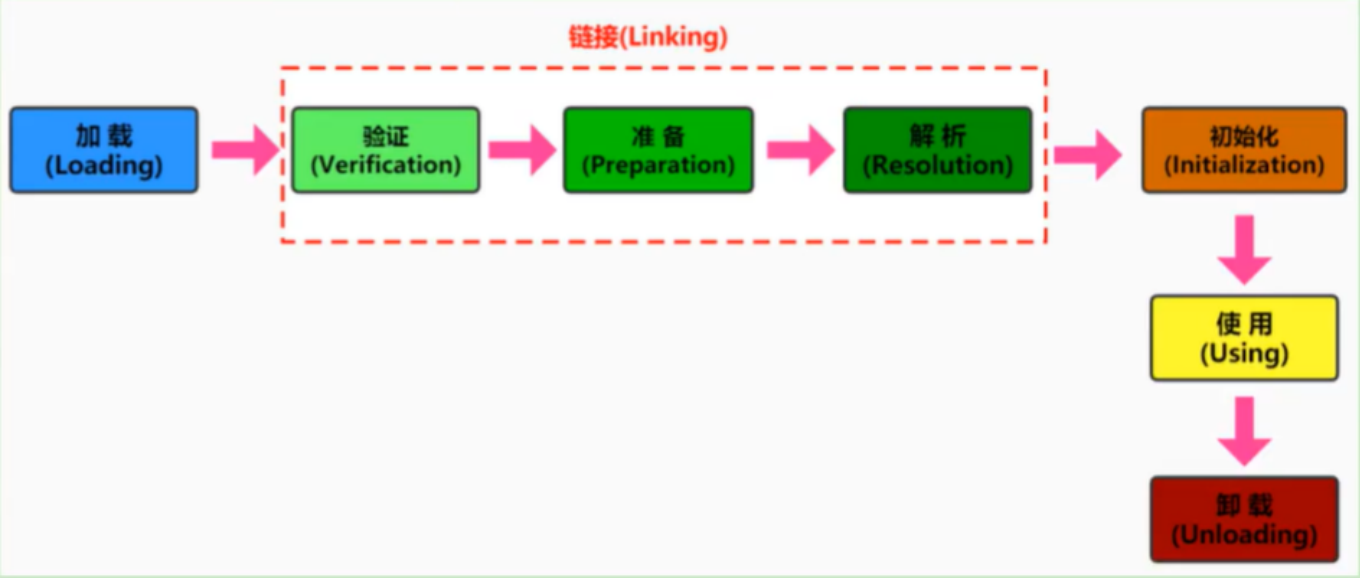
> 笔记来源：尚硅谷 JVM 全套教程，百万播放，全网巅峰（宋红康详解 java 虚拟机）
>
> 同步更新：https://gitee.com/vectorx/NOTE_JVM
>
> https://codechina.csdn.net/qq_35925558/NOTE_JVM
>
> https://github.com/uxiahnan/NOTE_JVM

[toc]

1. 概述

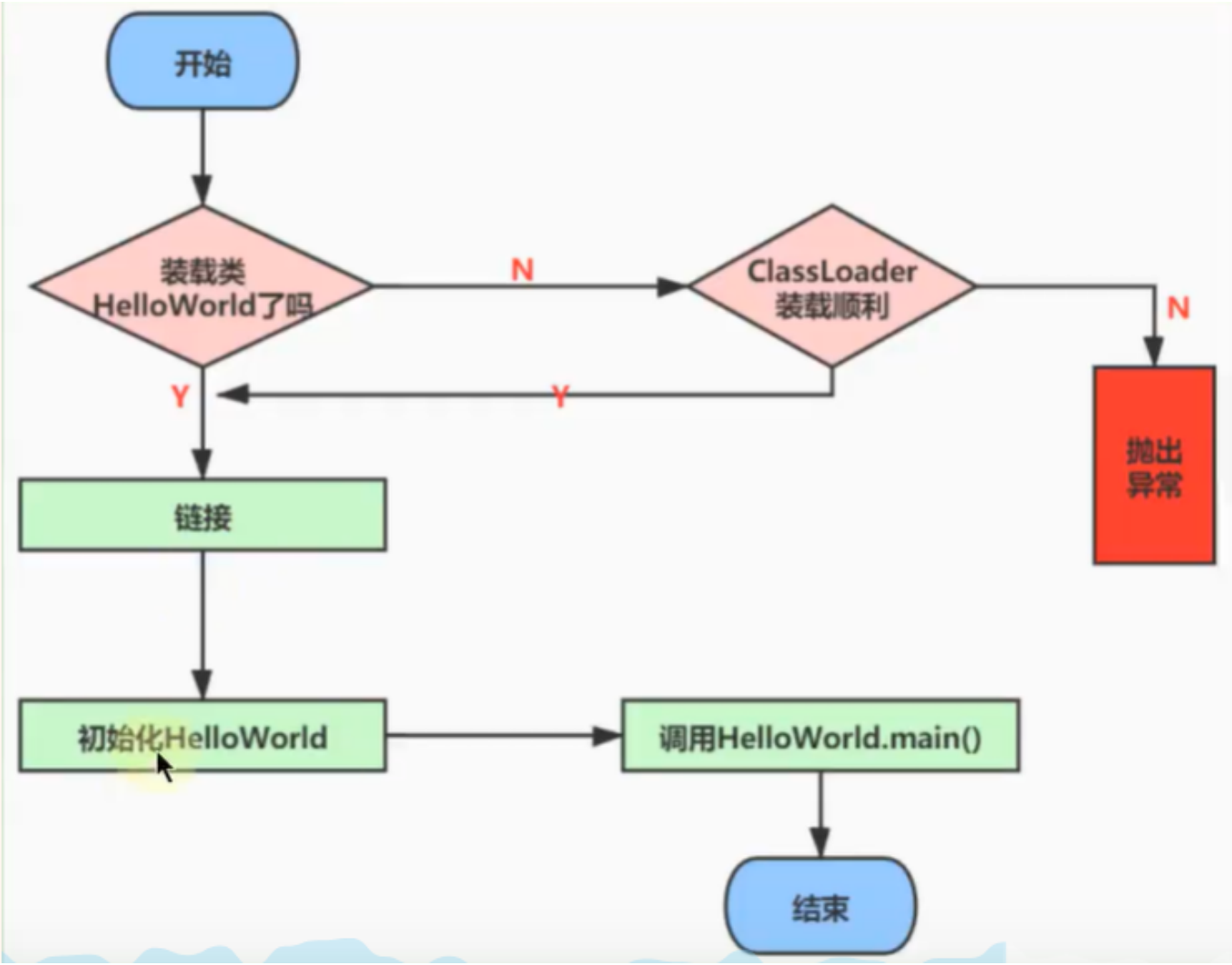
在 Java 中数据类型分为基本数据类型和引用数据类型。基本数据类型由虚拟机预先定义，引用数据类型则需要进行类的加载。

