

类加载运行全过程

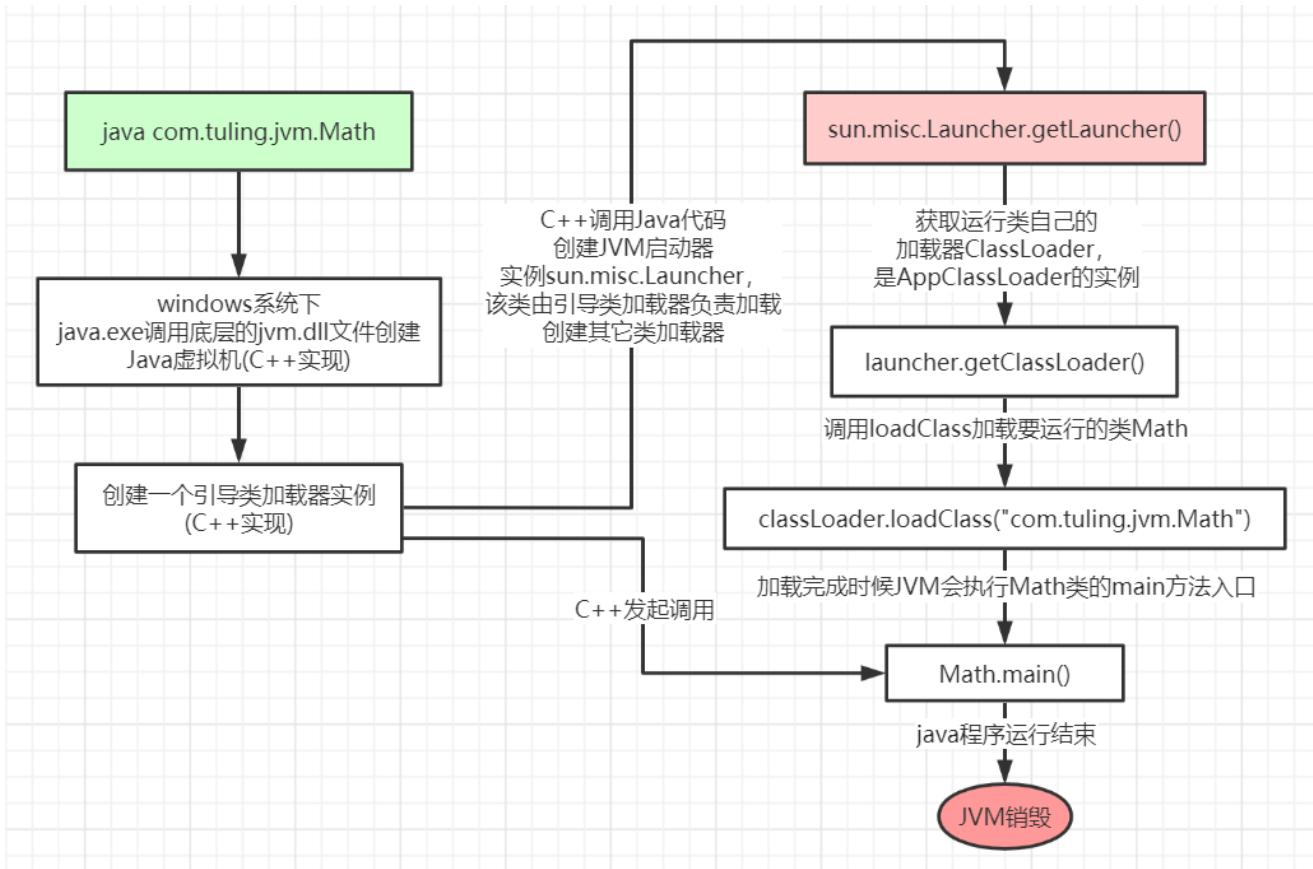
当我们用java命令运行某个类的main函数启动程序时，首先需要通过**类加载器**把主类加载到JVM。

```
1 package com.tuling.jvm;
2
3 public class Math {
4     public static final int initData = 666;
5     public static User user = new User();
6
7     public int compute() { //一个方法对应一块栈帧内存区域
8         int a = 1;
9         int b = 2;
10        int c = (a + b) * 10;
11        return c;
12    }
13
14    public static void main(String[] args) {
15        Math math = new Math();
16        math.compute();           一手微信study322
17    }
18
19 }
```

