1. 虚拟机类加载
2. 概述

在Class文件中描述的各类信息，最终都需要加载到虚拟机中才能够被运行和使用。虚拟机的类加载机制就是虚拟机将描述类的数据从Class文件加载到内存，并对数据进行校验，转换解析和初始化，最终形成可以被虚拟机直接使用的Java类型的过程。

1. Java类加载的优缺点

Java类型的加载，连接和初始化都是在程序运行期间完成的，这样Java语言在提前编译时会面临额外的困难，也会在类加载时增加一些开销。但是却为Java应用带来了极高的可扩展性和灵活性，Java的动态扩展就是依赖运行期动态加载和动态连接的特点实现的。

1. 类型的生命周期

一个类型从被加载到虚拟机内存开始，到卸载存储内存为止，它的整个生命周期将会经历加载，验证，准备，解析，初始化，使用，卸载七个将诶段，其中验证，准备，解析三个部分统称为连接。加载，验证，准备，初始化和卸载这5个阶段的顺序是确定的，类型的加载过程必须按照这种顺序开始，而解析阶段则有可能在初始化阶段之后再开始，为了支持Java的运行时绑定特性。

1. 初始化阶段的时机

《Java虚拟机规范》中没有强制规定加载阶段何时开始，而是严格规定了初始化阶段的开始情况：

1. 使用new关键字实例化对象时，读取或设置一个类型的静态字段时，调用一个类型的静态方法时，如果类型没有进行过初始化，则需要先触发其初始化阶段。
2. 使用java.lang.reflect包的方法对类型进行反射调用时，如果类型没有进行过初始化，则先触发其初始化。
3. 初始化类时，如果其父类没有初始化，则需要先触发其父类的初始化。
4. 当虚拟机启动时，指定主类先初始化。
5. 动态语言解析生成的方法句柄，如果该句柄所对应的类没有进行过初始化，则先触发初始化。
6. 接口中的默认方法的实现类发生了初始化，则该接口要在这之前初始化。

这六种情况是对一个类型的主动初始化，称为主动引用，其他的引用类型的方式都不会触发初始化，称为被动引用。

对于静态字段，只有直接定义这个字段的类才会被初始化，如果是通过该类的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。

通过数组定义来引用类A，不会触发对此类的初始化。而是让虚拟机自动生成了一个直接继承于java.lang.Object的子类B类，创建的动作由字节码指令newarray触发。这个类B代表了一个元素类型为类A的数组，数组中应有的属性和方法都实现在这个类中，之所以Java对数组的访问比C++要安全，就是因为这个类中包装了数组元素的访问，而C++中则是直接翻译为了对数组指针的移动。Java检查到数组越界时抛出ArrayIndexOutOfBoundsException异常，避免了非法内存访问。

对于调用其他类的常量，该常量在编译阶段会存入调用类的常量池中，本质上是没有直接引用到直接定义常量的类中，因此不会触发定义常量的类的初始化。

1. 接口的加载

接口也有初始化过程，但是接口不能使用static{}代码块，但是可以对成员变量赋值，因此编译器会为接口生成<clinit>()类构造器，用于初始化接口所定义的成员变量。

同时，接口在初始化时并不要求其父接口全部完成初始化，只有在真正使用到父接口时才会初始化。

1. 类加载的过程

Java虚拟机中类加载包括五个阶段，加载，验证，准备，解析和初始化。

1. 加载

在加载阶段，Java虚拟机需要完成三件事情：

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的访问入口。

1. 获取二进制流

规范中并没有规定该类的二进制字节流要从哪里获取，因此Java虚拟机的使用者们由此开发出了很多技术。如

从ZIP压缩包中获取，称为JAR，WAR包的基础；

运行时计算生成，用于动态代理技术，在reflect.Proxy中，为特定接口生成形式为“\*$Proxy”的代理类的二进制字节流；

等等

加载阶段是开发人员可控性最强的阶段，可以使用Java虚拟机内置的引导类加载器完成，也可以由用户自定义的类加载器完成，开发人员通过定义自己的类加载器控制字节流的获取方式（重写一个类加载器的findClass()或loadClass()方法），实现动态性。