按照 Java 虚拟机规范，从 class 文件到加载到内存中的类，到类卸载出内存为止，它的整个生命周期包括如下 7 个阶段：



其中，验证、准备、解析 3 个部分统称为链接（Linking）

从程序中类的使用过程看



大厂面试题

> 蚂蚁金服：
>
> 描述一下 JVM 加载 Class 文件的原理机制?
>
> 一面：类加载过程
>
> 百度：
>
> 类加载的时机
>
> java 类加载过程?
>
> 简述 java 类加载机制?
>
> 腾讯：
>
> JVM 中类加载机制，类加载过程?
>
> 滴滴：
>
> JVM 类加载机制
>
> 美团：

> Java 类加载过程
>
> 描述一下 jvm 加载 class 文件的原理机制
>
> 京东：
>
> 什么是类的加载？
>
> 哪些情况会触发类的加载？
>
> 讲一下 JVM 加载一个类的过程 JVM 的类加载机制是什么？

2. 过程一：Loading（加载）阶段

2.1. 加载完成的操作

加载的理解

$\color{red}{所谓加载，简而言之就是将Java类的字节码文件加载到机器内存中，并在内存中构建出Java类的原型——类模板对象。}$所谓类模板对象，其实就是 Java 类在]VM 内存中的一个快照，JVM 将从字节码文件中解析出的常量池、类字段、类方法等信息存储到类模板中，这样]VM 在运行期便能通过类模板而获取 Java 类中的任意信息，能够对 Java 类的成员变量进行遍历，也能进行 Java 方法的调用。

反射的机制即基于这一基础。如果 JVM 没有将 Java 类的声明信息存储起来，则 JVM 在运行期也无法反射。

加载完成的操作

$\color{red}{加载阶段，简言之，查找并加载类的二进制数据，生成Class的实例。}$

在加载类时，Java 虚拟机必须完成以下 3 件事情：

- 通过类的全名，获取类的二进制数据流。
- 解析类的二进制数据流为方法区内的数据结构（Java 类模型）
- 创建 java.lang.Class 类的实例，表示该类型。作为方法区这个类的各种数据的访问入口

2.2. 二进制流的获取方式

对于类的二进制数据流，虚拟机可以通过多种途径产生或获得。（只要所读取的字节码符合 JVM 规范即可）

- 虚拟机可能通过文件系统读入一个 class 后缀的文件$\color{red}{（最常见）}$
- 读入 jar、zip 等归档数据包，提取类文件。
- 事先存放在数据库中的类的二进制数据
- 使用类似于 HTTP 之类的协议通过网络进行加载
- 在运行时生成一段 class 的二进制信息等
- 在获取到类的二进制信息后，Java 虚拟机就会处理这些数据，并最终转为一个 java.lang.Class 的实例。

