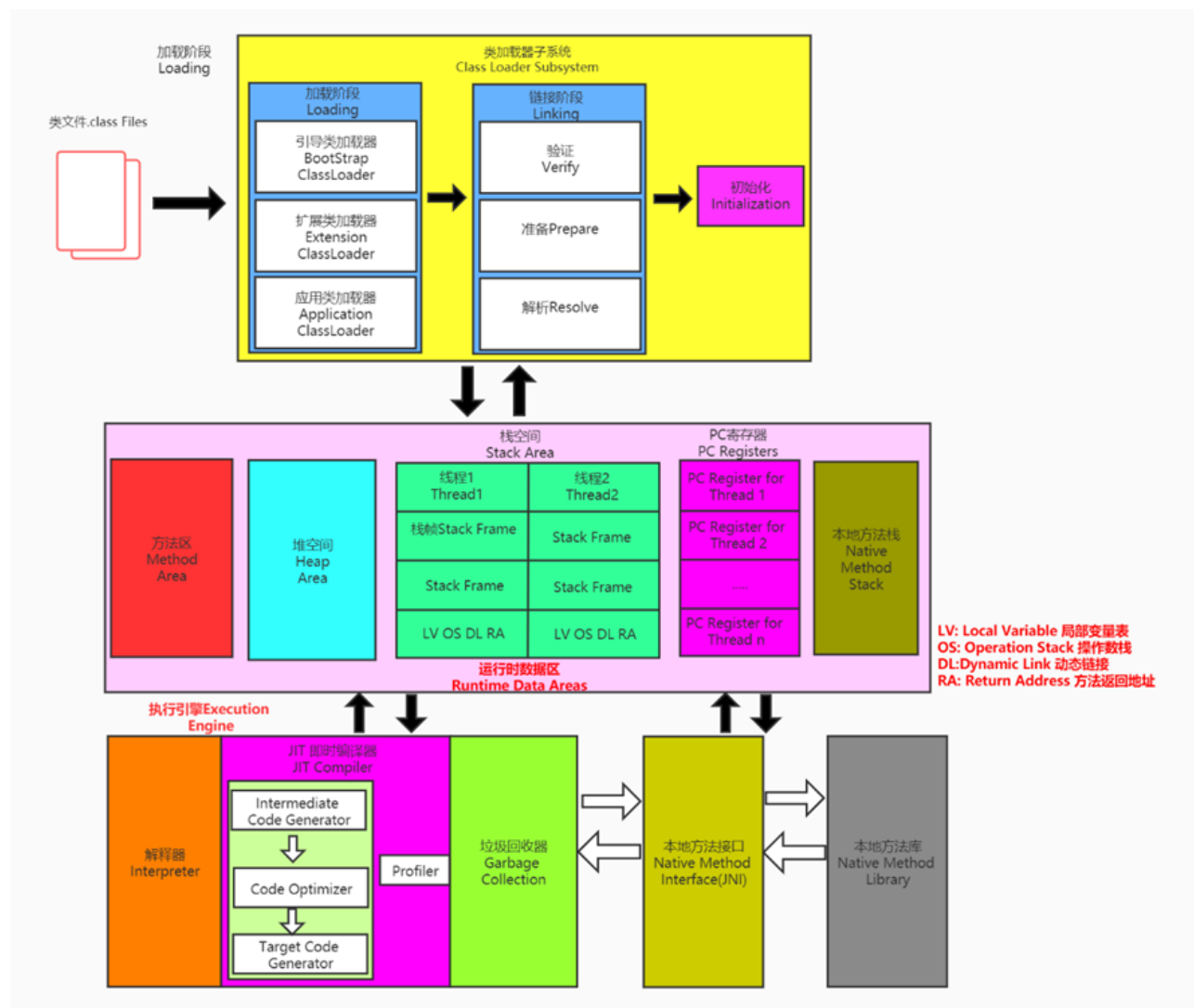


# 类加载子系统



内存结构

构建一个JVM，主要是需要类加载器与执行引擎

- 类加载器子系统负责从文件系统或者网络中加载Class文件，class文件在文件开头有特定的文件标识。(cafe babe)
- ClassLoader只负责class文件的加载，至于它是否可以运行，则由Execution Engine决定。
- 加载的类信息存放于一块称为**方法区**的内存空间。除了类的信息外，方法区中还会存放**运行时常量池信息**，可能还包括**字符串字面量和数字常量**（这部分常量信息是Class文件中常量池部分的内存映射）