通过Java命令执行代码的大体流程如下：

扫一扫 不怀孕



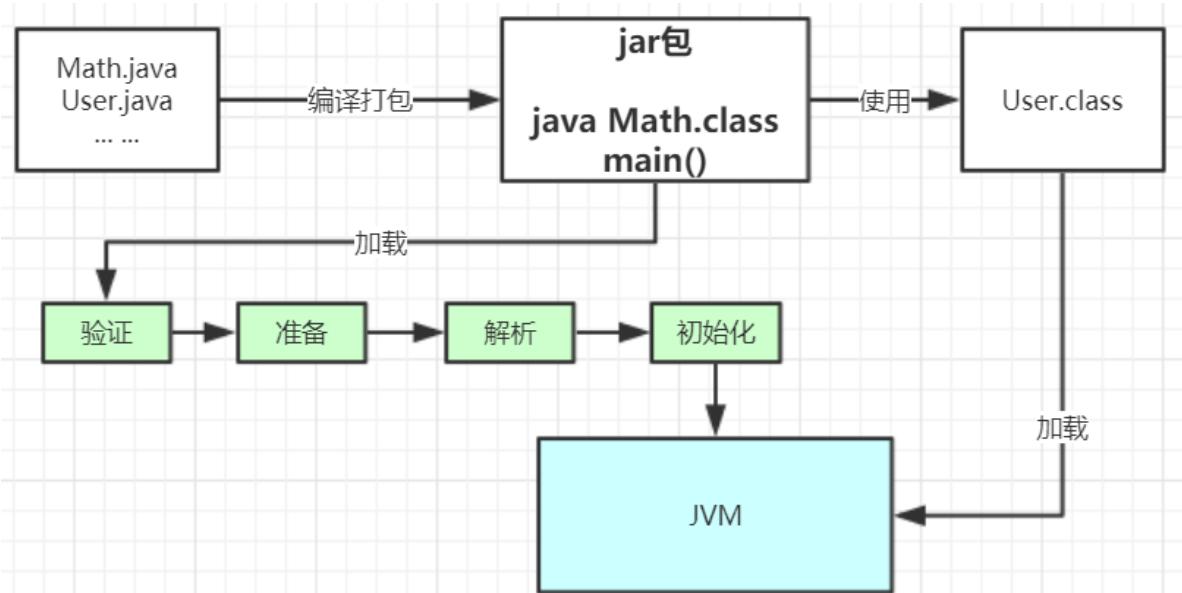


加载 >> 验证 >> 准备 >> 解析 >> 初始化 >> 使用 >> 卸载

- 加载：在硬盘上查找并通过IO读入字节码文件，使用到类时才会加载，例如调用类的main()方法，new对象等等，在加载阶段会在内存中生成一个**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
- 验证：校验字节码文件的正确性
- 准备：给类的静态变量分配内存，并赋予默认值
- 解析：将**符号引用**替换为直接引用，该阶段会把一些静态方法(符号引用，比如main()方法)替换为指向数据所存内存的指针或句柄等(直接引用)，这是所谓的**静态链接**过程(类加载期间完成)，**动态链接**是在程序运行期间完成的将符号引用替换为直接引用，下节课会讲到动态链接
- **初始化**：对类的静态变量初始化为指定的值，执行静态代码块

扫一扫 不怀孕





类被加载到方法区中后主要包含 **运行时常量池、类型信息、字段信息、方法信息、类加载器的引用、对应class实例的引用等信息。**

类加载器的引用：这个类到类加载器实例的引用

对应class实例的引用：类加载器在加载类信息放到方法区中后，会创建一个对应的Class类型的对象实例放到堆(Heap)中，作为开发人员访问方法区中类定义的入口和切入点。

一手微信 study322

注意，主类在运行过程中如果使用到其它类，会逐步加载这些类。

jar包或war包里的类不是一次性全部加载的，是使用到时才加载。

```

1 public class TestDynamicLoad {
2
3     static {
4         System.out.println("*****load TestDynamicLoad*****");
5     }
6
7     public static void main(String[] args) {
8         new A();
9         System.out.println("*****load test*****");
10        B b = null; //B不会加载，除非这里执行 new B()
11    }
12 }
13
14 class A {
15     static {
16         System.out.println("*****load A*****");
17     }
18
19     public A() {
20         System.out.println("*****initial A*****");

```

扫一扫 不怀孕



```

21 }
22 }
23
24 class B {
25     static {
26         System.out.println("*****load B*****");
27     }
28
29     public B() {
30         System.out.println("*****initial B*****");
31     }
32 }
33
34 运行结果:
35 *****load TestDynamicLoad*****
36 *****load A*****
37 *****initial A*****
38 *****load test*****

```

类加载器和双亲委派机制 一手微信study322

上面的类加载过程主要是通过类加载器来实现的，Java里有如下几种类加载器

- 引导类加载器：负责加载支撑JVM运行的位于JRE的lib目录下的核心类库，比如rt.jar、charsets.jar等
- 扩展类加载器：负责加载支撑JVM运行的位于JRE的lib目录下的ext扩展目录中的JAR类包
- 应用程序类加载器：负责加载ClassPath路径下的类包，主要就是加载你自己写的那些类
- 自定义加载器：负责加载用户自定义路径下的类包

看一个类加载器示例：

```

1 public class TestJDKClassLoader {
2
3     public static void main(String[] args) {
4         System.out.println(String.class.getClassLoader());
5         System.out.println(com.sun.crypto.provider.DESKeyFactory.class.getClassLoader()
6             .getClass().getName());
6         System.out.println(TestJDKClassLoader.class.getClassLoader().getClassName());
7
8         System.out.println();

