虚拟机类加载机制

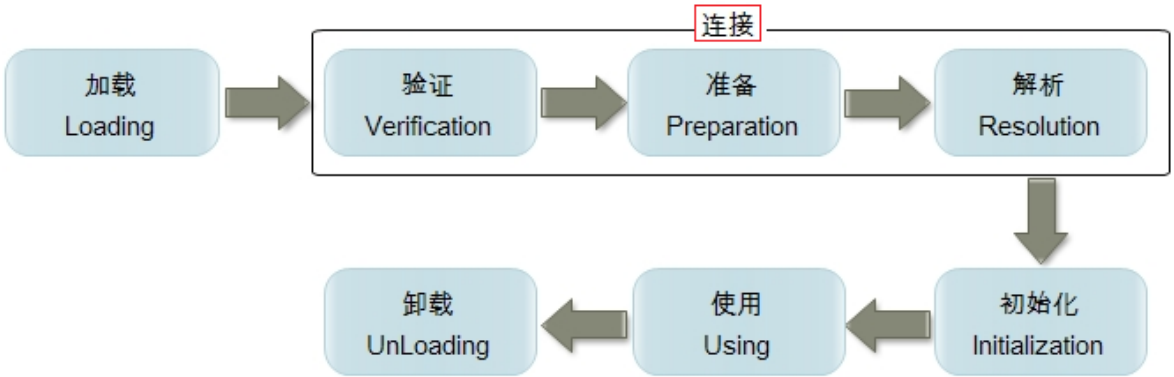
1 概述

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

与那些在编译时需要进行连接工作的语言不同，在Java语言里面，类型的加载、连接和初始化过程都是在程序运行期间完成的，这种策略虽然会令类加载时稍微增加一些性能开销，但是会为Java应用程序提供高度的灵活性，Java里天生可以动态扩展的语言特性就是依赖运行期动态加载和动态连接这个特点实现的。

2 类加载的时机

从类被加载到虚拟机内存中开始，到卸载出内存为止，类的生命周期包括加载（Loading）、验证（Verification）、准备（Preparation）、解析（Resolutino）、初始化（Initialization）、使用（Using）和卸载（Unloading）7个阶段。其中验证、准备、解析3个部分统称为连接（Linking），这7个阶段的发生顺序如下图所示：



加载、验证、准备、初始化和卸载这5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地**开始**（注意是开始，不是按部就班地“进行”或“完成”），这些阶段通常都是相互交叉地混合式进行的通常会在一个阶段的执行过程中调用或激活另一个阶段，但解析阶段则不一定，在某些情况下，解析阶段有可能在初始化阶段结束后开始，以支持Java的动态绑定。

Java虚拟机规范中并没有进行强制约束什么情况下需要开始类加载过程的第一个阶段：加载，这点可以交给虚拟机的具体实现来自由把握。但是对于初始化阶段，虚拟机规范则是严格规定了有且只有5中情况必须立即对类进行初始化（而加载、验证、准备自然需要在此之前开始）。

- 遇到new、getstatic、putstatic或invokestatic这4条字节码指令时，如果类没有进行过初始化，则需要先触发其初始化。生成这4条指令的最常见的Java代码场景是：使用new关键字实例化对象的时候、读取或设置一个类的静态字段（被final修饰，已在编译器把结果放入常量池的静态字段除外）的时候，以及调用一个类的静态方法的时候。
- 使用java.lang.reflect包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。
- 当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。
- 当虚拟机启动时，用户需要指定一个要执行的主类（包含main()的那个类），虚拟机会先初始化这个主类。

- 当使用JDK 1.7的动态语言支持时，如果一个java.lang.invoke.MethodHandle实例最后的解析结果REF_getStatic、REF_outStatic、REF_invokeStatic的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化。

  以上五种触发类进行初始化的场景，称为对一个类进行主动引用。除此之外，所有引用类的方式都不会触发初始化，称为被动引用。下面举3个被动引用的例子：

例1：

```
package org.fenixsoft.classloading;
//通过子类引用父类的静态字段，不会导致子类初始化
public class SuperClass{
    static{
        System.out.println("SuperClass init!");
    }
    public static int value = 123;
}

public class SubClass{
    static{
        System.out.println("SubClass init!");
    }
}
public class NotInitialization{
    public static void main(String[] args){
        System.out.println(SubClass.value);
    }

//OutPut:SuperClass init
//~
```

上述代码运行后没有输出"SubClass init!"。对于静态字段，只有直接定义这个字段的类才会被初始化，因此通过其子类来引用父类中定义的静态字段，只会触发父类的初始化而不会触发子类的初始化。