运行时的常量池

反编译: javap -v test.class

```
1  Classfile /Users/yangshu/Project/learn_myself/java源码解析/java8_demo/target/classes/out/out/demo/one/test.class
2
3  Last modified 2020-11-4; size 439 bytes
4  MD5 checksum 874d3e1dc2e73eee14a744214b28ad87
5  Compiled from "test.java"
6  public class demo.one.test
7      minor version: 0
8      major version: 52
9      flags: ACC_PUBLIC, ACC_SUPER
10 Constant pool:
11     #1 = Methodref          #3.#21          // java/lang/Object."<init>":()
12     V
13     #2 = Class              #22              // demo/one/test
14     #3 = Class              #23              // java/lang/Object
15     #4 = Utf8               <init>
16     #5 = Utf8               ()V
17     #6 = Utf8               Code
18     #7 = Utf8               LineNumberTable
19     #8 = Utf8               LocalVariableTable
20     #9 = Utf8               this
21     #10 = Utf8              Ldemo/one/test;
22     #11 = Utf8              main
23     #12 = Utf8              ([Ljava/lang/String;)V
24     #13 = Utf8              args
25     #14 = Utf8              [Ljava/lang/String;
26     #15 = Utf8              a
27     #16 = Utf8              I
28     #17 = Utf8              b
29     #18 = Utf8              k
30     #19 = Utf8              SourceFile
31     #20 = Utf8              test.java
32     #21 = NameAndType        #4:#5          // "<init>":()V
33     #22 = Utf8              demo/one/test
34     #23 = Utf8              java/lang/Object
35 {
36     public demo.one.test();
37         descriptor: ()V
38         flags: ACC_PUBLIC
39         Code:
40             stack=1, locals=1, args_size=1
41             0: aload_0
```

```

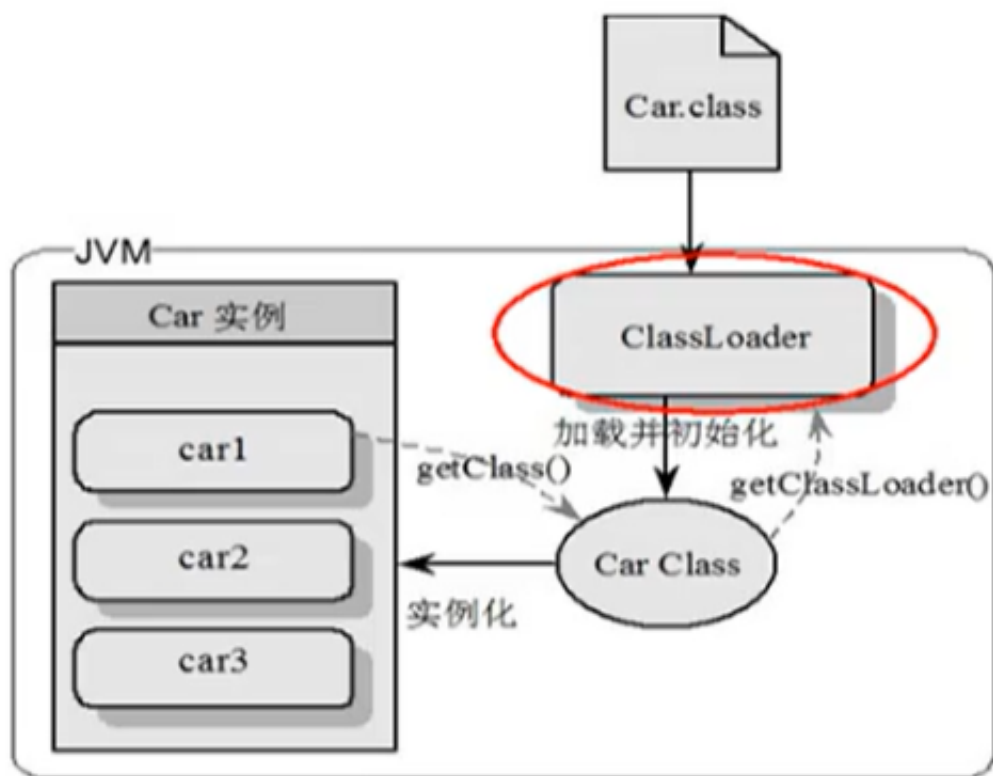
42         1: invokespecial #1                                // Method java/lang/Object.
43 "<init>":()V
44         4: return
45     LineNumberTable:
46         line 7: 0
47     LocalVariableTable:
48         Start Length Slot Name Signature
49         0      5      0 this Ldemo/one/test;
50
51     public static void main(java.lang.String[]);
52     descriptor: ([Ljava/lang/String;)V
53     flags: ACC_PUBLIC, ACC_STATIC
54     Code:
55         stack=2, locals=4, args_size=1
56         0: iconst_1
57         1: istore_1
58         2: iconst_2
59         3: istore_2
60         4: iload_1
61         5: iload_2
62         6: iadd
63         7: istore_3
64         8: return
65     LineNumberTable:
66         line 9: 0
67         line 10: 2
68         line 11: 4
69         line 12: 8
70     LocalVariableTable:
71         Start Length Slot Name Signature
72         0      9      0 args [Ljava/lang/String;
73         2      7      1 a I
74         4      5      2 b I
75         8      1      3 k I
76     }
77     SourceFile: "test.java"

```

## ClassLoader

- class file存在于本地硬盘上，可以理解为设计师画在纸上的模板，而最终这个模板在执行的时候是要加载到JVM当中来根据这个文件实例化出n个一模一样的实例。

- class file加载到JVM中，被称为DNA元数据模板，放在方法区。
- 在.class文件->JVM->最终成为元数据模板，此过程就要一个运输工具（类装载机Class Loader），扮演一个快递员的角色。

