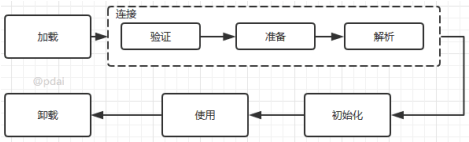
1、类的加载分几步？

按照 Java 虚拟机规范，从 **class 文件**到**加载到内存中的类**，到**类卸载出内存**为止，类的**整个生命周期**包括如下 7 个阶段：装载、链接（验证、准备、解析）、初始化、使用、卸载。

其中，类加载的过程包括了加载、验证、准备、解析、初始化五个阶段。

【注意】在这五个阶段中，**加载、验证、准备和初始化**这四个阶段发生的**顺序是确定的**，而**解析阶段则不一定**，它**在某些情况下**可以**在初始化阶段之后**开始，这是为了支持 Java 语言的**运行时绑定**(也称为**动态绑定**或晚期绑定)。另外注意这里的几个阶段是**按顺序开始**，而**不是按顺序进行或完成**，因为这些阶段通常都是互相交叉地混合进行的，通常在一个阶段执行的过程中调用或激活另一个阶段。



2、都有谁需要加载？（引用数据类型）

在 Java 中数据类型分为基本数据类型和引用数据类型。**基本数据类型**由虚拟机**预先定义**，**引用数据类型**则需要**进行类的加载**。

3、类的装载做了什么？

装载即将 **Java 类的字节码文件** *加载*到**机器内存中**，并**在内存中** *构建*出 **Java 类的原型——类模板对象**。将从字节码文件中**解析出的**常量池、类字段、类方法等**信息存储到类模板中**，方便 **JVM 在运行期**通过**类模板**而**获取 Java 类中的任意信息**。（类模板放在方法区）

装载的操作

查找并**加载**类的二进制数据，**生成** Class 实例。

在加载类时 Java 虚拟机必**完成以下 3 件事情**：

1）通过**类的全名**，*获取***类的二进制数据流**；

2）*解析***类的二进制数据流**为**方法区内的数据结构**（Java 类模型）；

3）*创建* java.lang.**Class 类的实例**表示该类型。作为方法区**这个类**的**各种数据**的**访问入口**。

4、Class 实例的位置在哪？

类将.class 文件加载至元空间后，会在**堆中创建**一个 Java.lang.**Class 对象**，用来**封装**类位于**方法区内的数据结构**，该 Class 对象是在加载类的过程中创建的，每个类都对应有一个 Class 类型的对象。（Class 类的**构造方法是私有的**，只有 **JVM 能够创建**。）

5、加载.class 文件的方式：

1）从**本地系统**中直接**加载**；

2）通过**网络下载**.class 文件；

3）从 **zip，jar 等归档文件**中**加载**.class 文件；

4）从**专有数据库**中提取.class 文件；

5）将 **Java 源文件动态编译**为.class 文件；

6、数组类的加载有什么不同？

数组类本身并**不是**由类加载器负责创建，而是由 **JVM 在运行时**根据需要而**直接创建**的，但**数组的元素类型**仍然**需要**依靠**类加载器去创建**。创建数组类（下述简称 A）的过程：

1）如果数组的元素类型是**引用类型**，那么就**遵循定义的加载过程**递归加载和创建数组 A 的元素类型；

2）**JVM** 使用指定的**元素类型**和**数组维度**来创建新的数组类。

3）如果数组的元素类型是**引用类型**，**数组类的可访问性**就由**元素类型的可访问性**决定。否则数组类的可访问性将**被缺省**定义为 **public**。

7、链接过程之验证阶段(Verification)

验证是链接操作的第一步，它的目的是**保证**加载的字节码是**合法、合理**并**符合规范**的。验证的内容则涵盖了类数据信息的**格式验证、语义检查、字节码验证**，以及**符号引用验证**等。

【整体说明】

1）**格式验证**会和**装载阶段一起执行**。**验证通过**之后，类加载器**才会成功**将类的二进制数据信息**加载到方法区**中。

2）格式验证之外的验证操作将会在**方法区中**进行。

3）在前面 3 次检查中，已经排除了文件格式错误、语义错误以及字节码的不正确性，但是依然不能确保类是没有问题的。还将在进行**符号引用的验证**。Class 文件在其**常量池会通过字符串**记录自己**将要使用的**其他**类或者方法**，确保解析动作能够被正常执行。

