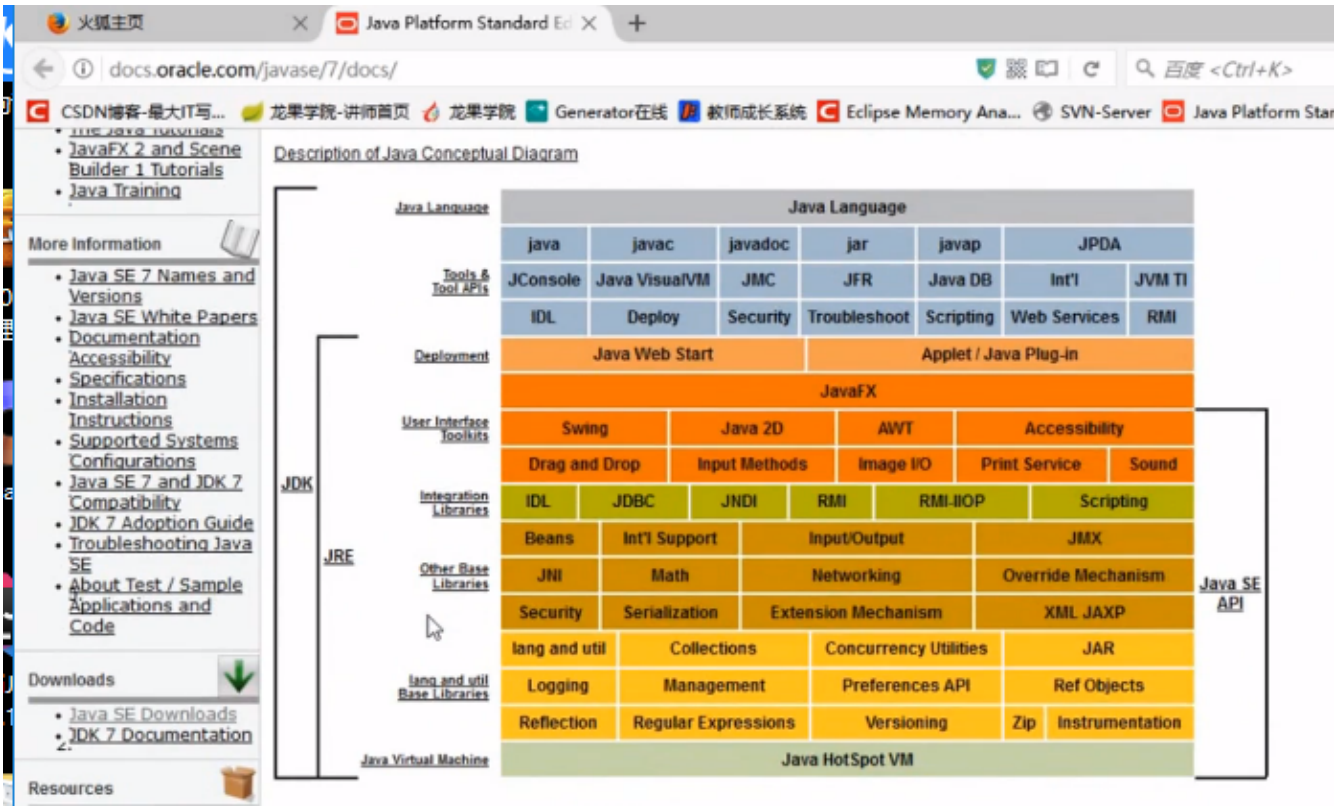
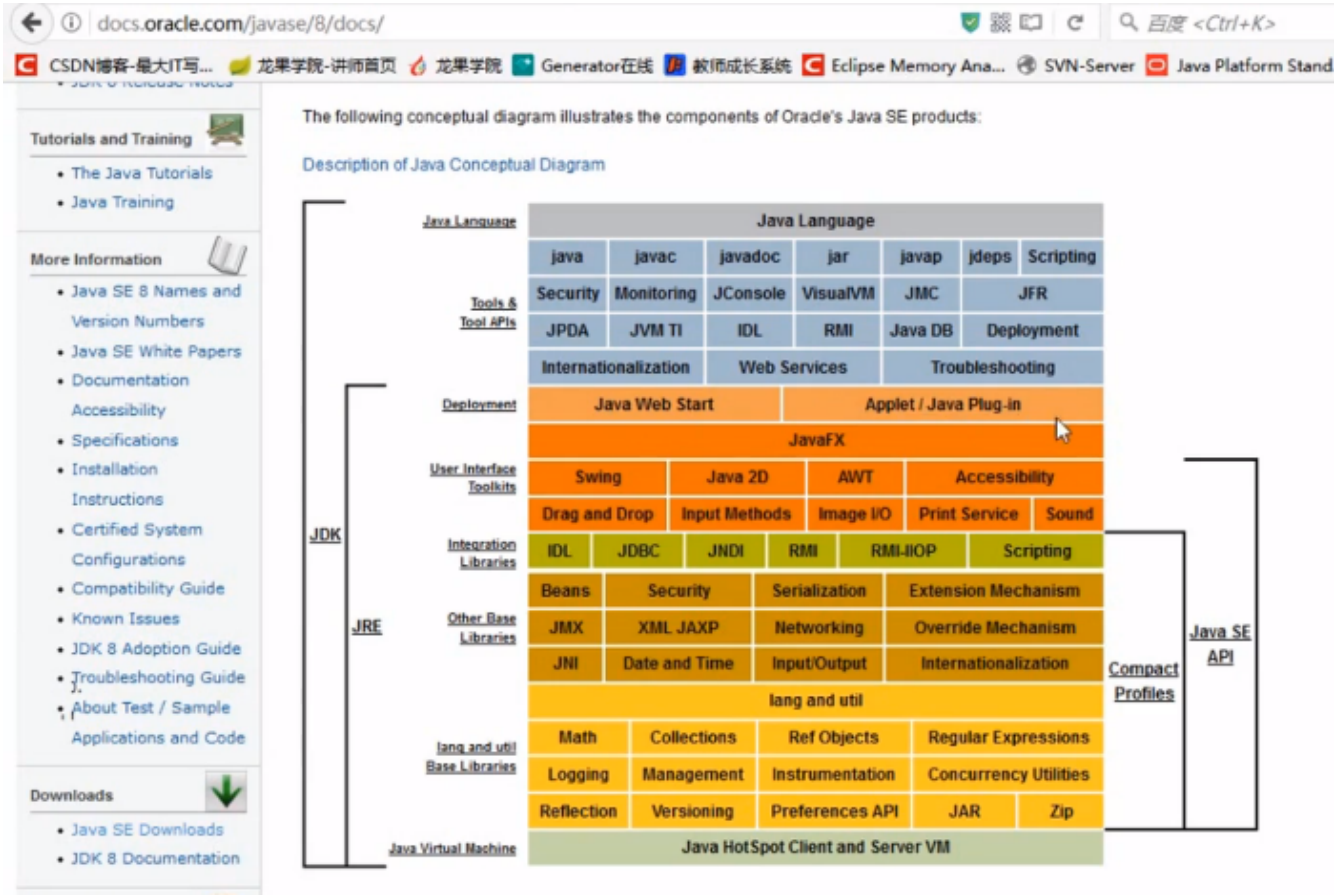