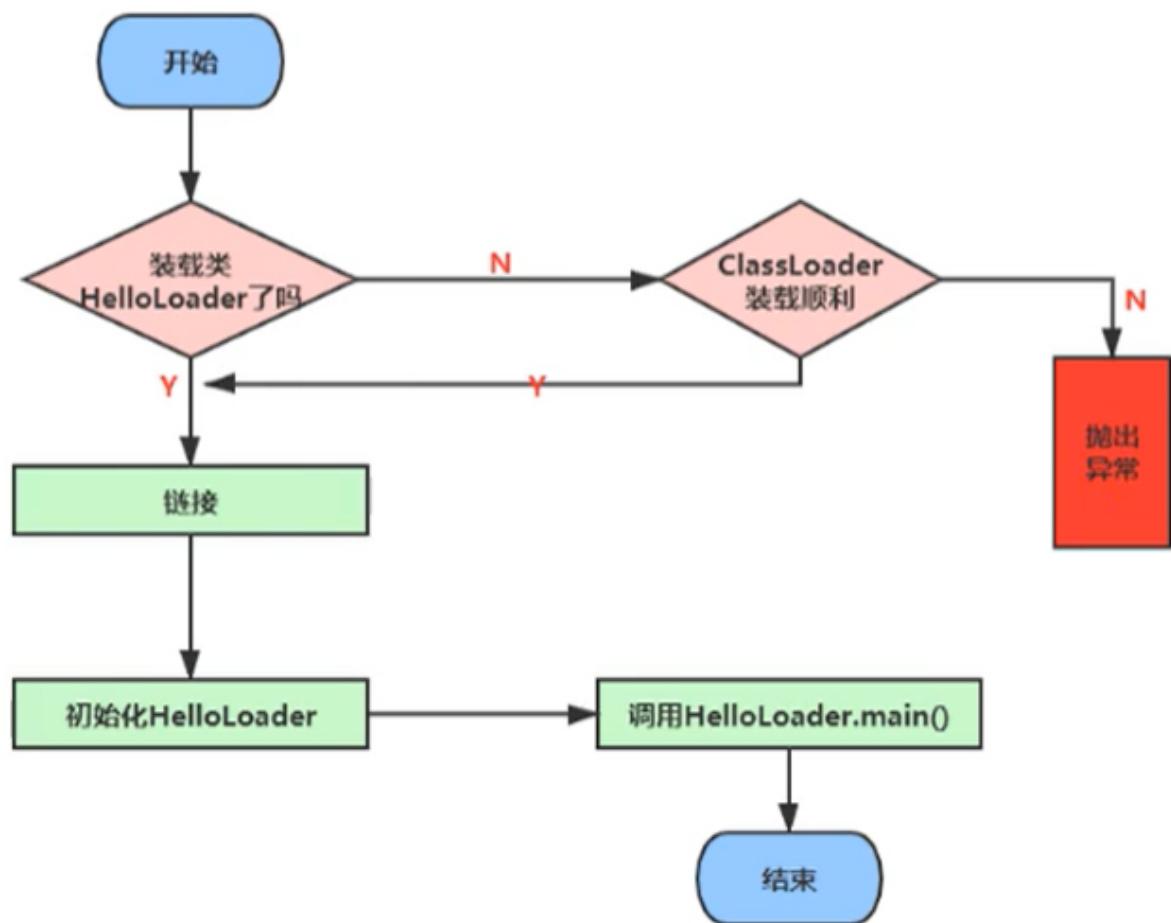


类加载的过程

## 类加载的过程

```
1 public class HelloLoader {
2     public static void main(String[] args) {
3         System.out.println("加载中.....");
4     }
5 }
6 }
```

- 在执行 main 方法之前，首先需要加载 main 方法的承载类 HelloLoader
- 加载成功后，在进行链接、初始化的操作，完成后调用 HelloLoader 中的main方法
- 加载失败则运行失败，会抛出异常



运行main的流程

完整的流程为：

加载 -> 链接(验证 -> 准备 -> 解析) -> 初始化

## 加载(loading)



- 通过一个类的**全限定名**获取定义此类的二进制字节流
  - 区分类的两个标志
    - 1. 全限类名是不是相等的
    - 2. 使用的类加载器是不是同一个
- 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

加载class文件的方式

1. 从本地系统中直接加载

2. 通过网络获取，典型场景：Web Applet
3. 从zip压缩包中读取，成为日后jar、war格式的基础
4. 运行时计算生成，使用最多的是：动态代理技术
5. 由其他文件生成，典型场景：JSP应用从专有数据库中提取.class文件，比较少见
6. 从加密文件中获取，典型的防Class文件被反编译的保护措施（需要用户自定义ClassLoader类，进行文件的解密）

## 链接(linking)

 类的加载过程 

链接

**验证(Verify) :**

- 目的在于确保Class文件的字节流中包含信息符合当前虚拟机要求，保证被加载类的正确性，不会危害虚拟机自身安全。
- 主要包括四种验证，文件格式验证，元数据验证，字节码验证，符号引用验证。

**准备(Prepare) :**

- 为类变量分配内存并且设置该类变量的默认初始值，即零值。
- 这里不包含用final修饰的static，因为final在编译的时候就会分配了，准备阶段会显式初始化；
- 这里不会为实例变量分配初始化，类变量会分配在方法区中，而实例变量是会随着对象一起分配到Java堆中。

**解析(Resolve) :**

- 将常量池内的符号引用转换为直接引用的过程。
- 事实上，解析操作往往会伴随着JVM在执行完初始化之后再执行。
- 符号引用就是一组符号来描述所引用的目标。符号引用的字面量形式明确定义在《java虚拟机规范》的Class文件格式中。直接引用就是直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄。
- 解析动作主要针对类或接口、字段、类方法、接口方法、方法类型等。对应常量池中的CONSTANT\_Class\_info、CONSTANT\_Fieldref\_info、CONSTANT\_Methodref\_info等。

### 链接

## 验证

- 目的在于确保Class文件的字节流中包含信息符合当前虚拟机要求，保证被加载类的正确性，不会危害虚拟机自身安全
- 主要包括四种验证，文件格式验证，元数据验证，字节码验证，符号引用验证。

字节码文件开头都是 CAFE BABE

## 准备（在虚拟机栈中有相应的讲解）

- 为类变量分配内存并且设置该类变量的默认初始值，即零值（所以在static中，没有声明的对

象可以先进行赋值，不会报错)

- 这里不包含用final修饰的static，因为final在编译的时候就会分配好了默认值，准备阶段会显式初始化
- 注意：这里不会为实例变量分配初始化，类变量会分配在方法区中，而实例变量是会随着对象一起分配到Java堆中

```
1 public class HelloApp {
2     private static int a = 1;    //prepare: a = 0 ---> initial : a = 1
3
4     public static void main(String[] args) {
5         System.out.println(a);
6     }
7 }
```

## 解析

在反编译之后可以查看符号引用

```
1 Constant pool:
2 #1 = Methodref          #3.#21          // java/lang/Object."<init>":()V
3 #2 = Class               #22              // demo/one/test
4 #3 = Class               #23              // java/lang/Object
5 #4 = Utf8                <init>
6 #5 = Utf8                ()V
7 #6 = Utf8                Code
8 #7 = Utf8                LineNumberTable
9 #8 = Utf8                LocalVariableTable
10 #9 = Utf8                this
11 #10 = Utf8               Ldemo/one/test;
12 #11 = Utf8               main
13 #12 = Utf8               ([Ljava/lang/String;)V
14 #13 = Utf8               args
15 #14 = Utf8               [Ljava/lang/String;
16 #15 = Utf8               a
17 #16 = Utf8               I
18 #17 = Utf8               b
19 #18 = Utf8               k
20 #19 = Utf8               SourceFile
21 #20 = Utf8               test.java
22 #21 = NameAndType        #4:#5           // "<init>":()V
23 #22 = Utf8               demo/one/test
```