```

扫一扫 不怀孕



```
9 ClassLoader appClassLoader = ClassLoader.getSystemClassLoader();
10 ClassLoader extClassLoader = appClassLoader.getParent();
11 ClassLoader bootstrapLoader = extClassLoader.getParent();
12 System.out.println("the bootstrapLoader : " + bootstrapLoader);
13 System.out.println("the extClassLoader : " + extClassLoader);
14 System.out.println("the appClassLoader : " + appClassLoader);
15
16 System.out.println();
17 System.out.println("bootstrapLoader加载以下文件: ");
18 URL[] urls = Launcher.getBootstrapClassPath().getURLs();
19 for (int i = 0; i < urls.length; i++) {
20     System.out.println(urls[i]);
21 }
22
23 System.out.println();
24 System.out.println("extClassLoader加载以下文件: ");
25 System.out.println(System.getProperty("java.ext.dirs"));
26
27 System.out.println();
28 System.out.println("appClassLoader加载以下文件: ");
29 System.out.println(System.getProperty("java.class.path"));
30
31 }
32 }
33
34 运行结果:
35 null
36 sun.misc.Launcher$ExtClassLoader
37 sun.misc.Launcher$AppClassLoader
38
39 the bootstrapLoader : null
40 the extClassLoader : sun.misc.Launcher$ExtClassLoader@3764951d
41 the appClassLoader : sun.misc.Launcher$AppClassLoader@14dad5dc
42
43 bootstrapLoader加载以下文件:
44 file:/D:/dev/Java/jdk1.8.0_45/jre/lib/resources.jar
45 file:/D:/dev/Java/jdk1.8.0_45/jre/lib/rt.jar
46 file:/D:/dev/Java/jdk1.8.0_45/jre/lib/sunrsasign.jar
47 file:/D:/dev/Java/jdk1.8.0_45/jre/lib/jsse.jar
48 file:/D:/dev/Java/jdk1.8.0_45/jre/lib/jce.jar
49 file:/D:/dev/Java/jdk1.8.0_45/jre/lib/charsets.jar
50 file:/D:/dev/Java/jdk1.8.0_45/jre/lib/jfr.jar
51 file:/D:/dev/Java/jdk1.8.0_45/jre/classes
```

扫一扫 不怀孕



```

52
53 extClassLoader加载以下文件:
54 D:\dev\Java\jdk1.8.0_45\jre\lib\ext;C:\Windows\Sun\Java\lib\ext
55
56 appClassLoader加载以下文件:
57 D:\dev\Java\jdk1.8.0_45\jre\lib\charsets.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\deploy.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\ext\access-bridge-64.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\ext\cldrdata.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\ext\dnsns.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\ext\jaccess.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\ext\ib\ext\jfxrt.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\ext\localizedata.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\ext\nashorn.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\ext\sunec.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\ext\sunjce_provider.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\ext\sunpkcs11.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\ext\zipfs.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\javaws.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\jce.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\jfr.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\jfxswt.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\jsse.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\management-agent.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\plugin.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\resources.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\rt.jar;D:\ideaProjects\project-all\target\classes;C:\Users\zhuge\.m2\repository\org\apache\zookeeper\zookeeper\3.4.12\zookeeper-3.4.12.jar;C:\Users\zhuge\.m2\repository\org\slf4j\slf4j-api\1.7.25\slf4j-api-1.7.25.jar;C:\Users\zhuge\.m2\repository\org\slf4j\slf4j-log4j12\log4j12\1.7.25\log4j-1.7.25.jar;C:\Users\zhuge\.m2\repository\log4j\log4j\1.2.17\log4j-1.2.17.jar;C:\Users\zhuge\.m2\repository\jline\jline\0.9.94\jline-0.9.94.jar;C:\Users\zhuge\.m2\repository\org\apache\yetus\audience-annotations\0.5.0\audience-annotations-0.5.0.jar;C:\Users\zhuge\.m2\repository\io\netty\netty\3.10.6.Final\netty-3.10.6.Final.jar;C:\Users\zhuge\.m2\repository\com\google\guava\guava\22.0\guava-22.0.jar;C:\Users\zhuge\.m2\repository\com\google\code\findbugs\jsr305\1.3.9\jsr305-1.3.9.jar;C:\Users\zhuge\.m2\repository\com\google\errorprone\error_prone_annotations\2.0.18\error_prone_annotations-2.0.18.jar;C:\Users\zhuge\.m2\repository\com\google\j2objc\j2objc-annotations\1.1\j2objc-annotations-1.1.jar;C:\Users\zhuge\.m2\repository\org\codehaus\mojo\animal-sniffer-annotations\1.14\animal-sniffer-annotations-1.14.jar;D:\dev\IntelliJ IDEA 2018.3.2\lib\idea_rt.jar