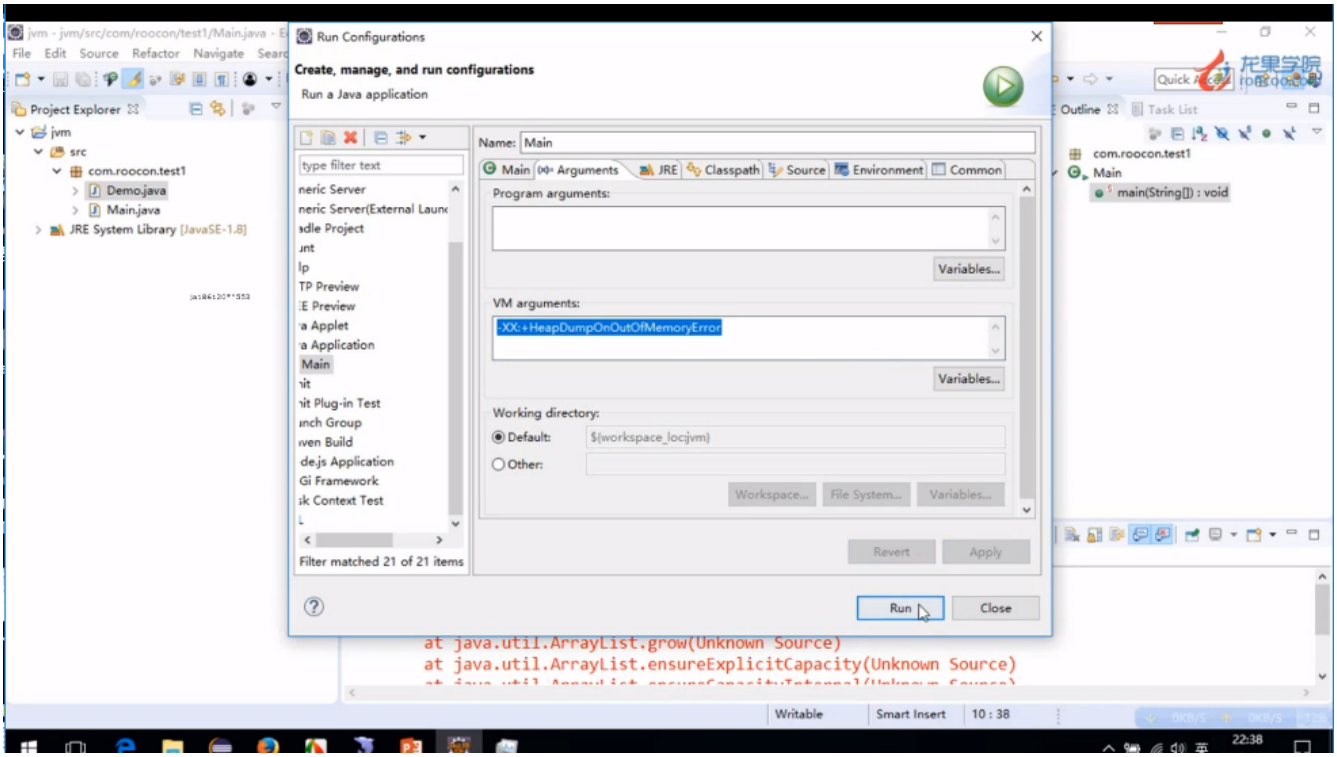
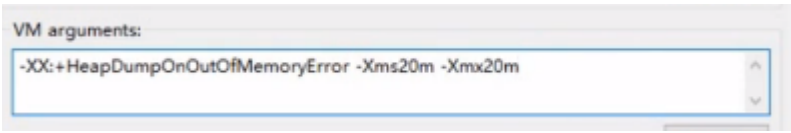