## 初始化

- 初始化阶段就是执行类构造器方法`clinit()`的过程
- 此方法不需定义，是 `javac` 编译器自动收集类中的所有类变量的赋值动作和静态代码块中的语句合并而来。也就是说，当我们代码中包含`static`变量的时候，就会有`clinit`方法
- `<clinit>()` 方法中的指令按语句在源文件中出现的顺序执行
- `<clinit>()` 不同于类的构造器。（关联：构造器是虚拟机视角下的 `<init>()`，类会有默认的构造器）
- 若该类具有父类，JVM会保证子类的`clinit()`执行前，父类的`clinit()`已经执行完毕（父类的类构造器会在子类之前被执行）
- 虚拟机必须保证一个类的 `<clinit>()` 方法在多线程下被同步加锁（在同一个时间内，只能被不同的线程访问一次）

只有在类中包含了`static`变量时，才会进行 `clinit()`

```
1 public class ClassInitTest {
2     private static int num = 1;
3     private static int number = 10;    //linking之prepare: number = 0 -
4     -> initial: 10 --> 20
5
6     static {
7         num = 2;
8         number = 20;
9         System.out.println(num);
10        System.out.println(number);
11    }
12
13    public static void main(String[] args) {
14        System.out.println(ClassInitTest.num); //2
15        System.out.println(ClassInitTest.number); //20
16    }
17 }
```



jclasslib: ClassInitTest.java x

一般信息

属性名索引: [cp\\_info #13](#) <Code>  
属性长度: 85

特有信息

字节码 异常表 杂项

```
1 0 iconst_1
2 1 putstatic #3 <demo/JVM/ClassInitTest.nu
3 4 bipush 10
4 6 putstatic #5 <demo/JVM/ClassInitTest.nu
5 9 iconst_2
6 10 putstatic #3 <demo/JVM/ClassInitTest.nu
7 13 bipush 20
8 15 putstatic #5 <demo/JVM/ClassInitTest.nu
9 18 getstatic #2 <java/lang/System.out>
10 21 getstatic #3 <demo/JVM/ClassInitTest.nu
11 24 invokevirtual #4 <java/io/PrintStream.p
12 27 getstatic #2 <java/lang/System.out>
13 30 getstatic #5 <demo/JVM/ClassInitTest.nu
14 33 invokevirtual #4 <java/io/PrintStream.p
15 36 return
```

在prepare的时候就会先进行初始化，与static执行的顺序没有关系

```
1 public class ClassInitTest {
2     private static int num = 1;
3
4     static{
5         num = 2;
```

```

6         number = 20;
7         System.out.println(num);
8         //System.out.println(number);    //报错：非法的前向引用（可以赋值，但不
9 能调用）
10    }
11
12    private static int number = 10;    //linking之prepare: number = 0 --
13    > initial: 20 --> 10
14
15    public static void main(String[] args) {
16        System.out.println(ClassInitTest.num);    //2
17        System.out.println(ClassInitTest.number);    //10
18    }
19 }

```

虚拟机必须保证一个类的()方法在多线程下被同步加锁

```

1  public class DeadThreadTest {
2      public static void main(String[] args) {
3          Runnable r = () -> {
4              System.out.println(Thread.currentThread().getName() + "开始")
5          };
6          DeadThread dead = new DeadThread();
7          System.out.println(Thread.currentThread().getName() + "结束")
8          ;
9          };
10
11         Thread t1 = new Thread(r, "线程1");
12         Thread t2 = new Thread(r, "线程2");
13
14         t1.start();
15         t2.start();
16     }
17 }
18
19 class DeadThread {
20     static {
21         if (true) {
22             System.out.println(Thread.currentThread().getName() + "初始化
23 当前类");
24             while (true) {
25
26

```

```
}  
}  
}
```

程序卡死，分析原因：

- 两个线程同时去加载 `DeadThread` 类，而 `DeadThread` 类中静态代码块中有一处死循环
- 先加载 `DeadThread` 类的线程抢到了同步锁，然后在类的静态代码块中执行死循环，而另一个线程在等待同步锁的释放
- 所以无论哪个线程先执行 `DeadThread` 类的加载，另外一个类也不会继续执行

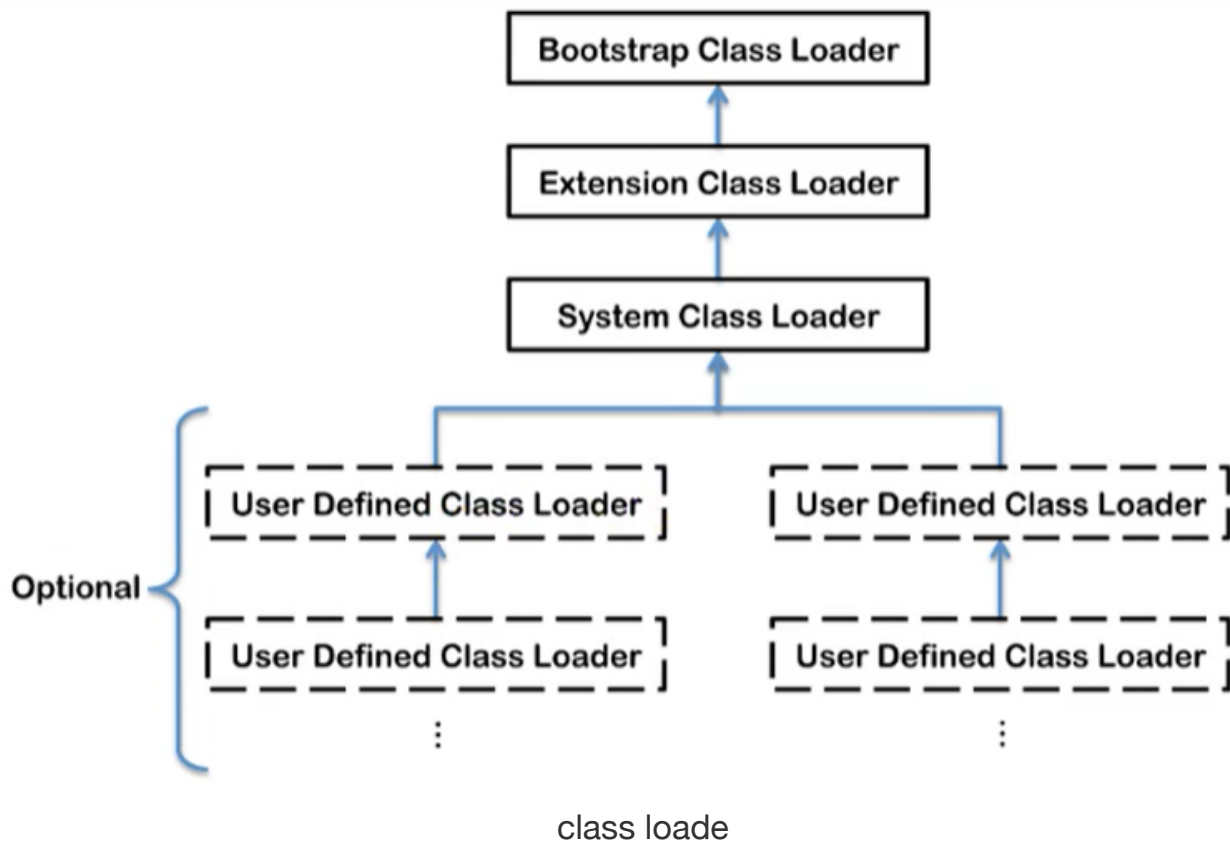
```
1 | 线程1开始  
2 | 线程2开始  
3 | 线程1初始化当前类
```

