**引言**

        最近有位细心的朋友在阅读笔者的文章时，对[**Java**](http://lib.csdn.net/base/javaee)类的生命周期问题有一些疑惑，笔者打开百度搜了一下相关的问题，看到网上的资料很少有把这个问题讲明白的，主要是因为目前国内java方面的教材大多只是告诉你“怎样做”，但至于“为什么这样做”却不多说，所以造成大家在基础和原理方面的知识比较匮乏，所以笔者今天就斗胆来讲一下这个问题，权当抛砖引玉，希望对在这个问题上有疑惑的朋友有所帮助，文中有说的不对的地方，也希望各路高手前来指正。

        首先来了解一下jvm（java虚拟机）中的几个比较重要的内存区域，这几个区域在java类的生命周期中扮演着比较重要的角色：

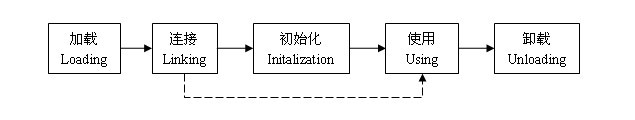
* **方法区：**在java的虚拟机中有一块专门用来存放已经加载的类信息、常量、静态变量以及方法代码的内存区域，叫做方法区。
* **常量池：**常量池是方法区的一部分，主要用来存放常量和类中的符号引用等信息。
* **堆区：**用于存放类的对象实例。
* **栈区：**也叫java虚拟机栈，是由一个一个的栈帧组成的后进先出的栈式结构，栈桢中存放方法运行时产生的局部变量、方法出口等信息。当调用一个方法时，虚拟机栈中就会创建一个栈帧存放这些数据，当方法调用完成时，栈帧消失，如果方法中调用了其他方法，则继续在栈顶创建新的栈桢。

        除了以上四个内存区域之外，jvm中的运行时内存区域还包括**本地方法栈**和**程序计数器**，这两个区域与java类的生命周期关系不是很大，在这里就不说了，感兴趣的朋友可以自己百度一下。

**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，如图所示：



下面我们就依次来说一说这五个阶段。

**加载**

|  |
| --- |
| 定义：在加载阶段，java虚拟机会做什么工作呢？其实很简单，就是找到需要加载的类并把类的信息加载到jvm的方法区中，然后在堆区中实例化一个java.lang.Class对象，作为方法区中这个类的信息的入口。 |

       在java中，我们经常会接触到一个词——**类加载**，它和这里的加载并不是一回事，通常我们说类加载指的是类的生命周期中加载、连接、初始化三个阶段。

类的加载方式比较灵活，最常用的加载方式有两种:

一种是根据类的全路径名找到相应的class文件，然后从class文件中读取文件内容；

另一种是从jar文件中读取。另外，还有下面几种方式也比较常用：

* 从网络中获取：比如10年前十分流行的Applet。
* 根据一定的规则实时生成，比如设计模式中的动态代理模式，就是根据相应的类自动生成它的代理类。
* 从非class文件中获取，其实这与直接从class文件中获取的方式本质上是一样的，这些非class文件在jvm中运行之前会被转换为可被jvm所识别的字节码文件。

       对于加载的时机，各个虚拟机的做法并不一样，但是有一个原则，就是当jvm“预期”到一个类将要被使用时，就会在使用它之前对这个类进行加载。比如说，在一段代码中出现了一个类的名字，jvm在执行这段代码之前并不能确定这个类是否会被使用到，于是，有些jvm会在执行前就加载这个类，而有些则在真正需要用的时候才会去加载它，这取决于具体的jvm实现。我们常用的hotspot虚拟机是采用的后者，就是说当真正用到一个类的时候才对它进行加载。

       加载阶段是类的生命周期中的第一个阶段，加载阶段之后，是连接阶段。有一点需要注意，就是有时连接阶段并不会等加载阶段完全完成之后才开始，而是交叉进行，可能一个类只加载了一部分之后，连接阶段就已经开始了。但是这两个阶段总的开始时间和完成时间总是固定的：加载阶段总是在连接阶段之前开始，连接阶段总是在加载阶段完成之后完成。

**连接**

|  |
| --- |
| 定义：连接阶段比较复杂，一般会跟加载阶段和初始化阶段交叉进行，这个阶段的主要任务就是做一些加载后的验证工作以及一些初始化前的准备工作，可以细分为三个步骤：验证、准备和解析。 |

