**第7章 虚拟机类加载机制**

虚拟机的类加载机制：

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，这就是虚拟机的类加载机制。在Java语言里，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。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。

eg：

（1）编写一个面向接口的应用程序，可以等到运行时再指定其实际的实现类；

（2）用户可以通过Java预定义的和自定义类加载器，让一个本地的应用程序可以在运行时从网络或其他地方加载一个二进制流作为程序代码的一部分，这种组装应用程序的方式目前已广泛应用于Java程序之中，如JSP、OSGi技术等，都使用了Java语言运行期类加载的特性。

类加载的时机

类字节码从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载（Loading）、验证（Verification）、准备（Preparation）、解析（Resolution）、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）。

类生命周期图

连接（Linking）

卸载

Unloading

初始化

Initialization

准备

Preparation

加载

Loading

验证

Verification

使用

Using

解析

Resolution

注：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”——强调这点是因为这些阶段通常都是**互相交叉地混合式**进行的，通常会在一个阶段执行的过程中调用、激活另外一个阶段。

而解析阶段则不一定：它在某些情况下可以在初始化阶段之后再开始，这是为了支持Java语言的运行时绑定（称为动态绑定/晚期绑定）。

JVM没有说明什么时候可以开始类加载的第一个阶段——加载，因为Java虚拟机规范中将这点交给虚拟机的具体实现来自由把握。

但是虚拟机规范严格规定了**有且只有5种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）：

（1）遇到new、getstatic、putstatic、invokestatic 这4条字节码指令时，如果类没有进行过初始化，则需要先触发初始化。

new指令对应的Java代码场景：**new关键字实例化对象**

getstatic指令对应的Java代码场景：**读取一个类的静态字段**（被final修饰、已在编译器把结果放入常量池的静态字段除外）

putstatic指令对应的Java代码场景：**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）

invokestatic指令对应的Java代码场景：**调用一个类的静态方法**时

（2）使用java.lang.reflect包的方法对类进行反射调用，如果类没有进行过初始化，则需要触发初始化。

（3）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。

（4）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），虚拟机会初始化这个主类。

（5）当使用JDK1.7的动态语言支持时，如果一个java.lang.invoke.Method

Handle实例最后的解析结果REF\_getStatic、REF\_putStatic、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。

这五种场景的行为称为对一个类进行主动引用，除此之外，其它引用类的方式都不会触发初始化，称为被动引用。

接口与类初始化过程稍有不同，主要体现在第（3）种行为：即一个接口在初始化时，不要求其父接口全部都完成了初始化，只有在该接口使用到父接口的时候（如引用接口中定义的常量）才会初始化父接口。

加载

加载阶段JVM需要完成以下3件事情：

（1）通过一个类的全限定名来获取此类的二进制字节流

（2）将这个字节流所代表的**静态**存储结构转化为内存方法区的运行时数据结构

（3）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。

非数组类加载阶段：

开发人员可控性最强，因为加载阶段可以使用系统提供的引导类加载器来完成，也可以由用户自定义的类加载器去完成，开发人员可以通过定义自己的类加载器去控制字节流的获取方式（即**重写一个类加载器的loadClass()方法**）

数组类加载阶段：

数组类本身不通过类加载器创建，它是由Java虚拟机直接创建的。但数组类与类加载器仍然有很密切的关系，因为数组类的元素类型（Element Type，指数组去掉所有维度的类型）最终是要靠类加载器去创建，一个数组类C创建过程需要遵循以下规则：

（1）如果数组的组件类型（Component Type，指数组去掉一个维度的类型）是引用类型，则递归采用JVM的加载过程去加载这个组件类型，数组C将在加载该组件类型的类加载器的类名称空间上被标识（一个类必须与类加载器一起确定唯一性）。

（2）如果数组的组件类型不是引用类型（如int[]数组），Java虚拟机将会把数组C标记为与引导类加载器关联。

（3）数组类的可见性与它的组件类型的可见性一致，如果组件类型不是引用类型，则数组类的可见性将默认为public。

加载阶段完成后，虚拟机外部的二进制字节流就按照虚拟机所需格式存储在方法区之中。然后在内存中实例化一个java.lang.Class类的对象（没有明确必须是在Java堆，因为对于HotSpot虚拟机而言，Class对象比较特殊，它虽然是对象，但是存放在方法区里），这个对象将作为程序访问方法区中的这些类型数据的外部接口。加载阶段与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。