对于数组类，其本身不通过类加载器创建，而是由Java虚拟机直接在内存中动态构造出来的，但是数组类中的元素类型依然要靠类加载器类完成加载。

1. 方法区存储

二进制字节流按照虚拟机所设定的格式存储在方法区中。

1. 实例化Class对象

类型数据存储在方法区中后，会在Java堆内存中实例化一个java.lang.Class类的对象，这个对象将作为程序访问方法区中类型数据的外部接口。

1. 验证
2. 概述

验证是连接阶段的第一步，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约束要求，保证这些信息别当做代码运行后不会危害虚拟机自身的安全。由于Class文件可以由任何途径产生，因此Java虚拟机需要检验二进制字节流，防止有恶意的字节码流攻击系统。验证阶段大致包含了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。

2. 文件格式验证

该阶段验证字节流是否符合Class文件格式的规范，并且能够被当前版本的虚拟机处理，如是否以魔数0xCAFEBABE开头等等。这个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，字节流才被允许进入Java虚拟机内存的方法区中存储。之后的三个验证阶段都是基于方法区的存储结构进行的，不会再直接读取，操作字节流。

1. 元数据验证

对字节码描述的信息进行语义分析，保证符合Java语言规范的要求，如这个类是否有父类，父类是否继承了不允许被继承的类等等。

1. 字节码验证

验证过程中最复杂的阶段，通过数据流分析和控制流分析，确定程序语义是合法的，符合逻辑的。上一阶段的元数据验证是对数据类型进行校验，这一阶段就是对类的方法体（Class文件中的Code属性）进行校验分析，保证被校验的方法在运行时不会做出危害虚拟机的行为。如保证任何跳转指令都不会跳转到方法体以外的字节码指令上等等。

但是，即使方法体通过了字节码验证，也不能保证它就是安全的，因为无法通过程序来准确判定另一段程序是否存在BUG。

由于数据流分析和控制流分析的复杂性，为了避免过多的执行时间消耗在字节码验证上，JDK6之后，Javac编译器进行了优化，将尽可能多的检验辅助措施移到Javac编译器汇总进行，即给方法体Code属性的属性表中添加了一项名为“StackMapTable”的属性，这个属性描述了方法体中所有的基本块（按照控制流拆分的代码块）开始时本地变量表和操作栈应有的状态，在字节码验证阶段，Java虚拟机就不需要根据程序推导这些状态的合法性，直接检查StackMapTable属性中的记录是否合法即可。这样，字节码验证的类型推导转变为了类型检查，节省了检验时间。

1. 符号引用验证

这个验证发生在连接的第三阶段（解析阶段），虚拟机将符号引用转化为直接引用时，符号引用验证是对类自身以外的各类信息进行匹配校验，检查该类是否缺少或者被禁止访问它所依赖的外部类，方法，字段等等。该验证主要是为了确保解析行为能够正常执行，如果无法通过符号引用验证，将会抛出IncompatibleClassChangeError的子类异常，如IllegalAccessError, NoSuchFieldError, NoSuchMethodError等。

对于验证阶段，如果程序运行的所有代码已经被反复使用过了，那么在生产环境的实施阶段就可以考虑关闭大部分的类验证措施，以缩短类加载的时间。

1. 准备

准备阶段是正式为类中定义的静态变量分配内存并且设置类变量的初始值的阶段，这些变量所使用的内存都应该在方法区中分配，注意方法区是一个逻辑上的区域，JDK7以前使用永久代实现方法区，JDK8以后，类变量存在于Java堆中。

注意，准备阶段只是初始化静态变量，而不包括实例变量，实例变量在对象实例化的时候随着对象一起分配在Java堆中。还有这里初始化的初始值只是数据类型的零值，而不是代码中自己赋予的值，因为这时并没有开始执行任何Java方法，代码中的赋值操作所对应的指令是在程序编译以后，存放与类构造器<clinit>()方法之中，因此代码中的赋值操作要在类的初始化阶段才会执行。

但是，对于某些特殊情况，如果类字段的字段属性表中存在ConstantValue属性，那么在准备阶段就会初始化为ConstantValue属性指定的初始值。当用final修饰字段时，编译时Javac会为该字段生成ConstantValue属性。

