JVM ClassLoader

ClassLoader的作用就是将class文件加载到JVM虚拟机中，在JVM中sun.misc.Launcher时JVM的虚拟机入口应用，如下所示：

*public Launcher() {*

*Launcher.ExtClassLoader var1;*

*try {*

*var1 = Launcher.ExtClassLoader.getExtClassLoader();*

*} catch (IOException var10) {*

*throw new InternalError("Could not create extension class loader", var10);*

*}*

*try {*

*this.loader = Launcher.AppClassLoader.getAppClassLoader(var1);*

*} catch (IOException var9) {*

*throw new InternalError("Could not create application class loader", var9);*

*}*

*//设置AppClassLoader为线程上下文类加载器*

*Thread.currentThread().setContextClassLoader(this.loader);*

*.....*

*var3 = (SecurityManager)this.loader.loadClass(var2).newInstance();*

*.....*

*}*

在Launcher中依次出现ExtClassLoader和AppClassLoader，在JVM中还默认启动启动类加载器Bootstrap Class Loader：

* 启动类加载器（Bootstrap ClassLoader），是Java类加载层次中最顶层的类加载，用C++实现的类加载器，是虚拟机的一部分，主要加载JDK中核心类库，如rt.jar,resources.jar等，完全由JVM控制，开发者无法访问
* 扩展类加载器（Extension ClassLoader），负责加载{JRE\_HOME}/lib/ext目录或者被java.ext.dirs系统变量指定的路径
* 应用程序加载器（App ClassLoader），负责加载classPath路径上指定的类库，如果程序中没有定义类加载器，一般作为默认的类加载器

下面是这三个类加载器的设置的参数列表：



# 自定义ClassLoader

除了Java默认提供的三个ClassLoader外，用户还可以根据需要定义自己的ClassLoader，这些自定义的ClassLoader都必须继承java.lang.ClassLoader类，也包括Java提供的Extension ClassLoader和App ClassLoader在内，但是Bootstrap ClassLoader不继承ClassLoader。ClassLoader的类结构如下所示：

//将byte字节流解析成JVM能够识别的Class对象，有了这个方法一位着我们

//不仅可以通过Class文件获得Class对象，其他的字节流都可以

*Class<?> defineClass ( byte[] , int , int )*

//实现类的加载规则，从而取得要加载类的字节码

*Class<?> findClass(String)*

//如果不想重新定义加载类额规则，也没有复杂的处理逻辑

//只是想能够一个加载一个自己指定的类，可以直接使用load

*Class<?> loadClass(String)*

//链接参数类，链接参照上面，不再赘述

*void resolveClass(Class<?>)*

其中核心的方式loadClass，通过制定的全限定类名加载class，调用loadClass(String,boolean)方法，一般步骤：

1. 执行findLoadedClass(String)，检测这个class是不是已经加载过
2. 执行父加载器的loadClass方法，如果父加载器为null，则JVM内置的加载器去替代（BootStrap ClassLoader）
3. 如果向上委托的父加载器没有加载成功，则通过findClass去查找

源码如下所示：



要编写一个ClassLoader子类，一般覆盖findClass方法，步骤如下：

1. 编写类继承ClassLoader抽象类
2. 复写findClass方法
3. 在findClass方法中调用defineClass方法，将二进制数据转换成类实例。

