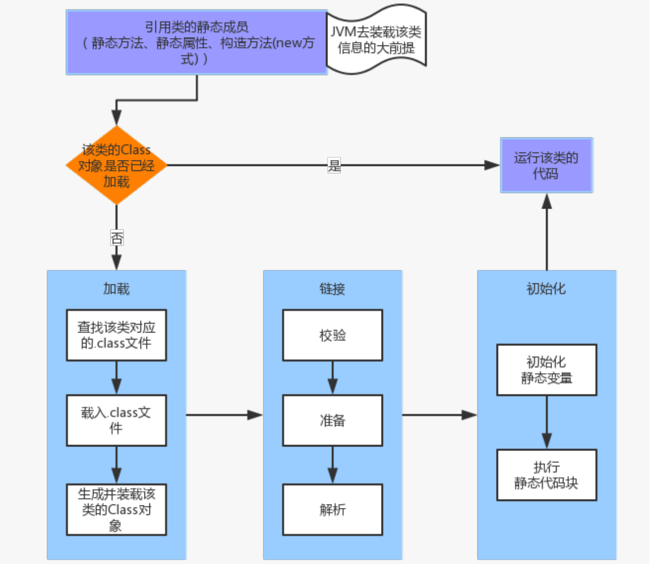
转载请说明出处：[Java面试相关（一）-- Java类加载全过程]()



引用类的静态成员（静态方法、静态属性、构造方法(new方式)）
JVM去装载该类信息的大前提
该类的Class对象是否已经加载
是
否
运行该类的代码
加载
查找该类对应的.class文件
载入.class文件
生成并装载该类的Class对象
链接
校验
准备
解析
初始化
初始化静态变量
执行静态代码块

JVM判断并装载类的过程

概述：

我们知道，Java中我们写类的代码，是存在于一个个.java文件中的，而这个后缀名也是让JVM识别编译的基础。可能有些Android开发者对几个ClassLoader（如：AppClassLoader等）比较熟悉，那么，整个类的加载过程：从未进行编译的.java文件，到类的初始化完毕并等待被实例化使用的过程，具体是怎么样的。

收集Java资料、看了几篇本人觉得较好的博文后，总结以下关于Java类的加载过程，掌握此过程，能够更加理解Java类的各个方法的执行顺序，以及JVM的工作和Java类生成的原理。读者笑纳～

类的加载过程分析

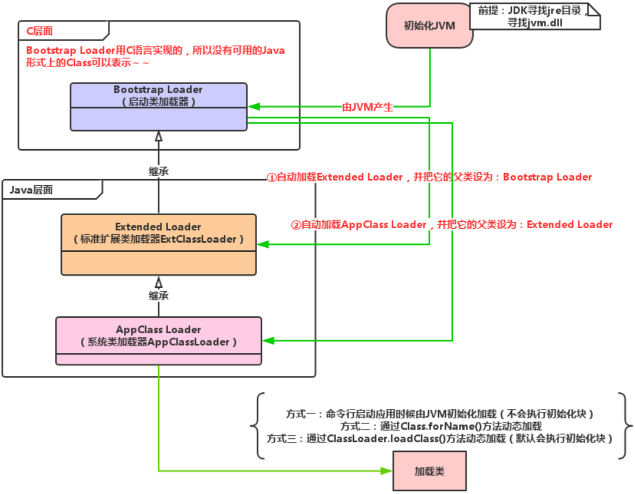
类从**.java文件到实际加载到内存中**，实际上是这样的一个**过程**：

.java文件 -> 通过你的JDK环境相关指令编译 -> .class文件 -> JVM初始化之后，如果有类的执行、调用等相关操作，JVM就会将.class文件加载到内存中，并开始下面的一系列处理：（链接->初始化）

一、关于ClassLoader

首先我们要搞清楚一点，ClassLoader是Java用于加载类的一个机制。等到程序运行时，JVM先初始化，在JVM初始化的过程中，JVM生成几个ClassLoader，JVM调用指定的ClassLoader去加载.class文件等各类路径、文件的类。