1. **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。
2. **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。
3. **解析：**这一阶段的任务就是把常量池中的符号引用（变量名）转换为直接引用（地址值）。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：比如安徽省黄山市余暇村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，而安徽省黄山市余暇村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。

        连接阶段完成之后会根据使用的情况（直接引用还是被动引用）来选择是否对类进行初始化。

**初始化**

       如果一个类被直接引用，就会触发类的初始化。在java中，直接引用的情况有：

* 通过new关键字实例化对象、读取或设置类的静态变量、调用类的静态方法。
* 通过反射方式执行以上三种行为。
* 初始化子类的时候，会触发父类的初始化。
* 作为程序入口直接运行时（也就是直接调用main方法）。

        除了以上四种情况，其他使用类的方式叫做被动引用，而被动引用不会触发类的初始化。请看主动引用的示例代码：

**[java]** [view plain](http://blog.csdn.net/zhengzhb/article/details/7517213) [copy](http://blog.csdn.net/zhengzhb/article/details/7517213)

1. **import** java.lang.reflect.Field;
2. **import** java.lang.reflect.Method;
4. **class** InitClass{
5. **static** {
6. System.out.println("初始化InitClass");
7. }
8. **public** **static** String a = **null**;
9. **public** **static** **void** method(){}
10. }
12. **class** SubInitClass **extends** InitClass{}
14. **public** **class** Test1 {
16. /\*\*
17. \* 主动引用引起类的初始化的第四种情况就是运行Test1的main方法时
18. \* 导致Test1初始化，这一点很好理解，就不特别演示了。
19. \* 本代码演示了前三种情况，以下代码都会引起InitClass的初始化，
20. \* 但由于初始化只会进行一次，运行时请将注解去掉，依次运行查看结果。
21. \* @param args
22. \* @throws Exception
23. \*/
24. **public** **static** **void** main(String[] args) **throws** Exception{
25. //  主动引用引起类的初始化一: new对象、读取或设置类的静态变量、调用类的静态方法。
26. //  new InitClass();
27. //  InitClass.a = "";
28. //  String a = InitClass.a;
29. //  InitClass.method();
31. //  主动引用引起类的初始化二：通过反射实例化对象、读取或设置类的静态变量、调用类的静态方法。
32. //  Class cls = InitClass.class;
33. //  cls.newInstance();
35. //  Field f = cls.getDeclaredField("a");
36. //  f.get(null);
37. //  f.set(null, "s");
39. //  Method md = cls.getDeclaredMethod("method");
40. //  md.invoke(null, null);
42. //  主动引用引起类的初始化三：实例化子类，引起父类初始化。
43. //  new SubInitClass();
45. }
46. }

        上面的程序演示了主动引用触发类的初始化的四种情况。

**类的初始化过程**是这样的：**按照顺序自上而下运行类中的变量赋值语句和静态语句，如果有父类，则首先按照顺序运行父类中的变量赋值语句和静态语句。**先看一个例子，首先建两个类用来显示赋值操作：

**[java]** [view plain](http://blog.csdn.net/zhengzhb/article/details/7517213) [copy](http://blog.csdn.net/zhengzhb/article/details/7517213)

1. **public** **class** Field1{
2. **public** Field1(){
3. System.out.println("Field1构造方法");   2
4. }
5. }
6. **public** **class** Field2{
7. **public** Field2(){
8. System.out.println("Field2构造方法");   //5
9. }
10. }

下面是演示初始化顺序的代码：

**[java]** [view plain](http://blog.csdn.net/zhengzhb/article/details/7517213) [copy](http://blog.csdn.net/zhengzhb/article/details/7517213)

1. **class** InitClass2{
2. **static**{
3. System.out.println("运行父类静态代码");   1
4. }
5. **public** **static** Field1 f1 = **new** Field1();   2
6. **public** **static** Field1 f2;    // 3,这里为其分配内存，并且设置默认为null,因为没有直接使用，所以不会初始化，知道连接阶段。
7. }
9. **class** SubInitClass2 **extends** InitClass2{
10. **static**{
11. System.out.println("运行子类静态代码");   //4
12. }
13. **public** **static** Field2 f2 = **new** Field2();  //5 这里才是真的使用，才进行初始化。
14. }
16. **public** **class** Test2 {
17. **public** **static** **void** main(String[] args) **throws** ClassNotFoundException{
18. **new** SubInitClass2();
19. }
20. }

