# [从一道面试题来认识java类加载时机与过程](https://blog.csdn.net/mashangyou/article/details/21384855?utm_medium=distribute.pc_relevant_t0.none-task-blog-BlogCommendFromMachineLearnPai2-1.nonecase&depth_1-utm_source=distribute.pc_relevant_t0.none-task-blog-BlogCommendFromMachineLearnPai2-1.nonecase)

**1  开门见山**

以前曾经看到过一个java的面试题,当时觉得此题很简单，可是自己把代码运行起来,可是结果并不是自己想象的那样。题目如下:

1. class SingleTon {
2. private static SingleTon singleTon = new SingleTon();
3. public static int count1;
4. public static int count2 = 0;
6. private SingleTon() {
7. count1++;
8. count2++;
9. }
11. public static SingleTon getInstance() {
12. return singleTon;
13. }
14. }
16. public class Test {
17. public static void main(String[] args) {
18. SingleTon singleTon = SingleTon.getInstance();
19. System.out.println("count1=" + singleTon.count1);
20. System.out.println("count2=" + singleTon.count2);
21. }
22. }

错误答案

count1=1

count2=1

 正确答案

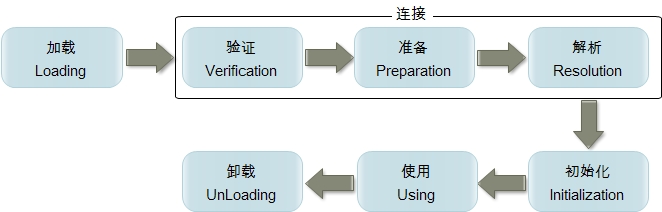
**count1=1**

**count2=0**

为神马？为神马？这要从java的类加载时机说起。

**2 类的加载时机**

类从被加载到虚拟机内存中开始，直到卸载出内存为止，它的整个生命周期包括了：**加载、验证、准备、解析、初始化、使用和卸载**这7个阶段。其中，**验证、准备和解析这三个部分统称为连接（linking）**。



其中，加载、验证、准备、初始化和卸载这五个阶段的顺序是确定的，类的加载过程必须按照这种顺序按部就班的“开始”（仅仅指的是开始，而非执行或者结束，因为这些阶段通常都是互相交叉的混合进行，通常会在一个阶段执行的过程中调用或者激活另一个阶段），而解析阶段则不一定（它在某些情况下可以在初始化阶段之后再开始，这是为了支持Java语言的运行时绑定。

**3 何时开始类的初始化**

什么情况下需要开始类加载过程的第一个阶段:"加载"。虚拟机规范中并没强行约束，这点可以交给虚拟机的的具体实现自由把握，但是对于初始化阶段虚拟机规范是严格规定了如下几种情况，如果类未初始化会对类进行初始化。

1. 创建类的实例
2. 访问类的静态变量(除常量【被final修辞的静态变量】原因:常量一种特殊的变量，因为编译器把他们当作值(value)而不是域(field)来对待。如果你的代码中用到了常变量(constant variable)，编译器并不会生成字节码来从对象中载入域的值，而是直接把这个值插入到字节码中。这是一种很有用的优化，但是如果你需要改变final域的值那么每一块用到那个域的代码都需要重新编译。
3. 访问类的静态方法
4. 反射如(**Class.forName("my.xyz.Test")**)
5. 当初始化一个类时，发现其父类还未初始化，则先出发父类的初始化
6. 虚拟机启动时，定义了main()方法的那个类先初始化

以上情况称为称对一个类进行**“主动引用”**，除此种情况之外，均不会触发类的初始化，称为**“被动引用”**

接口的加载过程与类的加载过程稍有不同。接口中不能使用static{}块。当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，只有真正在使用到父接口时（例如引用接口中定义的常量）才会初始化。

**4 被动引用例子**

1. **子类调用父类的静态变量，子类不会被初始化。只有父类被初始化。。对于静态字段，只有直接定义这个字段的类才会被初始化.**
2. **通过数组定义来引用类，不会触发类的初始化**
3. **访问类的常量，不会初始化类**
4. class SuperClass {
5. static {
6. System.out.println("superclass init");
7. }
8. public static int value = 123;
9. }
11. class SubClass extends SuperClass {
12. static {
13. System.out.println("subclass init");
14. }
15. }
17. public class Test {
18. public static void main(String[] args) {
19. System.out.println(SubClass.value);*// 被动应用1*
20. SubClass[] sca = new SubClass[10];*// 被动引用2*
21. }
22. }

程序运行输出    superclass init

                            123

从上面的输入结果证明了被动引用1与被动引用2

1. class ConstClass {
2. static {
3. System.out.println("ConstClass init");
4. }
5. public static final String HELLOWORLD = "hello world";
6. }
8. public class Test {
9. public static void main(String[] args) {
10. System.out.println(ConstClass.HELLOWORLD);*// 调用类常量*
11. }
12. }