1. 程序运行时类的加载实际过程
    - JDK执行指令去寻找jre目录，寻找jvm.dll，并初始化JVM；
    - 产生一个Bootstrap Loader（启动类加载器）；
    - BootstrapLoader自动加载ExtendedLoader（标准扩展类加载器），并将其父Loader设为Bootstrap Loader。
    - BootstrapLoader自动加载AppClassLoader（系统类加载器），并将其父Loader设为Extended Loader。
    - 最后由AppClassLoader加载HelloWorld类。
2. 各种ClassLoader及其特点
    - **BootstrapLoader（启动类加载器）**：加载System.getProperty("sun.boot.class.path")所指定的路径或jar
    - **ExtendedLoader（标准扩展类加载器ExtClassLoader）**：加载System.getProperty("java.ext.dirs")所指定的路径或jar。在使用Java运行程序时，也可以指定其搜索路径，例如：java -Djava.ext.dirs=d:\projects\testproj\classes HelloWorld
    - **AppClassLoader（系统类加载器AppClassLoader）**：加载System.getProperty("java.class.path")所指定的路径或jar。在使用Java运行程序时，也可以加上-cp来覆盖原有的Classpath设置，例如： java -cp ./lavasoft/classes HelloWorld
    - **特点**
        - ExtClassLoader和AppClassLoader在JVM启动后，会在JVM中保存一份，并且在程序运行中无法改变其搜索路径。如果想在运行时从其他搜索路径加载类，就要产生新的类加载器。
        - 运行一个程序时，总是由AppClassLoader（系统类加载器）开始加载指定的类
        - 在加载类时，每个类加载器会将加载任务上交给其父，如果其父找不到，再由自己去加载
        - BootstrapLoader（启动类加载器）是最顶级的类加载器了，其父加载器为null
3. 各类ClassLoader的关系图解（帮助理解）


C层面
Bootstrap Loader用C语言实现的，所以没有可用的Java形式上的Class可以表示~~
Bootstrap Loader
（启动类加载器）
初始化JVM
前提：JDK寻找jre目录寻找jvm.dll
由JVM产生
继承
Java层面
①自动加载Extended Loader，并把它的父类设为：Bootstrap Loader
Extended Loader
（标准扩展类加载器ExtClassLoader）
②自动加载AppClass Loader，并把它的父类设为：Extended Loader
继承
AppClass Loader
（系统类加载器AppClassLoader）
方式一：命令行启动应用时候由JVM初始化加载（不会执行初始化块）
方式二：通过Class.forName()方法动态加载
方式三：通过ClassLoader.loadClass()方法动态加载（默认会执行初始化块）
加载类

各个ClassLoader的作用以及他们之间的关系.png

注意：图解中可得，执行代码`c.getClassLoader().getParent().getParent()`为`null`，由于get不到BootstrapLoader，因为BootstrapLoader是C层次实现的。

4. 关于不同类加载器所处命名空间不同的问题理解

   Java当中，不同的类加载器加载的类在虚拟机中位于不同命名空间下，而不同命名空间下的类相互不可见。那么，有时我们会很不解，BootstrapLoader加载java.util.List类，而我们自己定义的类比如com.androidjp.MyClass 则由APPClassLoader来加载。从上面的图中我们可以看到，APPClassLoader等ClassLoader之间是继承与被继承的关系，而APPClassLoader本身可以作为我们自定义ClassLoader的父类，当默认调用APPClassLoader去加载某个类时，它先在它的缓存区查看要加载的这个类是否存在，存在则直接加载，不存在则它会让它的父类ExtendedLoader去加载，而ExtendedLoader又会调用进行同样的步骤，直到他的父亲BootstrapLoader，这种调用关系我们称之为“双亲委托机制”。在这种机制下，原本在JVM的BootstrapLoader类型表【JVM为每一个类加载器维护一个表，表中存放所有以这个类加载器为初始类加载器的类】中的java.util.List,就能够被这个MyClass所发现，因为他们两个类之间相互是‘融洽’的，换句话说，MyClass的初始类加载器的先辈所加载的类，也是我的亲人，我们之间的交互是安全的。所以， 才有了‘MyClass中可以调用加载List’的过程。详细可以点击这篇文章来参考。