4）在验证阶段，虚拟机就会**检查**这些**类或者方法**确实是**存在**的，并且当前类**有权限访问**这些数据。

8、链接过程之准备阶段(Preparation)

准备阶段为**类的静态变量**分配内存，并将其**初始化为默认值**。

假设一个类变量的定义为：public static int value = 3；那么变量 value 在准备阶段过后的初始值为 0，而不是 3，因为这时候尚未开始执行任何 Java 方法，而把 value 赋值为 3 的 put static 指令是在程序编译后，存放于类构造器<clinit>()方法之中的，所以把 value 赋值为 3 的动作将在初始化阶段才会执行。

【注意】Java 并不支持 boolean 类型，对于 boolean 类型，**内部实现是 int,**由于 int 的默认值是 0,故对应的，**boolean 的默认值**就是 **false**。

1）这里不包含基本数据类型的字段用 **static final 修饰的情况**，因为 **final 在编译**的时候就**会分配**了，**准备阶段会显式赋值**；

2）注意这里不会为**实例变量**分配初始化，实例变量是会**随着对象**一起**分配到 Java 堆**中；

3）在这个阶段并**不会**像初始化阶段中那样会**有初始化或者代码被执行**。

9、链接过程之解析阶段(Resolution)

解析就是将**符号引用**转为**直接引用**，也就是得到类、字段、方法**在内存中的指针**或者**偏移量**。因此，可以说，如果**直接引用存在**，那么**可以肯定系统中存在**该类、方法或者字段。但**只存在符号引用，不能确定**系统中**一定存在**该结构。

以方法为例，Java 虚拟机**为每个类**都准备了**一张方法表**，将**其所有的方法**都列在表中，当需要**调用一个类的方法**的时候，只要知道这个方法**在方法表中的偏移量**就可以**直接调用**该方法。通过**解析操作，符号引用**就可以转变为**目标方法在类中方法表中的位置**，从而使得方法被**成功调用**。

10、Initialization(初始化)阶段

初始化阶段为**类的静态变量赋**予正确的**初始值**。(显式初始化) 到了初始化阶段，才真正开始执行类中定义的 Java 程序代码。

初始化阶段执行类的初始化方法：<clinit>()。该方法仅能**由 Java 编译器生成**并由 **JVM 调用**。它是由**类静态成员的赋值语句**以及 **static 语句块**合并产生的。

11、哪些类不会生成<clinit>方法？

Java 编译器并不会为所有的类都产生<clinit>()初始化方法。不会包含<clinit>()方法的情况：

1）一个类中并**没有**声明任何的**类变量**，也**没有静态代码块**时；

2）一个类中**声明类变量**，但是**没有**明确使用**类变量的初始化语句**以及**静态代码块**来执行**初始化操作**时；

3）一个类中包含 **static final 修饰**的**基本数据类型**的**字段**，这些类字段**初始化语句**采用编译时**字面量**或**常量赋值**（链接阶段的准备环节）。【注意】如果初始化语句是是**方法或构造器**则还是初始化阶段赋值，会有<clinit>()。

12、<clinit>()的调用会死锁吗？

虚拟机会保证一个类的**<clinit>()方法在多线程环境中**被正确地**加锁、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<clinit>()方法，其他线程都需要阻塞等待，直到活动线程执行<clinit>()方法完毕，引发死锁。

13、类加载的时机（哪些情况触发类加载）？

1）当**创建一个类的实例**时，比如使用 new 关键字，或者通过反射、克隆、反序列化。

2）当调用**类的静态方法**时，即当使用了字节码 invokestatic 指令；

3）当使用**类、接口的静态变量**时(final 修饰特殊考虑)，比如，使用 getstatic 或者 putstatic 指令；

4）当使用 java.lang.reflect 包中**反射类的方法**时如：**Class.forName**("com.atguigu.java.Test")；

5）当**初始化子类**时，如果发现其父类还没有进行过初始化，则需要先**触发其父类的初始化**；【注意】一个**父接口**并不会因为它的**子接口**或者**实现类**的**初始化而初始化**。只有当程序**首次使用**特定接口的**静态字段**时，**才会**导致**该接口的初始化**。