```

58

类加载器初始化过程：

参见类运行加载全过程图可知其中会创建JVM启动器实例sun.misc.Launcher。

sun.misc.Launcher初始化使用了单例模式设计，保证一个JVM虚拟机内只有一个sun.misc.Launcher实例。

在Launcher构造方法内部，其创建了两个类加载器，分别是

sun.misc.Launcher.ExtClassLoader(扩展类加载器)和sun.misc.Launcher.AppClassLoader(应用类加载器)。

JVM默认使用Launcher的getClassLoader()方法返回的类加载器AppClassLoader的实

扫
一
扫
不
怀
孕

```

1 //Launcher的构造方法
2 public Launcher() {

```



```

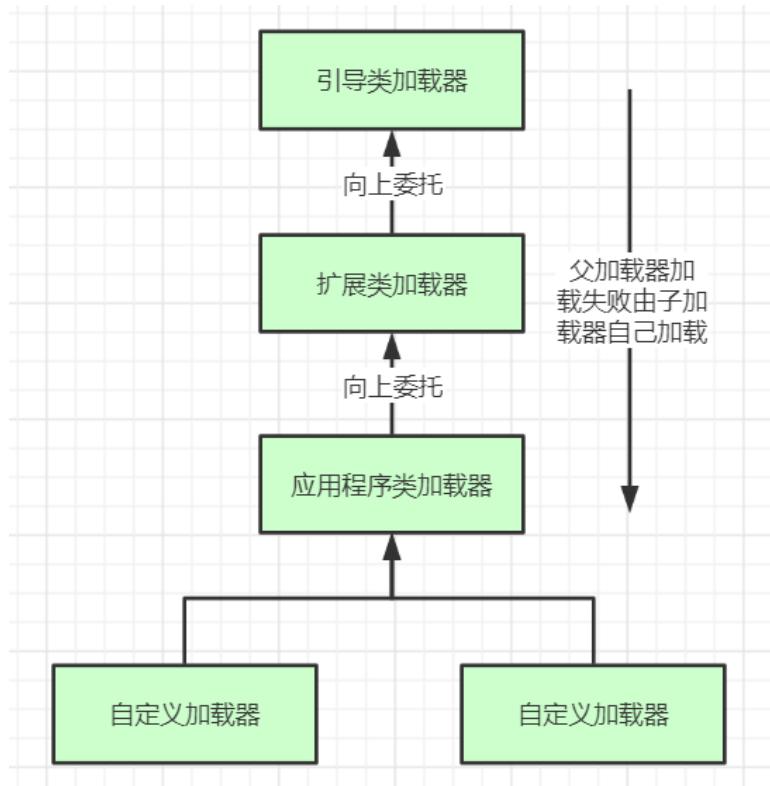
3  Launcher.ExtClassLoader var1;
4  try {
5      //构造扩展类加载器，在构造的过程中将其父加载器设置为null
6      var1 = Launcher.ExtClassLoader.getExtClassLoader();
7  } catch (IOException var10) {
8      throw new InternalError("Could not create extension class loader", var10);
9  }
10
11 try {
12     //构造应用类加载器，在构造的过程中将其父加载器设置为ExtClassLoader,
13     //Launcher的loader属性值是AppClassLoader，我们一般都是用这个类加载器来加载我们自己写的应用程序
14     this.loader = Launcher.AppClassLoader.getAppClassLoader(var1);
15 } catch (IOException var9) {
16     throw new InternalError("Could not create application class loader", var9);
17 }
18
19 Thread.currentThread().setContextClassLoader(this.loader);
20 String var2 = System.getProperty("java.security.manager");
21 . . . . . //省略一些不需关注代码
22
23 }

```

一手微信study322

双亲委派机制

JVM类加载器是有亲子层级结构的，如下图



扫一扫 不怀孕



这里类加载其实就有一个**双亲委派机制**，加载某个类时会先委托父加载器寻找目标类，找不到再委托上层父加载器加载，如果所有父加载器在自己的加载类路径下都找不到目标类，则在自己的类加载路径中查找并载入目标类。

比如我们的Math类，最先会找应用程序类加载器加载，应用程序类加载器会先委托扩展类加载器加载，扩展类加载器再委托引导类加载器，顶层引导类加载器在自己的类加载路径里找了半天没找到Math类，则向下退回加载Math类的请求，扩展类加载器收到回复就自己加载，在自己的类加载路径里找了半天也没找到Math类，又向下退回Math类的加载请求给应用程序类加载器，应用程序类加载器于是在自己的类加载路径里找Math类，结果找到了就自己加载了。。

双亲委派机制说简单点就是，先找父亲加载，不行再由儿子自己加载

我们来看下应用程序类加载器AppClassLoader加载类的双亲委派机制源码，AppClassLoader的loadClass方法最终会调用其父类ClassLoader的loadClass方法，该方法的大体逻辑如下：