二、类的加载方式

1. 方式一：命令行启动应用时候由JVM初始化加载
2. 方式二：通过Class.forName()方法动态加载（默认会执行初始化块，但如果指定

ClassLoader，初始化时不执行静态块 ）

3. 方式三：通过ClassLoader.loadClass()方法动态加载（不会执行初始化块 ）

解析：

方式一其实就是通过以下几种主动引用类的方式所触发的JVM的类加载和初始化过程。然后，其实这三种类加载方式，在java 层面上都是JVM调用了ClassLoader去加载类的过程，只是：方式一相对与方式二和方式三而言，属于静态方式的加载；而方式二和方式三的区别，在于`Class.ForName`源码中：

```
///Class.forname(String name)
public static Class<?> forName(String className) throws ClassNotFoundException {
    return forName(className, true, VMStack.getCallingClassLoader());
}
……………
///实际调用：
public static Class<?> forName(String className, boolean shouldInitialize,
        ClassLoader classLoader) throws ClassNotFoundException {
    if (classLoader == null) {
        classLoader = BootClassLoader.getInstance();
    }
      Class<?> result;
    try {
        result = classForName(className, shouldInitialize, classLoader);
    } catch (ClassNotFoundException e) {
        Throwable cause = e.getCause();
        if (cause instanceof LinkageError) {
            throw (LinkageError) cause;
        }
        throw e;
    }
    return result;
}
```

在源码当中可以看到，参数`boolean shouldInitialize`，在默认情况下的Class.forName(String)此参数默认为`true`，则默认情况下会进行初始化，

那么，初始化到时是怎么个操作过程，此过程又是怎么样去触发的呢？下面我们通过分析类的加载流程以及整体图解，来帮助说明。

三、详细分析整个类的加载流程

下面分析一下类的几种加载方式、ClassLoader对类加载的背后，是怎么个原理：

1. 类从编译、被使用，到卸载的全过程：

编译 -> 加载 -> 链接（验证+准备+解析）->初始化（使用前的准备）->使用-> 卸载

2. 类的初始化之前

加载（除了自定义加载）和链接的过程是完全由jvm负责的，包括：加载 -> 验证 -> 准备 -> 解析

> 这里的“自定义加载”可以理解为：自定义类加载器去实现自定义路径中类的加载，可以参考这篇文章。由于默认各个路径的类文件加载过程在JVM初始化的过程中就默认设定好了，也就是一般步骤下的加载过程，已经在JVM初始化过程中规定的AppClassLoader等加载器中规定了步骤，所以，按一般的加载步骤，就是按JVM规定的顺序，JVM肯定先负责了类的加载和链接处理，然后再进行类初始化。

1. 首先是**加载**：
   - 此过程由**类加载器**完成
   - 这一块JVM要完成3件事：
     1. 通过一个类的全限定名来获取定义此类的二进制字节流。
     2. 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。
     3. 在java堆中生成一个代表这个类的java.lang.Class对象，作为方法区这些数据的访问入口。
   - 这一步很灵活,很多技术都是在这里切入，因为它并没有限定二进制流从哪里来，那么我们可以用系统的类加载器，也可以用自己的方式写加载器来控制字节流的获取：
     1. 从class文件来->一般的文件加载
     2. 从zip包中来->加载jar中的类
     3. 从网络中来->Applet
   - 获取二进制流获取完成后会按照jvm所需的方式保存在方法区中，同时会在java堆中实例化一个java.lang.Class对象与堆中的数据关联起来。
2. 然后是**验证**（也称为**检验**）：

