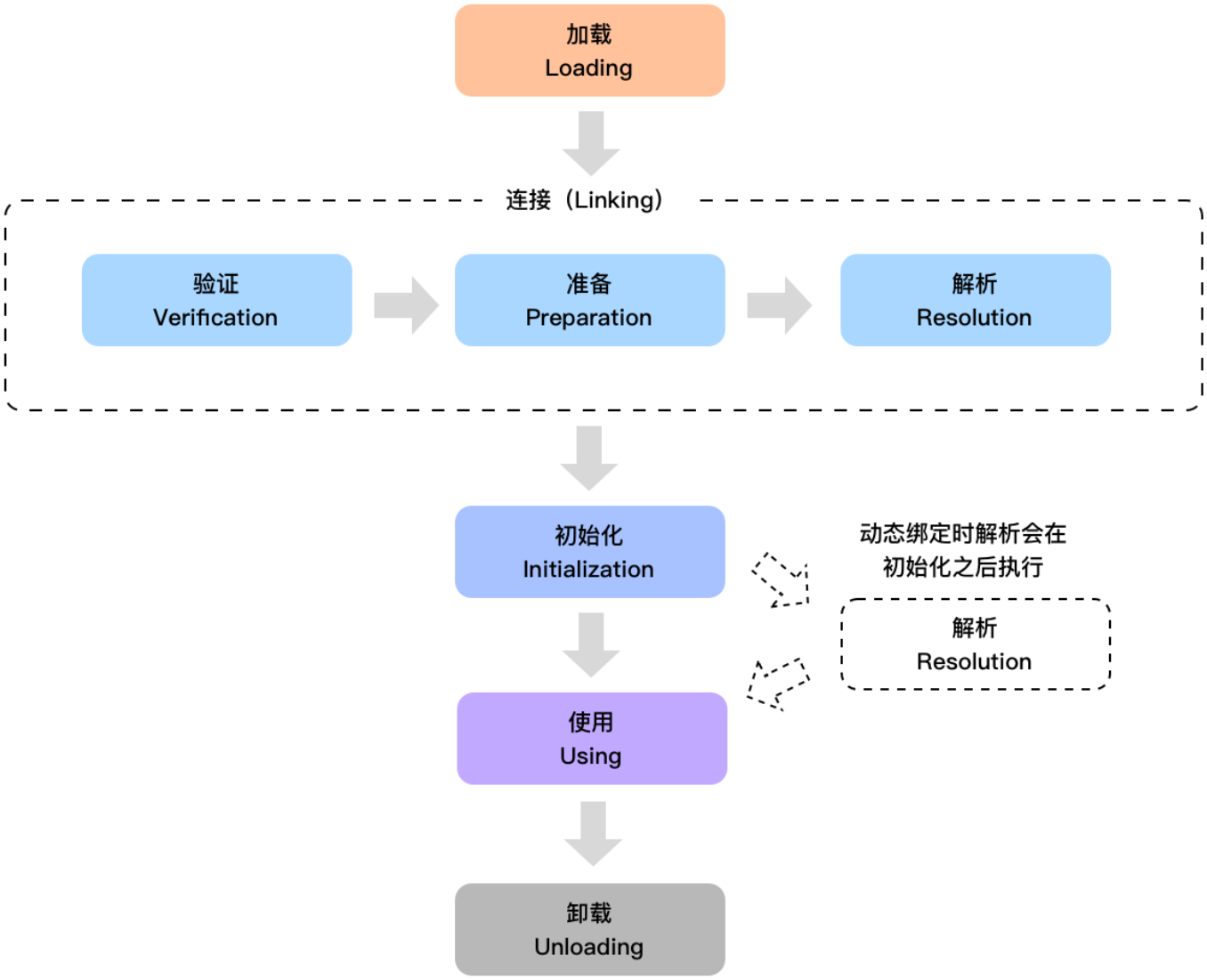
JVM类加载机制解析

JVM类加载过程

JVM把class文件加载到内存，并对数据进行校验、准备、解析、初始化，最终形成JVM可以直接使用的Java类型的过程。

类的生命周期

从类的生命周期而言，一个类包括如下阶段：



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进行，而解析阶段则不一定，它在某些情况下可能在初始化阶段后在开始，因为java支持运行时绑定。

1、加载（重要）

将class字节码文件加载到内存中，并将这些数据转换成方法区中的运行时数据（静态变量、静态代码块、常量池等），在堆中生成一个Class类对象代表这个类（反射原理），作为方法区类数据的访问入口。