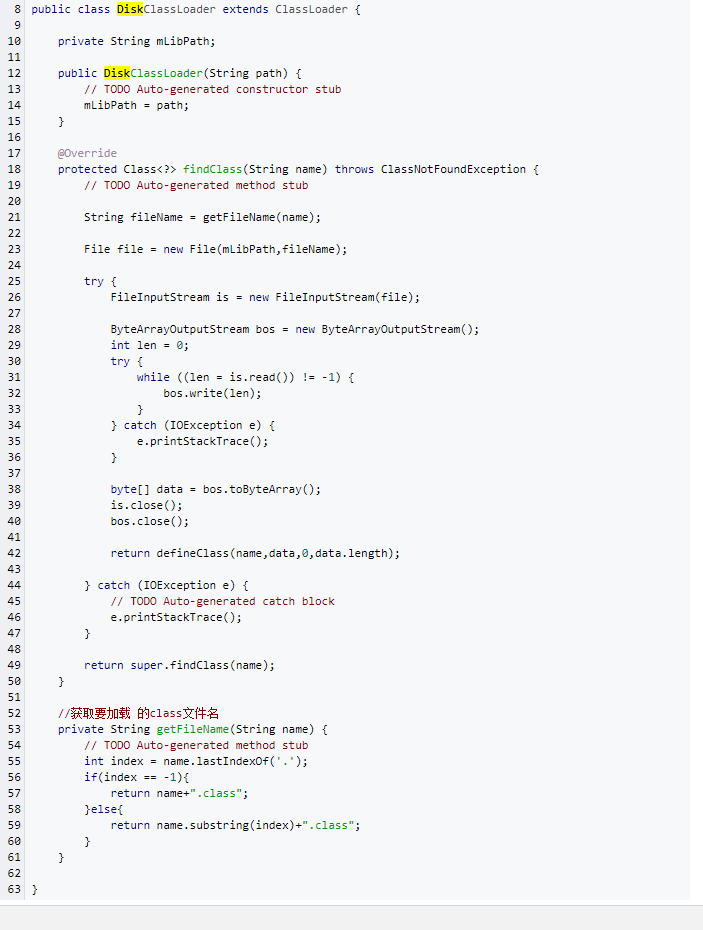
下面是自定义一个ClassLoader，默认加载路径D:\\lib下的jar包和资源：

* 测试用的类文件，Test.java



将其编译成class文件，放到D:\\lib路径下

* 自定义ClassLoader，DiskClassLoader



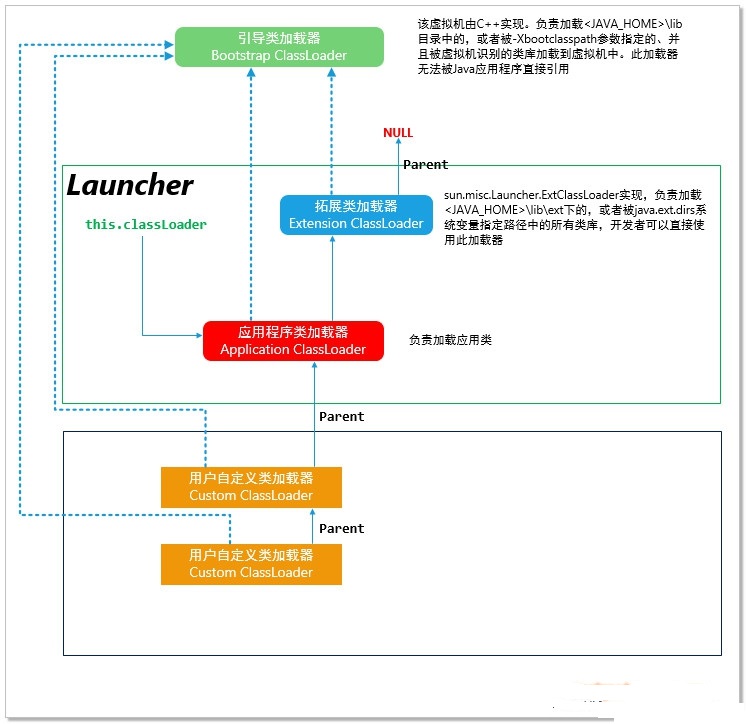
覆盖findClass方法，在其中定义查找class的方法，然后数据通过defineClass生成Class对象。

* 测试，编写测试代码，调用Test对象的say方法



# 2. ClassLoader加载类原理

ClassLoader使用的是双亲委托模型来搜索类，每个ClassLoader都有一个父类加载器的引用（除了Bootstrap ClassLoader）。当一个ClassLoader实例需要加载某个类时，在搜索类之前，先委托给父类加载器，这个过程由上至下以此检查，如下图所示：

