类加载过程

类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、验证、准备、解析、初始化、使用和卸载七个阶段。

其中类加载的过程包括了加载、验证、准备、解析、初始化五个阶段。在这五个阶段中，加载、验证、准备和初始化这四个阶段发生的顺序某些情况下可以在初始化阶段之后开始，这是为了支持Java语言的运行时绑定（也成为动态绑定或晚期绑定）。另外注意这里的几个阶段是成，因为这些阶段通常都是互相交叉地混合进行的，通常在一个阶段执行的过程中调用或激活另一个阶段。

这里简要说明下Java中的绑定：绑定指的是把一个方法的调用与方法所在的类(方法主体)关联起来，对java来说，绑定分为静态绑定和动态绑

静态绑定：即前期绑定。在程序执行前方法已经被绑定，此时由编译器或其它连接程序实现。针对java，简单的可以理解为程序编译期的绑定
private和构造方法是前期绑定的。

动态绑定：即晚期绑定，也叫运行时绑定。在运行时根据具体对象的类型进行绑定。在java中，几乎所有的方法都是后期绑定的。

类加载的条件：（加载、验证、准备、解析、初始化、使用和卸载）

Class只有在被使用的时候才会被加载，Java虚拟机不会无缘无故的加载一个Class。

Java虚拟机规范中并没有进行规定第一个阶段“加载”什么时候开始，但是对于“初始化“，Java虚拟机规定有下面几种情况必须对Class进行初始化之前就开始）。

- 当一个类进行一个实例化时，如new，反射，克隆，序列化
- 调用类的静态方法时，即使用了字节码invokestatic指令
- 当使用类的静态字段或者方法时，即使用了字节码的getstatic和putstatic指令的时候
- 子类进行初始化的时候发现父类没有初始化，会先初始化父类
- 启动Java虚拟机含有main方法的那个类

上面的情况称为 主动引用，只有这些情况会触发初始化。其他情况都属于被动引用，都不会发生初始化。下面代码示例：

```
public class SSClass
{
    static
    {
        System.out.println("SSClass");
    }
}
public class SuperClass extends SSClass
{
    static
    {
        System.out.println("SuperClass init!");
    }

    public static int value = 123;

    public SuperClass()
    {
        System.out.println("init SuperClass");
    }
}
public class SubClass extends SuperClass
{
    static
    {
        System.out.println("SubClass init");
    }

    static int a;

    public SubClass()
    {
        System.out.println("init SubClass");
    }
}
public class NotInitialization
{
    public static void main(String[] args)
    {
        System.out.println(SubClass.value);
    }
}
```

输出

```
SSClass
SuperClass init!
123
```

加载

加载阶段是类加载过程（加载、验证、准备、解析、初始化、使用和卸载）中的第一个阶段。虚拟机会需要完成三件事情：

- 通过一个类的全限定名来获取此类的二进制文件流
- 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

加载阶段和连接阶段（Linking）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动作，仍然属于连接阶段的内容，这两个阶段的开始时间仍然保持着固定的先后顺序。

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验证阶段大致会完成4个阶段的检验动作：

- 文件格式验证：验证字节流是否符合Class文件格式的规范；例如：是否以魔术0xCAFEBABE开头、主次版本号是否在当前虚拟机的有不被支持的类型
- 元数据验证：对字节码描述的信息进行语义分析（注意：对比javac编译阶段的语义分析），以保证其描述的信息符合Java语言规范除了java.lang.Object之外
- 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
- 符号引用验证：确保解析动作能正确执行

验证阶段是非常重要的，但不是必须的，它对程序运行期没有影响，如果所引用的类经过反复验证，那么可以考虑采用-Xverifynone参数来机类加载的时间

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。这时候进行内存分配的仅而不包括实例变量，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

```
public static int value=123;
```

那变量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指令是程序被所以把value赋值为123的动作将在初始化阶段才会执行

至于“特殊情况”是指：public static final int value=123，即当类字段的字段属性是ConstantValue时，会在准备阶段初始化为指定的值，所以段初始化为123而非0.