1. 首先，检查一下指定名称的类是否已经加载过，如果加载过了，就不需要再加载，直接返回。
2. 如果此类没有加载过，那么，再判断一下是否有父加载器；如果有父加载器，则由父加载器加载（即调用parent.loadClass(name, false);）或者调用bootstrap类加载器来加载。
3. 如果父加载器及bootstrap类加载器都没有找到指定的类，那么调用当前类加载器的findClass方法来完成类加载。

```
1 //ClassLoader的loadClass方法，里面实现了双亲委派机制
2 protected Class<?> loadClass(String name, boolean resolve)
3 throws ClassNotFoundException
4 {
5     synchronized (getClassLoadingLock(name)) {
6         // 检查当前类加载器是否已经加载了该类
7         Class<?> c = findLoadedClass(name);
8         if (c == null) {
9             long t0 = System.nanoTime();
10            try {
11                if (parent != null) { //如果当前加载器父加载器不为空则委托父加载器加载该类
12                    c = parent.loadClass(name, false);
13                } else { //如果当前加载器父加载器为空则委托引导类加载器加载该类
14                    c = findBootstrapClassOrNull(name);
15                }
16            } catch (ClassNotFoundException e) {
17                // ClassNotFoundException thrown if class not found
18                // from the non-null parent class loader
19            }
20        }
```

扫一扫 不怀孕



```

21 if (c == null) {
22     // If still not found, then invoke findClass in order
23     // to find the class.
24     long t1 = System.nanoTime();
25     //都会调用URLClassLoader的findClass方法在加载器的类路径里查找并加载该类
26     c = findClass(name);
27
28     // this is the defining class loader; record the stats
29     sun.misc.PerfCounter.getParentDelegationTime().addTime(t1 - t0);
30     sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);
31     sun.misc.PerfCounter.getFindClasses().increment();
32 }
33 }
34 if (resolve) { //不会执行
35     resolveClass(c);
36 }
37 return c;
38 }
39 }

```

一手微信study322

为什么要设计双亲委派机制？

- 沙箱安全机制：自己写的java.lang.String.class类不会被加载，这样便可以防止核心API库被随意篡改
- 避免类的重复加载：当父亲已经加载了该类时，就没有必要子ClassLoader再加载一次，**保证被加载类的唯一性**

看一个类加载示例：

```

1 package java.lang;
2
3 public class String {
4     public static void main(String[] args) {
5         System.out.println("*****My String Class*****");
6     }
7 }
8
9 运行结果：
10 错误：在类 java.lang.String 中找不到 main 方法，请将 main 方法定义为：
11     public static void main(String[] args)
12 否则 JavaFX 应用程序类必须扩展javafx.application.Application

```

扫一扫 不怀孕



全盘负责委托机制

“全盘负责”是指当一个ClassLoader装载一个类时，除非显示的使用另外一个ClassLoader，该类所依赖及引用的类也由这个ClassLoader载入。

自定义类加载器示例：

自定义类加载器只需要继承 java.lang.ClassLoader 类，该类有两个核心方法，一个是 loadClass(String, boolean)，实现了**双亲委派机制**，还有一个方法是 findClass，默认实现是空方法，所以我们自定义类加载器主要是重写 findClass 方法。

```
1 public class MyClassLoaderTest {  
2     static class MyClassLoader extends ClassLoader {  
3         private String classPath;  
4  
5         public MyClassLoader(String classPath) {  
6             this.classPath = classPath;  
7         }  
8  
9         private byte[] loadByte(String name) throws Exception {  
10            name = name.replaceAll("\\.", "/");  
11            FileInputStream fis = new FileInputStream(classPath + "/" + name  
12 + ".class");  
13            int len = fis.available();  
14            byte[] data = new byte[len];  
15            fis.read(data);  
16            fis.close();  
17            return data;  
18        }  
19  
20        protected Class<?> findClass(String name) throws ClassNotFoundException {  
21            try {  
22                byte[] data = loadByte(name);  
23                //defineClass将一个字节数组转为Class对象，这个字节数组是class文件读取后最终的字节  
24                return defineClass(name, data, 0, data.length);  
25            } catch (Exception e) {  
26                e.printStackTrace();  
27                throw new ClassNotFoundException();  
28            }  
29        }  
30  
31    }  
32}
```

扫一扫 不怀孕



```

33 public static void main(String args[]) throws Exception {
34     //初始化自定义类加载器，会先初始化父类ClassLoader，其中会把自定义类加载器的父加载器设置为应用程序类加载器AppClassLoader
35     MyClassLoader classLoader = new MyClassLoader("D:/test");
36     //D盘创建 test/com/tuling/jvm 几级目录，将User类的复制类User1.class丢入该目录
37     Class clazz = classLoader.loadClass("com.tuling.jvm.User1");
38     Object obj = clazz.newInstance();
39     Method method = clazz.getDeclaredMethod("sout", null);
40     method.invoke(obj, null);
41     System.out.println(clazz.getClassLoader().getClass().getName());
42 }
43 }
44
45 运行结果：
46 =====自己的加载器加载类调用方法=====
47 com.tuling.jvm.MyClassLoaderTest$MyClassLoader