如果输入数据不是 ClassFile 的结构，则会抛出 ClassFormatError。

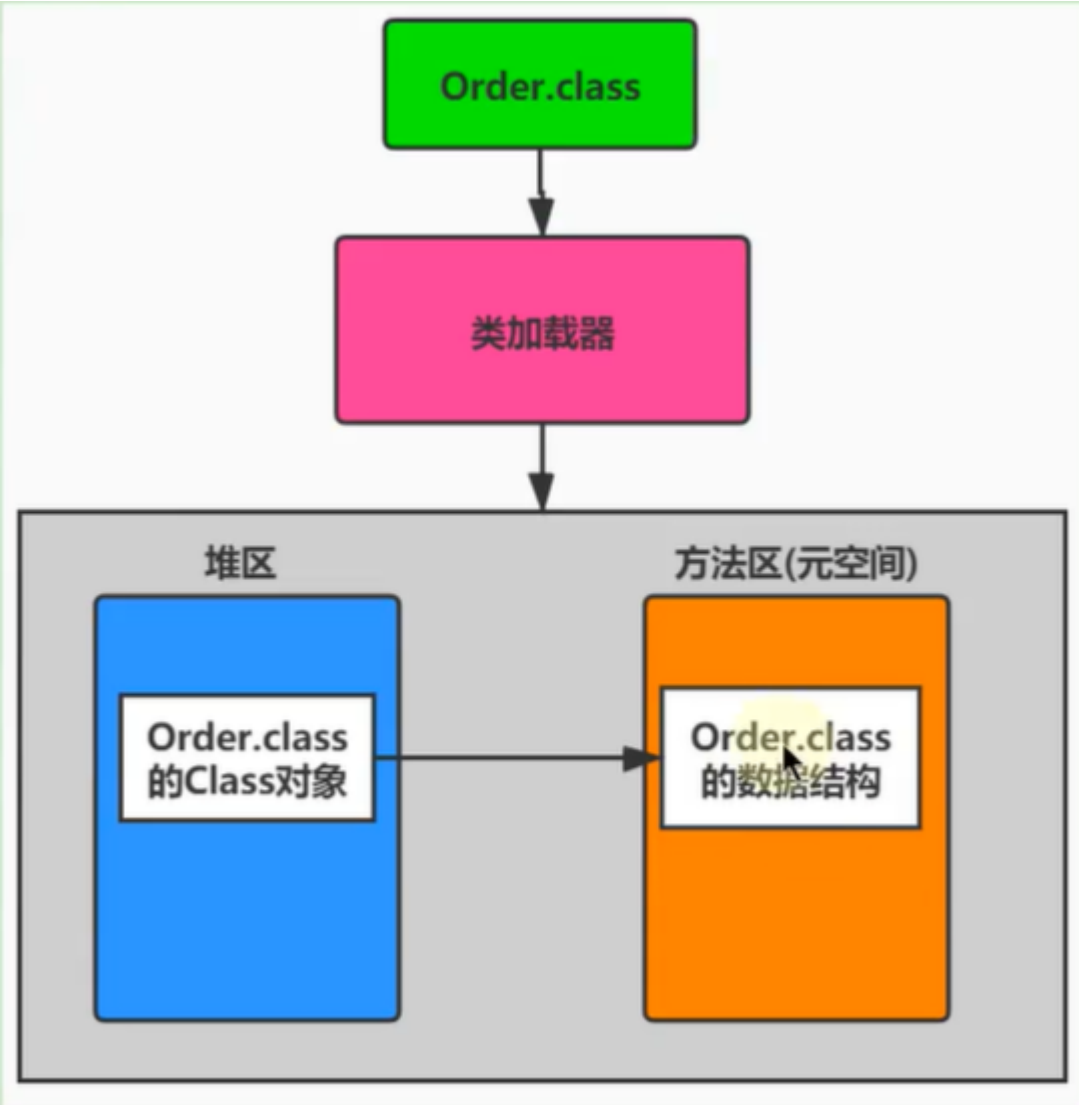
2.3. 类模型与 Class 实例的位置

类模型的位置

加载的类在 JVM 中创建相应的类结构，类结构会存储在方法区（JDKl.8 之前：永久代；J0Kl.8 及之后：元空间）。

Class 实例的位置

类将.class 文件加载至元空间后，会在堆中创建一个 Java.lang.Class 对象，用来封装类位于方法区内的数据结构，该 Class 对象是在加载类的过程中创建的，每个类都对应有一个 Class 类型的对象。




```
Class clazz = Class.forName("java.lang.String");
//获取当前运行时类声明的所有方法
Method[] ms = clazz.getDecla#FF0000Methods();
for (Method m : ms) {
    //获取方法的修饰符
    String mod = Modifier.toString(m.getModifiers());
    System.out.print(mod + "");
    //获取方法的返回值类型
    String returnType = (m.getReturnType()).getSimpleName();
    System.out.print(returnType + "");
    //获取方法名
    System.out.print(m.getName() + "(");
```

```
        //获取方法的参数列表
        Class<?>[] ps = m.getParameterTypes();
        if (ps.length == 0) {
            System.out.print(')');
        }
        for (int i = 0; i < ps.length; i++) {
            char end = (i == ps.length - 1) ? ')' : ',';
            //获取参教的类型
            System.out.print(ps[i].getSimpleName() + end);
        }
    }
```

2.4. 数组类的加载

创建数组类的情况稍微有些特殊，因为数组类本身并不是由类加载器负责创建，而是由 JVM 在运行时根据需要而直接创建的，但数组的元素类型仍然需要依靠类加载器去创建。创建数组类（下述简称 A）的过程：