   一句话：检查代码的：完整性、正确性、安全性
   - *主要经历几个步骤：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文件格式的规范并 验证其版本是否能被当前的jvm版本所处理。ok没问题后，字节流就可以进入内存的方法区进行保存了。后面的3个校验都是在方法区进行的。
     - *元数据验证*：对字节码描述的信息进行语义化分析，保证其描述的内容符合java语言的语法规范。
     - *字节码检验*：最复杂，对方法体的内容进行检验，保证其在运行时不会作出什么出格的事来。
     - *符号引用验证*：来验证一些引用的真实性与可行性，比如代码里面引了其他类，这里就要去检测一下那些来究竟是否存在；或者说代码中访问了其他类的一些属性，这里就对那些属性的可以访问行进行了检验。（这一步将为后面的解析工作打下基础）
   - 目的：确保class文件的字节流信息符合jvm的口味，不会让jvm感到不舒服。假如class文件是由纯粹的java代码编译过来的，自然不会出现类似于数组越界、跳转到不存在的代码块等不健康的问题，因为一旦出现这种现象，编译器就会拒绝编译了。但是，跟之前说的一样，Class文件流不一定是从java源码编译过来的，也可能是从网络或者其他地方过来的，甚至你可以自己用16进制写，假如jvm不对这些数据进行校验的话，可能一些有害的字节流会让jvm完全崩溃。

    验证阶段很重要，但也不是必要的，假如说一些代码被反复使用并验证过可靠性了，实施阶段就可以尝试用-Xverify:none参数来关闭大部分的类验证措施，以简短类加载时间。

3. 随后是**准备**：

   一句话：为静态域分配存储空间
   - 这阶段会为类变量（指那些静态变量）分配内存并设置类比那辆初始值的阶段，这些内存在方法区中进行分配。这里要说明一下，这一步只会给那些静态变量设置一个初始的值，而那些实例变量是在实例化对象时进行分配的。

     例如：
     - `public static int value=123;`此时value的值为0，不是123。
     - `private int i = 123;` 此时，i 还未进行初始化，因为这句代码还不能执行。
4. 最后是**解析**：

   一句话：符号引用 -> 直接引用
   - 是对类的字段，方法等东西进行转换，具体涉及到Class文件的格式内容。

3. 类的初始化条件(主动对类进行引用)

说明：要对类进行**初始化**，代码上可以理解为**‘为要初始化的类中的所有静态成员都赋予初始值、对类中所有静态块都执行一次，并且是按代码编写顺序执行’**。

如下代码：输出的是‘1’。如果①和②顺序调换，则输出的是‘123’。

```
public class Main {
   public static void main(String[] args){
        System.out.println(Super.i);
    }
}
class Super{
    //①
    static{
        i = 123;
    }
    //②
    protected static int i = 1;
}
```

1. 遇到new，getstatic，putstatic，invokestatic这4条字节码指令时，假如类还没进行初始化，则马上对其进行初始化工作。

   其实就是3种情况：
   - 用new实例化一个类时
   - 读取或者设置类的静态字段时（不包括被final修饰的静态字段，因为他们已经被塞进常量池了）
   - 执行静态方法的时候。

2. 使用java.lang.reflect.*的方法对类进行反射调用的时候，如果类还没有进行过初始化，马上对其进行。
3. 初始化一个类的时候，如果他的父亲还没有被初始化，则先去初始化其父亲。
4. 当jvm启动时，用户需要指定一个要执行的主类（包含static void main(String[] args)的那个类），则jvm会先去初始化这个类。
5. 用Class.forName(String className);来加载类的时候，也会执行初始化动作。