docs.oracle.com/javase/7/docs/
JavaFX 2 and Scene Builder 1 Tutorials
Java Training
More Information
Java SE 7 Names and Versions
Java SE White Papers
Documentation Accessibility
Specifications
Installation Instructions
Supported Systems Configurations
Java SE 7 and JDK 7 Compatibility
JDK 7 Adoption Guide
Troubleshooting Java SE
About Test / Sample Applications and Code
Downloads
Java SE Downloads
JDK 7 Documentation
Resources
Description of Java Conceptual Diagram
Java Language
Java Language
java
javac
javadoc
jar
javap
JPDA
Tools & Tool APIs
JConsole
Java VisualVM
JMC
JFR
Java DB
Int'l
JVM TI
IDL
Deploy
Security
Troubleshoot
Scripting
Web Services
RMI
Deployment
Java Web Start
Applet / Java Plug-in
JavaFX
User Interface Toolkits
Swing
Java 2D
AWT
Accessibility
Drag and Drop
Input Methods
Image I/O
Print Service
Sound
Integration Libraries
IDL
JDBC
JNDI
RMI
RMI-IIOP
Scripting
Other Base Libraries
Beans
Int'l Support
Input/Output
JMX
JNI
Math
Networking
Override Mechanism
Security
Serialization
Extension Mechanism
XML JAXP
lang and util Base Libraries
lang and util
Collections
Concurrency Utilities
JAR
Logging
Management
Preferences API
Ref Objects
Reflection
Regular Expressions
Versioning
Zip
Instrumentation
Java Virtual Machine
Java HotSpot VM
JDK
JRE
Java SE API