        上面的代码中，初始化的顺序是：第03行，第05行，第11行，第13行。第04行是声明操作，没有赋值，所以不会被运行。而下面的代码：

**[java]** [view plain](http://blog.csdn.net/zhengzhb/article/details/7517213) [copy](http://blog.csdn.net/zhengzhb/article/details/7517213)

1. **class** InitClass2{
2. **public** **static** Field1 f1 = **new** Field1();
3. **public** **static** Field1 f2;
4. **static**{
5. System.out.println("运行父类静态代码");
6. }
7. }
9. **class** SubInitClass2 **extends** InitClass2{
10. **public** **static** Field2 f2 = **new** Field2();
11. **static**{
12. System.out.println("运行子类静态代码");
13. }
14. }
16. **public** **class** Test2 {
17. **public** **static** **void** main(String[] args) **throws** ClassNotFoundException{
18. **new** SubInitClass2();
19. }
20. }

        初始化顺序为：第02行、第05行、第10行、第12行，各位可以运行程序查看结果。

**在类的初始化阶段，只会初始化与类相关的静态赋值语句和静态语句，也就是有static关键字修饰的信息，而没有static修饰的赋值语句和执行语句在实例化对象的时候才会运行。**

**使用**

       类的使用包括主动引用和被动引用，主动引用在初始化的章节中已经说过了，下面我们主要来说一下被动引用：

* 引用父类的静态字段，只会引起父类的初始化，而不会引起子类的初始化。
* 定义类数组，不会引起类的初始化。
* 引用类的常量，不会引起类的初始化。

被动引用的示例代码：

**[java]** [view plain](http://blog.csdn.net/zhengzhb/article/details/7517213) [copy](http://blog.csdn.net/zhengzhb/article/details/7517213)

1. **class** InitClass{
2. **static** {
3. System.out.println("初始化InitClass");
4. }
5. **public** **static** String a = **null**;
6. **public** **final** **static** String b = "b";
7. **public** **static** **void** method(){}
8. }
10. **class** SubInitClass **extends** InitClass{
11. **static** {
12. System.out.println("初始化SubInitClass");
13. }
14. }
16. **public** **class** Test4 {
18. **public** **static** **void** main(String[] args) **throws** Exception{
19. //  String a = SubInitClass.a;// 引用父类的静态字段，只会引起父类初始化，而不会引起子类的初始化
20. //  String b = InitClass.b;// 使用类的常量不会引起类的初始化
21. SubInitClass[] sc = **new** SubInitClass[10];// 定义类数组不会引起类的初始化
22. }
23. }

        最后总结一下使用阶段：**使用阶段包括主动引用和被动引用，主动饮用会引起类的初始化，而被动引用不会引起类的初始化。**

        当使用阶段完成之后，java类就进入了卸载阶段。

**卸载**

       关于类的卸载，笔者在[**单例模式讨论篇：单例模式与垃圾回收**](http://blog.csdn.net/zhengzhb/article/details/7331354)一文中有过描述，在类使用完之后，如果满足下面的情况，类就会被卸载：

* 该类所有的实例都已经被回收，也就是java堆中不存在该类的任何实例。
* 加载该类的ClassLoader已经被回收。
* 该类对应的java.lang.Class对象没有任何地方被引用，无法在任何地方通过反射访问该类的方法。

        如果以上三个条件全部满足，jvm就会在方法区垃圾回收的时候对类进行卸载，类的卸载过程其实就是在方法区中清空类信息，java类的整个生命周期就结束了。

**总结**

        做java的朋友对于对象的生命周期可能都比较熟悉，对象基本上都是在jvm的堆区中创建，在创建对象之前，会触发类加载（加载、连接、初始化），当类初始化完成后，根据类信息在堆区中实例化类对象，初始化非静态变量、非静态代码以及默认构造方法，当对象使用完之后会在合适的时候被jvm垃圾收集器回收。读完本文后我们知道，对象的生命周期只是类的生命周期中使用阶段的主动引用的一种情况（即实例化类对象）。而类的整个生命周期则要比对象的生命周期长的多。

如果有疑问或者发现本文中不对的地方，欢迎各位留言。本文doc文档版本的下载地址：

csdn资源：<http://download.csdn.net/detail/zhengzhb/4260878>

 百度文库：<http://wenku.baidu.com/view/5d2c8b6858fafab069dc023d.html?st=1>