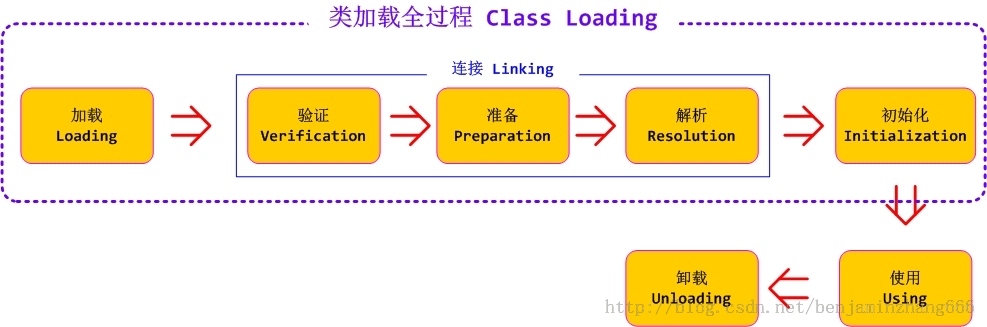


首先由顶层的类加载器Bootstrap ClassLoader试图加载，如果没加载到，则把任务转交给Extension ClassLoader加载，如果也没加载到，则转交给App ClassLoader 进行加载，如果它也没有加载得到的话，则返回给委托的发起者，由它到指定的文件系统或网络等URL中加载该类。如果它们都没有加载到这个类时，则抛出ClassNotFoundException异常。否则将这个找到的类生成一个类的定义，并将它加载到内存当中，最后返回这个类在内存中的Class实例对象。使用双亲委托模型的优点：

* 可以避免重复加载，当父类已经加载该类，子ClassLoader就没有必要再加载一次。
* 避免安全隐患，不使用委托模式，例如用户随时使用自定义的String来动态替代java核心api中的定义类型，存在非常大的安全隐患。

# 3. 类加载全过程

JVM把class文件加载到内存，并对数据进行校验、解析和初始化，最终形成JVM可以直接使用的Java对象组件的全过程如下所示：



主要步骤如下：

* 装载(Loading)，查找和导入class文件
* 链接(Linking)，执行校验、准备和解析步骤，解析步骤可以选择：

1. 校验，检查载入Class文件数据的正确性
2. 准备，给类的静态变量分配存储空间
3. 解析，将符号引用转成直接引用

* 初始化，对类的静态变量、静态代码块执行初始化工作。

1. **加载**

在加载阶段，虚拟机完成以下3个工作：

* 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换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个数据的访问入口。

相对于其他阶段，这个是开发人员可控的阶段，这个阶段可以使用系统提供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，开发人员可通过自定义的类加载器去控制字节流的获取方式（重写loadClass方法）。加载完成后，虚拟机外部的二进制字节流按照虚拟机所需的格式存储在方法区中。

记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，但是夹在加载阶段之中的动作，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。

1. **验证**

验证是连接阶段的第一步，这个阶段的目的是为了确保Class文件的字节流包含的信息符合当前虚拟机的要求，主要包括以下校验：

* 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处理范围之内、常量池中的常量是否有不被支持的类型。
* 元数据验证，对字节码描述的信息进行语义分析，以保证其描述的信息符合Java语义规范的要求，例如这个类是否有父类
* 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。

验证阶段非常重要，但不是必须的，对程序运行期没有影响，可以采用-Xverifynoe参数来关闭大部分的类验证措施，以缩短虚拟机类加载的时间。

1. **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。这个阶段分配的内存仅包括类变量（static修饰的变量），而不包括实例变量，实例变量在对象实例化时随着对象一起分配在堆中。

类变量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：

*public static int value=123;*

变量value在准备阶段后的初始值为0而不是123，这个时候尚未开始任何Java方法，而把value的赋值存放在类构造器中，但是特殊情况，例如：

*public final static int value=123;*

当类字段属性是ConstantValue时，会在准备阶段初始化为指定的值。

1. **解析**

解析阶段是虚拟机将常量池中的符号引用替换成直接引用的过程，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、方法句柄和调用点限定符7类符号引用。

1. **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，变量已经进行过一次系统要求的初始值，而在初始化阶段，则根据程序员通过程序制定的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。

<clinit>()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static{}中的语句合并产生的，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的，静态语句块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可以赋值，但是不能访问。例如：

*public class Test*

*{*

*static*

*{*

*i=0;*

*System.out.println(i);*

*//这句编译器会报错：Cannot reference a field before it is defined（非法向前应用）*

*}*

*static int i=1;*

*}*

<clinit>()方法与实例构造器<init>()方法不同，它不需要显示地调用父类构造器，虚拟机会保证在子类<init>()方法执行之前，父类的<clinit>()方法方法已经执行完毕。

1. **使用阶段和卸载阶段，不再详述**

**参考文献：**

http://www.importnew.com/18548.html

https://www.cnblogs.com/zhengbin/p/5631363.html

http://www.th7.cn/Program/java/201601/756452.shtml