docs.oracle.com/javase/8/docs/
Tutorials and Training
The Java Tutorials
Java Training
More Information
Java SE 8 Names and Version Numbers
Java SE White Papers
Documentation Accessibility
Specifications
Installation Instructions
Certified System Configurations
Compatibility Guide
Known Issues
JDK 8 Adoption Guide
Troubleshooting Guide
About Test / Sample Applications and Code
Downloads
Java SE Downloads
JDK 8 Documentation
The following conceptual diagram illustrates the components of Oracle's Java SE products:
Description of Java Conceptual Diagram
Java Language
Java Language
java
javac
javadoc
jar
javap
jdeps
Scripting
Tools & Tool APIs
Security
Monitoring
JConsole
VisualVM
JMC
JFR
JPDA
JVM TI
IDL
RMI
Java DB
Deployment
Internationalization
Web Services
Troubleshooting
Deployment
Java Web Start
Applet / Java Plug-in
JavaFX
User Interface Toolkits
Swing
Java 2D
AWT
Accessibility
Drag and Drop
Input Methods
Image I/O
Print Service
Sound
Integration Libraries
IDL
JDBC
JNDI
RMI
RMI-IIOP
Scripting
Other Base Libraries
Beans
Security
Serialization
Extension Mechanism
JMX
XML JAXP
Networking
Override Mechanism
JNI
Date and Time
Input/Output
Internationalization
lang and util
lang and util Base Libraries
Math
Collections
Ref Objects
Regular Expressions
Logging
Management
Instrumentation
Concurrency Utilities
Reflection
Versioning
Preferences API
JAR
Zip
Java Virtual Machine
Java HotSpot Client and Server VM
JDK
JRE
Compact Profiles
Java SE API