- 如果数组的元素类型是引用类型，那么就遵循定义的加载过程递归加载和创建数组 A 的元素类型；
- JVM 使用指定的元素类型和数组维度来创建新的数组类。

如果数组的元素类型是引用类型，数组类的可访问性就由元素类型的可访问性决定。否则数组类的可访问性将被缺省定义为 public。

---

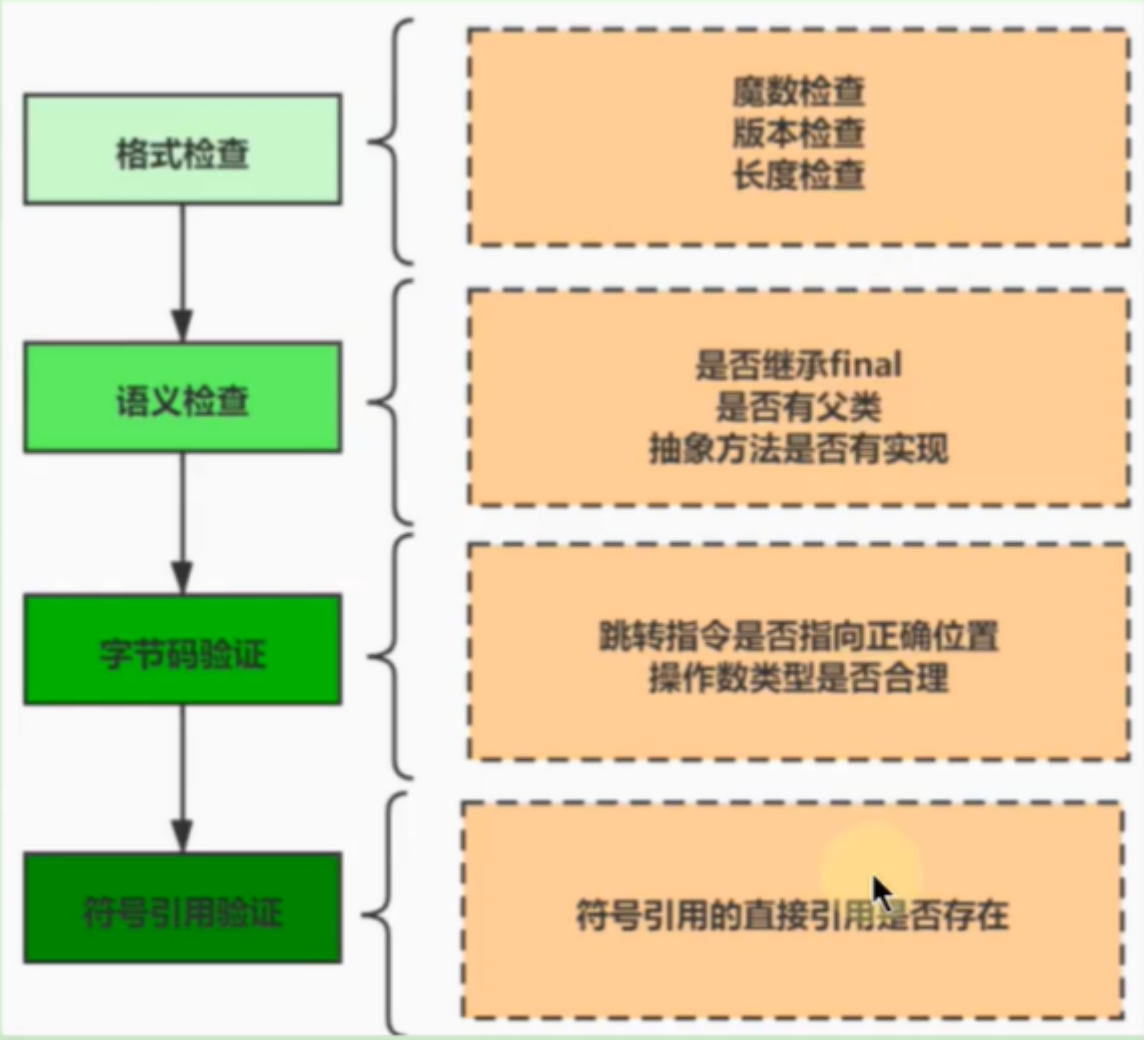
3. 过程二：Linking（链接）阶段

3.1. 环节 1：链接阶段之 Verification（验证）

当类加载到系统后，就开始链接操作，验证是链接操作的第一步。

$\color{red}{它的目的是保证加载的字节码是合法、合理并符合规范的。}$

验证的步骤比较复杂，实际要验证的项目也很繁多，大体上 Java 虚拟机需要做以下检查，如图所示。



整体说明：

验证的内容则涵盖了类数据信息的格式验证、语义检查、字节码验证，以及符号引用验证等。

- $\color{red}{其中格式验证会和加载阶段一起执行}$。验证通过之后，类加载器才会成功将类的二进制数据信息加载到方法区中。
- $\color{red}{格式验证之外的验证操作将会在方法区中进行}$。

链接阶段的验证虽然拖慢了加载速度，但是它避免了在字节码运行时还需要进行各种检查。（磨刀不误砍柴工）

具体说明：

1. 格式验证：是否以魔数 0XCAFEBABE 开头，主版本和副版本号是否在当前 Java 虚拟机的支持范围内，数据中每一个项是否都拥有正确的长度等。

2. 语义检查：Java 虚拟机会进行字节码的语义检查，但凡在语义上不符合规范的，虚拟机也不会给予验证通过。比如：

    - 是否所有的类都有父类的存在（在 Java 里，除了 object 外，其他类都应该有父类）
    - 是否一些被定义为 final 的方法或者类被重写或继承了
    - 非抽象类是否实现了所有抽象方法或者接口方法

3. 字节码验证：Java 虚拟机还会进行字节码验证，$\color{red}{字节码验证也是验证过程中最为复杂的一个过程}$。它试图通过对字节码流的分析，判断字节码是否可以被正确地执行。比如：