例2：

```
package org.fenixsoft.classloading;
//通过数组定义来引用类，不会触发此类的初始化
public class NotInitialization{
    public static void main(String[] args){
        SuperClass[] sca = new SuperClass[10];
    }
}
//Output：
//~
```

运行之后没有输出"SupClass init!"，说明并没有触发类org.fenixsoft.classloading.SuperClass的初始化阶段。但是这段代码里面触发了另外一个名为“[Lorg.fenixsoft.classloading.SuperClass”的类的初始化阶段，对于用户代码来说，这病不是一个合法的类名称，它是一个由虚拟机自动生成的、直接继承于java.lang.Object的子类，创建动作由字节码指令newarray触发。

例3：

```
package org.fenixsoft.classloading;
//常亮在编译阶段会存入调用类的常量池中，本质上并没有直接引用到定义常量的类，因此不会触发定义常量的类的初始化。
public class ConstClass{
    static{
        System.out.println("ConstClass init!");
    }
    public static final String HELLOWORLD = "hello world!";
}

public class NotInitialization{
    public static void main(String[] args){
        System.out.println(ConstClass.HELLOWORLD);
    }
}
```

上述代码运行之后，也没有输出“ConstClass init!”，这是因为虽然在Java源码中引用了ConstClass类中的常量HELLOWORLD，但其实在编译阶段通过常量传播优化，已经将此常量的值"hello world!"存储到了NotInitialization类的常量池中，以后NotInitialization对常量ConstClass.HELLOWORLD的引用实际都被转化为NotInitialization类对自身常量池的引用了。也就是说，实际上NotInitialization的Class文件之中并没有ConstClass类的符号引用入口，这两个类在编译成Class之后就不存在任何联系了。

3 类加载的过程

3.1 加载

“加载”是“类加载(Class Loading)”过程的一个阶段，在加载阶段虚拟机需要完成以下3件事情：

- 通过一个类的全限定名来获取定义此类的二进制字节流；
- 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；
- 在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据结构的访问入口。

加载阶段和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，但这些夹在夹在阶段之中进行的动作，仍然属于连接阶段的内容，这两个阶段的开始时间仍然保持着固定的先后顺序。

3.2 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。

从整体上看，验证阶段大致上会完成下面4个阶段的检验动作：文件格式验证、元数据验证、字节码验证、符号引用验证。

- **文件格式验证：**

  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。例如可能验证：是否以魔数0xCAFEBABE开头、主次版本号是否在当前虚拟机处理范围之内、常量池的常亮是否有不被支持的常亮类型等。该验证阶段的主要目的是保证输入的字节流能正确的解析并存储于方法区之内，格式上符合描述一个Java类型信息的要求。这阶段的验证是基于二进制字节流进行的，只有通过了这个阶段的验证后，字节流才会进入内存的方法区中进行存储，所以后面的3个验证阶段全部是基于方法区的存储结构进行的，不会再直接操作字节流。

- **元数据验证：**

  第二阶段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，这个阶段可能包括的验证点有：这个类是否有父类（除了java.lang.Object外，所有的类都应当有父类）、这个类的父类是否继承了不允许被继承的类（被final修饰的类）、如果这个类不是抽象类，是否实现了其父类或者接口之中要求实现的所有方法等等。

- **字节码验证：**

  第三阶段是整个验证过程中最复杂的一个阶段，主要目的是通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。在第二阶段对元数据信息中的数据类型做完校验后，这个阶段将对类的方法体进行校验分析，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事件，例如：

  - 保证任意时刻操作数栈的数据类型与指令代码序列都能配合工作；
  - 保证跳转指令不会跳转到方法体以外的字节码指令上；
  - 保证方法体中的类型转换是有效的。

- **符号引用验证：**

  最后一个阶段的校验发生在虚拟机将符号引用转化为直接引用的时候，这个转化动作将在链接的第三阶段——解析阶段中发生。符号引用验证可以看做是对类自身以外（常量池中的各种符号引用）的信息进行匹配性校验：

  - 符号引用中通过字符串描述的全限定名能否找到对应的类；
  - 在指定类中是否存在符合方法的字段描述符以及简单名称所描述的方法和字段；
  - 符号引用中的类、字段、方法的访问性是否可以被当前类访问。

    符号引用验证的目的是确保解析动作能正常执行。

3.3 准备

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，这些变量所使用的内存都将在方法区进行分配。这时候进行内存分配的仅包括类变量（被static修饰的变量），而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在堆中。其次，这里所说的初始值“通常情况”下是数据类型的零值，假设一个类变量的定义为:

```
public static int value = 123;
```