堆内存溢出

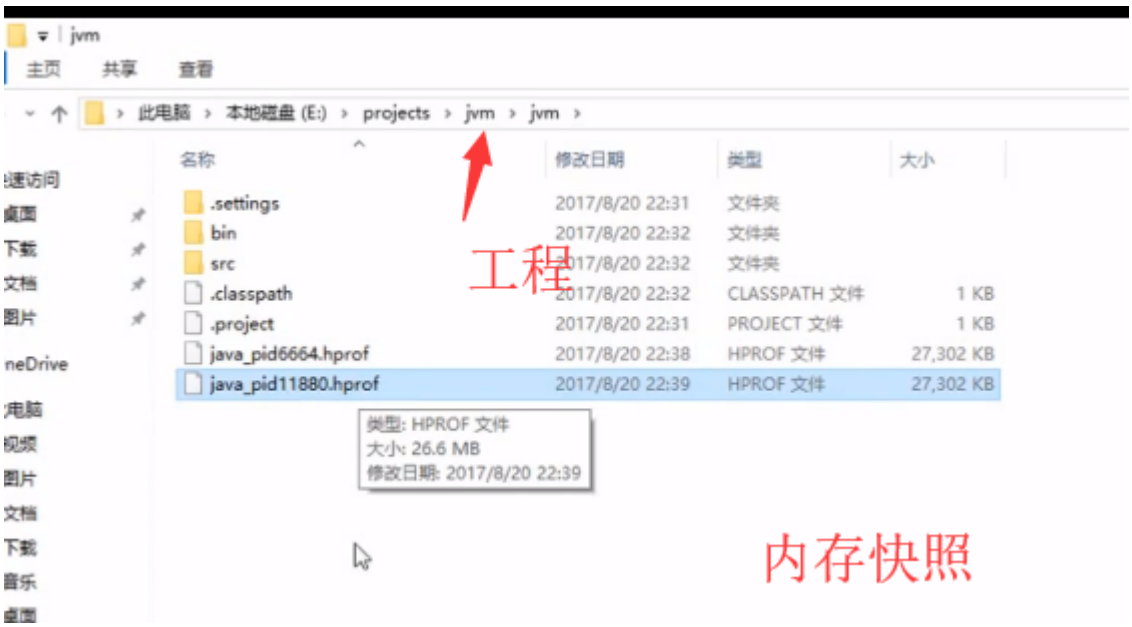




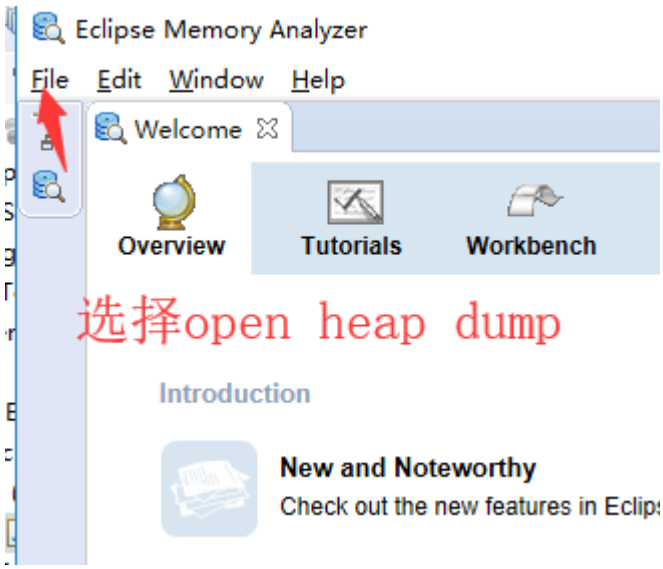
-Xms20m 设置虚拟机内存

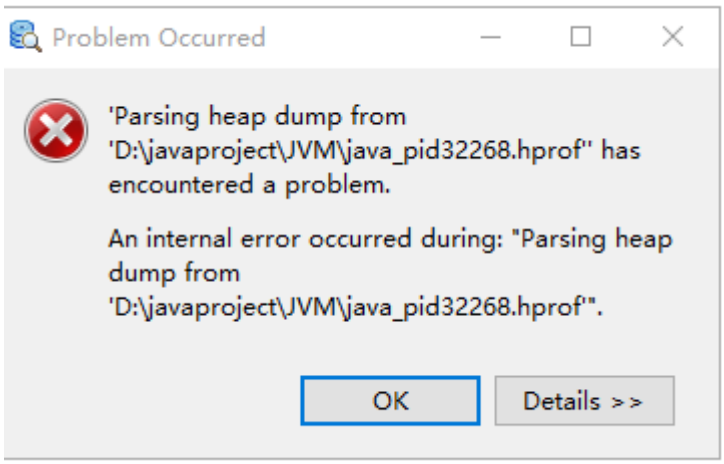
-XX:+HeapDumpOnOutOfMemoryError 保存虚拟机内存快照 通过jvm参数--XX:-HeapDumpOnOutOfMemoryError可以让JVM在出现内存溢出是Dump出当前的内存转储快照；

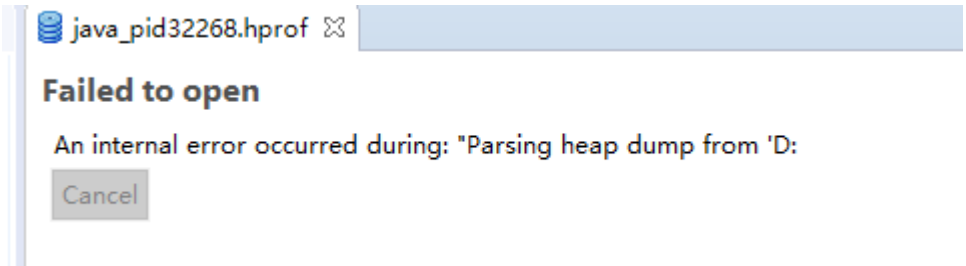
-XX:+HeapDumpOnOutOfMemoryError -Xms20m -Xms20m



使用内存分析工具打开eclipse memory analyzer 下载解压







open the MemoryAnalyzer.ini file；

2.change the default -Xmx1024m to a larger size.