```

打破双亲委派机制

再来一个沙箱安全机制示例，尝试打破双亲委派机制，用自定义类加载器加载我们自己实现的java.lang.String.class

一手微信study322

```

1 public class MyClassLoaderTest {
2     static class MyClassLoader extends ClassLoader {
3         private String classPath;
4
5         public MyClassLoader(String classPath) {
6             this.classPath = classPath;
7         }
8
9         private byte[] loadByte(String name) throws Exception {
10            name = name.replaceAll("\\.", "/");
11            FileInputStream fis = new FileInputStream(classPath + "/" + name
12 + ".class");
13            int len = fis.available();
14            byte[] data = new byte[len];
15            fis.read(data);
16            fis.close();
17            return data;
18        }
19    }
20
21    protected Class<?> findClass(String name) throws ClassNotFoundException {
22        try {

```

扫一扫 不怀孕



```
23 byte[] data = loadByte(name);
24 return defineClass(name, data, 0, data.length);
25 } catch (Exception e) {
26 e.printStackTrace();
27 throw new ClassNotFoundException();
28 }
29 }
30
31 /**
32 * 重写类加载方法，实现自己的加载逻辑，不委派给双亲加载
33 * @param name
34 * @param resolve
35 * @return
36 * @throws ClassNotFoundException
37 */
38 protected Class<?> loadClass(String name, boolean resolve)
39 throws ClassNotFoundException {
40 synchronized (getClassLoadingLock(name)) {
41 // First, check if the class has already been loaded
42 Class<?> c = findLoadedClass(name);  
一子微信study322
43
44 if (c == null) {
45 // If still not found, then invoke findClass in order
46 // to find the class.
47 long t1 = System.nanoTime();
48 c = findClass(name);
49
50 // this is the defining class loader; record the stats
51 sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);
52 sun.misc.PerfCounter.getFindClasses().increment();
53 }
54 if (resolve) {
55 resolveClass(c);
56 }
57 return c;
58 }
59 }
60 }
61
62 public static void main(String args[]) throws Exception {
63 MyClassLoader classLoader = new MyClassLoader("D:/test");
64 //尝试用自己改写类加载机制去加载自己写的java.lang.String.class
```

扫一扫 不怀孕



```
65 Class clazz = classLoader.loadClass("java.lang.String");
66 Object obj = clazz.newInstance();
67 Method method= clazz.getDeclaredMethod("sout", null);
68 method.invoke(obj, null);
69 System.out.println(clazz.getClassLoader().getClass().getName());
70 }
71 }
72
73 运行结果:
74 java.lang.SecurityException: Prohibited package name: java.lang
75 at java.lang.ClassLoader.preDefineClass(ClassLoader.java:659)
76 at java.lang.ClassLoader.defineClass(ClassLoader.java:758)
```

Tomcat打破双亲委派机制

以Tomcat类加载为例，Tomcat 如果使用默认的双亲委派类加载机制行不行？

我们思考一下：Tomcat是个web容器，那么它要解决什么问题：

1. 一个web容器可能需要部署两个应用程序，不同的应用程序可能会**依赖同一个第三方类库的不同版本**，不能要求同一个类库在同一个服务器只有一份，因此要保证每个应用程序的类库都是独立的，保证相互隔离。
2. 部署在同一个web容器中**相同的类库相同的版本可以共享**。否则，如果服务器有10个应用程序，那么要有10份相同的类库加载进虚拟机。
3. **web容器也有自己依赖的类库，不能与应用程序的类库混淆**。基于安全考虑，应该让容器的类库和程序的类库隔离开来。
4. web容器要支持jsp的修改，我们知道，jsp 文件最终也是要编译成class文件才能在虚拟机中运行，但程序运行后修改jsp已经是司空见惯的事情，web容器需要支持 jsp 修改后不用重启。