- 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。

通过类的权限定名来获取定义此类的二进制流，这里并没有明确指明要从哪里获取以及怎样获取，也就是说并没有明确规定一定要我们从一个 Class 文件中获取。基于此，在Java的发展过程中，充满创造力的开发人员在这个舞台上玩出了各种花样：

- 从zip包中获取，这就是以后jar、ear、war格式的基础
- 从网络中获取，典型应用就是Applet
- 运行时计算生成，典型应用就是动态代理技术
- 由其他文件生成，典型应用就是JSP，即由JSP生成对应的.class文件
- 从数据库中读取，在组件的开发过程中，我们可能会用到组件上传功能，这个时候就会将JAR等其它信息都存放到数据库，在应用初使化的时候，将组件的JAR从数据库中读出来，并一起加载到Classpath中

2、链接（理解）

将Java类的二进制代码合并到JVM的运行状态之中。

验证

确保加载的类信息符合JVM规范，没有安全方面的问题。

正如前面所说，二进制字节流可能有很多种来源，虚拟机如果不检查输入的字节流，对其完全信任的话，很可能会因为载入了有害的字节 流而导致系统崩溃，所以**验证是虚拟机对自身保护的一项重要工作**。

验证阶段大致会完成以下四个阶段的检验动作：

- 文件格式验证
- 元数据验证
- 字节码验证
- 符号引用验证

准备

为静态变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中进行分配。注意此时的设置初始值为默认值（引用类型的初始值为nul），具体赋值在初始化阶段完成。

各个数据类型对应的默认值如下：

数据类型	零值
int	0
long	0L
short	(short)0
chart	'\u0000'
byte	(byte)0
boolean	false
float	0.0f
double	0.0d
reference	null

比如，定义 public static int value = 123 。那么在准备阶段过后，value 的值是 0 而不是 123,把 value 赋值为123是在程序被编译后，存放在类的构造器方法之中，是在初始化阶段才会被执行。但是有一种特殊情况，通过**final**修饰的属性，比如 定义 public final static int value = 123，那么在准备阶段过后，value 就被赋值为123了。

解析

虚拟机常量池内的符号引用替换为直接引用（地址引用）的过程。

符号引用（Symbolic References）

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的定位到目标即可。

下面以简单的代码来理解符号引用：

```
public class Demo1 {
    public static String serial;
    private int count;

    public static void calculate() {
    }

    public int getCount() {
        return count;
    }
}
```

使用`javap -verbose SymbolClass`反编译一下这个类，我们主要看看常量池部分：

```
Constant pool:
   #1 = Methodref          #4.#25         // java/lang/Object."<init>":()V
   #2 = Fieldref           #3.#26         // com/zephyr/demo/SymbolClass.count:I
```

```
   #3 = Class              #27            // com/zephyr/demo/SymbolClass
   #4 = Class              #28            // java/lang/Object
   #5 = Utf8               serial
   #6 = Utf8               Ljava/lang/String;
   #7 = Utf8               count
   #8 = Utf8               I
   #9 = Utf8               <init>
  #10 = Utf8               ()V
  #11 = Utf8               Code
  #12 = Utf8               LineNumberTable
  #13 = Utf8               LocalVariableTable
  #14 = Utf8               this
  #15 = Utf8               Lcom/zephyr/demo/SymbolClass;
  #16 = Utf8               calculate
  #17 = Utf8               getCount
  #18 = Utf8               ()I
  #19 = Utf8               main
  #20 = Utf8               ([Ljava/lang/String;)V
  #21 = Utf8               args
  #22 = Utf8               [Ljava/lang/String;
  #23 = Utf8               SourceFile
  #24 = Utf8               SymbolClass.java
  #25 = NameAndType        #9:#10         // "<init>":()V
  #26 = NameAndType        #7:#8          // count:I
  #27 = Utf8               com/zephyr/demo/SymbolClass
  #28 = Utf8               java/lang/Object
```

上面带`Utf8`的那一行就是符号引用，每行最前面的就是符号，后面就是引用的值。对于变量来说都会有两行成对出现，比如#7 是 count，#8就是 count 的类型 Integer(常量池里简写为 I)。方法如果有返回值，方法和返回值也会成对出现，比如 #17 和 #18，分别代表的方法和返回值类型。

简单理解符号引用就是**对于类、变量、方法的描述**。并且符号引用与虚拟机实现的内存布局无关，引用的目标不一定已经加载到内存中。

直接引用（Direct References）

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内存布局相关的，同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同。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。

解析过程

Java 本身是一个静态语言，但后面又加入了动态加载特性，因此我们理解解析阶段需要从这两方面来考虑。

如果不涉及动态加载，那么一个符号的解析结果是可以缓存的，这样可以避免多次解析同一个符号，因为第一次解析成功后面多次解析也必然成功，第一次解析异常后面重新解析也会是同样的结果。