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。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

初始化

类初始化阶段是类加载过程的最后一步，到了初始化阶段，才真正开始执行类中定义的java程序代码。在准备阶段，变量已经赋过一次系统
造器()方法的过程。

- clinit>()方法是由编译器自动收集类中的所有变量的赋值动作和静态语句块static中的语句合并产生的，编译器收集的顺序一定是clini
- clinit>()方法与类的构造函数（实例构造init()方法）不同，不需要显示的调用父类构造器，虚拟机会保证子类的clinit>()方法执行之前
- 由于父类的clinit>()方法先执行，那么父类的静态块语句优先于子类的变量赋值操作.
- 虚拟机会保证一个类的clinit>()方法在多线程环境中被正确的加锁和同步，如果多个线程同时初始化一个类，那么只会有一个线程执行
线程执行完毕clinit>()方法之后，其他线程就不会再去执行clinit>()方法了。一个类的clinit>()方法只会执行一次。

很有意思的一个题

```
package jvm.classload;

public class StaticTest
{
    public static void main(String[] args)
    {
        staticFunction();
    }

     static StaticTest st = new StaticTest();

     static
     {
         System.out.println("1");
     }

     {
         System.out.println("2");
     }

     StaticTest()
     {
         System.out.println("3");
         System.out.println("a="+a+",b="+b);
     }

     public static void staticFunction(){
         System.out.println("4");
     }

     int a=110;
     static int b =112;
}
```

类加载器

Java虚拟机团队把类加载阶段中的“通过一个类的全限定名来获取此类的二进制字节流”这个动作放到了Java虚拟机的外部去实现，以便让应

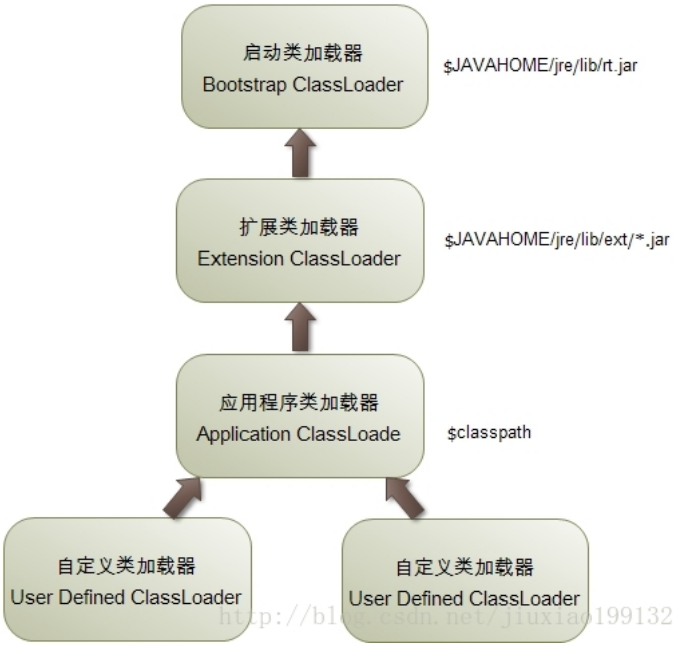
类。实现这个动作的代码模块被称为“类加载器”。

在Java中类加载器是核心组件，所有的类都是由类加载器加载的。类加载器将获取到的二进制字节流读入系统，然后交给Java虚拟机进行连

类加载器虽然只用于实现类的加载动作，但它在Java程序中的作用却远远不限于类加载阶段。对于任意一个类，都需要由加载它的类加载器中的唯一性。通俗一点就是：比较两个类是否“相等”，只有在这两个类由同一个类加载器加载的前提下才有意义。否则，即使这两个类是来的类加载器不同，那么这两个类必定不相等。
类加载器的分类

从Java虚拟机的角度，只存在两种不同的类加载器：一种是启动类加载器（Bootstrap ClassLoader），这个类加载器由C++语言实现，是虚有的其他类加载器，这些类加载器由Java语言实现，独立于虚拟机外部，并且全部继承抽象类java.lang.ClassLoader。