https://www.cnblogs.com/ZuoAndFutureGirl/p/9029419.html

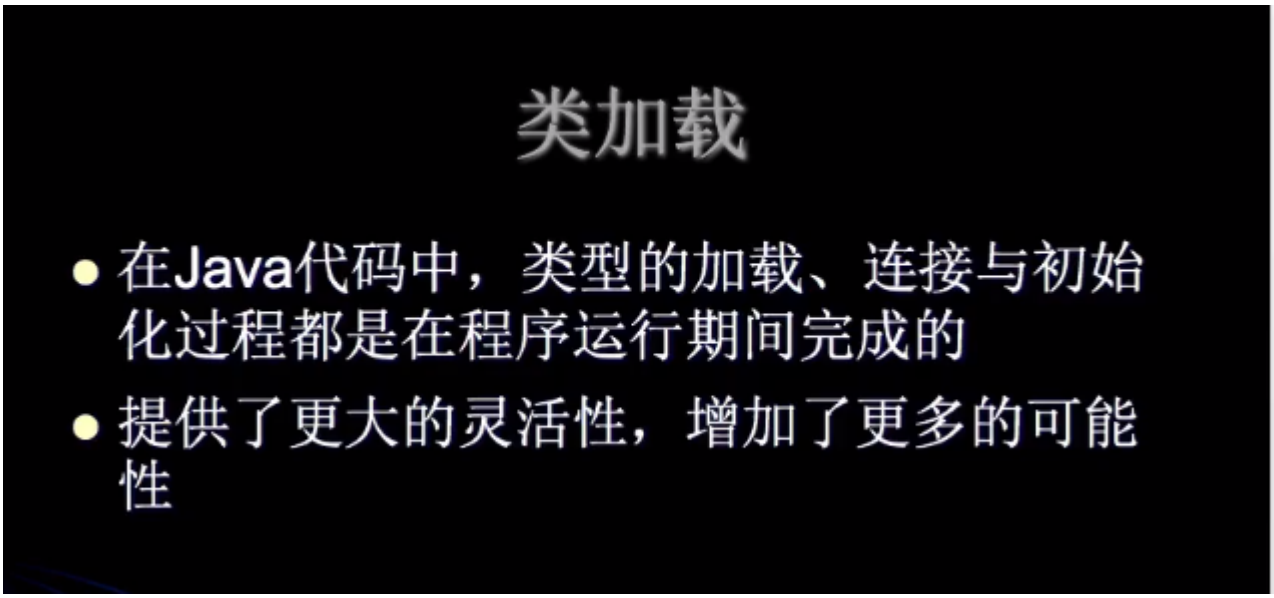
懂了很多道理，却依然过不好这一生。