6. 如果一个**接口定义了 default 方法**，那么直接实现或者间接实现**该接口的类的初始化**，**该接口**要在其之前被**初始化**。

7）当**虚拟机启动时**，用户需要**指定一个要执行的主类**（包含 main()方法的那个类），虚拟机会**先初始化**这个主类。【注意】这个类在 **main()方法之前**被链接和初始化。

14、被动使用的情况

并非在代码中出现的类，就一定会被加载或者初始化。如果**不符合主动使用的条件**，类就**不会初始化**。被动使用不会引起类的初始化。

1）当**访问一个静态字段**时，只有**真正声明**这个字段**的类**才会被初始化。如：当通过**子类引用父类的静态变量，不会**导致子类初始化

2）通过**数组定义类引用**，不会触发此类的初始化；

3）**引用常量**不会触发此类或接口的初始化。因为常量在链接阶段就已经被显式赋值了。

4）**调用 ClassLoader 类**的 **loadClass()方法**加载一个类，并不是对类的主动使用，**不会**导致类的初始化。

15、类的卸载

Java 虚拟机将结束生命周期的几种情况

1）执行了 **System.exit()方法**；

2）程序**正常执行结束**；

3）程序在执行过程中遇到了**异常或错误而异常终止**；

4）由于**操作系统出现错误**而导致 Java 虚拟机进程终止。

16、什么是类加载器？

类加载器是 JVM 执行类加载机制的前提。它负责通过各种方式将 Class 信息的二进制数据流读入 JVM 内部，转换为一个与目标类对应的 java.lang.Class 对象实例。然后交给 Java 虚拟机进行链接、初始化等操作。

17、类加载的显式加载与隐式加载

指 JVM 加载 class 文件到内存的方式：

1）显式加载：指的是在代码中通过**调用 ClassLoader** 加载 class 对象，如直接使用 **Class.forName**(name)【默认会执行初始化块】，**this.getClass().getClassLoader().loadClass()**【不执行初始化块】加载 class 对象。

2）隐式加载：则是**不直接在代码中调用 ClassLoader 的方法**加载 class 对象，而是通过**虚拟机自动加载到内存**中，如在加载某个类的 class 文件时，**该类的 class 文件中引用了另外一个类的对象**，此时**额外引用的类**将通过 **JVM 自动加载到内存中**。

18、Class.forName()和 ClassLoader.loadClass()

1）**Class.forName()**：将类的.class 文件**加载到 jvm 中**，还会**对类进行解释，执行**类中的 **static 块**；

2）**ClassLoader.loadClass()**：只干一件事情，就是**将.class 文件加载到 jvm 中，不会执行 static** 中的内容,只有**在 newInstance 才会**去执行 static 块。

3）**Class.forName(name, initialize, loader)**：带参函数也可**控制是否加载 static 块**。并且只有调用了 newInstance()方法采用调用构造函数，创建类的对象 。

19、什么为类的唯一性？

对于任意一个类，都需要由**加载它的类加载器**和**这个类本身**一同**确认**其在 Java 虚拟机中的**唯一性**。**每一个类加载器**，都**拥有**一个**独立**的**类名称空间**：*比较两个类是否相等*，只有在这**两个类**是**由同一个类加载器加载**的前提下才有意义。否则，即使这两个类源自同一个 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**。

20、子父类加载器的关系？

除了顶层的**启动类加载器**外，**其余的类加载器**都应当**有**自己的“**父类**”**加载器**。

【注意】不同类加载器**看似是继承关系，实际上是包含关系**。在**下层加载器**中，**形参中**包含着**上层加载器的引用**。

21、类的加载器有哪些？

1）启动类加载器 Bootstrap ClassLoader：**负责加载**存放在 JDK\jre\lib(JDK 代表 JDK 的安装目录，下同)下，或被-Xbootclasspath 参数指定的路径中的**能被虚拟机识别的类库**(如 rt.jar，所有的 java.* 开头的类均被 Bootstrap ClassLoader 加载)。启动类加载器是**无法被 Java 程序直接引用的**。

2）扩展类加载器 Extension ClassLoader：该加载器由 sun.misc.Launcher$ExtClassLoader 实现，它**负责加载** JDK\jre\lib\ext 目录中，或者由 java.ext.dirs 系统变量指定的路径中的**所有类库**(如 javax.*开头的类)，开发者**可以直接使用扩展类加载器**。

3）应用程序类加载器 Application ClassLoader：由 sun.misc.Launcher$AppClassLoader 来实现，它**负责加载**用户类路径(ClassPath)**所指定的类**，开发者**可以直接使用**该类加载器，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**。

