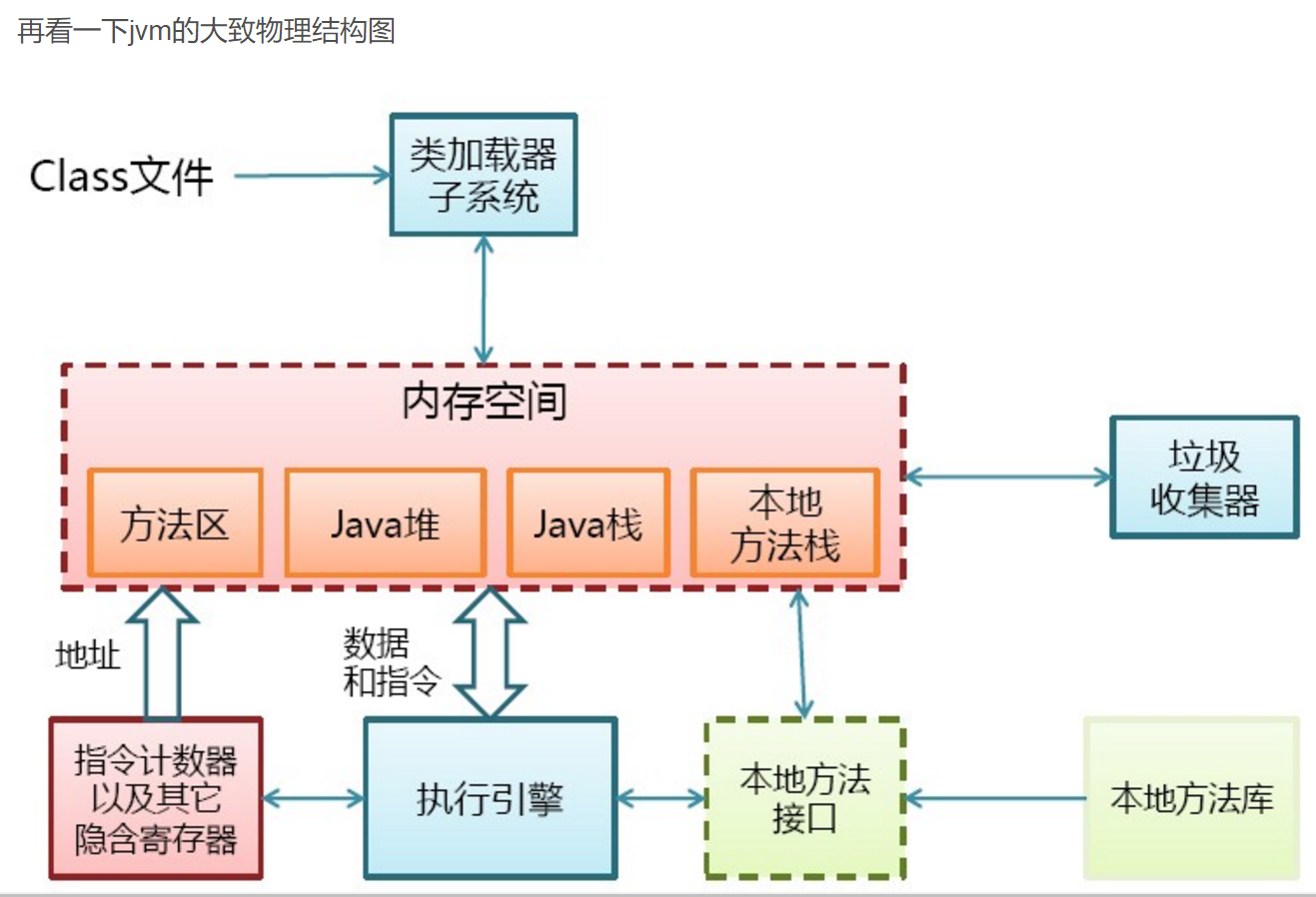
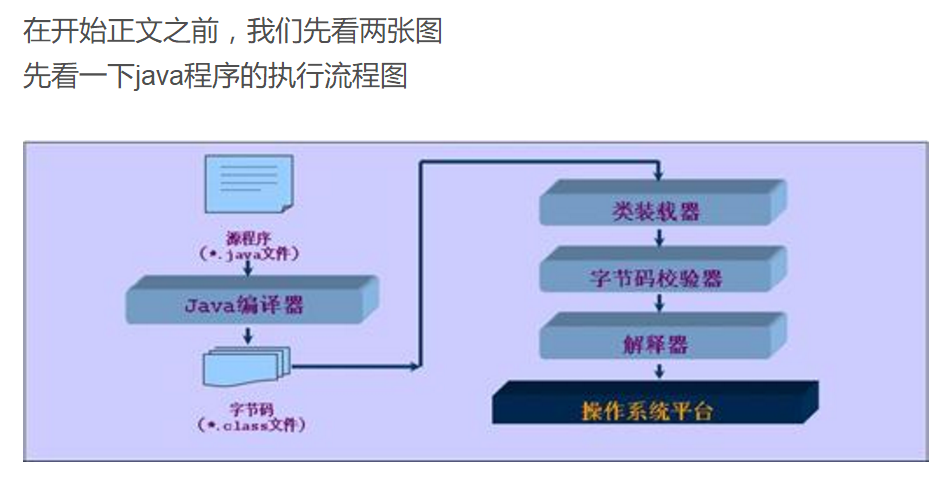
**深入理解类加载机制** 