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。

验证阶段大致上会完成下面4个阶段的检验动作：

（1）文件格式验证

检验字节流是否符合Class文件格式的规范，并且能被当前版本的虚拟机处理。这阶段的验证是基于二进制字节流进行的，只有通过这个阶段的验证后，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。

（2）元数据验证

对字节码描述的信息（类的元数据信息）进行语义分析，以保证其描述的信息符合Java语言规范要求。

（3）字节码验证

最复杂的阶段。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。

（4）符号引用验证

发生在虚拟机将符号引用转化为直接引用的时候，这个转化动作将在连接阶段的第三阶段——解析阶段中发生。符号引用验证可以认为是对类自身以外（常量池中的各种符号引用）的信息进行匹配性校验。

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中进行分配。

应该注意的是，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Java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;

则变量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，所以把value赋值为123的动作将在初始化阶段后才会执行。

|  |  |
| --- | --- |
| 基本数据类型的零值 | |
| 数据类型 | 零值 |
| int | 0 |
| long | 0L |
| short | (short)0 |
| char | ‘\u0000’ |
| byte | (byte)0 |
| boolean | false |
| float | 0.0f |
| double | 0.0d |
| reference | null |

如果类字段的字段属性表存在ConstantValue属性，则在准备阶段变量value就会被初始化为ConstantValue属性所指定的值，如：

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。

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**，符号引用在Class文件中以CONSTANT\_Class\_info、CONSTANT\_Fieldref\_info、CONSTANT\_Methodref\_info等类型的常量出现。

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地定位到目标即可。符号引用与虚拟机实现的内存布局无关，引用的目标并不一定已经加载到内存中。各种虚拟机实现的内存布局可以各不相同，但是它们能接受的符号引用必须都是一致的。

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。直接引用是和虚拟机实现的内存布局相关的，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，那引用的目标必定已经在内存中存在。

用于操作符号引用的字节码指令

|  |  |  |  |
| --- | --- | --- | --- |
| anewarray | checkcast | getfield | getstatic |
| instanceof | invokedynamic | invokeinterface | invokespecial |
| invokestatic | invokevirtual | ldc | ldc\_w |
| multianewarray | new | putfield | putstatic |

除invokedynamic指令外，虚拟机实现可以对第一次解析的结果进行缓存（在运行时常量池中记录直接引用，并把常量标识为已解析状态）从而避免解析动作重复进行。无论是否真正执行了多次解析动作，虚拟机需要保证的是在同一个实体中，如果一个符号引用之前已经被成功解析过，那么后续的引用解析请求就应当一致成功；同样如果第一次解析失败，那么其他指令对这个符号的解请求也应当收到相同的异常。

对于invokedynamic指令，上述规则不成立。当碰到某个前面已经由invo

kedynamic指令触发过解析的符号引用时，并不意味着这个解析结果对于其他invokedynamic指令也同样生效。因为invokedynamic指令的目的本来就是用于动态语言支持，它所对应的引用称为“动态调用点限定符”（Dynamic Call Site Specifier），这里“动态”的含义是必须等到程序实际运行到这条指令时，解析动作才能进行。相对的，其余可触发解析的指令都是“静态”的，可以在刚刚完成加载阶段，还没有开始执行代码时就进行解析。

