[**类的生命周期**](http://www.cnblogs.com/chy2055/p/5124520.html)

　　类加载器，顾名思义，类加载器(class loader)用来加载Java类到Java虚拟机中。

　　一般来说，Java虚拟机使用Java类的方式如下：

**Java源程序(.java 文件)在经过Java编译器编译之后就被转换成Java字节代码(.class 文件)。类加载器负责读取Java字节代码，并转换成java.lang.Class类的一个实例。**

　　每个这样的实例用来表示一个Java类。通过此实例的newInstance()方法就可以创建出该类的一个对象。实际的情况可能更加复杂，比如Java字节代码可能是通过工具动态生成的，也可能是通过网络下载的。

　　基本上所有的类加载器都是java.lang.ClassLoader类的一个实例。

　　其实我们研究类加载器主要研究的就是类的生命周期。

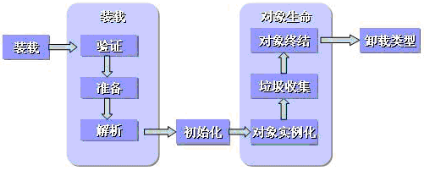
　　Java虚拟机中的几个比较重要的内存区域：

**堆区：**  用于存放类的对象实例。

**栈区：**也叫java虚拟机栈，是由一个一个的栈帧组成的后进先出的栈式结构，栈桢中存放方法运行时产生的局部变量、方法出口等信息。当调用一个方法时，虚拟机栈中就会创建一个栈帧存放这些数据，当方法调用完成时，栈帧消失，如果方法中调用了其他方法，则继续在栈顶创建新的栈桢。

**常量池：**常量池是方法区的一部分，主要用来存放常量和类中的符号引用等信息。

**方法区：** 在java虚拟机中有一块专门用来存放已加载的类信息、常量、静态变量以及方法代码的内存区域。

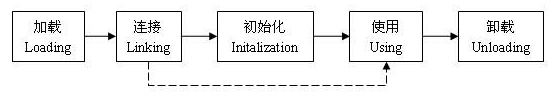


**类的生命周期**

**当我们编写一个java的源文件后，经过编译会生成一个后缀名为class的文件，这种文件叫做字节码文件，只有这种字节码文件才能够在java虚拟机中运行。**

**java类的生命周期就是指一个class文件从加载到卸载的全过程。**

**一个java类的完整的生命周期会经历加载、连接、初始化、使用、和卸载五个阶段，当然也有在加载或者连接之后没有被初始化就直接被使用的情况，这里我们主要来研究类加载器所执行的部分，也就是加载，链接和初始化。**如图所示：



下面我先简单看一下类加载器所执行的三部分的简单介绍

**1、加载：**查找并加载类的二进制数据

**2、连接**

**–验证：**确保被加载的类的正确性

**–准备：**为类的静态变量分配内存，并将其初始化为默认值

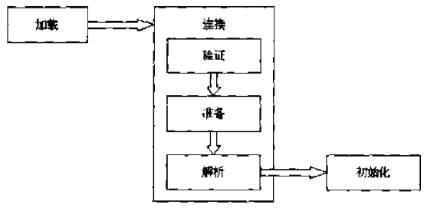
**–解析：**把类中的符号引用转换为直接引用

**3、初始化：**为类的静态变量赋予正确的初始值

　　从上边我们可以看出类的静态变量赋了两回值。

　　这是为什么呢？原因是，**在连接过程中时为静态变量赋值为默认值，也就是说，只要是你定义了静态变量，不管你开始给没给它设置，系统都为他初始化一个默认值。到了初始化过程，系统就检查是否用户定义静态变量时有没有给设置初始化值，如果有就把静态变量设置为用户自己设置的初始化值，如果没有还是让静态变量为初始化值**