程序输出结果

hello world

从上面的输出结果证明了被动引用3

**5 类的加载过程**

**5.1 加载**

  “加载”(Loading)阶段是“类加载”(Class Loading)过程的第一个阶段，在此阶段，虚拟机需要完成以下三件事情：

        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。

       加载阶段即可以使用系统提供的类加载器在完成，也可以由用户自定义的类加载器来完成。加载阶段与连接阶段的部分内容(如一部分字节码文件格式验证动作)是交叉进行的，加载阶段尚未完成，连接阶段可能已经开始。

**5.2 验证**

       验证是连接阶段的第一步，这一阶段的目的是为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

       Java语言本身是相对安全的语言，使用Java编码是无法做到如访问数组边界以外的数据、将一个对象转型为它并未实现的类型等，如果这样做了，编译器将拒绝编译。但是，Class文件并不一定是由Java源码编译而来，可以使用任何途径，包括用十六进制编辑器(如UltraEdit)直接编写。如果直接编写了有害的“代码”(字节流)，而虚拟机在加载该Class时不进行检查的话，就有可能危害到虚拟机或程序的安全。

      不同的虚拟机，对类验证的实现可能有所不同，但大致都会完成下面四个阶段的验证：文件格式验证、元数据验证、字节码验证和符号引用验证。

       1、文件格式验证，是要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。如验证魔数是否0xCAFEBABE；主、次版本号是否正在当前虚拟机处理范围之内；常量池的常量中是否有不被支持的常量类型……该验证阶段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中，经过这个阶段的验证后，字节流才会进入内存的方法区中存储，所以后面的三个验证阶段都是基于方法区的存储结构进行的。

       2、元数据验证，是对字节码描述的信息进行语义分析，以保证其描述的信息符合Java语言规范的要求。可能包括的验证如：这个类是否有父类；这个类的父类是否继承了不允许被继承的类；如果这个类不是抽象类，是否实现了其父类或接口中要求实现的所有方法……

       3、字节码验证，主要工作是进行数据流和控制流分析，保证被校验类的方法在运行时不会做出危害虚拟机安全的行为。如果一个类方法体的字节码没有通过字节码验证，那肯定是有问题的；但如果一个方法体通过了字节码验证，也不能说明其一定就是安全的。

       4、符号引用验证，发生在虚拟机将符号引用转化为直接引用的时候，这个转化动作将在“解析阶段”中发生。验证符号引用中通过字符串描述的权限定名是否能找到对应的类；在指定类中是否存在符合方法字段的描述符及简单名称所描述的方法和字段；符号引用中的类、字段和方法的访问性(private、protected、public、default)是否可被当前类访问

验证阶段对于虚拟机的类加载机制来说，不一定是必要的阶段。如果所运行的全部代码确认是安全的，可以使用**-Xverify：none**参数来关闭大部分的类验证措施，以缩短虚拟机类加载时间。

**5.3 准备**

       准备阶段是为类的静态变量分配内存并将其初始化为默认值，这些内存都将在方法区中进行分配。准备阶段不分配类中的实例变量的内存，实例变量将会在对象实例化时随着对象一起分配在Java堆中。

        public static int value=123;//在准备阶段value初始值为0 。在初始化阶段才会变为123 。

**5.4 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。

       符号引用（Symbolic Reference）：符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。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加载到内存中。

       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，如果有了直接引用，那么引用的目标必定已经在内存中存在。

**5.5 初始化**

       类初始化是类加载过程的最后一步，前面的类加载过程，除了在加载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。

        初始化阶段是执行类构造器<clinit>()方法的过程。<clinit>()方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块(static{}块)中的语句合并产生的。

**6 题目分析**

上面很详细的介绍了类的加载时机和类的加载过程，通过上面的理论来分析本文开门见上的题目

1. class SingleTon {
2. private static SingleTon singleTon = new SingleTon();
3. public static int count1;
4. public static int count2 = 0;
6. private SingleTon() {
7. count1++;
8. count2++;
9. }
11. public static SingleTon getInstance() {
12. return singleTon;
13. }
14. }
16. public class Test {
17. public static void main(String[] args) {
18. SingleTon singleTon = SingleTon.getInstance();
19. System.out.println("count1=" + singleTon.count1);
20. System.out.println("count2=" + singleTon.count2);
21. }
22. }

分析:

1:SingleTon singleTon = SingleTon.getInstance();调用了类的SingleTon调用了类的静态方法，触发类的初始化  
2:类加载的时候在准备过程中为类的静态变量分配内存并初始化默认值 singleton=null count1=0,count2=0  
3:类初始化化，为类的静态变量赋值和执行静态代码快。singleton赋值为new SingleTon()调用类的构造方法  
4:调用类的构造方法后count=1;count2=1  
5:继续为count1与count2赋值,此时count1没有赋值操作,所有count1为1,但是count2执行赋值操作就变为0