22、JVM 类加载机制

1）全盘负责：当**一个类加载器**负责**加载某个 Class 时，该 Class** 所依赖的和引用的**其他 Class** 也将**由该类加载器**负责载入，**除非显示使用另外一个**类加载器来载入；

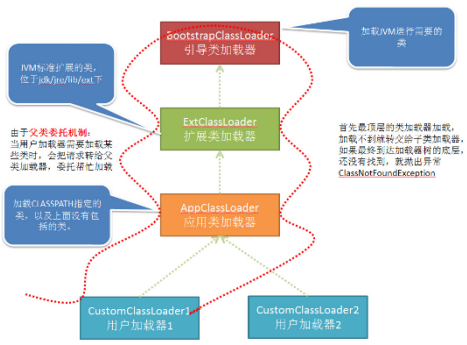
2）父类委托：**先让父类加载器**试图加载该类，只有在父类加载器**无法加载该类时才尝试从自己**的类路径中加载该类；

3）缓存机制：缓存机制将会保证**所有加载过的 Class 都会被缓存**，当程序中需要**使用某个 Class 时**，类加载器**先从缓存区寻找该 Class**，只有缓存区**不存在，系统才会读取该类**对应的二进制数据，并将其转换成 Class 对象，**存入缓存区**。这就是为什么修改了 Class 后，必须重启 JVM，程序的修改才会生效；

4）双亲委派机制：如果一个类加载器收到了类加载的请求，它**首先不会自己**去尝试加载这个类，而是**把请求委托给父加载器**去完成，***依次向上***，因此，所有的类加载请求**最终都**应该**被传递到顶层的启动类加载器中**，只有当**父加载器**在它的搜索范围中**没有找到所需的类时**，即无法完成该加载，**子加载器才会尝试自己**去加载该类。

23、双亲委派机制过程？

当 AppClassLoader 加载一个 class 时，它首先不会自己去尝试加载这个类，而是把类加载请求委派给父类加载器 ExtClassLoader 去完成。当 ExtClassLoader 加载一个 class 时，它首先也不会自己去尝试加载这个类，而是把类加载请求委派给 BootStrapClassLoader 去完成。如果 BootStrapClassLoader 加载失败(例如在$JAVA_HOME/jre/lib 里未查找到该 class)，会使用 ExtClassLoader 来尝试加载；若 ExtClassLoader 也加载失败，则会使用 AppClassLoader 来加载，如果 AppClassLoader 也加载失败，则会报出异常。



24、双亲委派机制优势和弊端

1）优势：

-避免类重复加载，确保一个类的全局唯一性；

-保护程序安全，防止核心 API 被随意篡改。

2）弊端：

检查类是否加载的**委托过程是单向的**，这个方式虽然从结构上说比较清晰，使各个 ClassLoader 的**职责非常明确**，但是**顶层的 ClassLoader** 无法访问**底层的 ClassLoader** 所加载的类。

25、双亲委派机制可以打破吗？怎么打破？

可以，双亲委派模型**并不是**一个**具有强制性约束的模型**，而是 Java 设计者**推荐**给开发者们**的类加载器实现方式**。

方法 1：不用 loadClass()，重写 findClass()。

设计者在 java.lang.**ClassLoader 中**添加一个新的 **protected 方法 findClass()**，并引导用户编写的类加载逻辑时尽可能去**重写这个方法，而不是在 loadClass()中编写代码**。(双亲委派的具体逻辑就实现在 loadClass()方法里面，按照 loadClass()方法的逻辑，如果父类加载失败，会自动调用自己的 findClass()方法完成加载)

方法 2 ：线程上下文类加载器。

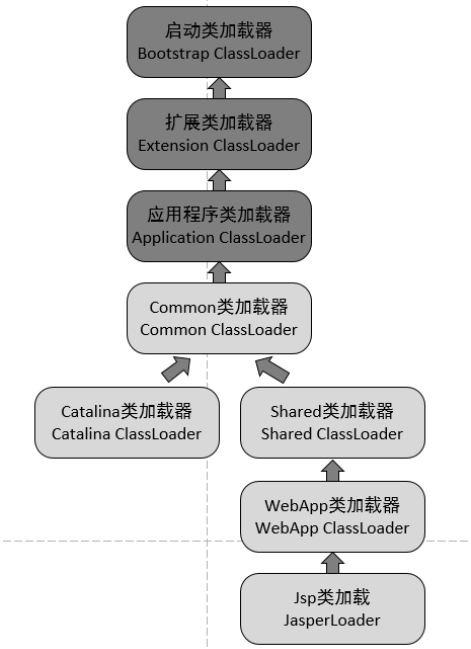
这类加载器**默认是**应用程序类加载器，通过 java.lang.Thread 类的 setContextClassLoader()方法进行设置。这是一种**父类加载器去请求子类加载器完成类加载**的行为，这种行为实际上是打通了双亲委派模型的层次结构来逆向使用类加载器。