从java开发人员的角度来看，类加载器可以划分的更加细致一些。绝大部分java程序都会使用到下面三种系统提供的类加载器：



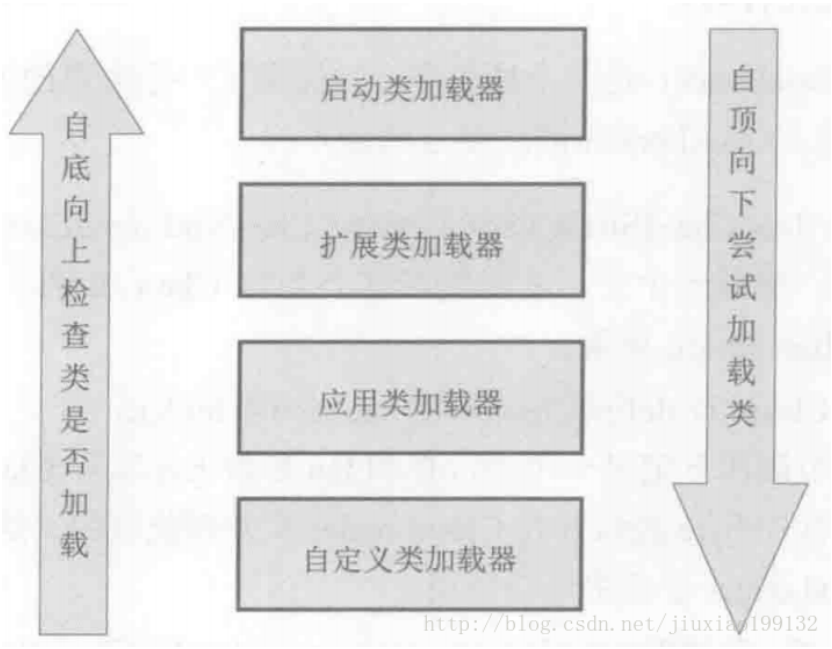
- 启动类加载器（Bootstrap ClassLoader）：使用C++语言实现，并非ClassLoader的子类。主要负责加载存放在JAVA_HOME / jre / lib Xbootclasspath参数所指定路径中以rt.jar命名的文件

- 扩展类加载器（Extension ClassLoader）：由sun.misc.Launcher$ExtClassLoader实现，它负责加载AVA_HOME / lib / ext目录中的，路径中的所有类库

- 应用程序类加载器（Application ClassLoader）：由sun.misc.Launcher$AppClassLoader实现，它负责加载classpath对应的jar及目类加载器

此外还有每个应用程序还可以自定义类加载器来扩展获取Class的能力

类加载器双亲委派模型



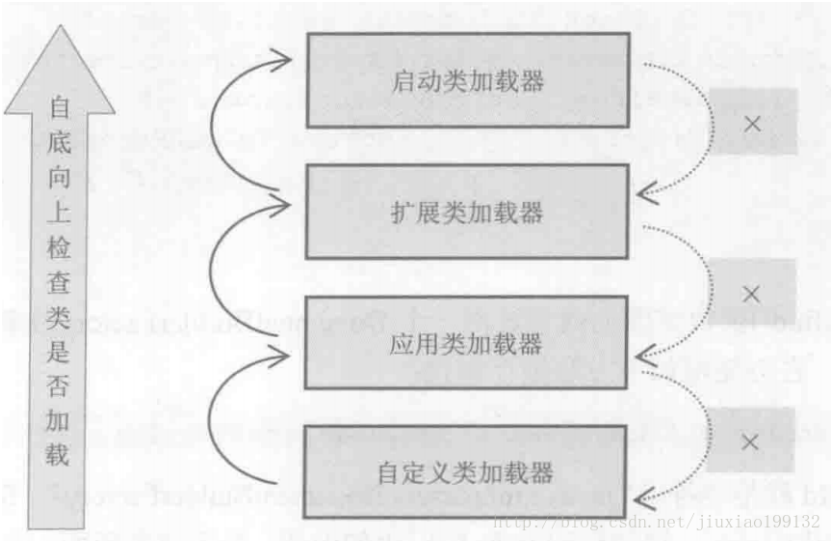
类加载器会先判断这个类是否被加载，如果加载了直接返回。否则判断是否有父级类加载器，如果有就由父级类加载器加载，父级类加载器载。

