**类加载机制**

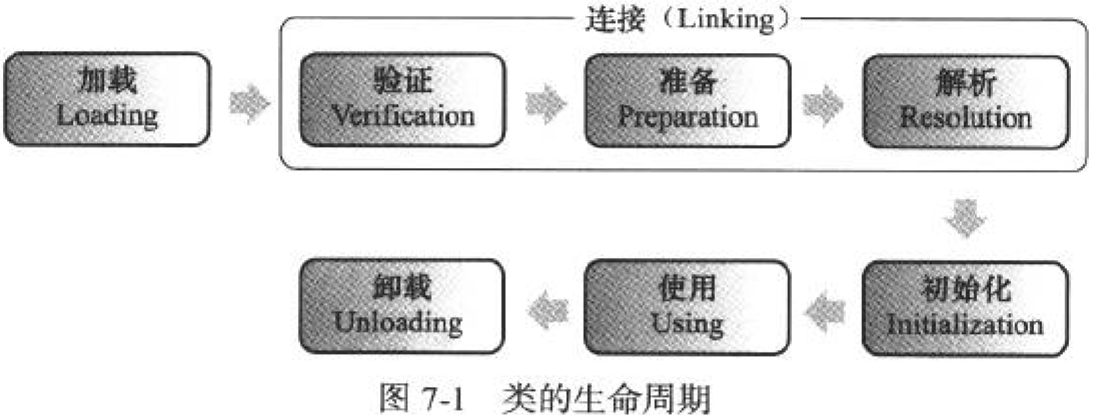
虚拟机把描述类的数据从Class文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成被虚拟机直接使用的Java类型，这就是虚拟机的类加载机制。

缺点：类型的加载、链接、初始化都是在运行期间完成，这样会增加一部分性能开销。

优点：Java天生可以动态扩展的语言特性，就是依赖运行期间的动态加载和动态链接特点实现的，可以提高程序的灵活性，例如，编写一个面向接口的应用程序，可以在运行时指定其具体的实现类。还可以通过Java预定义和自定义类加载器，让应用程序在运行期间通过网络或者在其它地方加载一个二进制流作为程序代码的一部分。

* **类加载的时机**

类从被加载到虚拟机内存开始，到卸载出内存为止，它的整个生命周期包括：加载、验证、准备、分析、初始化、使用、卸载7个阶段。其中验证、准备、分析可以合并为连接，如下如：



这些阶段会按部就班的**开始**，并不是按部就班的“**进行**”或“**完成**”，因为它们会相互交叉的进行，它们会在一个阶段执行的过程中调用、激活另一个阶段。

注意：解析阶段在某种情况下会在初始化阶段之后执行，这是为了支持Java语言的运行时绑定（也称为动态绑定）。

* **类加载过程**

类加载的全过程包括：加载、验证、准备、解析、初始化，这5个阶段的全过程。

**加载**

“加载”只是“类加载过程”中的一个阶段而已，所以要注意这个概念的区别。加载阶段只做以下3件事：

1. 通过一个类的全限定名（包名+类名）来获取这个类的二进制字节流。
2. 将这个字节流中的静态存储结构转化为方法区的运行时数据结构。
3. 在内存中生成一个这个类的java.lang.Class对象（存放在方法区中），作为方法区中这个类的各种数据的访问入口。

对于数组而言，数组本身不通类过加载器创建，它是由Java虚拟机直接创建。

**验证**

验证是连接阶段的第一步，验证的目的是为了保证Class文件的字节流中包含的字符信息符合当前虚拟机的要求，并且不会危害虚拟机的自身安全。

**准备**

参考：<https://blog.csdn.net/honjane/article/details/51835636>

准备阶段仅仅为类变量（被static修饰的成员变量）分配内存，设置初始值。这里说的设置初始值是指“零”值：

public static int a = 123;