方法 3：代码热替换、模块热部署。

每一个程序模块（OSGi 中称为 Bundle）都有**一个自己的类加载器**，当需要**更换一个 Bundle 时**，就把 **Bundle** *连同***类加载器**一起**换掉**以实现代码的热替换。在 OSGi 环境下，类加载器***不再***双亲委派模型推荐的**树状结构**，而是进一步发展为**更加复杂的网状结构**。

26、什么是 tomcat 类加载机制？

Tomcat 的类加载机制是违反了双亲委托原则的，对于一些**未加载的非基础类，各个 web** 应用**自己的类加载器**(WebAppClassLoader)会**优先查看自己的仓库**加载，**加载不到时**再交给 commonClassLoader **走双亲委托**。



当 tomcat 启动时，会创建几种类加载器：

1）Bootstrap 引导类加载器：

加载 **JVM 启动所需的类**，以及**标准扩展类**（位于 jre/lib/ext 下）

2）System 系统类加载器：

加载 **tomcat 启动的类**，比如 bootstrap.jar，通常在 catalina.bat 或者 catalina.sh 中指定。位于 CATALINA_HOME/bin 下。

3）CommonClassLoader、CatalinaClassLoader、SharedClassLoader 和 WebappClassLoader

这些是 **Tomcat 自己定义的类加载器**，它们分别加载/common/*、/server/*、/shared/*（在 tomcat 6 之后已经合并到根目录下的 lib 目录下）和/WebApp/WEB-INF/*中的 Java 类库。其中 **WebApp 类加载器**和 **Jsp 类加载器**通常**会存在多个实例**，**每一个** Web 应用程序**对应一个** WebApp 类加载器，**每一个** JSP 文件**对应一个** Jsp 类加载器。

从图中的委派关系中可以看出：

1）CommonClassLoader 能加载的类都可以被 Catalina ClassLoader 和 SharedClassLoader 使用，从而实现了公有类库的共用，而 CatalinaClassLoader 和 Shared ClassLoader 自己能加载的类则与对方相互隔离。

2）WebAppClassLoader 用 SharedClassLoader 加载到的类，但各个 WebAppClassLoader 实例之间相互隔离。

3）而 **JasperLoader 的加载范围**仅仅是这个 **JSP 文件所编译出来的那一个.class 文件**，它出现的**目的就是为了被丢弃**：当 Web 容器检测到 **JSP 文件被修改**时，**会替换掉目前的 JasperLoader 的实例**，并通过再**建立一个新的 Jsp 类加载器**来实现 JSP 文件的 HotSwap 功能。

【注意】当应用需要到某个类时，则会按照下面的顺序进行类加载：

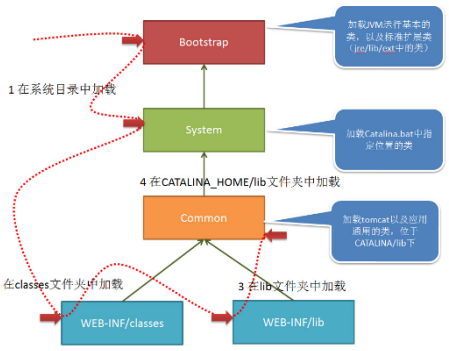
1）使用 **bootstrap 引导类加载器**加载；

2）使用 **system 系统类加载器**加载；

3）使用**应用类加载器**在 WEB-INF/classes 中加载；

4）使用**应用类加载器**在 WEB-INF/lib 中加载；

5）使用 **common 类加载器**在 CATALINA_HOME/lib 中加载。



【总结】tomcat 为了**实现隔离性**，没有遵守这个约定，**每个 webappClassLoader 加载自己的目录下**的 class 文件，**不会传递给父类**加载器。