## 不同的类加载器

类加载器从大类看可以分为两类：

- 引导类加载器（Bootstrap ClassLoader）是 `native` 方法，底层由C/C++实现
- 自定义加载器（User-Defined ClassLoader）继承于 `ClassLoader` 接口，使用java进行编写，可以用户进行自定义
  - 将所有派生于抽象类 `ClassLoader` 的类加载器都划分为自定义类加载器

常见的类加载器一般有三种：



上面类加载器的关系是包含关系，不是上下级的关系，也不是继承的关系，类似于文件夹的形式（例如：a/b/c）

AppClassLoader -> ExtClassLoader -> BootstrapClassLoader

AppClassLoader 与 ExtClassLoader 两个类加载器都是 `sun.misc.Launcher` 的内部类

```
1 public class ClassLoaderTest {
2     public static void main(String[] args) {
3
4         //获取系统类加载器
5         ClassLoader systemClassLoader = ClassLoader.getSystemClassLoader(
6     );
7         System.out.println(systemClassLoader); //sun.misc.Launcher$AppClass
8 loader@18b4aac2
9
10        //获取其上层：扩展类加载器
11        ClassLoader extClassLoader = systemClassLoader.getParent();
12        System.out.println(extClassLoader); //sun.misc.Launcher$ExtClassLo
13 ader@1540e19d
14    }
```

```

15 //获取其上层：获取不到引导类加载器
16 ClassLoader bootstrapClassLoader = extClassLoader.getParent();
17 System.out.println(bootstrapClassLoader);//null
18
19 //对于用户自定义类来说：默认使用系统类加载器进行加载
20 ClassLoader classLoader = ClassLoaderTest.class.getClassLoader();
21 System.out.println(classLoader);//sun.misc.Launcher$AppClassLoader@18b4aac2
22
23
24 //String类使用引导类加载器进行加载的。---> Java的核心类库都是使用引导类加载器进行加载的。
25 ClassLoader classLoader1 = String.class.getClassLoader();
    System.out.println(classLoader1);//null
}

```

- 尝试获取引导类加载器，获取到的值为 null，这并不代表引导类加载器不存在，因为引导类加载器是 C/C++ 语言，我们获取不到
- 两次获取系统类加载器的值都相同： `sun.misc.Launcher$AppClassLoader@18b4aac2`，这说明系统类加载器是全局唯一的
- Java的核心类库都是使用引导类加载器进行加载的。(String)

## 启动类加载器 (BootstrapClassLoader)

- 这个类加载器使用 C/C++ 语言实现的，嵌套在 JVM 内部
- 它用来加载 Java 的核心库（ `JAVA_HOME/jre/lib/rt.jar`、`resources.jar` 或 `sun.boot.class.path` 路径下的内容），用于提供 JVM 自身需要的类
- 并不继承自 `java.lang.ClassLoader`，没有父加载器
- 加载扩展类 and 应用程序类加载器，并作为他们的父类加载器
- 出于安全考虑，Bootstrap 启动类加载器只加载包名为 `java`、`javax`、`sun` 等开头的类

## 扩展类加载器 (Extension ClassLoader)

- Java 语言编写，由 `sun.misc.Launcher$ExtClassLoader` 实现
- 派生于 `ClassLoader` 类
- 父类加载器为启动类加载器
- 从 `java.ext.dirs` 系统属性所指定的目录中加载类库，或从 JDK 的安装目录的 `jre/lib/ext` 子目录（扩展目录）下加载类库。如果用户创建的 JAR 放在此目录下，也会

## 系统类加载器（AppClassLoader）

- Java语言编写，由 `sun.misc.LaunchersAppClassLoader` 实现
- 派生于 `ClassLoader` 类
- 父类加载器为扩展类加载器
- 它负责加载环境变量 `classpath` 或系统属性 `java.class.path` 指定路径下的类库
- 该类加载是程序中默认类加载器，一般来说，Java应用的类都是由它来完成加载
- 通过 `ClassLoader.getSystemClassLoader()` 方法可以获取到该类加载器

```

1  public class ClassLoaderTest1 {
2      public static void main(String[] args) {
3
4          System.out.println("*****启动类加载器*****");
5          //获取BootstrapClassLoader能够加载的api的路径
6          URL[] urLs = sun.misc.Launcher.getBootstrapClassPath().getURLs();
7          for (URL element : urLs) {
8              System.out.println(element.toExternalForm());
9          }
10         //从上面的路径中随意选择一个类,来看看他的类加载器是什么:引导类加载器
11         ClassLoader classLoader = Provider.class.getClassLoader();
12         System.out.println(classLoader);//null
13
14         System.out.println("*****扩展类加载器*****");
15         String extDirs = System.getProperty("java.ext.dirs");
16         for (String path : extDirs.split(";")) {
17             System.out.println(path);
18         }
19
20         //从上面的路径中随意选择一个类,来看看他的类加载器是什么:扩展类加载器
21         ClassLoader classLoader1 = CurveDB.class.getClassLoader();
22         System.out.println(classLoader1);//sun.misc.Launcher$ExtClassLoad
23         er@1540e19d
24     }
25 }