解析动作主要针对以下7类符号引用进行：

① 类或接口（CONSTANT\_Class\_info）

② 字段（CONSTANT\_Fieldref\_info）

③ 类方法（CONSTANT\_Methodref\_info）

④ 接口方法（CONSTANT\_InterfaceMethodref\_info）

⑤ 方法类型（CONSTANT\_MethodType\_info）

⑥ 方法句柄（CONSTANT\_MethodHandle\_info）

⑦ 调用点限定符（InvokeDynamic\_info）

类或接口解析

假设当前代码所处的类为D，要把一个从未解析过的符号引用N（CONSTANT

\_Class\_info）解析为一个类或接口C的直接引用，则虚拟机完成整个解析过程需要以下3个步骤：

1）如果C不是数组类型，则JVM将会把N的全限定名传递给D的类加载器去加载类C。在加载过程中，由于元数据验证、字节码验证的需要，又可能触发其他相关类的加载动作。

2）如果C是数组类型，并且数组的元素类型为对象，即N的描述符会是类似于“[Ljava/lang/Integer”的形式，那将会按照第1点的规则加载数组元素类型。如果N的描述符如前面所假设的形式，需要加载的元素类型就是“java.lang.Integer”，接着由虚拟机生成一个代表此数组维度和元素的数组对象。

3）如果1）、2）没有出现异常，则C在虚拟机中实际上已经成为了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，确认D是否具备对C的访问权限，如果发现不具备访问权限，将抛出java.lang.IllegalAccessEr

ror异常。

？？？

字段解析

解析一个从未被解析过的字段符号引用，首先对字段表内class\_index项中索引的CONSTANT\_Class\_info符号引用进行解析，也就是字段所属的类或接口的符号引用。如果解析成功，将该字段所属的类或接口用C表示，虚拟机规范要求按照如下步骤对C进行后续字段的搜索。

1）如果**C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，查找结束

2）否则，**如果在C中实现了接口，将会按照继承关系从下往上递归搜索各个接口和它的父接口**，如果接口中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，查找结束。

3）否则，**如果C不是java.lang.Object的话，将会按照继承关系从下往上递归搜索其父类**，如果在父类中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，查找结束。

4）否则，查找失败，抛出java.lang.NoSuchFieldError异常

如果查找过程成功返回了直接引用，将会对这个字段进行权限验证，如果发现不具备对字段的访问权限，将抛出java.lang.IllegalAccessError异常。

类方法解析

解析一个从未被解析过的类方法符号引用，首先对方法表内class\_index项中索引的CONSTANT\_Class\_info符号引用进行解析，也就是字段所属的类或接口的符号引用。如果解析成功，将该字段所属的类用C表示，虚拟机规范要求按照如下步骤对C进行后续字段的搜索。

1）类方法和接口方法符号引用的常量类型定义是分开的。如果在类方法表中发现class\_index中索引的C是个接口，就直接抛出java.lang.Incompatible

ClassChangeError

2）否则，在**类C中查找是否有简单名称和描述符都与目标相匹配的方法**，如果有则返回这个方法的直接引用，查找结束。

3）否则，在**类C的父类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**，如果有则返回这个方法的直接引用，查找结束。

4）否则，**在类C实现的接口列表及它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，如果存在匹配的方法，说明类C是一个抽象类**（抽象类可以实现接口，但接口中方法可以不用实现，或者可以只部分实现），这时查找结束，抛出java.lang.AbstractMethodError

5）否则，宣告方法查找失败，抛出java.lang.NoSuchMethodError

最后，如果查找过程成功返回了直接引用，将会对这个方法进行权限验证，如果不具备对此方法的访问权限，将抛出java.lang.IllegalAccessError

接口方法解析

解析一个从未被解析过的接口方法符号引用，首先对接口方法表内class\_

index项中索引的CONSTANT\_Class\_info符号引用进行解析，也就是字段所属的类或接口的符号引用。如果解析成功，将该字段所属接口用C表示，虚拟机规范要求按照如下步骤对C进行后续字段的搜索。

1）如果在接口方法表中发现class\_index中索引的C是个类，就直接抛出java.lang.IncompatibleClassChangeError

2）否则，在**接口C中查找是否有简单名称和描述符都与目标相匹配的方法**，如果有则返回这个方法的直接引用，查找结束。

3）否则，在**接口C的父接口中递归查找，直到java.lang.Object类（查找范围会包括Object类）为止**，如果有则返回这个方法的直接引用，查找结束。

4）否则，宣告方法查找失败，抛出java.lang.NoSuchMethodError异常。

由于接口中的所有方法默认都是public的，所以不存在访问权限的问题。

初始化

初始化是类加载过程的最后一步。初始化阶段，才真正开始执行类中定义的Java程序代码（字节码）。

在准备阶段，变量已经完成了系统的初始值赋值。而在初始化阶段，JVM会根据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**。