如果使用了动态加载，前面使用动态加载解析过的符号后面重新解析结果可能会不同。使用动态加载时解析过程发生在在程序执行到这条指令的时候，这就是为什么前面讲的动态加载时解析会在初始化后执行。

整个解析阶段主要做了下面几个工作：

- 类或接口的解析
- 类方法解析
- 接口方法解析
- 字段解析

3、初始化（重要）

初始化阶段是执行类构造器()方法的过程。类构造器()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。

- 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、则需要先初始化其父类。
- 虚拟机会保证一个类的()方法在多线程环境中被正确加锁和同步。

Java程序初始化顺序

1. 父类的静态变量
2. 父类的静态代码块
3. 子类的静态变量
4. 子类的静态代码块
5. 父类的非静态变量
6. 父类的非静态代码块
7. 父类的构造方法
8. 子类的非静态变量
9. 子类的非静态代码块
10. 子类的构造方法

下面用代码演示加载顺序：

```
public class Demo02 {
    public static void main(String[] args) {
        B b = new B();
    }
}

class A{

    static String str1 = "父类A的静态变量";
    String str2 = "父类A的非静态变量";

    static {
        System.out.println("执行了父类A的静态代码块");
    }

    {
        System.out.println("执行了父类A的非静态代码块");
    }

    public A(){
        System.out.println("执行了父类A的构造方法");
    }
```

```
}

class B extends A{

    static String str1 = "子类B的静态变量";
    String str2 = "子类B的非静态变量";

    static {
        System.out.println("执行了子类B的静态代码块");
    }

    {
        System.out.println("执行了子类B的非静态代码块");
    }

    public B(){
        System.out.println("执行了子类B的构造方法");
    }
}
```

控制台输出内容：

```
执行了父类A的静态代码块
执行了子类B的静态代码块
执行了父类A的非静态代码块
执行了父类A的构造方法
执行了子类B的非静态代码块
执行了子类B的构造方法
```

什么情况下，类会被初始化?

对于初始化阶段，虚拟机严格规范了有且只有5种情况下，必须对类进行初始化：

- 当遇到 new、getstatic、putstatic或invokestatic 这4条直接码指令时，比如 new 一个类，读取一个静态字段(未被 final 修饰)、或调用一个类的静态方法时。
- 使用 java.lang.reflect 包的方法对类进行反射调用时 ，如果类没初始化，需要触发其初始化。
- 初始化一个类，如果其父类还未初始化，则先触发该父类的初始化。
- 当虚拟机启动时，用户需要定义一个要执行的主类 (包含 main 方法的那个类)，虚拟机会先初始化这个类。
- 当使用 JDK1.7 的动态动态语言时，如果一个 MethodHandle 实例的最后解析结构为 REFgetStatic、REFputStatic、REF_invokeStatic、的方法句柄，并且这个句柄没有初始化，则需要先触发器初始化。

什么情况下，类不会被初始化?

- 当访问一个静态域时，只有真正声明这个域的类才会被初始化。例如：通过子类引用父类的静态变量，不会导致子类初始化。
- 通过数组定义类引用，不会触发此类的初始化。
- 引用常量不会触发此类的初始化（常量在编译阶段就存入调用类的常量池中了）。

```
public class Demo3 {
```

```
    public static void main(String[] args) throws ClassNotFoundException {
        //主动引用：new一个类的对象
//        People people = new People();
        //主动引用：调用类的静态成员(除了final常量)和静态方法
//        People.getAge();
//        System.out.println(People.age);
        //主动调用：使用java.lang.reflect包的方法对类进行反射调用
//        Class.forName("pri.xiaowd.classloader.People");


        //被动引用:当访问一个静态域时，只有真正声明这个域的类才会被初始化。
        System.out.println(WhitePeople.age);
        //被动引用:通过数组定义引用，不会初始化
//        People[] people = new People[10];
        //被动引用:引用常量不会触发此类的初始化
//        System.out.println(People.num);
    }
    //主动调用：先启动main方法所在的类
//    static {
//        System.out.println("main方法所在的类在虚拟机启动时就加载");
//    }
}

class People{

    static int age = 3;
    static final int num = 20;

    static {
        System.out.println("People被初始化了！");
    }

    public People() {
    }

    public static int getAge() {
        return age;
    }

    public static void setAge(int age) {
        People.age = age;
    }
}

class WhitePeople extends People{

    static {
        System.out.println("WhitePeople被初始化了！");
    }
```

```
}
```