```

运行结果

```
1 *****启动类加载器*****
2 file:/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/jre
3 /lib/resources.jar
4 file:/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/jre
5 /lib/rt.jar
6 file:/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/jre
7 /lib/sunrsasign.jar
8 file:/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/jre
9 /lib/jsse.jar
10 file:/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/jre
11 /lib/jce.jar
12 file:/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/jre
13 /lib/charsets.jar
    file:/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/jre
    /lib/jfr.jar
    file:/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/jre
    /classes
    null
    *****扩展类加载器*****
    /Users/yangshu/Library/Java/Extensions:/Library/Java/JavaVirtualMachines/
    jdk1.8.0_181.jdk/Contents/Home/jre/lib/ext:/Library/Java/Extensions:/Netw
    ork/Library/Java/Extensions:/System/Library/Java/Extensions:/usr/lib/java
    sun.misc.Launcher$ExtClassLoader@3b9a45b3
```

## 用户自定义类加载器

### 为什么要去定义

在Java的日常应用程序开发中，类的加载几乎是由上述3种类加载器相互配合执行的，在必要时，我们还可以自定义类加载器，来定制类的加载方式。那为什么还需要自定义类加载器？

- 隔离加载类
- 修改类加载的方式
- 扩展加载源
- 防止源码泄漏

### 如何自定义

- 可以通过继承抽象类 `java.lang.ClassLoader` 类的方式，实现自己的类加载器，以满足一些特殊的需求
- 在JDK1.2之前，在自定义类加载器时，总会去继承`ClassLoader`类并重写 `loadClass()` 方法，从而实现自定义的类加载类，但是在JDK1.2之后已不再建议用户去覆盖 `loadClass()` 方法，而是建议把自定义的类加载逻辑写在 `findclass()` 方法中
- 在编写自定义类加载器时，如果没有太过于复杂的需求，可以直接继承 `URIClassLoader` 类，这样就可以避免自己去编写 `findclass()` 方法及其获取字节码流的方式，使自定义类加载器编写更加简洁。

```
1 public class CustomClassLoader extends ClassLoader {
2     @Override
3     protected Class<?> findClass(String name) throws ClassNotFoundException
4 on {
5
6         try {
7             byte[] result = getClassFromCustomPath(name);
8             if (result == null) {
9                 throw new FileNotFoundException();
10            } else {
11                return defineClass(name, result, 0, result.length);
12            }
13        } catch (FileNotFoundException e) {
14            e.printStackTrace();
15        }
16
17        throw new ClassNotFoundException(name);
18    }
19
20    private byte[] getClassFromCustomPath(String name) {
21        //从自定义路径中加载指定类:细节略
22        //如果指定路径的字节码文件进行了加密，则需要在此方法中进行解密操作。
23        return null;
24    }
25
26    public static void main(String[] args) {
27        CustomClassLoader customClassLoader = new CustomClassLoader();
28        try {
29            Class<?> clazz = Class.forName("One", true, customClassLoader
30 );
31            Object obj = clazz.newInstance();
32            System.out.println(obj.getClass().getClassLoader());
```

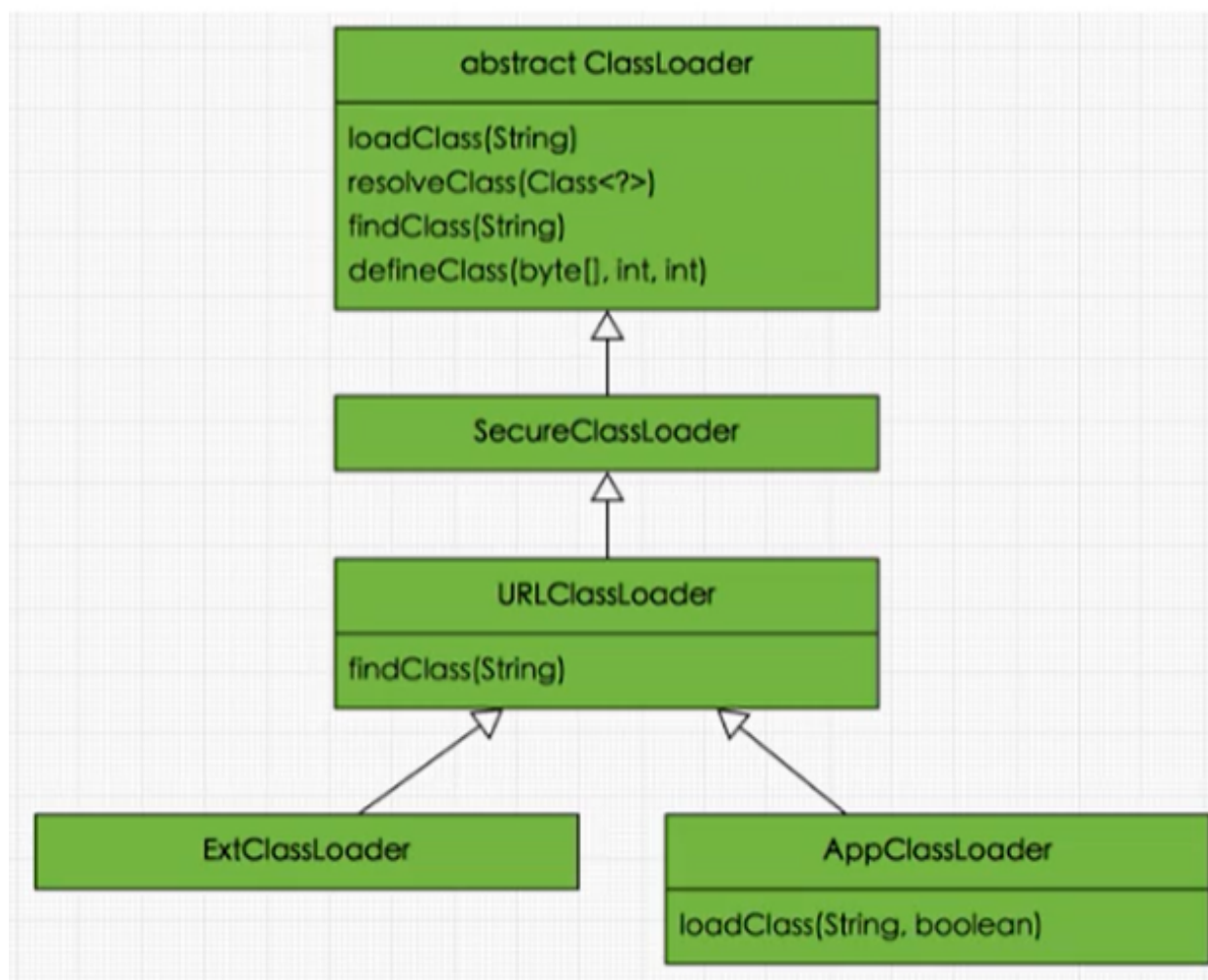


```

33         } catch (Exception e) {
34             e.printStackTrace();
35         }
    }
}