　　类的加载、连接和初始化



**类的加载：**

**类的加载指的是将类的.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。**

**这里的class对象其实就像一面镜子一样，外面是类的源程序，里面是class对象，它实时的反应了类的数据结构和信息。**

**加载.class文件的方式：**

**1、从本地系统中直接加载**

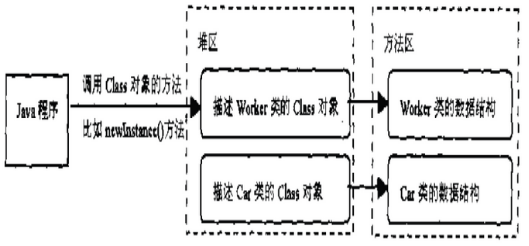
**2、通过网络下载.class文件**

**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**

**类的加载过程**



**结论：**

**1、类的加载的最终产品是位于堆区中的Class对象**

**2、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**

**Java虚拟机给我们提供了两种类加载器：**

**1、Java虚拟机自带的加载器**

**a. 根类加载器**(使用C++编写，程序员无法在Java代码中获得该类)

**b. 扩展加载器，使用Java代码实现**

**c. 系统加载器(应用加载器)，使用Java代码实现**

**2、用户自定义的类加载器**

**java.lang.ClassLoader的子类**

     　　   用户可以定制类的加载方式

我们看一下API对ClassLoader的介绍：

　　类加载器是负责加载类的对象。

　　ClassLoader 类是一个抽象类。如果给定类的二进制名称，那么类加载器会试图查找或生成构成类定义的数据。一般策略是将名称转换为某个文件名，然后从文件系统读取该名称的“类文件”。每个class对象都包含一个对定义它的 ClassLoader 的引用。

我们再来看一下Class类的一个方法getClassLoader

　　public [ClassLoader](http://blog.csdn.net/csh624366188/java/lang/ClassLoader.html) getClassLoader()

　　返回该类的类加载器。有些实现可能使用null来表示根类加载器。如果该类由根类加载器加载，则此方法在这类实现中将返回null。

**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。**大家在做web开发的时候有可能会出现这种问题，比如我们在做测试的时候是用的jdk1.6，而我们在部署的时候我们用的是jdk1.5.这时候就很可能汇报LinkageError错误，版本不兼容。

**类的连接:**

　　类被加载后，就进入连接阶段。

**连接就是将已经读入到内存的类的二进制数据合并到虚拟机的运行时环境中去。**

**验证：**

**当一个类被加载之后，必须要验证一下这个类是否合法，比如这个类是不是符合字节码的格式、变量与方法是不是有重复、数据类型是不是有效、继承与实现是否合乎标准等等。总之，这个阶段的目的就是保证加载的类是能够被jvm所运行。**

　　很多人都感觉，既然这个类都通过编译加载到内存里了，那肯定就是合法的了，为什么还要验证呢，这是因为这里的验证时为了避免有人恶意编写class文件，也就是说并不是通过编译得到的class文件。所以这里验证其实是检查的class文件的内部结构是否符合字节码的要求。

**准备：**

**准备阶段的工作就是为类的静态变量分配内存并设为jvm默认的初值，对于非静态的变量，则不会为它们分配内存。**

　　有一点需要注意，这时候，**静态变量的初值为jvm默认的初值，而不是我们在程序中设定的初值。**

　　jvm默认的初值是这样的：

　　基本类型(int、long、short、char、byte、boolean、float、double)的默认值为0。引用类型的默认值为null。

　　常量的默认值为我们程序中设定的值，比如我们在程序中定义final static int a = 100，则准备阶段中a的初值就是100。

**解析：**这一阶段的任务就是把常量池中的符号引用转换为直接引用。

那么什么是符号引用，什么又是直接引用呢？

　　我们来举个例子：我们要找一个人，我们现有的信息是这个人的身份证号是1234567890。只有这个信息我们显然找不到这个人，但是通过公安局的身份系统，我们输入1234567890这个号之后，就会得到它的全部信息：比如山东省滨州市滨城区18号张三，通过这个信息我们就能找到这个人了。这里，123456790就好比是一个符号引用，而山东省滨州市滨城区18号张三就是直接引用。

　　在内存中也是一样，比如我们要在内存中找一个类里面的一个叫做show的方法，显然是找不到。但是在解析阶段，jvm就会把show这个名字转换为指向方法区的的一块内存地址，比如c17164，通过c17164就可以找到show这个方法具体分配在内存的哪一个区域了。这里show就是符号引用，而c17164就是直接引用。在解析阶段，jvm会将所有的类或接口名、字段名、方法名转换为具体的内存地址。

**类的初始化:**

　　在类的生命周期执行完加载和连接之后就开始了类的初始化。

　　在类的初始化阶段，java虚拟机执行类的初始化语句，为类的静态变量赋值。

在程序中，类的初始化有两种途径：

　　(1) 在变量的声明处赋值

　　(2) 在静态代码块处赋值

　　比如下面的代码，a就是第一种初始化，b就是第二种初始化

　　　　public class Test{

　　　　　　public static int a = 0;

　　　　　　public static int b;

　　　　　　static {

　　　　　　　　b = 2;