类加载机制概念

Java虚拟机把描述类的数据从Class文件加载到内存，并对数据进行校验、准备、解析*(将常量池里的符号引用转化为直接引用)*和初始化，最终形成可以被虚拟机直接使用的Java类型，这就是虚拟机的加载机制。

Class文件由类装载器装载后，在JVM中将形成一份描述Class结构的元信息对象，通过该元信息对象可以获知Class的结构信息：如构造函数，属性和方法等，Java允许用户借由这个Class相关的元信息对象间接调用Class对象的功能,这里就是我们经常能见到的Class类。

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括了：加载（Loading）、验证（Verification）、准备（Preparation）、解析（Resolution）、初始化（Initialization）、使用（using）、和卸载（Unloading）七个阶段。其中验证、准备和解析三个部分统称为连接（Linking），这七个阶段的发生顺序如下图所示：

## 工作机制

类装载器就是寻找类的字节码文件，并构造出类在JVM内部表示的对象组件。在Java中，类装载器把一个类装入JVM中，要经过以下步骤：

(1) 装载：查找和导入Class文件；

(2) 链接：把类的二进制数据合并到JRE中；

(a)校验：检查载入Class文件数据的正确性；

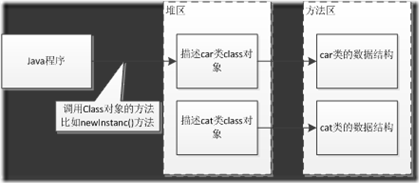
(b)准备：给类的静态变量分配存储空间；

(c)解析：将符号引用转成直接引用；

(3) 初始化：对类的静态变量，静态代码块执行初始化操作

## 1. 装载(加载) **（重点）**

### 什么是类的装载

类的装载指的是将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个java.lang.Class对象，用来封装类在方法区内的数据结构。类的加载的最终产品是位于堆区中的Class对象，Class对象封装了类在方法区内的数据结构，并且向Java程序员提供了**访问方法区内的数据结构的接口（这个应该就是符号引用了）**。   
   
 类加载器并不需要等到某个类被“首次主动使用”时再加载它，JVM规范允许类加载器在预料某个类将要被使用时就预先加载它，如果在预先加载的过程中遇到了.class文件缺失或存在错误，类加载器必须在程序首次主动使用该类时才报告错误（LinkageError错误）如果这个类一直没有被程序主动使用，那么类加载器就不会报告错误。

加载.class文件的方式有:

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

在了解了什么是类的加载后，回头来再看jvm进行类加载阶段都做了什么。虚拟机需要完成以下三件事情：

1.通过一个类的全限定名称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。

相对于类加载过程的其他阶段，加载阶段是开发期相对来说可控性比较强，**该阶段既可以使用系统提供的类加载器完成，也可以由用户自定义的类加载器来完成**，开发人员可以通过定义自己的类加载器去控制字节流的获取方式。关于这个过程的更多细节，我会在下一节细说，类的加载。   
加载阶段完成后，虚拟机外部的 **二进制字节流就按照虚拟机所需的格式存储在方法区之中，而且在Java堆中也创建一个java.lang.Class类的对象**，这样便可以通过该对象访问方法区中的这些数据。