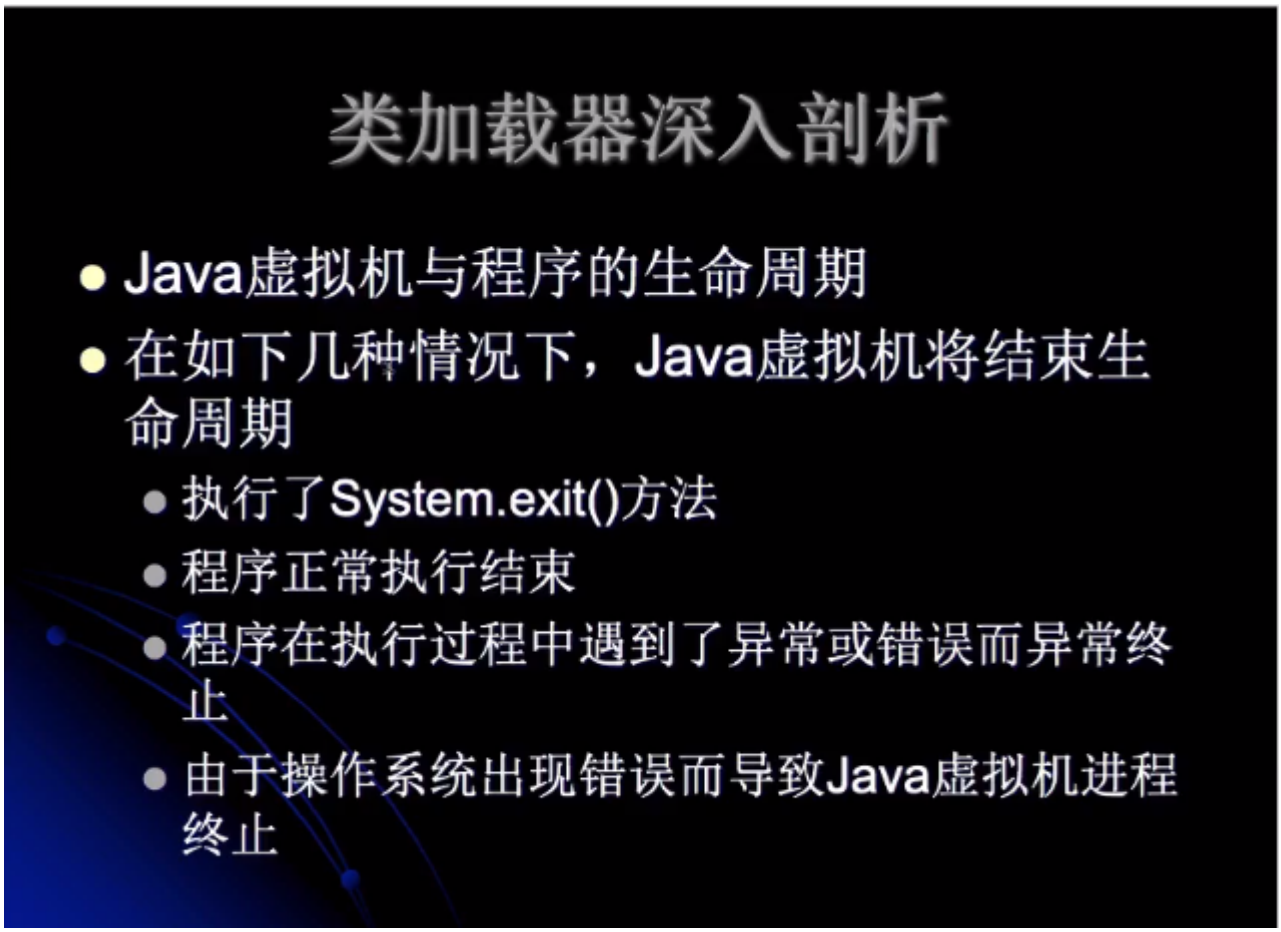
JVM原理

http://www.iprogramming.cn/

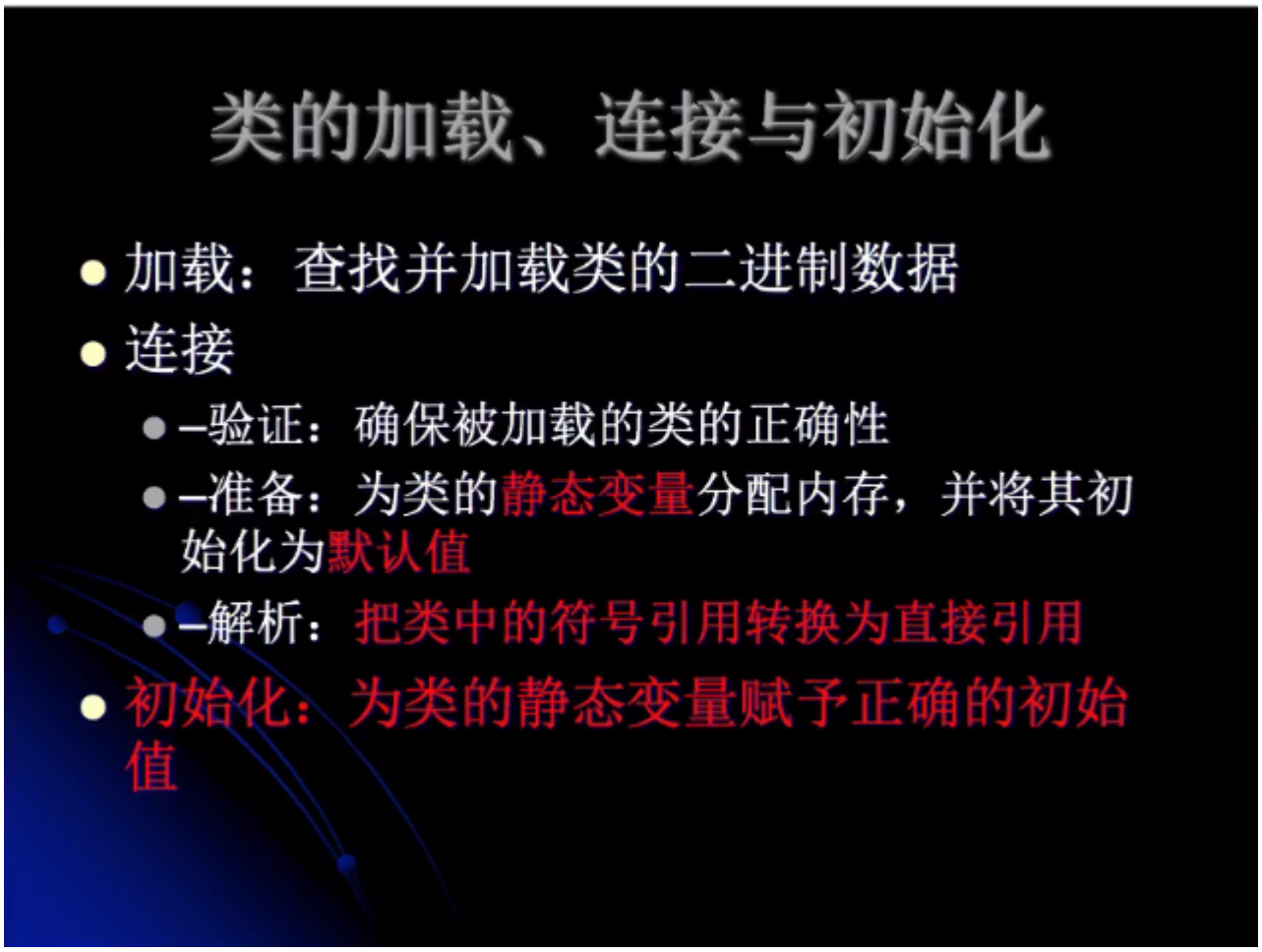
类加载