```

`sun.misc.Launcher` 它是一个java虚拟机的入口应用



## 获取ClassLoader途径

方式一：获取当前类的ClassLoader

```
clazz.getClassLoader()
```

方式二：获取当前线程上下文的ClassLoader

```
Thread.currentThread().getContextClassLoader()
```

方式三：获取系统的ClassLoader

```
ClassLoader.getSystemClassLoader()
```

方式四：获取调用者的ClassLoader

```
DriverManager.getCallerClassLoader()
```

获取ClassLoader途径

```
1 public class ClassLoaderTest2 {
2     public static void main(String[] args) {
3         try {
4
5             //1. Class.forName().getClassLoader()
6             ClassLoader classLoader = Class.forName("java.lang.String").g
7 etClassLoader();
8             System.out.println(classLoader); // String 类由启动类加载器加载
9 , 我们无法获取
10
11             //2. Thread.currentThread().getContextClassLoader()
12             ClassLoader classLoader1 = Thread.currentThread().getContextC
13 lassLoader();
14             System.out.println(classLoader1);
15
16             //3. ClassLoader.getSystemClassLoader().getParent()
17             ClassLoader classLoader2 = ClassLoader.getSystemClassLoader()
18 ;
19             System.out.println(classLoader2);
20
21         } catch (ClassNotFoundException e) {
22             e.printStackTrace();
23         }
24     }
25 }
```

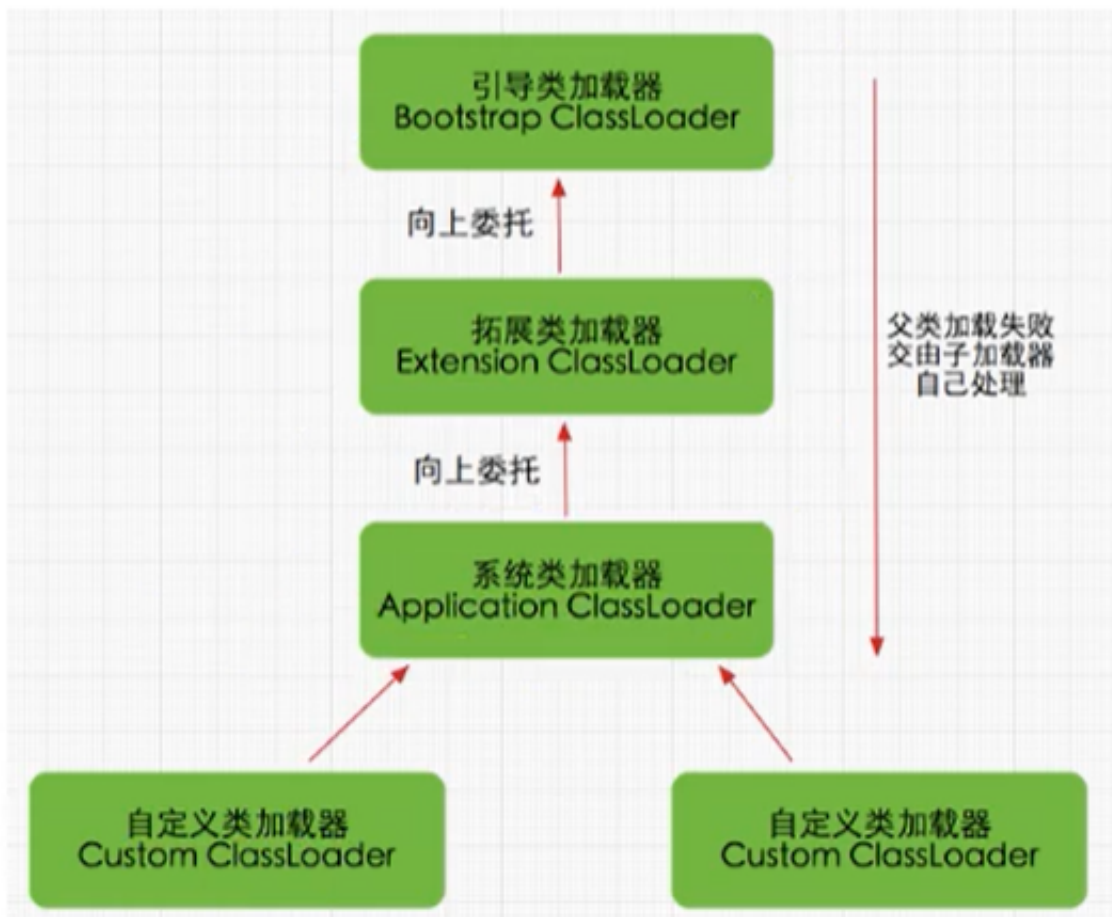
```
// 输出  
null  
sun.misc.Launcher$AppClassLoader@18b4aac2  
sun.misc.Launcher$AppClassLoader@18b4aac2
```

## 双亲委派机制

### 原理

Java虚拟机对class文件采用的是**按需加载**的方式，也就是说**当需要使用该类时才会将它的class文件加载到内存生成class对象**。而且加载某个类的class文件时，Java虚拟机采用的是**双亲委派模式**，即把请求交由父类处理，它是一种任务委派模式