1. 解析
2. 概述

解析阶段是Java虚拟机将常量池内的符号引用替换为直接引用的过程。

1. 符号引用

符号引用以一组符号来描述所引用点的目标，符号可以是任何形式的字面量，只要使用时能够定位到目标即可，它与虚拟机实现的内存布局无关，符号引用的字面量形式定义在Class文件格式当中。

1. 直接引用

直接引用是可以直接指向目标的指针，相对偏移量或者能间接定位到目标的句柄。直接引用与虚拟机的内存布局直接相关，同一个符号引用在不同的虚拟机上翻译出来的直接引用都不会相同。如果有了直接引用，那么引用的目标在虚拟机的内存上必定存在。

1. 解析时间

解析阶段发生的时间是不确定的，因为解析用于转换符号引用，在执行用于操作符号引用的字节码指令之前解析符号引用即可，而这些指令的执行时间也是不确定的。

1. 多次解析请求

对于同一个符号引用进行多次解析请求，虚拟机可以对第一次解析的结果进行缓存，避免解析动作重复进行。但是对于invokedynamic指令，它对应的引用称为“动态调用点限定符”，动态的含义是只有在程序实际运行到这条指令时，解析动作才能够运行，因此每次解析都可能不一样。

1. 解析过程

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这7类符号引用进行。

1. 类或接口的解析

首先将符号引用的全限定名传递给类的加载器来加载这个类，加载过程中，有元数据校验，字节码校验的需要，有可能触发其他相关类的加载动作。如果加载过程没有异常，就会生成一个有效的类或接口，但是在解析完成之前，还要进行符号引用验证，验证当前的类是否对解析生成的类或接口有访问权限。

1. 字段解析

首先对字段A所属的类或接口B的符号引用进行解析，如果解析完成无异常，对B进行搜索，如果B本身就包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用；否则，按照继承关系从下往上递归搜索各个接口和它的父接口，或者父类。

若最终查找失败，则抛出NoSuchFieldError异常，如果找到了并返回了引用，则对这个字段进行权限验证，如果无权限，则抛出IllegalAccessError异常。这样，Java虚拟机能够获得字段唯一的解析结果，但是真实的Javac编译器会更加严格，如果多个同名字段，则会拒绝编译为Class文件，

1. 方法解析

首先解析出方法A所属的类或接口B的符号引用，成功后，在B中搜索，由于Class文件格式中类的方法和接口的方法符号引用的常量类型定义是分开的，因此如果在类的方法表中发现是个接口的话，直接抛出IncompatibleClassChangeError异常。否则，在B，B的父类，B实现的接口列表以及他们的父接口之中递归查找是否有简单名称和描述符都与目标相匹配的方法，同样要进行权限验证。

1. 接口方法解析

与方法解析类似，但是接口是允许多重继承的，如果不同的父接口中存有多个简单名称和描述符都与目标相匹配的方法，就会从这些方法中返回其中一个并结束查找。严格的Javac编译器同样可能会拒绝这种不确定的做法。

1. 初始化

类的初始化阶段是类加载的最后一个阶段，在该阶段，Java虚拟机才真正开始执行类中编写的Java程序代码，将主导权移交给程序。在该阶段，会根据代码中的赋值初始化静态变量和其他资源，即执行类构造器<clinit>()方法的过程，该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中的语句合并生成的。该方法不需要显示地调用父类构造器，Java虚拟机保证在子类的<clinit>()方法执行前，父类的<clinit>()方法已经执行完毕了，因此Java虚拟机中第一个被执行的<clinit>()方法的类型肯定是java.lang.Object。而且该方法也非必须的，只要类中赋值和静态代码块就不会生成该方法。同时，一个类的<clinit>()方法一定是线程安全的，类的初始化只会进行一次。

接口中只要有变量赋值操作，就能够生成<clinit>()方法，执行接口的<clinit>()方法不需要先执行父接口的<clinit>()方法，因为只有当父接口中定义的变量被使用时，父接口才会被初始化。

静态语句块只能访问到定义在它之前的变量，对于在它之后的变量，静态语句块中只能赋值不能访问。同时父类中的静态语句块一定是先于子类的变量赋值操作的