   【注意:ClassLoader的loadClass(String className);方法只会加载并编译某类，并不会对其执行初始化】

   说明：“主动对类进行引用”指的就是以上五种JVM规定的判定初始化与否的预处理条件。

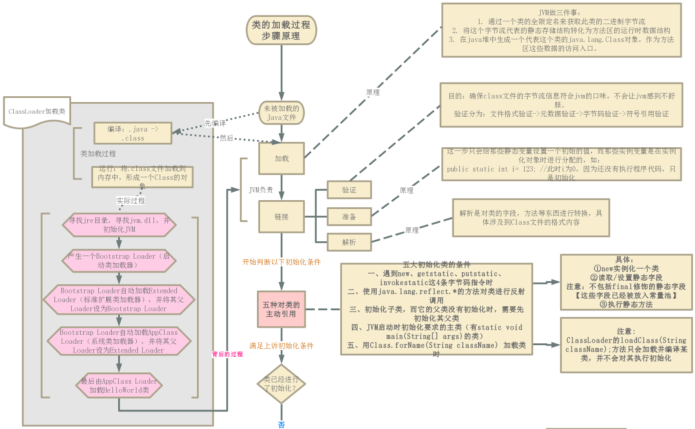
   那么，其他的方式，都可归为‘类被动引用’的方式，这些方式是不会引起JVM去初始化相关类的：

   1. 子类调用父类的静态变量（子类不会进行初始化，父类会初始化）
   2. 通过数组引用类的情况（类Main不会被初始化）

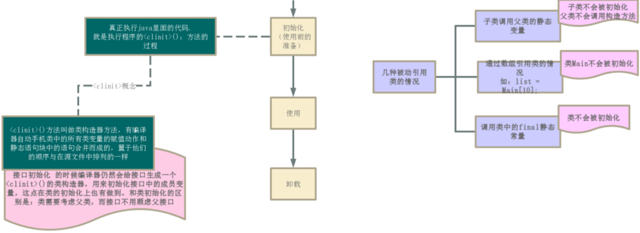
      如：list = Main[10];
   3. 调用类中的final静态常量（类不会被初始化）

四、原理分析图解

类加载中每个部分详细的原理说明，可以查看这篇文章。以下的图解为本人总结，算比较全地对每个步骤的原理过程一目了然：



类的加载过程步骤原理
未被加载的Java文件
先编译
然后
加载
原理
JVM做三件事：
1. 通过一个类的全限定名来获取此类的二进制字节流
2. 将这个字节流代表的静态存储结构转化为方法区的运行时数据结构
3. 在java堆中生成一个代表这个类的java.lang.Class对象，作为方法区这些数据的访问入口。
JVM负责
链接
验证
准备
解析
目的：确保class文件的字节流信息符合jvm的口味，不会让jvm感到不舒服。
验证分为：文件格式验证->元数据验证->字节码验证->符号引用验证
原理
这一步只会给那些静态变量设置一个初始的值，而那些实例变量是在实例化对象时进行分配的，如：
public static int i= 123; //此时i为0，因为还没有执行程序代码，只是初始化
原理
解析是对类的字段，方法等东西进行转换，具体涉及到Class文件的格式内容
ClassLoader加载类
编译：.java -> .class
类加载过程
运行：将.class文件加载到内存中，形成一个Class的对象
实际过程
寻找jre目录，寻找jvm.dll，并初始化JVM
产生一个Bootstrap Loader（启动类加载器）
Bootstrap Loader自动加载Extended Loader（标准扩展类加载器），并将其父Loader设为Bootstrap Loader
Bootstrap Loader自动加载AppClass Loader（系统类加载器），并将其父Loader设为Extended Loader
最后由AppClass Loader加载HelloWorld类
背后的过程
开始判断以下初始化条件
五种对类的主动引用
五大初始化类的条件
一、遇到new、getstatic、putstatic、invokestatic这4条字节码指令时
二、使用java.lang.reflect.*的方法对类进行反射调用
三、初始化子类，而它的父类没有初始化时，需要先初始化其父类
四、JVM启动时初始化要求的主类（有static void main(String[] args)的类）
五、用Class.forName(String className) 加载类时
具体：
①new实例化一个类
②读取/设置静态字段
注意：不包括final修饰的静态字段【这些字段已经被放入常量池】
③执行静态方法
注意：
ClassLoader的loadClass(String className);方法只会加载并编译某类，并不会对其执行初始化
满足上诉初始化条件
类已经进行了初始化?
否


