JVM类加载相关笔记

此文档主要记录以下部分相关笔记：

1. java.exe开始类加载运行过程
2. JVM类加载器
3. 双亲委派机制
4. 手写类加载器打破双亲委派
5. Tomcat类加载机制
6. 类加载过程

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。

package com.test;

import com.test.User;

public class Math {

public static final int initData = 666;

public static User user = new User();

public int compute() { //一个方法对应一块栈帧内存区域

int a = 1;

int b = 2;

int c = (a + b) \* 10;

return c;

}

public static void main(String[] args) {

Math math = new Math();

math.compute();

}

}

通过Java命令执行代码的大体流程如下：



其中 类加载的过程如下



其中loadClass的类加载过程有如下几步：

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

1. 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
2. 验证：校验字节码文件的正确性
3. 准备：给类的静态变量分配内存，并赋予默认值（静态变量赋值为默认值，比如Integer是0，Boolean是false，复杂数据类型为null，final修饰的直接赋值）
4. 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，比如对象的方法链接指向了另一个符号引用，在调用这个对象方法时会调用被指向的符号引用
5. 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

使用javap -v Math.class 命令查看class字节码如下：

Classfile /D:/笔记/notes/code/target/classes/com/test/Math.class

Last modified 2021-2-28; size 744 bytes

MD5 checksum c235e43e071a43ac30c22cc28b383dc1

Compiled from "Math.java"

public class com.test.Math

minor version: 0

major version: 52

flags: ACC\_PUBLIC, ACC\_SUPER

Constant pool:

#1 = Methodref #8.#35 // java/lang/Object."<init>":()V

#2 = Class #36 // com/test/Math

#3 = Methodref #2.#35 // com/test/Math."<init>":()V

#4 = Methodref #2.#37 // com/test/Math.compute:()I

#5 = Class #38 // com/test/User

#6 = Methodref #5.#35 // com/test/User."<init>":()V

#7 = Fieldref #2.#39 // com/test/Math.user:Lcom/test/User;

#8 = Class #40 // java/lang/Object

#9 = Utf8 initData

#10 = Utf8 I

#11 = Utf8 ConstantValue

#12 = Integer 666

#13 = Utf8 user

#14 = Utf8 Lcom/test/User;

#15 = Utf8 <init>

#16 = Utf8 ()V

#17 = Utf8 Code

#18 = Utf8 LineNumberTable

#19 = Utf8 LocalVariableTable

#20 = Utf8 this

#21 = Utf8 Lcom/test/Math;

#22 = Utf8 compute

#23 = Utf8 ()I

#24 = Utf8 a

#25 = Utf8 b

#26 = Utf8 c

#27 = Utf8 main

#28 = Utf8 ([Ljava/lang/String;)V

#29 = Utf8 args

#30 = Utf8 [Ljava/lang/String;

#31 = Utf8 math

#32 = Utf8 <clinit>

#33 = Utf8 SourceFile

#34 = Utf8 Math.java

#35 = NameAndType #15:#16 // "<init>":()V

#36 = Utf8 com/test/Math

#37 = NameAndType #22:#23 // compute:()I

#38 = Utf8 com/test/User

#39 = NameAndType #13:#14 // user:Lcom/test/User;

#40 = Utf8 java/lang/Object

{

public static final int initData;

descriptor: I

flags: ACC\_PUBLIC, ACC\_STATIC, ACC\_FINAL

ConstantValue: int 666

public static com.test.User user;

descriptor: Lcom/test/User;

flags: ACC\_PUBLIC, ACC\_STATIC

public com.test.Math();

descriptor: ()V

flags: ACC\_PUBLIC

Code:

stack=1, locals=1, args\_size=1

0: aload\_0

1: invokespecial #1 // Method java/lang/Object."<init>":()V

4: return

LineNumberTable:

line 2: 0

LocalVariableTable:

Start Length Slot Name Signature

0 5 0 this Lcom/test/Math;

public int compute();

descriptor: ()I

flags: ACC\_PUBLIC

Code:

stack=2, locals=4, args\_size=1

0: iconst\_1

1: istore\_1

2: iconst\_2

3: istore\_2

4: iload\_1

5: iload\_2

6: iadd

7: bipush 10

9: imul

10: istore\_3

11: iload\_3

12: ireturn

LineNumberTable:

line 6: 0

line 7: 2

line 8: 4

line 9: 11

LocalVariableTable:

Start Length Slot Name Signature

0 13 0 this Lcom/test/Math;