变量value在准备阶段过后的初始值为0而不是123，因为这时候尚未开始执行任何java方法，而把value赋值为123的putstatic指令是程序被编译后，存放于类构造器<clinit>()方法之中没所以把value赋值为123的动作酱紫初始化阶段才会执行。

至于”特殊情况“是指：如果类字段的属性表中存在ConstantValue属性，那在准备阶段变量value就会被初始化为ConstantVlue属性所指定的值，假设上面类变量value的定义变为：

```
public static final int value = 123;
```

编译时Javac将会为value生成ConstantValue属性，在准备阶段虚拟机就会根据ConstantValue的设置将value赋值为123。

3.4 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。解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行。

3.5 初始化

类初始化阶段是类加载过程的最后一步，到了初始化阶段，才真正开始执行类中定义的java程序代码。在准备阶段，变量已经赋过一次系统要求的初始值，而在初始化阶段，则更具程序员通过程序制定的主观计划去初始化类变量和其他资源，或者说，初始化阶段是执行类构造器<clinit>()方法的过程。下面先看一下<clinit>()方法执行过程中一些可能会影响程序运行行为的特点和细节。

- <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没编译器收集的顺序是由语句在源文件中出现的顺序所决定的，静态语句块中只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句块可以赋值，但是不能访问。

```
public class Test{
    static{
        i = 0;//给变量赋值可以正常编译通过
        System.out.print(i);//编译器会提示“非法向前引用”
    }
    static int i = 1;
}
```

- <clinit>()方法与类的构造函数（或者说实例构造器<init>()方法）不同，它不需要显示地调用父类构造器，虚拟机会保证在子类的<clinit>()方法执行之前，父类的<clinit>()方法已经执行完毕。因此在虚拟机中第一个被执行的<clinit>()方法的类肯定是java.lang.Object。
- 由于父类的<clinit>()方法先执行，也就意味着父类中定义的静态语句块要优先于子类的变量赋值操作。

```
static class Parent{
    public static int A = 1;
    static{
        A = 2;
    }
}
static class Sub extends Parent{
    public static int B = A;
}
public static void main(String[] args){
    System.out.println(Sub.B);
}
//字段B的值是2而不是1
```

一个类中的初始化顺序为：

> **{静态变量，静态初始化块}**（按照出现的顺序）
>
> **{变量/初始化块}**（按照出现的顺序）
>
> **构造方法**

考虑类的继承，则初始化顺序为：

> **父类静态变量、静态初始化块**
>
> **子类静态变量、静态初始化块**
>
> **子类main方法**
>
> **父类变量、初始化块**
>
> **父类构造方法**
>
> **子类变量、初始化块**
>
> **子类构造器**

- <clinit>()方法对于类或者接口来说并不是必需的，如果一个类中没有静态语句块，也没有对变量的赋值操作，name编译器可以不为这个类生产<clinit>()方法。
- 接口中不能使用静态语句块，但仍然有变量初始化的赋值操作，因此接口与类一样都会生成<clinit>()方法。但接口与类不同的是，执行接口的<clinit>()方法不需要先执行父接口的<clinit>()方法。只有当父接口中定义的变量使用时，父接口才会初始化。另外，接口的实现类子初始化时也一样不会执行接口的<clinit>()方法。
- 虚拟机会保证一个类的<clinit>()方法在多线程环境中能被正确的枷锁、同步，如果多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的<clinit>()方法，其他线程都需要阻塞等待，直到活动线程执行<clinit>()方法完毕。如果在一个类的<clinit>()方法中有耗时很长的操作，就可能造成多个线程阻塞。在实际应用中这种阻塞往往是隐藏的。

```
package jvm.classload;

public class DealLoopTest
{
    static class DeadLoopClass
    {
        static
        {
            if(true)
            {
                System.out.println(Thread.currentThread()+"init DeadLoopClass");
                while(true)
                {
                }
            }
        }
    }

    public static void main(String[] args)
    {
        Runnable script = new Runnable(){

            public void run()
```

```
            {
                System.out.println(Thread.currentThread()+" start");
                DeadLoopClass dlc = new DeadLoopClass();
                System.out.println(Thread.currentThread()+" run over");
            }
        };

        Thread thread1 = new Thread(script);
        Thread thread2 = new Thread(script);
        thread1.start();
        thread2.start();
    }
}
```

运行结果：（即一条线程在死循环以模拟长时间操作，另一条线程在阻塞等待）

```
Thread[Thread-0,5,main] start
Thread[Thread-1,5,main] start
Thread[Thread-0,5,main]init DeadLoopClass
```

