步，如果找不到则执行第7步。
6. 从文件中载入Class，成功后跳至第8步。
7. 抛出ClassNotFountException异常。
8. 返回对应的java.lang.Class对象。

双亲委派机制

双亲委派模型是一种组织类加载器之间关系的一种规范，它的工作原理是:如果一个类加载器收到了类加载的请求，它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。

这样层层递进，最终所有的加载请求都被传到最顶层的启动类加载器中，只有当父类加载器无法完成这个加载请求(它的搜索范围内没有找到所需的类)时，才会交给子类加载器去尝试加载。

这样的好处是，java类随着它的类加载器一起具备了带有优先级的层次关系。这是十分必要的，比如java.lang.Object，它存放在\jre\lib\rt.jar中，它是所有java类的父类，因此无论哪个类加载都要加载这个类，最终所有的加载请求都汇总到顶层的启动类加载器中。

Object类会由启动类加载器来加载，所以加载的都是同一个类，如果不使用双亲委派模型，由各个类加载器自行去加载的话，系统中就会出现不止一个Object类，出现没必要的重复加载的情况。

长按订阅更多面经分享