## 2. 验证**（了解）**

验证的目的是为了确保Class文件中的字节流包含的信息符合当前虚拟机的要求，而且不会危害虚拟机自身的安全。不同的虚拟机对类验证的实现可能会有所不同，但大致都会完成以下四个阶段的验证：**文件格式的验证、元数据的验证、字节码验证和符号引用验证。**

1）文件格式的验证：验证字节流是否符合Class文件格式的规范，并且能被当前版本的虚拟机处理，该验证的主要目的是保证输入的字节流能正确地解析并存储于方法区之内。经过该阶段的验证后，字节流才会进入内存的方法区中进行存储，后面的三个验证都是基于方法区的存储结构进行的。

2）元数据验证：对类的元数据信息进行语义校验（其实就是对类中的各数据类型进行语法校验），保证不存在不符合Java语法规范的元数据信息。

3）字节码验证：该阶段验证的主要工作是进行数据流和控制流分析，对类的方法体进行校验分析，以保证被校验的类的方法在运行时不会做出危害虚拟机安全的行为。

4）符号引用验证：这是最后一个阶段的验证，它发生在虚拟机将符号引用转化为直接引用的时候（解析阶段中发生该转化，后面会有讲解），主要是对类自身以外的信息（常量池中的各种符号引用）进行匹配性的校验。

## 3. 准备**（了解）**

准备阶段是正式为**类变量分配内存**并设置类变量初始值的阶段，这些内存都将在**方法区中进行分配。**注：  
1）这时候进行内存分配的仅包括类变量（静态变量static），而不包括实例变量，**实例变量**会在对象实例化时随着对象一块**分配在Java堆**中。

2）这里所设置的初始值通常情况下是数据类型默认的零值（如0、0L、null、false等），而不是被在Java代码中被显式地赋予的值。

## 4. 解析**（了解）**

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。

符号引用（Symbolic Reference）：符号引用以一组符号来描述所引用的目标，符号引用可以是任何形式的字面量，符号引用与虚拟机实现的内存布局无关，引用的目标并不一定已经在内存中。

直接引用（Direct Reference） ：直接引用可以是直接**指向目标的指针**、相对偏移量或是一个能**间接定位到目标的句柄**。直接引用是与虚拟机实现的内存布局相关的，同一个符号引用在不同的虚拟机实例上翻译出来的直接引用一般都不相同，如果有了直接引用，那引用的目标必定已经在内存中存在。

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

2、字段解析：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。

3、类方法解析：对类方法的解析与对字段解析的搜索步骤差不多，只是多了判断该方法所处的是类还是接口的步骤，而且对类方法的匹配搜索，是先搜索父类，再搜索接口。

4、接口方法解析：与类方法解析步骤类似，只是接口不会有父类，因此，只递归向上搜索父接口就行了。

## 5. 初始化**（了解）**

类初始化阶段是类加载过程的最后一步，前面的类加载过程中，除了加载（Loading）阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。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，为类的静态变量赋予正确的初始值，JVM负责对类进行初始化，主要对类变量进行初始化（静态变量static）。在Java中对类变量进行初始值设定有两种方式：

①声明类变量时指定初始值

②使用静态代码块为类变量指定初始值

JVM初始化步骤

1、假如这个类还没有被加载和连接，则程序先加载并连接该类

2、假如该类的直接父类还没有被初始化，则先初始化其直接父类

3、假如类中有初始化语句，则系统依次执行这些初始化语句

**初始化阶段是执行类构造器()方法的过程。**

类初始化的触发条件:**只有当对类的主动使用的时候才会导致类的初始化。**

(1) 创建类的实例，也就是new的方式

(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（如Class.forName(“com.shengsiyuan.Test”)）

(5) 初始化某个类的子类，则其父类也会被初始化

(6) Java虚拟机启动时被标明为启动类的类（Java Test），直接使用java.exe命令来运行某个主类

除此以外，所有其他方式都不会触发初始化，称为被动引用。

# 结束生命周期

在以下情况的时候，Java虚拟机会结束生命周期   
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

最后加载完可以使用的流程：

运行类过程：方法区找到方法--堆中实例化对象--调用栈（指向堆中实例）