需要注意的是，其他线程虽然会被阻塞，但如果执行<clinit>()方法的那条线程退出<clinit>()方法后，其他线程唤醒之后不会再次进入<clinit>()方法。同一个类加载器下，一个类型只会初始化一次。

4 类加载器

虚拟机设计团队把类加载阶段中的“通过一个类的全限定名来获取描述此类的二进制字节流”这个动作放到Java虚拟机外部去实现，以便让应用程序自己决定如何去获取所需要的类。实现这动作的代码模块称为“类加载器”。

4.1 类与类加载器

类加载器虽然只用于实现类的加载动作，但它在Java程序中起到的作用却远远不限于类加载阶段。对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立其在Java虚拟机中的唯一性，每一个类加载器，都拥有一个独立的类名称空间。也就是说：比较两个类是否“相等”，只有在这两个类是由同一个类加载器加载的前提下才有意义，否则，即使这两个类来源于同一个Class文件，被同一个虚拟机加载，只要加载它们的类加载器不同，那这两个类就必定不相等。

这里所指的“相等”，包括代表类的Class对象的equals()方法、isAssignableFrom()方法、isInstance()方法的返回结果，也包括使用instanceof关键字做对象所属关系判定等情况。下面是一个例子：

```
public class ClassLoaderTest{
    public static void main(String[] args){

        ClassLoader myLoader = new ClassLoader(){
            @override
            public Class<?> loadClass(String name) throws ClassNotFoundException{
                try{
                    String fileName = name.substring(name.lastIndexOf(".")+1) + ".class";
                    InputStream is = getClass().getResourceAsStream(fileName);
                    if(is == null){

                        return super.loadClass(name);
```

```
                    }
                    byte[] b = new byte[is.available];
                    is.read(b);
                    return defineClass(name,b,0,b.length());
                } catch(IOException e){
                    throw new ClassNotFoundException(name);
                }
            }
        };
        Object obj =
myLoader.loadClass("org.fenixsoft.classloading.ClassLoaderTest").newInstance();
        System.out.println(obj.getClass);
        System.out.println(obj instanceof org.fenixsoft.classloading.ClassLoaderTest);
    }
}
```

输出结果：

```
class org.fenixsoft.classloading.ClassLoaderTest
false
```

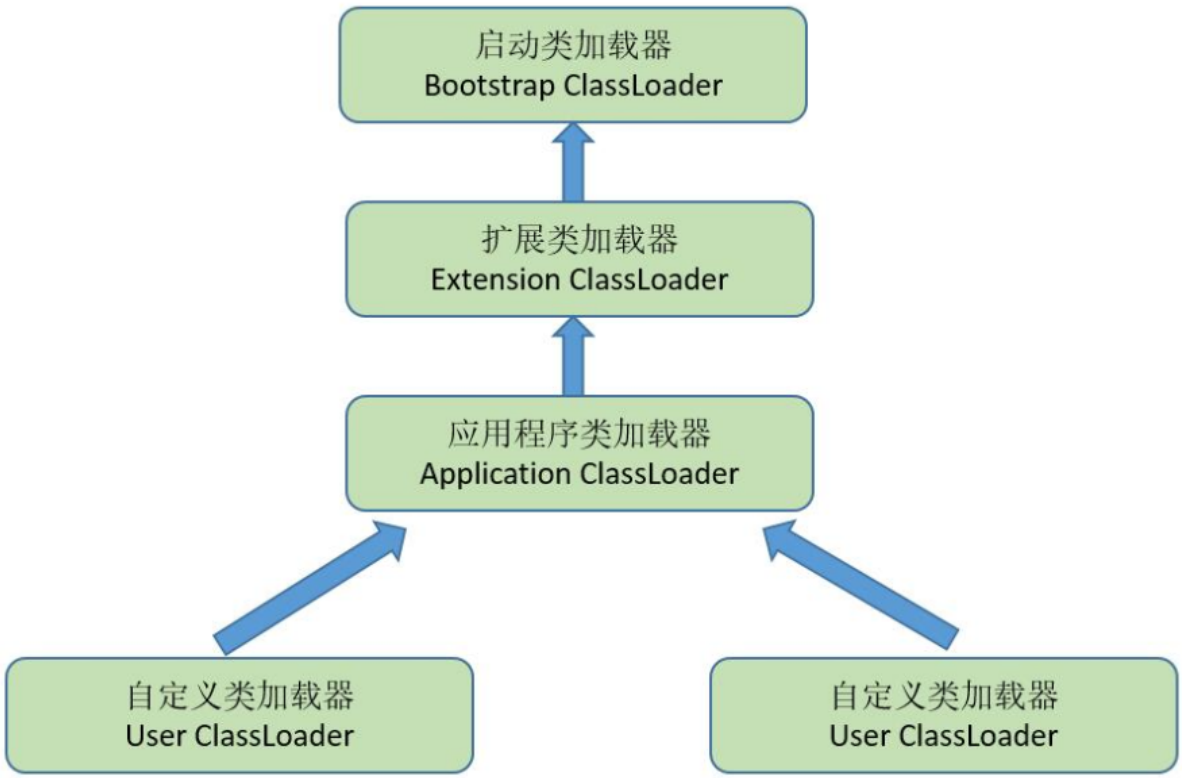
上面代码实现了一个简单的类加载器。可以加载与自己在同一路径下的Class文件，可以从结果中看出，obj对象确实是类org.fenixsoft.classloading.ClassLoaderTest实例化出来的对象，但这个对象与类org.fenixsoft.classloading.ClassLoaderTest做属性类型检查的时候却返回了false，这是因为在虚拟机中存在了两个ClassLoaderTest类，一个是由系统应用程序类加载器加载的，另外一个是由自定义的类加载器加载的，虽然来自同一个Class文件，但依然是两个独立的类，做对象所属类型检查时结果自然为false。