再看看我们的问题：**Tomcat 如果使用默认的双亲委派类加载机制行不行？**

答案是不行的。为什么？

第一个问题，如果使用默认的类加载器机制，那么是无法加载两个相同类库的不同版本的，默认的类加载器是不管你是什么版本的，只在乎你的全限定类名，并且只有一份。

第二个问题，默认的类加载器是能够实现的，因为他的职责就是**保证唯一性**。

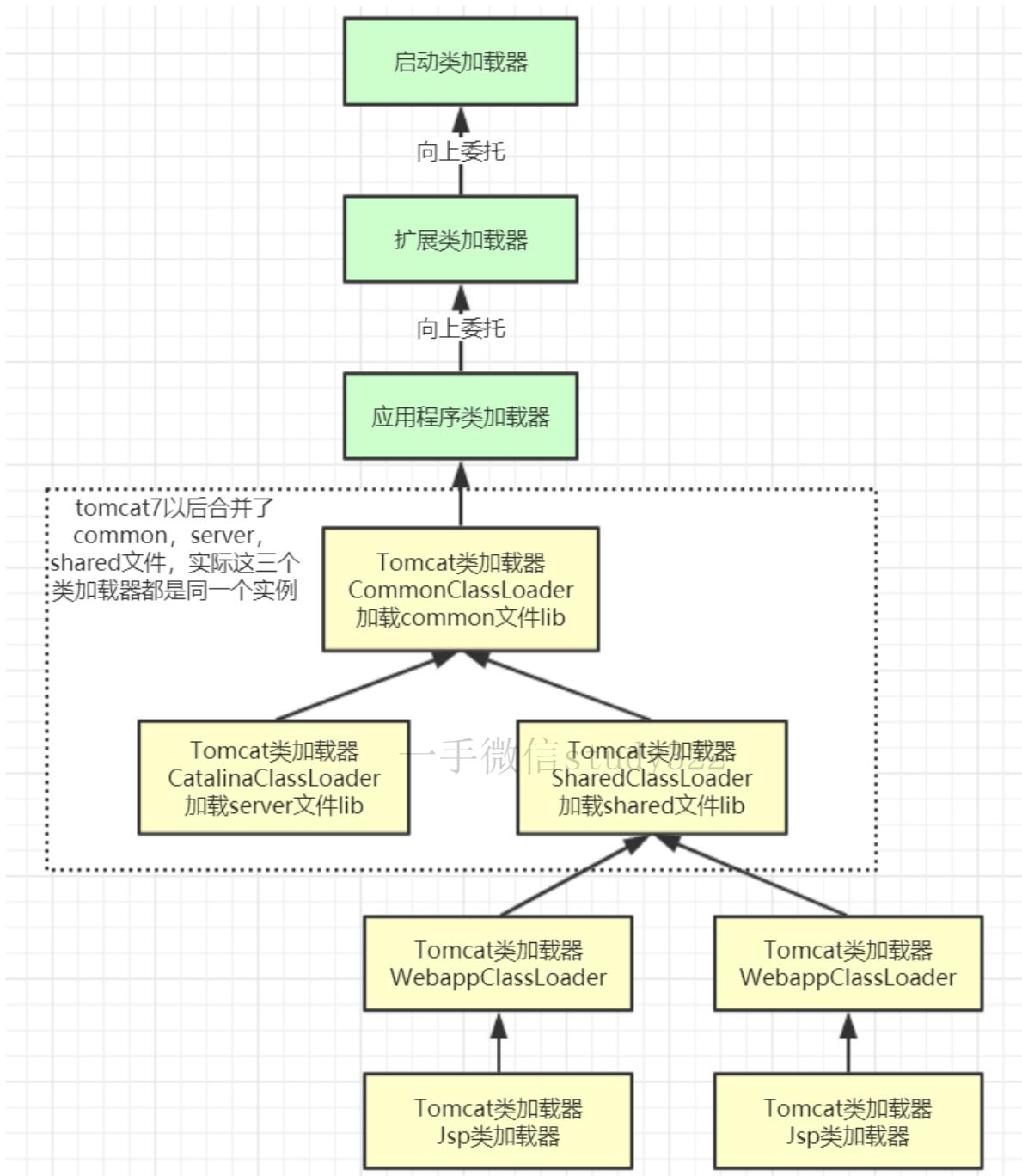
第三个问题和第一个问题一样。

我们再看第四个问题，我们想我们要怎么实现jsp文件的热加载，jsp 文件其实也就是class文件，那么如果修改了，但类名还是一样，类加载器会直接取方法区中已经存在的，修改后的jsp是不会重新加载的。那么怎么办呢？我们可以直接卸载掉这jsp文件的类加载器，所以到了，每个jsp文件对应一个唯一的类加载器，当一个jsp文件修改了，就直接卸载这个器。重新创建类加载器，重新加载jsp文件。



扫一扫 不怀孕

Tomcat自定义加载器详解



tomcat的几个主要类加载器：

- commonLoader: Tomcat最基本的类加载器，加载路径中的class可以被Tomcat容器本身以及各个Webapp访问；
- catalinaLoader: Tomcat容器私有的类加载器，加载路径中的class对于Webapp不可见；
- sharedLoader: 各个Webapp共享的类加载器，加载路径中的class对于所有Webapp可见，但是对于Tomcat容器不可见；
- WebappClassLoader: 各个Webapp私有的类加载器，加载路径中的class对于每个Webapp可见，比如加载war包里相关的类，每个war包应用都有自己的

扫一扫 不怀孕



WebappClassLoader，实现相互隔离，比如不同war包应用引入了不同的spring版本，这样实现就能加载各自的spring版本；

从图中的委派关系中可以看出：

CommonClassLoader能加载的类都可以被CatalinaClassLoader和SharedClassLoader使用，从而实现了公有类库的共用，而CatalinaClassLoader和SharedClassLoader自己能加载的类则与对方相互隔离。