真正执行java里面的代码.
就是执行程序的<clinit>()：方法的过程
初始化（使用前的准备）
<clinit>概念
使用
卸载
<clinit>()方法叫做类构造器方法，有编译器自动手机类中的所有类变量的赋值动作和静态语句块中的语句合并而成的，置于他们的顺序与在源文件中排列的一样
接口初始化 的时候编译器仍然会给接口生成一个<clinit>()的类构造器，用来初始化接口中的成员变量，这点在类的初始化上也有做到。和类初始化的区别是：类需要考虑父类，而接口不用顾虑父接口
几种被动引用类的情况
子类调用父类的静态变量
子类不会被初始化
父类不会调用构造方法
通过数组引用类的情况
如：list = Main[10];
类Main不会被初始化
调用类中的final静态常量
类不会被初始化

Java类加载过程.png

说明： 图解左下角说的`<clinit>()`方法，概念上是一个方法块，这个<clinit>(){……}方法块在初始化过程中执行，可以用下面代码理解：

```
class Parent{
public static int A=1;
static{
A=2;
}
}
---相当于---->
class Parent{
<clinit>(){
   public static int A=1;
   static{
     A=2;
   }
}
}
```

相当于把静态变量的赋值和静态代码块等操作顺序串连成一个方法。

注意：

- 对于类，会生成<clinit>(){……}方法体：去包含静态变量的赋值和静态块代码
- 而对于接口，也会生成<clinit>(){……}方法体：去初始化接口中的成员变量
- 接口和类初始化过程的区别：类的初始化执行之前要求父类全部都初始化完成了，但接口的初始化貌似对父接口的初始化不怎么感冒，也就是说，子接口初始化的时候并不要求其父接口也完成初始化，只有在真正使用到父接口的时候它才会被初始化（比如引用接口上的常量的时候啦）

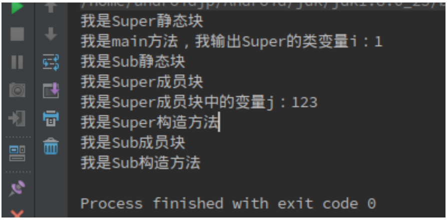
五、简单代码示例说明

这里，用一个java代码示例，来根据输出得到的各个方法和块的执行顺序，去更加形象地理解整个类的加载和运行过程：

```
public class Main {
    public static void main(String[] args){
```

```
        System.out.println("我是main方法，我输出Super的类变量i："+Sub.i);
        Sub sub  = new Sub();
    }
}
class Super{
    {
        System.out.println("我是Super成员块");
    }
    public Super(){
        System.out.println("我是Super构造方法");
    }
    {
        int j = 123;
        System.out.println("我是Super成员块中的变量j："+j);
    }
    static{
        System.out.println("我是Super静态块");
        i = 123;
    }
    protected static int i = 1;
}
class Sub extends Super{
    static{
        System.out.println("我是Sub静态块");
    }
    public Sub(){
        System.out.println("我是Sub构造方法");
    }
    {
        System.out.println("我是Sub成员块");
    }
}
```

得到结果为：



```
我是Super静态块
我是main方法，我输出Super的类变量i：1
我是Sub静态块
我是Super成员块
我是Super成员块中的变量j：123
我是Super构造方法
我是Sub成员块
我是Sub构造方法

Process finished with exit code 0
```

说明：

1. 静态代码块和静态变量的赋值 是 **先于** main方法的调用执行的。
2. 静态代码块和静态变量的赋值是按顺序执行的。
3. 子类调用父类的类变量成员，是不会触发子类本身的初始化操作的。
4. 使用new方式创建子类，对于类加载而言，是先加载父类、再加载子类（注意：此时由于父类已经在前面初始化了一次，所以，这一步，就只有子类初始化，父类不会再进行初始化）
5. 不论成员块放在哪个位置，它都 **先于** 类构造方法执行。