- 在字节码的执行过程中，是否会跳转到一条不存在的指令
- 函数的调用是否传递了正确类型的参数
- 变量的赋值是不是给了正确的数据类型等

栈映射帧（StackMapTable）就是在这个阶段，用于检测在特定的字节码处，其局部变量表和操作数栈是否有着正确的数据类型。但遗憾的是，100%准确地判断一段字节码是否可以被安全执行是无法实现的，因此，该过程只是尽可能地检查出可以预知的明显的问题。如果在这个阶段无法通过检查，虚拟机也不会正确装载这个类。但是，如果通过了这个阶段的检查，也不能说明这个类是完全没有问题的。

$\color{red}{在前面3次检查中，已经排除了文件格式错误、语义错误以及字节码的不正确性。但是依然不能确保类是没有问题的。}$

4. 符号引用的验证：校验器还将进符号引用的验证。Class 文件在其常量池会通过字符串记录自己将要使用的其他类或者方法。因此，在验证阶段，$\color{red}{虚拟机就会检查这些类或者方法确实是存在的}$，并且当前类有权限访问这些数据，如果一个需要使用类无法在系统中找到，则会抛出 NoClassDefFoundError，如果一个方法无法被找到，则会抛出 NoSuchMethodError。此阶段在解析环节才会执行。

3.2. 环节 2：链接阶段之 Preparation（准备）

$\color{red}{准备阶段（Preparation），简言之，为类的静态变分配内存，并将其初始化为默认值。}$

当一个类验证通过时，虚拟机就会进入准备阶段。在这个阶段，虚拟机就会为这个类分配相应的内存空间，并设置默认初始值。Java 虚拟机为各类型变量默认的初始值如表所示。

类型	默认初始值
byte	(byte)0
short	(short)0
int	0
long	0L
float	0.0f
double	0.0
char	\u0000
boolean	false
reference	null

Java 并不支持 boolean 类型，对于 boolean 类型，内部实现是 int，由于 int 的默认值是 0，故对应的，boolean 的默认值就是 false。

注意

- $\color{red}{这里不包含基本数据类型的字段用static final修饰的情况，因为final在编译的时候就会分配了，准备阶段会显式赋值。}$

```
// 一般情况：static final修饰的基本数据类型、字符串类型字面量会在准备阶段赋值
private static final String str = "Hello world";
// 特殊情况：static final修饰的引用类型不会在准备阶段赋值，而是在初始化阶段赋值
private static final String str = new String("Hello world");
```

- 注意这里不会为实例变量分配初始化，类变量会分配在方法区中，而实例变量是会随着对象一起分配到 Java 堆中。
- 在这个阶段并不会像初始化阶段中那样会有初始化或者代码被执行。

3.3. 环节 3：链接阶段之 Resolution（解析）

在准备阶段完成后，就进入了解析阶段。解析阶段（Resolution），简言之，将类、接口、字段和方法的符号引用转为直接引用。

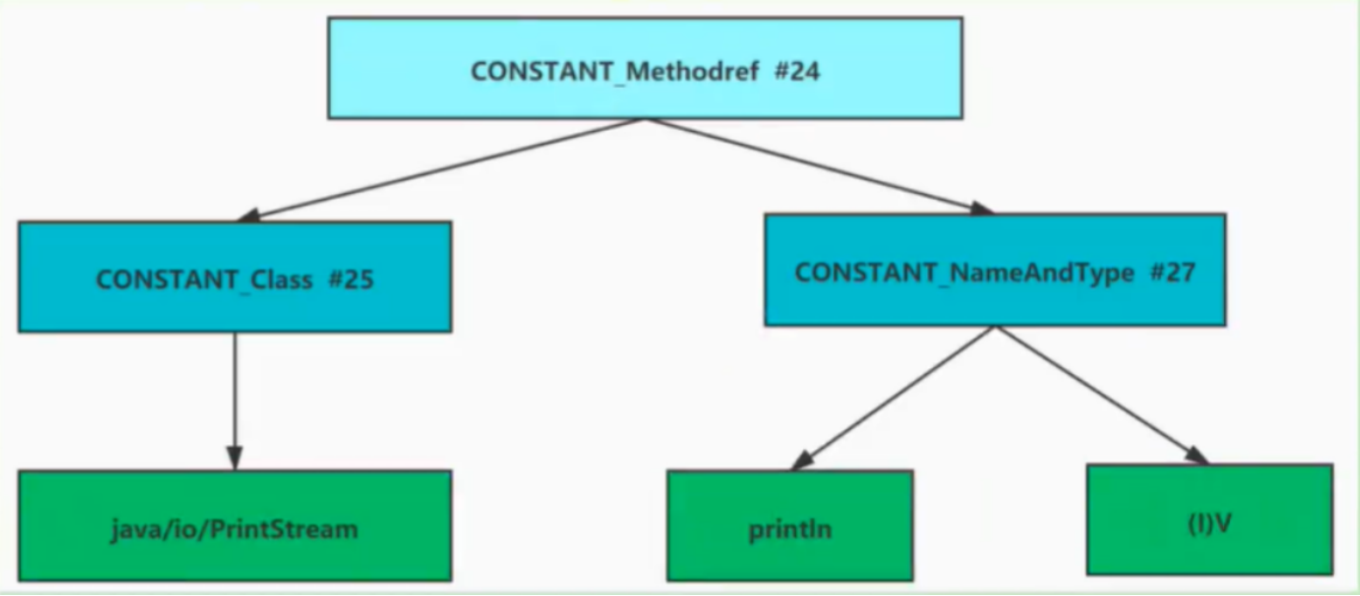
具体描述：

符号引用就是一些字面量的引用，和虚拟机的内部数据结构和和内存布局无关。比较容易理解的就是在 Class 类文件中，通过常量池进行了大量的符号引用。但是在程序实际运行时，只有符号引用是不够的，比如当如下 println()方法被调用时，系统需要明确知道该方法的位置。