- 如果一个类加载器收到了类加载请求，它并不会自己先去加载，而是把这个请求委托给父类的加载器去执行；
- 如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，请求最终将**到达顶层的启动类加载器**；
- 如果父类加载器可以完成类加载任务，就成功返回，倘若父类加载器无法完成此加载任务，**子加载器才会尝试自己去加载，这就是双亲委派模式**。
- 父类加载器一层一层往下分配任务，如果子类加载器能加载，则加载此类，如果将加载任务分配至系统类加载器也无法加载此类，则抛出异常



双亲委派机制

## 举例

1

建立一个 `java.lang.String` 类，写上 `static` 代码块

```
1 package java.lang;
2
3 public class String {
4     static{
5         System.out.println("我是自定义的String类的静态代码块");
6     }
7 }
```

另外的程序中加载 `String` 类

```

1 public class StringTest {
2
3     public static void main(String[] args) {
4         java.lang.String str = new java.lang.String();
5         System.out.println("hello,atguigu.com");
6
7         StringTest test = new StringTest();
8         System.out.println(test.getClass().getClassLoader());
9     }
10 }

```

程序并没有输出我们静态代码块中的内容，可见仍然加载的是 JDK 自带的 String 类，找到了最上层的启动类加载器 `BootstrapClassLoader`，加载自带的 String 类

## 2

自定义的 String 中添加 main 方法

```

1 package java.lang;
2
3 public class String {
4     static{
5         System.out.println("我是自定义的String类的静态代码块");
6     }
7     //错误：在类 java.lang.String 中找不到 main 方法
8     public static void main(String[] args) {
9         System.out.println("hello,String");
10    }
11 }

```

由于双亲委派机制找到的是 JDK 自带的 String 类，在那个 String 类中并没有 main() 方法，会抛出异常

```

1 错误：在类 java.lang.String 中找不到 main 方法，请将 main 方法定义为：
2     public static void main(String[] args)
3 否则 JavaFX 应用程序类必须扩展javafx.application.Application

```

## 3

在 `java.lang` 包下自定义类

```

1 package java.lang;
2
3 public class ShkStart {
4     public static void main(String[] args) {
5         System.out.println("hello!");
6     }
7 }

```

出于**保护机制**，java.lang 包下不允许我们自定义类

```

1 Error: A JNI error has occurred, please check your installation and try a
2 gain
3 Exception in thread "main" java.lang.SecurityException: Prohibited packag
4 e name: java.lang
5     at java.lang.ClassLoader.preDefineClass(ClassLoader.java:662)
6     at java.lang.ClassLoader.defineClass(ClassLoader.java:761)
7     at java.security.SecureClassLoader.defineClass(SecureClassLoader.java
8 :142)
9     at java.net.URLClassLoader.defineClass(URLClassLoader.java:467)
10    at java.net.URLClassLoader.access$100(URLClassLoader.java:73)
11    at java.net.URLClassLoader$1.run(URLClassLoader.java:368)
12    at java.net.URLClassLoader$1.run(URLClassLoader.java:362)
13    at java.security.AccessController.doPrivileged(Native Method)
14    at java.net.URLClassLoader.findClass(URLClassLoader.java:361)
15    at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
16    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:349)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
    at sun.launcher.LauncherHelper.checkAndLoadMain(LauncherHelper.java:4
95)

```

## 4

当加载jdbc.jar 用于实现数据库连接的时候

- jdbc.jar是基于SPI接口进行实现的
- 所以在加载的时候，会进行双亲委派，最终从根加载器中加载 SPI核心类，然后再加载SPI接口类
- 接着在进行反向委托，通过线程上下文类加载器进行实现类 jdbc.jar的加载。

## 双亲委派机制的优势

- 避免类的重复加载
- 保护程序安全，防止核心API被随意篡改
  - 自定义类：`java.lang.String` 不会进行加载
  - 自定义类：`java.lang.ShkStart`（报错：阻止创建 `java.lang` 开头的类）

## 沙箱安全机制

---

- 自定义String类时：在加载自定义String类的时候会率先使用引导类加载器加载，而引导类加载器在加载的过程中会先加载jdk自带的文件（rt.jar包中java.lang.String.class），报错信息说没有main方法，就是因为加载的是rt.jar包中的String类。
- 这样可以保证对java核心源代码的保护，这就是沙箱安全机制。

## 如何判断两个class对象是否相同？

---

在JVM中表示两个class对象是否为同一个类存在两个必要条件：

- 类的完整类名必须一致，包括包名（全限定名相等）
- 加载这个类的ClassLoader（指ClassLoader实例对象）必须相同
  - 换句话说，在JVM中，即使这两个类对象（class对象）来源同一个Class文件，被同一个虚拟机所加载，但只要加载它们的ClassLoader实例对象不同，那么这两个类对象也是不相等的

## 对类加载器的引用（引用信息保存在方法区中）

---

- JVM必须知道一个类型是由启动加载器加载的还是由用户类加载器加载的
- 如果一个类型是由用户类加载器加载的，那么JVM会将这个类加载器的一个引用作为类型信息的一部分保存在方法区中
- 当解析一个类型到另一个类型的引用的时候，JVM需要保证这两个类型的类加载器是相同的

## 类的主动使用和被动使用

---

Java程序对类的使用方式分为：主动使用和被动使用。主动使用，又分为七种情况：

1. 创建类的实例
2. 访问某个类或接口的静态变量，或者对该静态变量赋值
3. 调用类的静态方法
4. 反射（比如：Class.forName("java.lang.String")）
5. 初始化一个类的子类
6. Java虚拟机启动时被标明为启动类的类
7. JDK7开始提供的动态语言支持：java.lang.invoke.MethodHandle实例的解析结果  
REF\_getStatic、REF putStatic、REF\_invokeStatic句柄对应的类没有初始化，则初始化

除了以上七种情况，其他使用Java类的方式都被看作是对类的被动使用，都不会导致类的初始化，即不会执行初始化阶段（不会调用 clinit() 方法和 init() 方法）