[类加载顺序](https://www.zhihu.com/question/49196023)

作者：潜龙勿用  
链接：https://www.zhihu.com/question/49196023/answer/258247412  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

当一个类使用new关键字来创建新的对象的时候，比如Person per = new Person();JVM根据Person()寻找匹配的类，然后找到这个类相匹配的构造方法，这里是无参构造，如果程序中没有给出任何构造方法，则JVM默认会给出一个无参构造。当创建一个对象的时候一定对调用该类的构造方法，构造方法就是为了对对象的数据进行初始化。JVM会对给这个对象分配内存空间，也就是对类的成员变量进行分配内存空间，如果类中在定义成员变量就赋值的话，就按初始值进行运算，如果只是声明没有赋初始值的话，JVM会按照规则自动进行初始化赋值。而成员方法是在对象调用这个方法的时候才会从方法区中加载到栈内存中，用完就立即释放内存空间。

我来谈谈学到类与对象时的一个关键点：初始化顺序。

先假设一个类，如图中一样各组成都有，那么他的初始化顺序是

1、初始化类变量（即static修饰的成员变量），并未赋值。不管写的位置在哪里，只要是类变量，系统总会先找到它进行变量初始化。

2、执行静态代码块和类变量定义式，两者根据写的位置来决定先后，先写先执行。其实从某种角度上看，可以把类变量定义赋值视为两部分：一部分是定义变量，一部分赋值。而这个赋值部分可以看做是一个静态代码块。两个静态代码块的执行顺序自然是看写的位置的先后了。

3、初始化实例变量（即未被static修饰的成员变量），并未赋值。同样的，不管写的位置在哪里，在创建对象时执行到这步时，系统总会找到它进行变量初始化。

4、执行构造代码块和实例变量定义赋值式，两者同样根据写的位置先后来决定执行顺序先后，同样可以按2中所写来理解。但是，这里要注意的就是构造代码块是可以调用静态变量的，实例变量定义赋值式可以看做是只对实例变量进行赋值的构造代码块。

5、执行构造函数。构造函数同样可以调用静态变量和实例变量。

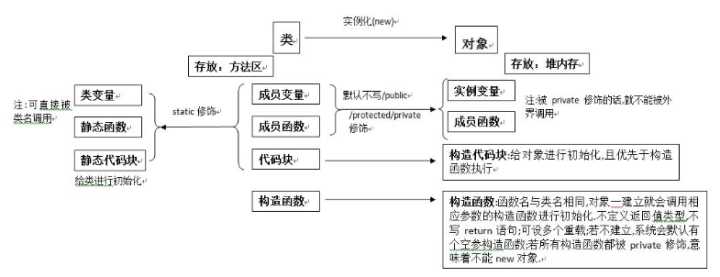
初始化结束。

这里说明一点：这是初始化顺序，不等同于语句程序的执行过程（毕老师的视频里有个很详细的例子讲这个执行过程，不知道的一定要去看）。因此在上面的初始化顺序里没有成员函数（静态或者非静态都没有），这是因为成员函数都是调用了才执行，虽然静态函数已经被加载进了方法区，但初始化过程中并没有执行过。

关于这个初始化顺序，其实一句话可以概括：

先初始化类变量然后赋值，再初始化实例变量然后赋值。

由于静态代码块可以调用静态变量，构造代码块和构造函数可以调用实例变量和静态变量，这块很容易来个看似复杂的代码，将一个变量变来变去的，弄明白这个初始化顺序就会解决很快了。



接下来，看几个例子来验证下：

第一个：

public class JustForTest {

public static void main(String[] args) {

Car c=new Car();

sop("i="+c.i);

}

static void sop(Object obj){

System.out.println(obj);

}

}

class Car{

static int i=1; //定义赋值

static { //静态代码块

i=4;

}

}

运行结果为:i=4.

只改写Car的内部，让静态代码块和静态变量的定义赋值互换位置，其他保持不变：

class Car{

static { //静态代码块

i=4;

}

static int i=1; //定义赋值

}

运行结果为：i=1.

最后来个综合点的，把Car再改写一下：

class Car{

static int i=1; //静态变量定义赋值

Car(){ //构造函数

i=2;

}

static { //静态代码块

i=4;

}

{ //构造代码块

i=3;

}

}

运行结果是：i=2.

按初始化顺序，构造函数是最后初始化的。

a. 加载类

1. 为父类静态属性分配内存并赋值 / 执行父类静态代码段 （按代码顺序）
2. 为子类静态属性分配内存并赋值 / 执行子类静态代码段 （按代码顺序）

b. 创建对象

1. 为父类实例属性分配内存并赋值 / 执行父类非静态代码段 （按代码顺序）
2. 执行父类构造器
3. 为子类实例属性分配内存并赋值 / 执行子类非静态代码段 （按代码顺序）
4. 执行子类构造器

作者：appple  
链接：https://www.zhihu.com/question/49196023/answer/114859346  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。