举例：

输出操作 System.out.println()对应的字节码：

```
invokevirtual #24 <java/io/PrintStream.println>
```



以方法为例，Java 虚拟机为每个类都准备了一张方法表，将其所有的方法都列在表中，当需要调用一个类的方法的时候，只要知道这个方法在方法表中的偏移量就可以直接调用该方法。$\color{red}{通过解析操作，符号引用就可以转变为目标方法在类中方法表中的位置，从而使得方法被成功调用。}$

---

4. 过程三：Initialization（初始化）阶段

4.1. static 与 final 的搭配问题

说明：使用 static+ final 修饰的字段的显式赋值的操作，到底是在哪个阶段进行的赋值?

- 情况 1：在链接阶段的准备环节赋值
- 情况 2：在初始化阶段<clinit>()中赋值

结论： 在链接阶段的准备环节赋值的情况：

- 对于基本数据类型的字段来说，如果使用 static final 修饰，则显式赋值(直接赋值常量，而非调用方法通常是在链接阶段的准备环节进行
- 对于 String 来说，如果使用字面量的方式赋值，使用 static final 修饰的话，则显式赋值通常是在链接阶段的准备环节进行
- 在初始化阶段<clinit>()中赋值的情况： 排除上述的在准备环节赋值的情况之外的情况。

最终结论：使用 static+final 修饰，且显示赋值中不涉及到方法或构造器调用的基本数据类到或 String 类型的显式财值，是在链接阶段的准备环节进行。

```
public static final int INT_CONSTANT = 10;                                    // 在链
接阶段的准备环节赋值
public static final int NUM1 = new Random().nextInt(10);                       // 在初
始化阶段clinit>()中赋值
public static int a = 1;                                                       // 在初
始化阶段<clinit>()中赋值

public static final Integer INTEGER_CONSTANT1 = Integer.valueOf(100);          // 在初
始化阶段<clinit>()中赋值
public static Integer INTEGER_CONSTANT2 = Integer.valueOf(100);                // 在初
始化阶段<clinit>()中概值

public static final String s0 = "helloworld0";                                 // 在链
接阶段的准备环节赋值
public static final String s1 = new String("helloworld1");                     // 在初
始化阶段<clinit>()中赋值
public static String s2 = "hellowrold2";                                       // 在初
始化阶段<clinit>()中赋值
```

4.2. <clinit>()的线程安全性

对于<clinit>()方法的调用，也就是类的初始化，虚拟机会在内部确保其多线程环境中的安全性。

虚拟机会保证一个类的()方法在多线程环境中被正确地加锁、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。

正是因为$\color{red}{函数()带锁线程安全的}$，因此，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，引发死锁。并且这种死锁是很难发现的，因为看起来它们并没有可用的锁信息。

如果之前的线程成功加载了类，则等在队列中的线程就没有机会再执行<clinit>()方法了。那么，当需要使用这个类时，虚拟机会直接返回给它已经准备好的信息。

4.3. 类的初始化情况：主动使用 vs 被动使用

Java 程序对类的使用分为两种：主动使用和被动使用。

主动使用

Class 只有在必须要首次使用的时候才会被装载，Java 虚拟机不会无条件地装载 Class 类型。Java 虚拟机规定，一个类或接口在初次使用前，必须要进行初始化。这里指的“使用”，是指主动使用，主动使用只有下列几种情况：（即：如果出现如下的情况，则会对类进行初始化操作。而初始化操作之前的加载、验证、准备已经完成。

1. 实例化：当创建一个类的实例时，比如使用 new 关键字，或者通过反射、克隆、反序列化。

```
/**
 * 反序列化
 */
Class Order implements Serializable {
    static {
        System.out.println("Order类的初始化");
    }
}

public void test() {
    ObjectOutputStream oos = null;
    ObjectInputStream ois = null;
    try {
        // 序列化
        oos = new ObjectOutputStream(new FileOutputStream("order.dat"));
        oos.writeObject(new Order());
        // 反序列化
        ois = new ObjectInputStream(new FileOutputStream("order.dat"));
        Order order = ois.readObject();
    }
    catch (IOException e){
        e.printStackTrace();
    }
    catch (ClassNotFoundException e){
        e.printStackTrace();
    }
    finally {
        try {
            if (oos != null) {
                oos.close();
            }
            if (ois != null) {
                ois.close();
```

```
            }
        }
        catch (IOException e){
            e.printStackTrace();
        }
    }
}
```

2. 静态方法：当调用类的静态方法时，即当使用了字节码 invokestatic 指令。

3. 静态字段：当使用类、接口的静态字段时（final 修饰特殊考虑），比如，使用 getstatic 或者 putstatic 指令。（对应访问变量、赋值变量操作）

```
public class ActiveUse {
    @Test
    public void test() {
        System.out.println(User.num);
    }
}

class User {
    static {
        System.out.println("User类的初始化");
    }
    public static final int num = 1;
}
```

4. 反射：当使用 java.lang.reflect 包中的方法反射类的方法时。比如：Class.forName("com.atguigu.java.Test")

