**一、Java的类加载机制回顾与总结：**

我们知道一个Java类要想运行，必须由jvm将其装载到内存中才能运行，装载的目的就是把Java字节代码转换成JVM中的java.lang.Class类的对象。这样Java就可以对该对象进行一系列操作，装载过程有两个比较重要的特征：**层次组织结构和代理模式**。**层次组织结构**指的是每个类加载器都有一个父类加载器，通过getParent()方法可以获取到。类加载器通过这种父亲-后代的方式组织在一起，形成树状层次结构。**代理模式**则指的是一个类加载器既可以自己完成Java类的定义工作，也可以代理给其它的类加载器来完成。由于代理模式的存在，启动一个类的加载过程的类加载器和最终定义这个类的类加载器可能并不是一个。ClassLoader的加载类过程主要使用loadClass方法，该方法中封装了中加载机制：**双亲委派模式**。

一般来说，父类优先的策略就足够好了。在某些情况下，可能需要采取相反的策略，即先尝试自己加载，找不到的时候再代理给父类加载器。这种做法在Java的Web容器中比较常见，也是Servlet规范推荐的做法。比如，Apache Tomcat为每个Web应用都提供一个独立的类加载器，使用的就是自己优先加载的策略。IBM WebSphere Application Server则允许Web应用选择类加载器使用的策略。  
类加载器的一个重要用途是在JVM中为相同名称的Java类创建隔离空间。在JVM中，判断两个类是否相同，不仅是根据该类的二进制名称，还需要根据两个类的定义类加载器。只有两者完全一样，才认为两个类的是相同的。因此，即便是同样的Java字节代码，被两个不同的类加载器定义之后，所得到的Java类也是不同的。如果试图在两个类的对象之间进行赋值操作，会抛出java.lang.ClassCastException。这个特性为同样名称的Java类在JVM中共存创造了条件。在实际的应用中，可能会要求同一名称的Java类的不同版本在JVM中可以同时存在。通过类加载器就可以满足这种需求。这种技术在OSGi中得到了广泛的应用

Java类的加载过程：

1.通过类的全名产生对应类的二进制数据流。（如果没找到对应类文件，只有在类实际使用时才抛出错误。）  
2.分析并将这些二进制数据流转换为方法区(JVM 的架构：方法区、堆，栈，本地方法栈，pc 寄存器)特定的数据结构（这些数据结构是实现有关的，不同 JVM 有不同实现）。这里处理了部分检验，比如类文件的魔数的验证，检查文件是否过长或者过短，确定是否有父类（除了 Obecjt 类）。  
3.创建对应类的 java.lang.Class 实例（注意，有了对应的 Class 实例，并不意味着这个类已经完成了加载链链接！）。

**Java类的链接**

Java类的链接指的是将Java类的二进制代码合并到JVM的运行状态之中的过程。在链接之前，这个类必须被成功加载。  
链接的过程比加载过程要复杂很多，这是实现java的动态性的重要一步！分为三部分：verification （**检测**）， preparation（**准备**） 和 resolution（**解析**）

**1.verification（**检测**）：**  
**验证**是用来确保Java类的二进制表示在结构上是完全正确的。如果验证过程出现错误的话，会抛出java.lang.VerifyError错误。

linking的resolve会把类中成员方法、成员变量、类和接口的符号引用替换为直接引用，而在这之前，需要检测被引用的类型正确性和接入属性是否正确（就是public ,private的的问题）诸如，检查final class 没有被继承，检查静态变量的正确性等等。  
验证是用来确保Java类的二进制表示在结构上是完全正确的。如果验证过程出现错误的话，会抛出java.lang.VerifyError错误。

**2.preparation（准备）：**

准备过程则是创建Java类中的静态域，并将这些域的值设为默认值。准备过程并不会执行代码。在一个Java类中会包含对其它类或接口的形式引用，包括它的父类、所实现的接口、方法的形式参数和返回值的Java类等。

对类的成员变量分配空间。虽然有初始值，但这个时候不会对他们进行初始化（因为**这里不会执行任何 Java 代码**）。具体如下：  
所有原始类型的值都为 0。如 float: 0f, int: 0, boolean: 0(注意 boolean 底层实现大多使用 int)，引用类型则为 null。值得注意的是，JVM 可能会在这个时期给一些有助于程序运行效率提高的数据结构分配空间。