这样有一个好处就是安全性：

因为在此机制下，用户自定义的类加载器不可能加载应该由父加载器加载的可靠类，从而防止不可靠甚至恶意的代码代替由父加载器加

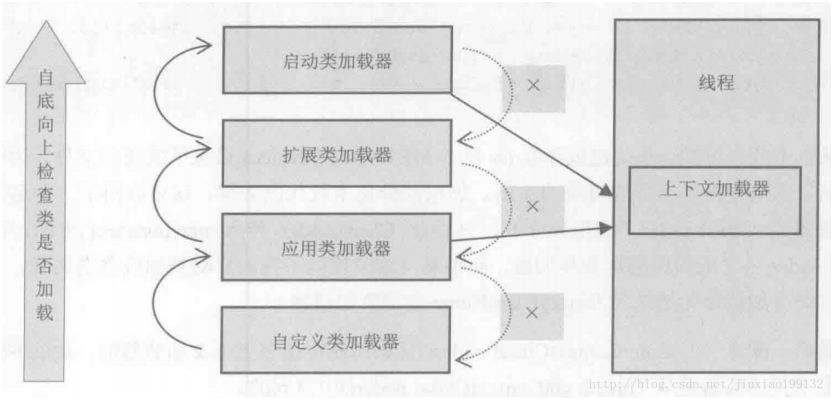
例如，java.lang.Object类总是由根类加载器加载，其他任何用户自定义的类加载器都不可能加载含有恶意代码的java.lang.Object类

双亲委派模型的弊端



上层的类加载器无法访问下层的类加载器，在应用类访问系统类没有问题，但是系统类访问应用类就无法访问了

双亲委派模型的”破坏”（补充）



双亲委派模型的第一次“被破坏”其实发生在双亲委派模型出现之前–即JDK1.2发布之前。由于双亲委派模型是在JDK1.2之后才被引入的，而
java.lang.ClassLoader则是JDK1.0时候就已经存在，面对已经存在 的用户自定义类加载器的实现代码，Java设计者引入双亲委派模型时不得
JDK1.2之后的java.lang.ClassLoader添加了一个新的proceted方法findClass()，在此之前，用户去继承java.lang.ClassLoader的唯一目的就是
行类加载的时候会调用加载器的私有方法loadClassInternal()，而这个方法的唯一逻辑就是去调用自己的loadClass()。JDK1.2之后已不再提倡
把自己的类加载逻辑写到findClass()方法中，在loadClass()方法的逻辑里，如果父类加载器加载失败，则会调用自己的findClass()方法来完
加载器是符合双亲委派模型的。

双亲委派模型的第二次“被破坏”是这个模型自身的缺陷所导致的，双亲委派模型很好地解决了各个类加载器的基础类统一问题(越基础的类由
所以被称为“基础”，是因为它们总是作为被调用代码调用的API。但是，如果基础类又要调用用户的代码，那该怎么办呢。

这并非是不可能的事情，一个典型的例子便是JNDI服务，它的代码由启动类加载器去加载(在JDK1.3时放进rt.jar)，但JNDI的目的就是对资源
立厂商实现部部署在应用程序的classpath下的JNDI接口提供者(SPI, Service Provider Interface)的代码，但启动类加载器不可能“认识”之些
为了解决这个困境，Java设计团队只好引入了一个不太优雅的设计：线程上下文件类加载器(Thread Context ClassLoader)。这个类加载器可
setContextClassLoader()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
程序类加载器。了有线程上下文类加载器，JNDI服务使用这个线程上下文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
上就是打通了双亲委派模型的层次结构来逆向使用类加载器，已经违背了双亲委派模型，但这也是无可奈何的事情。Java中所有涉及SPI的加
JNDI,JDBC,JCE,JAXB和JBI等。

双亲委派模型的第三次“被破坏”是由于用户对程序的动态性的追求导致的，例如OSGi的出现。在OSGi环境下，类加载器不再是双亲委派模
状结构