JVM-类文件结构以及类加载机制

一，Class类文件的结构

Class文件是一组以8位字节为基础单位的二进制流。

- 每个Class文件的头4个字节成为魔数，它的唯一作用就是确定这个文件是否为一个能被虚拟机接受的Class文件。为啥不用文件的扩展名来标识，因为文件的扩展名是可以被改变的。
- 紧接着魔数的4个字节是Class文件的版本号：第5个字节和第6个字节是次版本号，第7，8个字节是主版本号，用来标记Class文件能否被当前JDK所能运行。高版本的JDK可以兼容以前版本的Clss文件，但是不能运行以后版本的Class文件。
- 紧接着就是常量池的入口，它可以理解为Class文件的资源仓库。只要存储的是字面量和符号引用。字面量比较接近于Java语言层面的常量概念，如文本字符串，声明为final的常量值等。符号引用就是包含了三点，类和接口的全限定名，字段的名称和描述符，方法的名称和描述符。在Class文件中不会保存各个方法，字段的最终的内存信息，它会在类加载的时候对符号引用进行翻译，获取到最终的内存地址。
- 紧接着就是访问标志，占用了两个字节，这个标志用于识别一些类或者接口层次的访问信息，包括这个Class是类还是接口，是否定义为public，是否定义为abstract类型，如果是类的话，是否被声明为final。
- 接下来是类索引，父类索引与接口索引集合，由这个区域的内容来确定该类的继承关系。
- 字段表集合，用于描述接口或类中声明的变量。字段的作用域，是否被static所修饰等等。
- 方法表集合，与字段表类似。

二，Class加载的大致过程

类的生命周期基本上包括加载，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，解析部分称为连接。解析阶段有可能在初始化之后才开始这是为了支持Java语言的运行时绑定。

- 加载

  在加载阶段需要完成三件事情

  - 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去的运行时的数据结构。
  - 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的访问接口。

  在加载过程完成之后，虚拟机外部的二进制字节流就按照虚拟机所需要的格式存储在方法区之中，然后在内存中实例化了一个java.lang.Class对象(并没有明确在Java堆中，对于HotSpot虚拟机而言，Class对象比较特殊，他虽然是对象，但是存放在方法区中)，将这个对象作为程序访问方法区中这类型数据的外部接口。

- 验证

  这一阶段的目的是确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并不会危害虚拟机自身的安全。避免恶意代码的攻击

  - 文件格式的验证

    魔数，当前JDK版本是否支持主版本号等

  - 元数据的验证

    进行语义分析，这个类是否继承了不允许继承的类，是或否实现了继承接口中的所有方法等。摆正不存在不符合java的元数据信息

  - 字节码验证

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。主要是对类方法进行校验分析。

- 准备

  该阶段是正式为类变量分配内存，并设置类变量(被static修饰的变量)被初始值(零值)阶段，但是如果是被final修饰的话，在准备阶段就会被赋值成等号右边的值。

- 解析

  该阶段是将常量池中的符号引用替换成直接引用的过程。主要是对类或接口，字段，类方法等进行解析

  - 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，只要使用的时能无歧义地确定目标即可。符号引用与虚拟机实现地内存布局无关，引用的目标不一定已加载到内存当中。
  - 直接引用：可以是直接指向目标的指针，相对偏移量或者一个可以能间接定位到目标地句柄，直接引用时和虚拟机实现的内存布局是相关的，如果有了直接引用，那么引用的目标必然已经在内存中出现，

- 初始化

  对于初始化阶段，虚拟机规范严格规定了有且仅有5种情况必须对类进行初始化操作。

  1，遇到new,getstatic,putstatic,invokestatic字节码指令时。

  2，使用java.lang.reflect包的方法对类进行反射调用的时候。

  3，初始化一个类的时候，发现其父类还未初始化，则先触发其父类的初始化。

  4，当虚拟机启动时，用户需要执行的主类，即包含Main方法的那个类。

  5，略

  被动引用的情况

  1，当子类仅仅调用父类的静态字段的时候，只会触发父类的初始化，而不会触发子类的初始化。

```
class A{
    static {
        System.out.println("A init()");
    }
    public static int value=1;
}
class B extends A{
    static {
        System.out.println("B init()");
    }
}
public class Test{
    public static void main(String[] args){
        System.out.println(B.value);
    }
}
/*
A init()
1
*/
```

  2,通过数组来引用类，不会触发类的初始化

```
public class A{
    static {
        System.out.println("A init()");
    }
}
public class Test{
    public static void main(String[] args){
        A[] x = new A[10];
    }
}
/*
结果中并不会出现A init()
*/
```

3，通过引用类的final常量，不会引用类的初始化

```
public class A{
    static {
        System.out.println("A init()");
    }
    public static final int value=1;
}
public class Test{
    public static void main(String[] args){
        System.out.println(A.value);
    }
}
/*
1
并不会出现A init()
*/
```

总结：

1，类继承接口得时候，必须初始化父接口。

2，接口初始化时，对于该接口得父接口并不要求初始化。

类的初始化是类加载过程的最后一步，在准备阶段，变量已经赋值过一次(零值),而在初始化阶段，则根据程序员通过程序制定的主主观计划去初始化变量的值，该过程时执行类构造器()方法的过程。是由编译器主动收集类中得所有类变量得赋值动作和静态语句块中得语句合并产生的。

clinit和init的区别

init和clinit方法执行时机不同

init是对象构造器方法，也就是说在程序执行 new 一个对象调用该对象类的 constructor 方法时才会执行init方法，而clinit是类构造器方法，也就是在jvm进行类加载—-验证—-解析—-初始化，中的初始化阶段jvm会调用clinit方法。

- 类加载器

- 类加载器的类别
  - 启动类加载器：加载<JAVA_HOME>\lib目录中的
  - 扩展类加载器：加载<JAVA_HOME>\lib\ext目录中的
  - 应用程序类加载器：负责加载用户类路径上所指定的类库
- 双亲委派模型
  - 工作过程：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每个层次的类加载器都是如此，因此所有的的加载请求最终应该传到顶层的启动类加载器中，只有当父加载器无法加载该类的时候，子加载器才会尝试自己去加载。
  - 意义

    可以保证我们的类有一个合适的优先级，例如Object类，它是我们系统中所有类的根类，采用双亲委派模型以后，不管是哪个类加载器来加载Object类，哪怕这个加载器是自定义类加载器，通过双亲委派模型，最终都是由启动类加载器去加载的，这样就可以保证Object这个类在程序的各个类加载器环境中都是同一个类。在虚拟机里觉得一个类是不是唯一有两个因素，第一个就是这个类本身，第二个就是加载这个类的类加载器，如果同一个类由不同的类加载器去加载，在虚拟机看来，这两个类是不同的类。
  - 如何破坏双亲委派模型

    重写loadClass()方法。