**3.resolution（解析）：**

解析的过程就是确保这些被引用的类能被正确的找到。解析的过程可能会导致其它的Java类被加载。

为类、接口、方法、成员变量的符号引用定位直接引用（如果符号引用先到常量池中寻找符号，再找先应的类型，无疑会耗费更多时间），完成内存结构的布局。  
这一步是可选的。可以在符号引用第一次被使用时完成，即所谓的延迟解析(late resolution)。但对用户而言，这一步永远是延迟解析的，即使运行时会执行 early resolution，但程序不会显示的在第一次判断出错误时抛出错误，而会在对应的类第一次主动使用的时候抛出错误！  
另外，这一步与之后的类初始化是不冲突的，并非一定要所有的解析结束以后才执行类的初始化。不同的 JVM 实现不同。  
看下面一段代码：

public class LinkTest {

public static void main(String[] args) {

ToBeLinked toBeLinked = null;

System.out.println("Test link.");

}

}

类 LinkTest引用了类ToBeLinked，但是并没有真正使用它，只是声明了一个变量，并没有创建该类的实例或是访问其中的静态域。如果把编译好的ToBeLinked的Java字节代码删除之后，再运行LinkTest，程序不会抛出错误。这是因为ToBeLinked类没有被真正用到。链接策略使得ToBeLinked类不会被加载，因此也不会发现ToBeLinked的Java字节代码实际上是不存在的。如果把代码改成ToBeLinked toBeLinked = new ToBeLinked();之后，再按照相同的方法运行，就会抛出异常了。因为这个时候ToBeLinked这个类被真正使用到了，会需要加载这个类。

**三、Java类的初始化**

开发 Java 时，接触最多的是对象的初始化。实际上类也是有初始化的。相比对象初始化，类的初始化机制要简单不少。  
类的初始化也是延迟的，直到类第一次被主动使用(active use)，JVM 才会初始化类。  
当一个Java类第一次被真正使用到的时候，JVM会进行该类的初始化操作。初始化过程的主要操作是执行静态代码块和初始化静态域。在一个类被初始化之前，它的直接父类也需要被初始化。但是，一个接口的初始化，不会引起其父接口的初始化。在初始化的时候，会按照源代码中从上到下的顺序依次执行静态代码块和初始化静态域。

public class StaticTest {

public static int X = 10;

public static void main(String[] args) {

System.out.println(Y); //输出60

}

static {

X = 30;

}

public static int Y = X \* 2;

}

在上面的代码中，在初始化的时候，静态域的初始化和静态代码块的执行会从上到下依次执行。因此变量X的值首先初始化成10，后来又被赋值成30；而变量Y的值则被初始化成60。  
类的初始化分两步：

1.如果基类没有被初始化，初始化基类。  
2.有类构造函数，则执行类构造函数。

类构造函数是由 Java 编译器完成的。它把类成员变量的初始化和 static 区间的代码提取出，放到一个<clinit>方法中。这个方法不能被一般的方法访问（注意，static final 成员变量不会在此执行初始化，它一般被编译器生成 constant 值）。同时，<clinit>中是不会显示的调用基类的<clinit>的，因为 1 中已经执行了基类的初始化。**该初始化过程是由 Jvm 保证线程安全的。。**

Java类和接口的初始化只有在特定的时机才会发生，这些时机包括：

创建一个Java类的实例。如

MyClass obj = new MyClass()

调用一个Java类中的静态方法。如

MyClass.sayHello()

给Java类或接口中声明的静态域赋值。如

MyClass.value = 10

访问Java类或接口中声明的静态域，并且该域不是常值变量。如

int value = MyClass.value

在顶层Java类中执行assert语句。

通过Java反射API也可能造成类和接口的初始化。需要注意的是，当访问一个Java类或接口中的静态域的时候，只有真正声明这个域的类或接口才会被初始化。考虑下面的代码：

class B {

static int value = 100;

static {

System.out.println("Class B is initialized."); //输出

}

}

class A extends B {

static {

System.out.println("Class A is initialized."); //不会输出

}

}

public class InitTest {

public static void main(String[] args) {

System.out.println(A.value); //输出100

}

}

在上述代码中，类InitTest通过A.value引用了类B中声明的静态域value。由于value是在类B中声明的，只有类B会被初始化，而类A则不会被初始化。

（全文完）