关于<clinit>()方法的执行特点和细节：

（1）<clinit>()方法是由编译器自动收集类中的**所有类变量（静态变量）的赋值**动作和**静态语句块（static{}块）中的语句的执行**合并产生的，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，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中可以赋值，但不能访问。

（2）<clinit>()方法与类的构造函数（实例构造器<init>()）不同。它不需要显式调用父类的类构造器， 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。所以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。

（3）<clinit>()方法对于类或接口并不是必需的。如果一个类中没有静态语句块，也没有对类变量（静态变量）进行赋值操作，则编译器可以选择不为这个类生成<clinit>()方法。

（4）接口中不能使用静态语句块，但仍然有变量初始化赋值（public static final）操作，因此接口与类一样都会生成<clinit>()方法。但接口与类的区别是，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。另外接口的实现类在初始化时也一样不会执行接口的<clinit>()方法。

（5）虚拟机会保证一个类的<clinit>()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，知道活动线程执行<clinit>()方法完毕。

类加载器

“通过一个类的全限定名来获取描述此类的二进制字节流”，实现这个动作的代码模块称为“类加载器”。**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**，每个类加载器，都拥有一个独立的类名称空间。即判断两个类是否“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，这两个类就必定不相等。判断两个类对象是否相等的方法有：

（1）Class对象的equals()方法，

（2）isAssignanleForm()方法、

（3）isInstance()方法，

（4）使用instanceof关键字做对象所属关系判定

等等...

双亲委派模型  
 从Java虚拟机的角度上讲，**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；令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**全都继承于抽象类java.lang.ClassLoader**。

绝大部分Java程序都会使用到如下3种系统提供的类加载器：

**（1）启动类加载器（Bootstrap ClassLoader）**：

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（仅按照文件名识别，如rt.jar，名字不符合的类库即使放在lib目录下也不会被加载）类库加载到虚拟机内存中。

开发者**无法在Java程序中直接调用启动类加载器，所以开发者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，就直接使用null替代即可（return null）。**

**（2）扩展类加载器（Extension 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**。

**（3）应用程序类加载器（Application ClassLoader）**：

这个加载器由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，所以一般也称它为系统类加载器。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**。

自定义类加载器

User ClassLoader

自定义类加载器

User ClassLoader

应用程序类加载器

Application ClassLoader

启动类加载器

Bootstrap ClassLoader

扩展类加载器

Extension ClassLoader

类加载器双亲委派模型

（Parents Delegation Model）

双亲委派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当有自己的父类加载器。需要注意的是，这里类加载器之间的父子关系不是以继承（Inheritance）的关系来实现，而是使用组合（Composition）关系来复用父类加载器的代码。

类加载器的双亲委派模型目前广泛应用于所有的Java程序中，但它并非一个强制性的约束模型，而是Java设计者推荐给开发者的一种类加载器实现方式。

使用双亲委派模型来组织类加载器之间的关系，好处就是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，它存放在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启动类加载器进行加载，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，如果没有使用双亲委派模型，由各个类加载器自行去加载，如果用户自己编写了一个称为java.lang.Object的类，并放在程序的ClassPath中，那系统中将会出现多个不同的Object类，Java类型体系中最基础的行为也就无法保证，应用程序将会变得一片混乱。

**双亲委派模型的实现源码**

**（java.lang.ClassLoader.loadClass()）**

protected Class<?> loadClass(String name, boolean resolve)

throws ClassNotFoundException

{

synchronized (getClassLoadingLock(name)) {

//首先，检查请求的类是否已经被当前类加载器加载过

Class<?> c = findLoadedClass(name);

if (c == null) {

long t0 = System.nanoTime();

try {

if (parent != null) {

//若当前父加载器不为空，往上传递加载请求，递归调用父加载器进行加载

c = parent.loadClass(name, false);

} else {

//直到父加载器为空，则当前最终父类加载器为启动类加载器（说明当前类为Extension ClassLoader）

c = findBootstrapClassOrNull(name);

