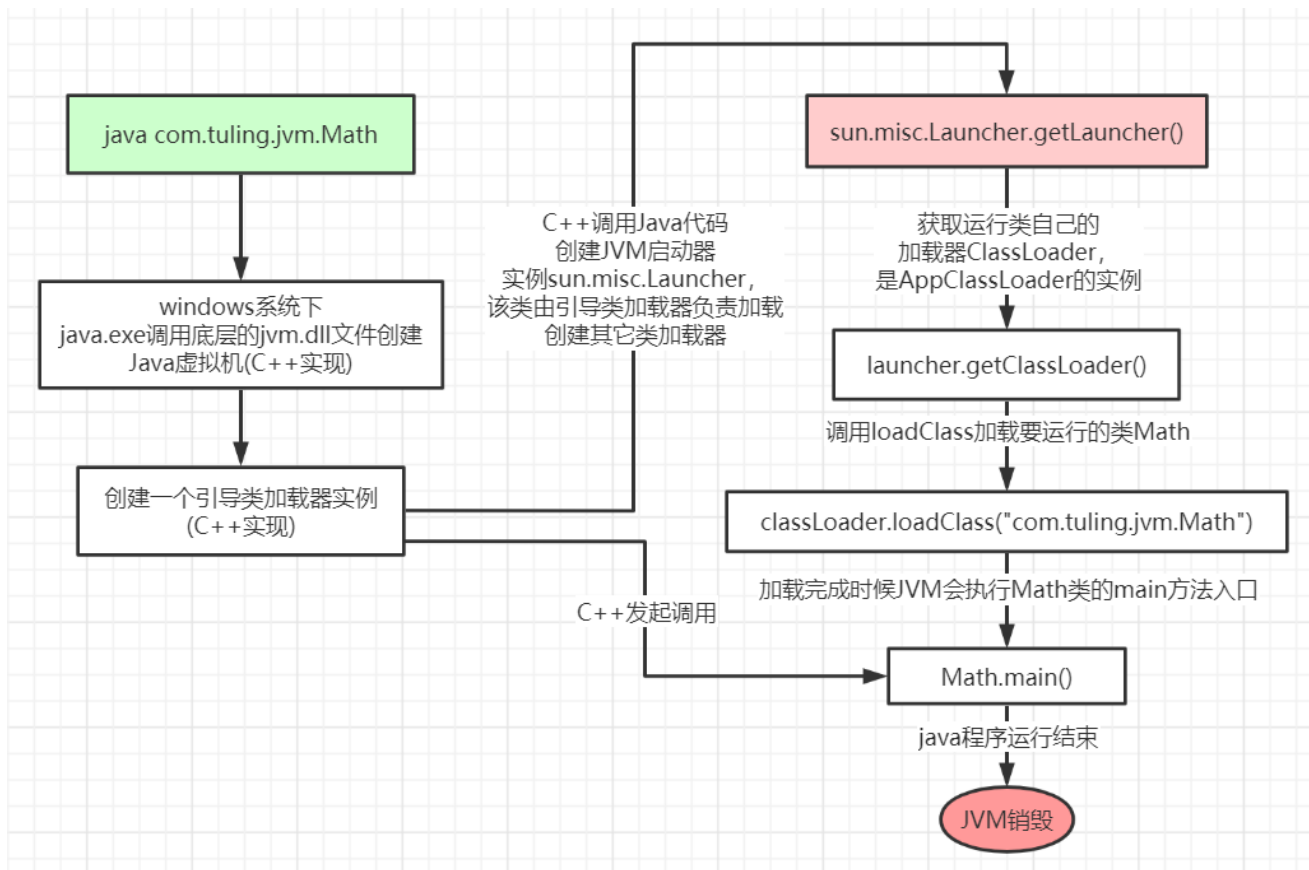


## 类加载运行全过程

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();
17    }
18
19 }
```

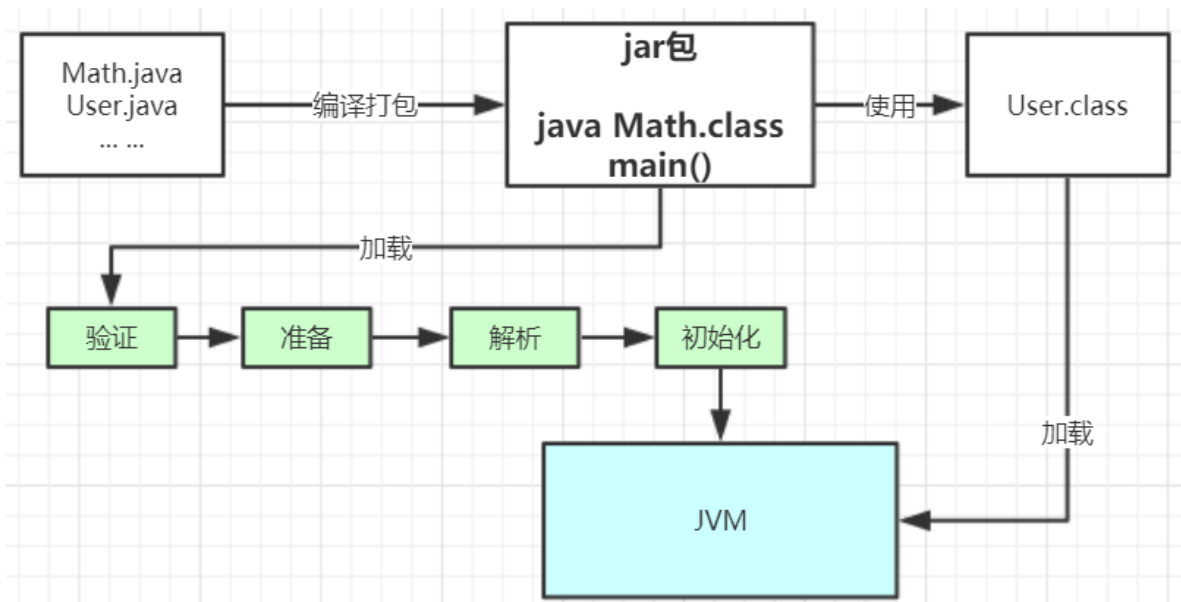
通过Java命令执行代码的大体流程如下：



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**

- **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，例如调用类的主方法，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



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。

```

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*****");
  
```

```

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

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().getClass().getName());
6          System.out.println(TestJDKClassLoader.class.getClassLoader().getClass().getName());
7
8          System.out.println();

```

```
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
```

```

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\dnssn
.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\ext\jaccess.jar;D:\dev\Java\jdk1.8.0_45\jre\l
ib\ext\jfxrt.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\ext\localedata.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\sunmscapi.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\ext\sunpkcs11.jar;D
ev\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\js
se.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\projec