4.2 双亲委派模型

从Java虚拟机的角度来讲，只存在两种不同的类加载器：一种是启动类加载器（Bootstrap ClassLoader），这个类加载器使用C++语言实现，是虚拟机自身的一部分；另一种就是所有其他的类加载器，这些类加载器都由Java语言实现，独立于虚拟机外部，并且都继承自抽象类java.lang.ClassLoader。

绝大部分Java程序都会使用到以下3系统提供的类加载器：

- **启动类加载器（Bootstrap ClassLoader）：**这个类加载器负责将存放在<JAVA_HOME>\lib目录中的，或者被-Xbootclasspath参数所指定的路径中的，并且是虚拟机识别的类库加载到虚拟机内存中。启动类加载器无法被Java程序直接引用，用户在编写自定义类加载器时，如果需要把加载请求委派给引导类加载器，直接使用null代替即可。
- **扩展类加载器（Extendion ClassLoader）：**这个加载器由sun.misc.Launcher$ExtClassLoader实现，它负责加载<JAVA_HOME>\lib\ext目录中的，或者被java.ext.dirs系统变量所指定的路径中的所有类库，开发者可以直接使用扩展类加载器。
- **应用程序类加载器（Application ClassLoader）：**这个类加载器由sun.misc.Launcher$AppClassLoader实现。由于这个类加载器是ClassLoader中的getSystemClassLoader()方法的返回值，所以一般也称为系统加载器。它负责加载用户类路径（ClassPath）上所指定的类库，开发者可以直接使用这个类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。



上图展示的类加载器之间的这种层次关系成为类记载器的双亲委派模型（Parents Delegation Model）。该模型要求除了顶层的启动类加载器外，其余的类加载器都应该有自己的父类加载器，而这种父子关系一般通过**组合（Composition）关系**来实现，而不是通过继承（Inheritance）。

双亲委派模型的工作过程是：某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，依次递归，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成此加载任务时，才自己去加载。

使用双亲委派模型的好处在于**Java类随着它的类加载器一起具备了一种带有优先级的层次关系。**例如类java.lang.Object，它存在在rt.jar中，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的Bootstrap ClassLoader进行加载，因此Object类在程序的各种类加载器环境中都是同一个类。相反，如果没有双亲委派模型而是由各个类加载器自行加载的话，如果用户编写了一个java.lang.Object的同名类并放在ClassPath中，那系统中将会出现多个不同的Object类，程序将混乱。

双亲委派模型的实现：

```
protected synchronized Class<?> loadClass(String name,boolean resolve)throws
ClassNotFoundException{
    //check the class has been loaded or not
    Class c = findLoadedClass(name);
    if(c == null){
        try{
            if(parent != null){
                c = parent.loadClass(name,false);
            }else{
                c = findBootstrapClassOrNull(name);

            }
```

```
            }catch(ClassNotFoundException e){
                //if throws the exception ,the father can not complete the load
            }
            if(c == null){
                c = findClass(name);
            }
        }
        if(resolve){
            resolveClass(c);
        }
        return c;
    }
```

双亲委派模型是Java设计者推荐给开发者的类加载器的实现方式，并不是强制规定的。大多数的类加载器都遵循这个模型，但是JDK中也有较大规模破坏双亲模型的情况，例如线程上下文类加载器（Thread Context ClassLoader）的出现。