代码编译的结果从本地机器码转变成字节码，是存储格式的发展， 是编程语言发展的一大步。

在class文件中描述的各种信息， 只有加载到JVM之中后才能运行和使用 。 JVM把描述类的class文件加载到内存， 并对数据进行校验， 转换解析 和初始化，最终可以形成被JVM直接使用的Java类型， 这就是虚拟机的加载机制 。 Java类加载都是动态运行时候进行的，尽管降低了一点点性能， 但是可以在真正运行时才制定实际运行的类， 极大地提高了灵活度。

类的生存周期： 加载 => 验证=> 准备 => 解析（连接） => 初始化 => 运行 => 卸载，按部就班的走，但是有的会交叉执行。

7.1 类加载的时机

什么时候开始类的加载过程的第一个阶段？加载。 JVM规范并没有强制规定， 这一点可以交付给具体的JVM自由把握， 但是 严格规定

当且仅当 5种情况必须立即对类初始化：

主动引用：

1） 遇到new , getstatic , putstatic , invokestatic这4条字节码指令。 常见场景为： 使用new实例化一个对象， 读取或者设置一个类的静态字段（final static 的除外， 已在编译时就放入常量池） 。

2）使用 java.lang.reflect.包中的方法对类的反射进行调用的时候 。

3）初始化子类之前必须初始化父类。

4）当JVM启动时候， 必须首先初始化那个 main()函数的那个类

5）当使用JDK1.7的动态语言支持时， 如果一个java.lang.invoke.MethodHandle实例最后的解析结果为REF-getstatic，REF-putstatic.
REF-invokestatic的方法句柄。

被动引用：

1）子类引用父类的static字段， 子类不会初始化。对于静态字段，只有直接定义这个字段的类才会初始化。

2）通过数组来定义引用类， 不会导致类的初始化。尽管如此， 但是它会触发另一个类， 这个类是JVM自动生成， 直接继承与java.lang.object的子类， 这个子类由newarray触发。

3）final static 在编译时就已经放入到类的常量池， 引用时不会触发类的初始化

接口与类的初始化最大不同在于： 父接口只有在真正使用时候才会初始化， 不会一定先于子接口之前初始化。

7. 2类的加载的过程

1）加载。记载和类的加载， 不是一个东西哈，这是类的记载的第一个阶段， 但是与连接过程交叉执行， 主要完成：

a. 从一个类的全限定名获取此类的二进制代码流（可以是zip ,jar； 可以是网络中； 可以运行中计算； 其他文件， 数据库） 。开发人员可以自己重写一个classloader()方法来定义获取控制字节流。

2） 验证 。 保证class文件的包含信息 符合 当前JVM的要求， 比如 魔数， 版本号， 文件格式； 元数据（这个类的信息， 比如是否有父类， 父类是否继承了final类， 是否实现了接口和父类的要求方法。。。。） ； 字节码， 检查程序语义是合法的， 符合逻辑的；发生在解析阶段的 符号引用验证（把符号引用转化成直接引用）。

3）准备。 为类变量（static 修饰， 但是无final）分配内存， 并且设置零值（真正赋值是在初始化阶段！），

4）解析 。 把常量池中的符号引用替换成直接引用。 注意优先级 this， super， this（super） ， super(this).....

5) 初始化 。

a. 初始化阶段执行类构造器 <clinit>() 方法（由类中的static块构成）.编译器收集顺序， 是按照源代码中 static块的顺序。

b. JVM 保证 父类的<clinit>() 方法先于子类执行。一次第一个初始化的类一定是object类。

c. <clinit>() 对于类和接口 来说， 并不是必须的，如果没有static块，那么可以JVM可以不生成<clinit>() 方法。

d. 接口中不能有static语句， 但是变量初始化的赋值操作， 因此接口也会有<clinit>() 方法， 但是与类不同， 父接口只有在使用时才
会初始化。

e.JVM保证一个类的<clinit>() 方法在多线程下正确的加锁， 同步， 如果多个线程去初始化一个类， 那么只有一个线程去执行这个类的<clinit>() 方法， 其他的线程需要阻塞等待， 这就隐含着多个进程的阻塞问题 。

7. 4类加载器

虚拟机设计团队把类加载器的加载阶段的“通过一个类的全限定名称来获取描述此类的二进制字节流”这个动作放到JVM外部去执行， 以便让程序自己决定如何去获取所需要的类 。实现这个动作的代码就叫做 类加载器。

类加载器可以说是Java语言的一项创新， 但是作用远不限于类加载阶段， 比如 类加载器和 类本身就可以唯一的确认一个类

双亲委派模型：

启动类加载器 <= 扩展类记载器 <= 应用程序类加载器 <= 自定义的用户加载器 。

工作原理： 不以继承，以组合的方式来复用父加载器的代码 。 如果一个类加载器 受到了类加载的请求， 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， 每一层的类加载器都是如此 。 因此所有的请求其实应该传到最顶层的启动类加载器 。 只有当父classloader反馈无法完成这个请求时（他的搜索范围没有找到所需要的类）， 子classloader 才会尝试自己去加载。这样使得classloader 一起具备了一种带有优先级的层次关系。