　　　　　　}

　　　　}

 静态变量的声明和静态代码块的初始化都可以看做静态变量的初始化，类的静态变量的初始化是有顺序的。顺序为类文件从上到下进行初始化，想到这，想起来一个很无耻的面试题，分享给大家看一下：

[复制代码](javascript:void(0);)

class Singleton{

　　　　private static Singleton singleton = new Singleton();

　　　　Public static int counter1;

　　　　Public static int counter2 = 0;

　　　　Private Singleton(){

　　　　　　Counter1++;

　　　　　　Counter2++;

　　　　}

　　　Public static Singleton getInstance(){

　　　　　　Return singleton;

　　　　}

Public static void main(String[] args){

Singleton singleton = Singleton.getInstance();

System.out.println(“counter1= ” + singleton.counter1);

System.out.println(“counter2= ” + singleton.counter2);

}

}

[复制代码](javascript:void(0);)

 大家先看看这里的程序会输出什么？

不知道大家的答案是什么，如果不介意的话可以把你的答案写到评论上，看看有多少人的答案和你一样的。我先说说我刚开始的答案吧。我认为会输出：

　　counter1 = 1

　　Counter2 = 1

不知道大家的答案是不是这个，反正我的是。下面我们来看一下正确答案：

　　Counter1=1

　　Counter2=0

 不知道你做对没有，反正我刚开始做错了。

好，现在我来解释一下为什么会是这个答案。在给出解释之前，我们先来看一个概念：

**Java程序对类的使用方式可分为两种：**

**(1) 主动使用**

**(2) 被动使用**

**所有的Java虚拟机实现必须在每个类或接口被Java程序“首次主动使用”时才初始化他们.**

**主动使用:**

**① 创建类的实例**

**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.alibaba.Test”))**

**⑤ 初始化一个类的子类**

**⑥ Java虚拟机启动时被标明为启动类的类(Java Test)**

　　OK，我们开始解释一下上面的答案，程序开始运行，首先执行main方法，执行main方法第一条语句，调用Singleton类的静态方法，这里调用Singleton类的静态方法就是主动使用Singleton类。所以开始加载Singleton类。在加载Singleton类的过程中，首先对静态变量赋值为默认值，

　　Singleton=null

　　counter1 = 0

　　Counter2 = 0

　　给他们赋值完默认值值之后，要进行的就是对静态变量初始化，对声明时已经赋值的变量进行初始化。我们上面提到过，初始化是从类文件从上到下赋值的。所以首先给Singleton赋值，给它赋值，就要执行它的构造方法，然后执行counter1++;counter2++;所以这里的counter1 = 1;counter2 = 1;执行完这个初始化之后，然后执行counter2的初始化，我们声明的时候给他初始化为0 了，所以counter2 的值又变为了0.

初始化完之后执行输出。所以这里是：

　　counter1 = 1

　　counter2 = 0

**类初始化步骤：**

　　(1) 假如一个类还没有被加载或者连接，那就先加载和连接这个类

　　(2) 假如类存在直接的父类，并且这个父类还没有被初始化，那就先初始化直接的父类

　　(3) 假如类中存在初始化语句，那就直接按顺序执行这些初始化语句

 　　在上边我们我们说了java虚拟机实现必须在每个类或接口被Java程序“首次主动使用”时才初始化他们，上面也举出了六种主动使用的说明。除了上述六种情形，其他使用Java类的方式都被看作是被动使用，不会导致类的初始化。程序中对子类的“主动使用”会导致父类被初始化；但对父类的“主动”使用并不会导致子类初始化(不可能说生成一个Object类的对象就导致系统中所有的子类都会被初始化)

注：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。

　　当java虚拟机初始化一个类时，要求它的所有的父类都已经被初始化，但这条规则并不适用于接口。

　　在初始化一个类时，并不会先初始化它所实现的接口.

　　在初始化一个接口时，并不会先初始化它的父接口.

 　　因此，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变量时，才会导致该接口的初始化。只有当程序访问的静态变量或静态方法确实在当前类或当前接口中定义时，才可以认为是对类或接口的主动使用 。如果是调用的子类的父类属性，那么子类不会被初始化。