2 11 1 a I

4 9 2 b I

11 2 3 c I

public static void main(java.lang.String[]);

descriptor: ([Ljava/lang/String;)V

flags: ACC\_PUBLIC, ACC\_STATIC

Code:

stack=2, locals=2, args\_size=1

0: new #2 // class com/test/Math

3: dup

4: invokespecial #3 // Method "<init>":()V

7: astore\_1

8: aload\_1

9: invokevirtual #4 // Method compute:()I

12: pop

13: return

LineNumberTable:

line 12: 0

line 13: 8

line 14: 13

LocalVariableTable:

Start Length Slot Name Signature

0 14 0 args [Ljava/lang/String;

8 6 1 math Lcom/test/Math;

static {};

descriptor: ()V

flags: ACC\_STATIC

Code:

stack=2, locals=0, args\_size=0

0: new #5 // class com/test/User

3: dup

4: invokespecial #6 // Method com/test/User."<init>":()V

7: putstatic #7 // Field user:Lcom/test/User;

10: return

LineNumberTable:

line 4: 0

}

SourceFile: "Math.java"

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。

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。

package com.test;

public class TestDynamicLoad {

static {

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*load TestDynamicLoad\*\*\*\*\*\*\*\*\*\*\*\*");

}

public static void main(String[] args) {

new A();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*load test\*\*\*\*\*\*\*\*\*\*\*\*");

B b = null; //B不会加载，除非这里执行 new B()

}

}

class A {

static {

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*load A\*\*\*\*\*\*\*\*\*\*\*\*");

}

public A() {

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*initial A\*\*\*\*\*\*\*\*\*\*\*\*");

}

}

class B {

static {

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*load B\*\*\*\*\*\*\*\*\*\*\*\*");

}

public B() {

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*initial B\*\*\*\*\*\*\*\*\*\*\*\*");

}

}

运行结果：  
 \*\*\*\*\*\*\*\*\*\*\*\*\*load TestDynamicLoad\*\*\*\*\*\*\*\*\*\*\*\*  
 \*\*\*\*\*\*\*\*\*\*\*\*\*load A\*\*\*\*\*\*\*\*\*\*\*\*  
 \*\*\*\*\*\*\*\*\*\*\*\*\*initial A\*\*\*\*\*\*\*\*\*\*\*\*  
 \*\*\*\*\*\*\*\*\*\*\*\*\*load test\*\*\*\*\*\*\*\*\*\*\*\*

类加载器和双亲委派机制

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

1. 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
2. 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
3. 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
4. 自定义加载器：负责加载用户自定义路径下的类包

package com.test;

import sun.misc.Launcher;

import java.net.URL;

public class TestJDKClassLoader {

public static void main(String[] args) {

System.out.println(String.class.getClassLoader());

System.out.println(com.sun.crypto.provider.DESKeyFactory.class.getClassLoader().getClass().getName());

System.out.println(TestJDKClassLoader.class.getClassLoader().getClass().getName());

System.out.println();

ClassLoader appClassLoader = ClassLoader.getSystemClassLoader();

ClassLoader extClassloader = appClassLoader.getParent();

ClassLoader bootstrapLoader = extClassloader.getParent();

System.out.println("the bootstrapLoader : " + bootstrapLoader);

System.out.println("the extClassloader : " + extClassloader);

System.out.println("the appClassLoader : " + appClassLoader);

System.out.println();

System.out.println("bootstrapLoader加载以下文件：");

URL[] urls = Launcher.getBootstrapClassPath().getURLs();

for (int i = 0; i < urls.length; i++) {

System.out.println(urls[i]);

}

System.out.println();

System.out.println("extClassloader加载以下文件：");

System.out.println(System.getProperty("java.ext.dirs"));

System.out.println();

System.out.println("appClassLoader加载以下文件：");

System.out.println(System.getProperty("java.class.path"));