WebAppClassLoader可以使用SharedClassLoader加载到的类，但各个WebAppClassLoader实例之间相互隔离。

而JasperLoader的加载范围仅仅是这个JSP文件所编译出来的那一个.Class文件，它出现的目的就是为了被丢弃：当Web容器检测到JSP文件被修改时，会替换掉目前的JasperLoader的实例，并通过再建立一个新的Jsp类加载器来实现JSP文件的热加载功能。

tomcat 这种类加载机制违背了java 推荐的双亲委派模型了吗？答案是：违背了。

很显然，tomcat 不是这样实现，tomcat 为了实现隔离性，没有遵守这个约定，**每个**

webappClassLoader加载自己的目录下的class文件，不会传递给父类加载器，打破了双亲委派机制。

一手微信study322

模拟实现Tomcat的webappClassLoader加载自己war包应用内不同版本类实现相互共存与隔离

```
1 public class MyClassLoaderTest {  
2     static class MyClassLoader extends ClassLoader {  
3         private String classPath;  
4  
5         public MyClassLoader(String classPath) {  
6             this.classPath = classPath;  
7         }  
8  
9         private byte[] loadByte(String name) throws Exception {  
10            name = name.replaceAll("\\.", "/");  
11            FileInputStream fis = new FileInputStream(classPath + "/" + name  
12 + ".class");  
13            int len = fis.available();  
14            byte[] data = new byte[len];  
15            fis.read(data);  
16            fis.close();  
17            return data;  
18        }  
19    }  
}
```

扫一扫 不怀孕



```
20
21     protected Class<?> findClass(String name) throws ClassNotFoundException {
22         try {
23             byte[] data = loadByte(name);
24             return defineClass(name, data, 0, data.length);
25         } catch (Exception e) {
26             e.printStackTrace();
27             throw new ClassNotFoundException();
28         }
29     }
30
31     /**
32      * 重写类加载方法，实现自己的加载逻辑，不委派给双亲加载
33      * @param name
34      * @param resolve
35      * @return
36      * @throws ClassNotFoundException
37     */
38     protected Class<?> loadClass(String name, boolean resolve)
39     throws ClassNotFoundException {
40         synchronized (getClassLoadingLock(name)) {
41             // First, check if the class has already been loaded
42             Class<?> c = findLoadedClass(name);
43
44             if (c == null) {
45                 // If still not found, then invoke findClass in order
46                 // to find the class.
47                 long t1 = System.nanoTime();
48
49                 //非自定义的类还是走双亲委派加载
50                 if (!name.startsWith("com.tuling.jvm")){
51                     c = this.getParent().loadClass(name);
52                 }else{
53                     c = findClass(name);
54                 }
55
56                 // this is the defining class loader; record the stats
57                 sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);
58                 sun.misc.PerfCounter.getFindClasses().increment();
59             }
60             if (resolve) {
61                 resolveClass(c);

```

扫一扫 不怀孕



```

62 }
63 return c;
64 }
65 }
66 }
67
68 public static void main(String args[]) throws Exception {
69 MyClassLoader classLoader = new MyClassLoader("D:/test");
70 Class clazz = classLoader.loadClass("com.tuling.jvm.User1");
71 Object obj = clazz.newInstance();
72 Method method= clazz.getDeclaredMethod("sout", null);
73 method.invoke(obj, null);
74 System.out.println(clazz.getClassLoader());
75
76 System.out.println();
77 MyClassLoader classLoader1 = new MyClassLoader("D:/test1");
78 Class clazz1 = classLoader1.loadClass("com.tuling.jvm.User1");
79 Object obj1 = clazz1.newInstance();
80 Method method1= clazz1.getDeclaredMethod("sout", null);
81 method1.invoke(obj1, null);
82 System.out.println(clazz1.getClassLoader());
83 }
84 }
85
86 运行结果:
87 =====自己的加载器加载类调用方法=====
88 com.tuling.jvm.MyClassLoaderTest$MyClassLoader@266474c2
89
90 =====另外一个User1版本：自己的加载器加载类调用方法=====
91 com.tuling.jvm.MyClassLoaderTest$MyClassLoader@66d3c617

```

注意：同一个JVM内，两个相同包名和类名的类对象可以共存，因为他们的类加载器可以不一样，所以看两个类对象是否是同一个，除了看类的包名和类名是否都相同之外，还需要他们的类加载器也是同一个才能认为他们是同一个。

附下User类的代码：

```

1 package com.tuling.jvm;
2
3 public class User {
4
5     private int id;
6     private String name;
7

```

扫一扫 不怀孕



```
8  public User() {
9  }
10
11 public User(int id, String name) {
12 super();
13 this.id = id;
14 this.name = name;
15 }
16
17 public int getId() {
18 return id;
19 }
20
21 public void setId(int id) {
22 this.id = id;
23 }
24
25 public String getName() {
26 return name;
27 }
28 一手微信study322
29 public void setName(String name) {
30 this.name = name;
31 }
32
33 public void sout() {
34 System.out.println("=====自己的加载器加载类调用方法=====");
35 }
36 }
```

文档: 01-VIP-从JDK源码级别彻底剖析JVM类加载机制

链接: <http://note.youdao.com/noteshare?>

id=35faf7c95e69943cdbff4642fcfd5318&sub=F6E1EB8E778044EC9BB87BA05DCE5E4B

扫一扫 不怀孕