}

} catch (ClassNotFoundException e) {

// 如果父类加载器抛出ClassNotFoundException

// 说明父类加载器无法完成加载请求

}

if (c == null) {

// 在父类加载器无法加载的时候

// 再调用本身的findClass方法来进行类加载，找到class文件并把字节码加载到内存中

long t1 = System.nanoTime();

c = findClass(name);

// this is the defining class loader; record the stats

sun.misc.PerfCounter.getParentDelegationTime().addTime(t1 - t0);

sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);

sun.misc.PerfCounter.getFindClasses().increment();

}

}

if (resolve) {

resolveClass(c);

//加载完字节码后，会根据需要进行验证、解析

}

return c;

}

}

破坏双亲委派模型

双亲委派模型并非是一个强制性的约束模型，而是Java设计者推荐给开发者的类加载器实现方式。所以到目前，双亲委派模型出现过3种较大规模的“被破坏”情况。

（1）第一次被破坏：

虚拟机在进行类加载的时候，会调用加载器的私有方法loadClassInte

rnal()，而这个方法的唯一逻辑就是去调用自己的loadClass()。用户继承java.lang.ClassLoader后，如果重写loadClass()方法，就会破坏双亲委派模型，因为双亲委派的具体逻辑就实现在这个方法之中。因此JDK1.2之后不提倡用户再去覆盖loadClass()方法，而是把自己的类加载逻辑写到findClass()方法中，在loadClass()方法的逻辑里如果父类加载失败，则会调用自己的findClass()方法来完成加载，这样就可以保证新写出来的类加载器是符合双亲委派原则的。

（2）第二次被破坏：

由于模型自身缺陷所导致。双亲委派模型可以很好解决各个类加载器的基础类的统一问题（越基础的类由越上层的加载器进行加载），但如果基础类需要调用回下层用户代码，那就会产生问题。例如JNDI服务，它的代码（rt.jar）是由启动类加载器去加载的。JNDI的目的是对资源进行集中管理和查找，它需要调用由一些独立厂商实现并部署在应用程序的ClassPath下的JNDI接口提供者（SPI）的代码，但启动类加载器的加载范围是不会加载这些代码的。

因此Java设计团队引入了“线程上下文类加载器（Thread Context ClassLoader）”。这个类加载器可以通过java.lang.Thread类的setContext

ClassLoader()方法进行设置，如果创建线程时还未设置，它将会从父线程中继承一个，如果在应用程序的全局范围内都没有设置过的话，那这个类加载器默认就是应用程序类加载器。

有了线程上下文类加载器，JNDI服务就可以使用这个类加载器去加载所需要的SPI代码，也就是父类加载器请求子类加载器去完成类加载的动作，打通双亲委派模型的层次结构来逆向使用类加载器。Java中所有涉及SPI的加载动作基本上都采用这种方式，比如JNDI、JDBC、JCE、JAXB和JBI等等。

（3）第三次被破坏：

由于用户对程序动态性的追求而导致的，这里所说的动态性是指：代码热替换（HotSwap）、模块热部署（Hot Deployment）等。目前，OSGi技术已经成为业界的Java模块化标准，而OSGi实现模块化热部署的关键是它自定义的类加载器机制的实现。每一个程序模块（Bundle）都有一个自己的类加载器，当需要更换一个Bundle时，就把Bundle连同类加载器一起换掉以实现代码的热替换。

在OSGi环境下，类加载器不再是双亲委派模型中的树状结构，而是进一步发展为更加复杂的网状结构。当收到类加载请求时，OSGi将按照以下顺序进行类搜索。

**满足双亲委**

**派模型原则**

1）将以java.\*开头的类委派给父类加载器加载

2）否则，将委派列表名单内的类委派给父类加载器加载

3）否则，将Import列表中的类委派给Export这个类的Bundle的类加载器加载

4）否则，查找当前Bundle的ClassPath，使用自己的类加载器加载

5）否则，查找类是否在自己的Fragment Bundle中，如果在，则委派给Fragment Bundle的类加载器加载

6）否则，查找Dynamic Import列表的Bundle，委派给对应Bundle的类加载器加载

7）否则，类查找失败