5. 继承：当初始化子类时，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 Java 虚拟机初始化一个类时，要求它的所有父类都已经被初始化，但是这条规则并不适用于接口。
>
> - 在初始化一个类时，并不会先初始化它所实现的接口
> - 在初始化一个接口时，并不会先初始化它的父接口
> - 因此，一个父接口并不会因为它的子接口或者实现类的初始化而初始化。只有当程序首次使用特定接口的静态字段时，才会导致该接口的初始化。

6. default 方法：如果一个接口定义了 default 方法，那么直接实现或者间接实现该接口的类的初始化，该接口要在其之前被初始化。

```
interface Compare {
    public static final Thread t = new Thread() {
        {
            System.out.println("Compare接口的初始化");
        }
```

```
    }
}
```

7. main 方法：当虚拟机启动时，用户需要指定一个要执行的主类（包含 main()方法的那个类），虚拟机会先初始化这个主类。

> VM 启动的时候通过引导类加载器加载一个初始类。这个类在调用 public static void main(String[])方法之前被链接和初始化。这个方法的执行将依次导致所需的类的加载，链接和初始化。

8. MethodHandle：当初次调用 MethodHandle 实例时，初始化该 MethodHandle 指向的方法所在的类。（涉及解析 REF getStatic、REF_putStatic、REF invokeStatic 方法句柄对应的类）

被动使用

除了以上的情况属于主动使用，其他的情况均属于被动使用。$\color{red}{被动使用不会引起类的初始化。}$

也就是说：$\color{red}{并不是在代码中出现的类，就一定会被加载或者初始化。}$如果不符合主动使用的条件，类就不会初始化。

1. 静态字段：当通过子类引用父类的静态变量，不会导致子类初始化，只有真正声明这个字段的类才会被初始化。

```
public class PassiveUse {
    @Test
    public void test() {
        System.out.println(Child.num);
    }
}

class Child extends Parent {
    static {
        System.out.println("Child类的初始化");
    }
}

class Parent {
    static {
        System.out.println("Parent类的初始化");
    }

    public static int num = 1;
}
```

2. 数组定义：通过数组定义类引用，不会触发此类的初始化

```
Parent[] parents= new Parent[10];
System.out.println(parents.getClass());
```

```
// new的话才会初始化
parents[0] = new Parent();
```

3. 引用常量：引用常量不会触发此类或接口的初始化。因为常量在链接阶段就已经被显式赋值了。

```
public class PassiveUse {
    public static void main(String[] args) {
        System.out.println(Serival.num);
        // 但引用其他类的话还是会初始化
        System.out.println(Serival.num2);
    }
}

interface Serival {
    public static final Thread t = new Thread() {
        {
            System.out.println("Serival初始化");
        }
    };

    public static int num = 10;
    public static final int num2 = new Random().nextInt(10);
}
```

4. loadClass 方法：调用 ClassLoader 类的 loadClass()方法加载一个类，并不是对类的主动使用，不会导致类的初始化。

```
Class clazz =
ClassLoader.getSystemClassLoader().loadClass("com.test.java.Person");
```

扩展

> -XX:+TraceClassLoading：追踪打印类的加载信息

5. 过程四：类的 Using（使用）

任何一个类型在使用之前都必须经历过完整的加载、链接和初始化 3 个类加载步骤。一旦一个类型成功经历过这 3 个步骤之后，便“万事俱备只欠东风”，就等着开发者使用了。

开发人员可以在程序中访问和调用它的静态类成员信息（比如：静态字段、静态方法），或者使用 new 关键字为其创建对象实例。

6. 过程五：类的 Unloading（卸载）

6.1. 类、类的加载器、类的实例之间的引用关系

在类加载器的内部实现中，用一个 Java 集合来存放所加载类的引用。另一方面，一个 Class 对象总是会引用它的类加载器，调用 Class 对象的 getClassLoader()方法，就能获得它的类加载器。由此可见，代表某个类的 Class 实例与其类的加载器之间为双向关联关系。