}

}

输出：

null

sun.misc.Launcher$ExtClassLoader

sun.misc.Launcher$AppClassLoader

the bootstrapLoader : null

the extClassloader : sun.misc.Launcher$ExtClassLoader@4b67cf4d

the appClassLoader : sun.misc.Launcher$AppClassLoader@18b4aac2

bootstrapLoader加载以下文件：

file:/C:/JAVA/JDK8/jre/lib/resources.jar

file:/C:/JAVA/JDK8/jre/lib/rt.jar

file:/C:/JAVA/JDK8/jre/lib/sunrsasign.jar

file:/C:/JAVA/JDK8/jre/lib/jsse.jar

file:/C:/JAVA/JDK8/jre/lib/jce.jar

file:/C:/JAVA/JDK8/jre/lib/charsets.jar

file:/C:/JAVA/JDK8/jre/lib/jfr.jar

file:/C:/JAVA/JDK8/jre/classes

extClassloader加载以下文件：

C:\JAVA\JDK8\jre\lib\ext;C:\WINDOWS\Sun\Java\lib\ext

appClassLoader加载以下文件：

C:\JAVA\JDK8\jre\lib\charsets.jar;C:\JAVA\JDK8\jre\lib\deploy.jar;C:\JAVA\JDK8\jre\lib\ext\access-bridge-64.jar;C:\JAVA\JDK8\jre\lib\ext\cldrdata.jar;C:\JAVA\JDK8\jre\lib\ext\dnsns.jar;C:\JAVA\JDK8\jre\lib\ext\jaccess.jar;C:\JAVA\JDK8\jre\lib\ext\jfxrt.jar;C:\JAVA\JDK8\jre\lib\ext\localedata.jar;C:\JAVA\JDK8\jre\lib\ext\nashorn.jar;C:\JAVA\JDK8\jre\lib\ext\sunec.jar;C:\JAVA\JDK8\jre\lib\ext\sunjce\_provider.jar;C:\JAVA\JDK8\jre\lib\ext\sunmscapi.jar;C:\JAVA\JDK8\jre\lib\ext\sunpkcs11.jar;C:\JAVA\JDK8\jre\lib\ext\zipfs.jar;C:\JAVA\JDK8\jre\lib\javaws.jar;C:\JAVA\JDK8\jre\lib\jce.jar;C:\JAVA\JDK8\jre\lib\jfr.jar;C:\JAVA\JDK8\jre\lib\jfxswt.jar;C:\JAVA\JDK8\jre\lib\jsse.jar;C:\JAVA\JDK8\jre\lib\management-agent.jar;C:\JAVA\JDK8\jre\lib\plugin.jar;C:\JAVA\JDK8\jre\lib\resources.jar;C:\JAVA\JDK8\jre\lib\rt.jar;D:\新的学习空间\笔记\notes\code\target\classes;C:\Program Files\JetBrains\IntelliJ IDEA 2020.2.3\lib\idea\_rt.jar

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的实例加载我们的应用程序。

Launcher类的构造方法

public Launcher() {

Launcher.ExtClassLoader var1;

try { // /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

var1 = Launcher.ExtClassLoader.getExtClassLoader();

} catch (IOException var10) {

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

}

try {//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，

//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

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

} catch (IOException var9) {

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

}

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

String var2 = System.getProperty("java.security.manager");

if (var2 != null) {

SecurityManager var3 = null;

if (!"".equals(var2) && !"default".equals(var2)) {

try {

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

} catch (IllegalAccessException var5) {

} catch (InstantiationException var6) {

} catch (ClassNotFoundException var7) {

} catch (ClassCastException var8) {

}

} else {

var3 = new SecurityManager();

}

if (var3 == null) {

throw new InternalError("Could not create SecurityManager: " + var2);

}

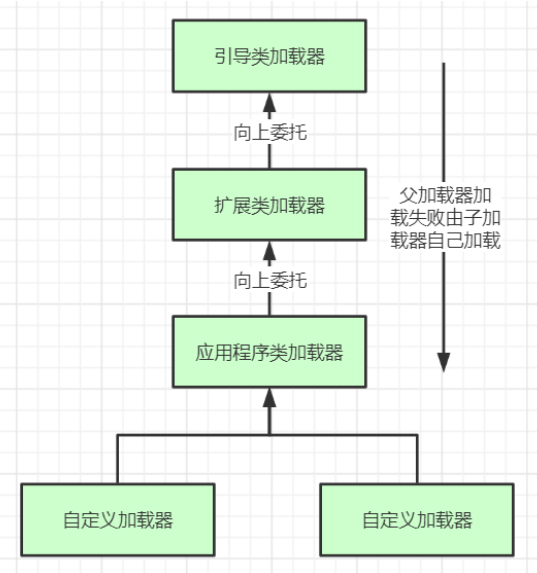
System.setSecurityManager(var3);

}

}

双亲委派机制

JVM类加载器是有亲子层级结构的，如下图



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);）.或者是调用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。

为什么要设计双亲委派机制？

1. 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
2. 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

全盘负责委托机制

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