在准备阶段设置的初始值是0，而不是123。假如这个类变量a同时被final修饰，那么在准备阶段会被赋值为123，并存储在方法区的常量池中。

注意：只有被**static final**修饰的**基本类型和String**才会在准备阶段被赋值，其他的变量都将在类构造器中进行初始化。

**解析**

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。

**初始化**

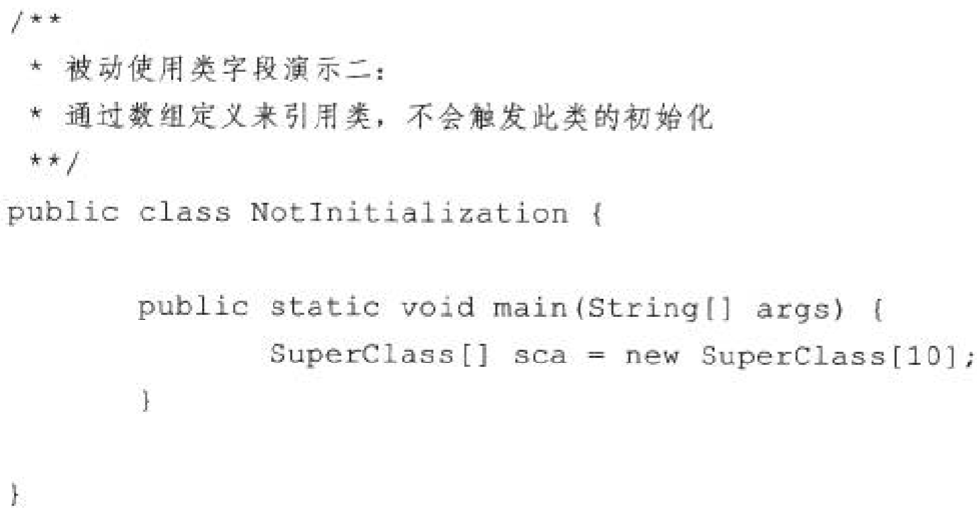
初始化是执行类构造器<clinit>()方法的过程。类构造器的执行包括两部分：类变量（static修饰的变量）赋值和静态语句块的执行，它也是由这两部分合并产生的。如果一个类中没有静态语句块，也没有对类变量的赋值操作，那么编译器可以不为这个类生成<clinit>()方法。

对于初始化阶段，虚拟机规范中规定了有且只有一下5中情况才会触发初始化过程（而加载、验证、准备自然要在此之前开始）：

1. 使用new关键字实例化一个对象、读取或设置一个类的静态字段（除了被final修饰的字段）、调用一个类的静态方法。
2. 使用java.lang.reflect包方法对类进行反射调用的时候。
3. 初始化一个类的时候如果它的父类没有初始化，则需要先初始化其父类。
4. 当虚拟机启动时，用户需要指定一个主类（包含main()方法的类）。
5. 使用动态语言支持的类（包含main()方法的类）。

上面这5种场景中的行为被称为对一个类的主动引用。除此之外的引用方式成为被动引用，被动引用是不会出发类的初始化的。下面几个案例来说明什么是被动引用：

1. 通过子类引用父类的静态字段，不会导致子类初始化。只有直接定义这个静态字段的类才会被初始化。至于是否会出发子类的加载和验证在虚拟机规范中没有给出明确规定。
2. 通过数组定义来引用这个类，不会触发此类初始化：



1. 引用类常量（被static final修饰的变量）时，不会定义常量类的初始化。常量在编译阶段会被加载到调用类的常量池中，以后调用类对常量的引用都会被转化为调用类对自身常量池的引用了：



另外，但是一个接口在初始化时，并不要求其父接口全部都完成了初始化，只有在真正使用到父类接口的时候（比如引用接口中定义的常量）才会初始化。接口中也不可以存在静态语句块的。

**卸载**

除非使用自定义类加载器，或者关闭应用，否则类不会卸载。类的卸载需要同时满足以下3个条件：

1. 该类的所有实例都已经被回收，java堆中不存在该类的任何实例。
2. 该类对应的java.lang.Class对象已经没有被任何地方引用，并且无法通过反射来访问该类的方法。
3. 加载该类的ClassLoader被回收（只能是自定义类加载器，因为启动类加载器永远不会被回收）。