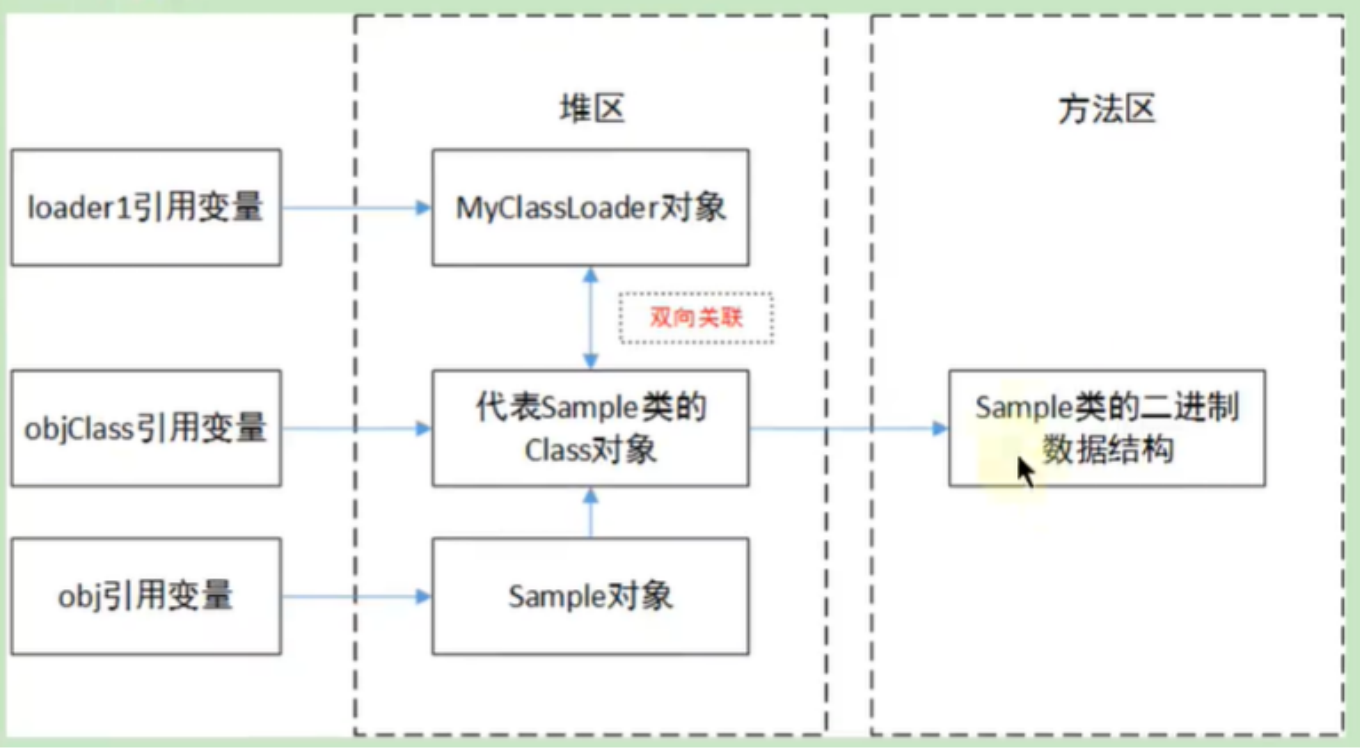
一个类的实例总是引用代表这个类的 Class 对象。在 Object 类中定义了 getClass()方法，这个方法返回代表对象所属类的 Class 对象的引用。此外，所有的 java 类都有一个静态属性 class，它引用代表这个类的 Class 对象。

6.2.类的生命周期

当 Sample 类被加载、链接和初始化后，它的生命周期就开始了。当代表 Sample 类的 Class 对象不再被引用，即不可触及时，Class 对象就会结束生命周期，Sample 类在方法区内的数据也会被卸载，从而结束 Sample 类的生命周期。

$\color{red}{一个类何时结束生命周期，取决于代表它的Class对象何时结束生命周期。}$

6.3. 具体例子



loader1 变量和 obj 变量间接应用代表 Sample 类的 Class 对象，而 objClass 变量则直接引用它。

如果程序运行过程中，将上图左侧三个引用变量都置为 null，此时 Sample 对象结束生命周期，MyClassLoader 对象结束生命周期，代表 Sample 类的 Class 对象也结束生命周期，Sample 类在方法区内的二进制数据被卸载。

当再次有需要时，会检查 Sample 类的 Class 对象是否存在，如果存在会直接使用，不再重新加载；如果不存在 Sample 类会被重新加载，在 Java 虚拟机的堆区会生成一个新的代表 Sample 类的 Class 实例（可以通过哈希码查看是否是同一个实例）

6.4. 类的卸载

（1）启动类加载器加载的类型在整个运行期间是不可能被卸载的（jvm 和 jls 规范）

（2）被系统类加载器和扩展类加载器加载的类型在运行期间不太可能被卸载，因为系统类加载器实例或者扩展类的实例基本上在整个运行期间总能直接或者间接的访问的到，其达到 unreachable 的可能性极小。

（3）被开发者自定义的类加载器实例加载的类型只有在很简单的上下文环境中才能被卸载，而且一般还要借助于强制调用虚拟机的垃圾收集功能才可以做到。可以预想，稍微复杂点的应用场景中（比如：很多时候用户在开发自定义类加载器实例的时候采用缓存的策略以提高系统性能），被加载的类型在运行期间也是几乎不太可能被卸载的（至少卸载的时间是不确定的）。

综合以上三点，一个已经加载的类型被卸载的几率很小至少被卸载的时间是不确定的。同时我们可以看的出来，开发者在开发代码时候，不应该对虚拟机的类型卸载做任何假设的前提下，来实现系统中的特定功能。

回顾：方法区的垃圾回收

方法区的垃圾收集主要回收两部分内容：常量池中废弃的常量和不再使用的类型。

HotSpot 虚拟机对常量池的回收策略是很明确的，只要常量池中的常量没有被任何地方引用，就可以被回收。

判定一个常量是否“废弃”还是相对简单，而要判定一个类型是否属于“不再使用的类”的条件就比较苛刻了。需要同时满足下面三个条件：

- $\color{blue}{该类所有的实例都已经被回收。也就是Java堆中不存在该类及其任何派生子类的实例。}$
- $\color{blue}{加载该类的类加载器已经被回收。这个条件除非是经过精心设计的可替换类加载器的场景，如OSGi、JSP的重加载等，否则通常是很难达成的。}$
- $\color{blue}{该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。}$

Java 虚拟机被允许对满足上述三个条件的无用类进行回收，这里说的仅仅是“被允许”，而并不是和对象一样，没有引用了就必然会回收。