t-all\target\classes;C:\Users\zhuge\.m2\repository\org\apache\zookeeper\zookeep
er\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\1.7.25\slf4j-log4j12-
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
o\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\go
ogle\code\findbugs\jsr305\1.3.9\jsr305-1.3.9.jar;C:\Users\zhuge\.m2\repository\co
m\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\j2
objc-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 IDE
A 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的实例加载我们的应用程序。

```

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

```

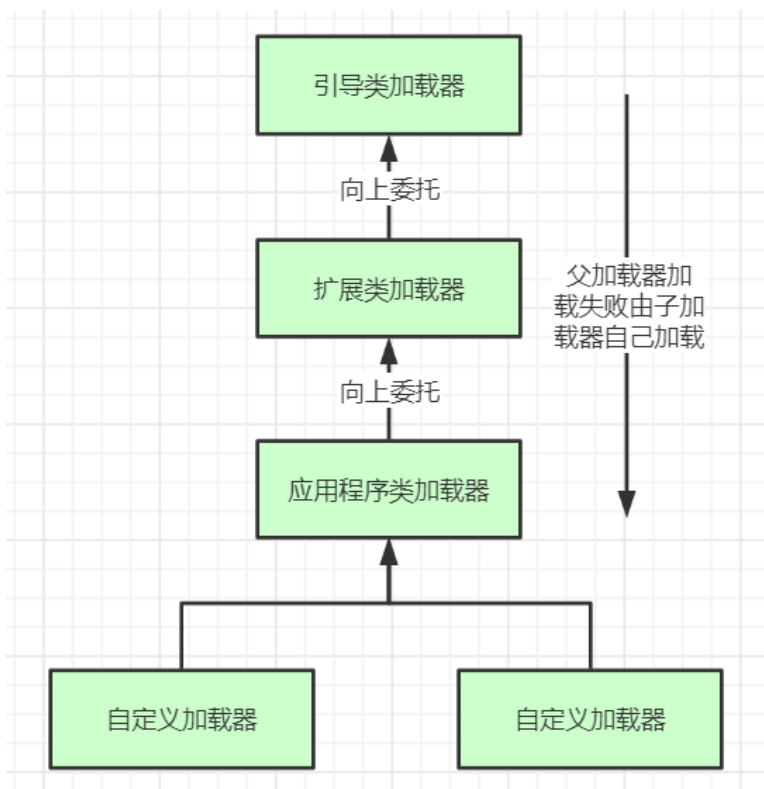
```

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 }

```

## 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。

```
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
```



```

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  }

```

## 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

```

## 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                 //数组。
25                 return defineClass(name, data, 0, data.length);
26             } catch (Exception e) {
27                 e.printStackTrace();
28                 throw new ClassNotFoundException();
29             }
30         }
31     }
32 }
```