类型的加载，连接与初始化过程都是在程序运行期间完成的。必备





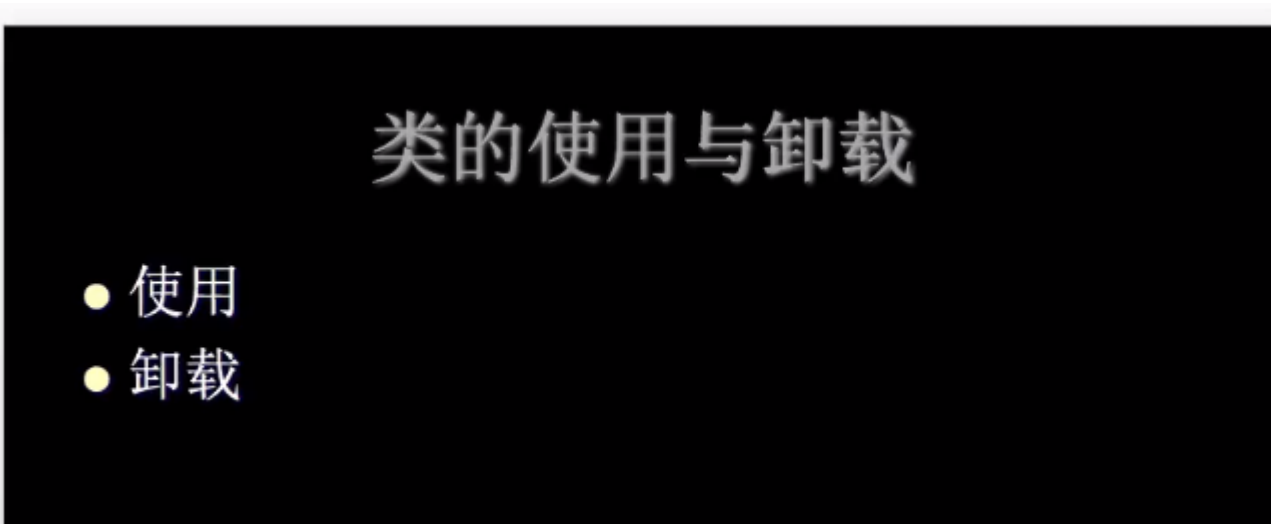
加载，连接，与初始化

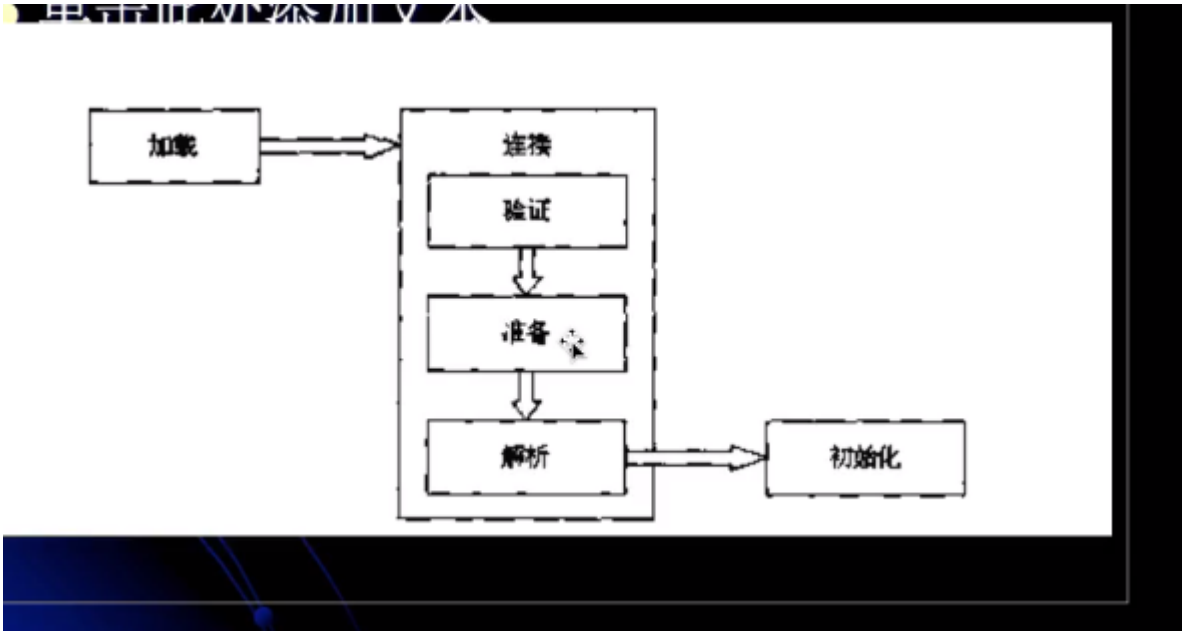


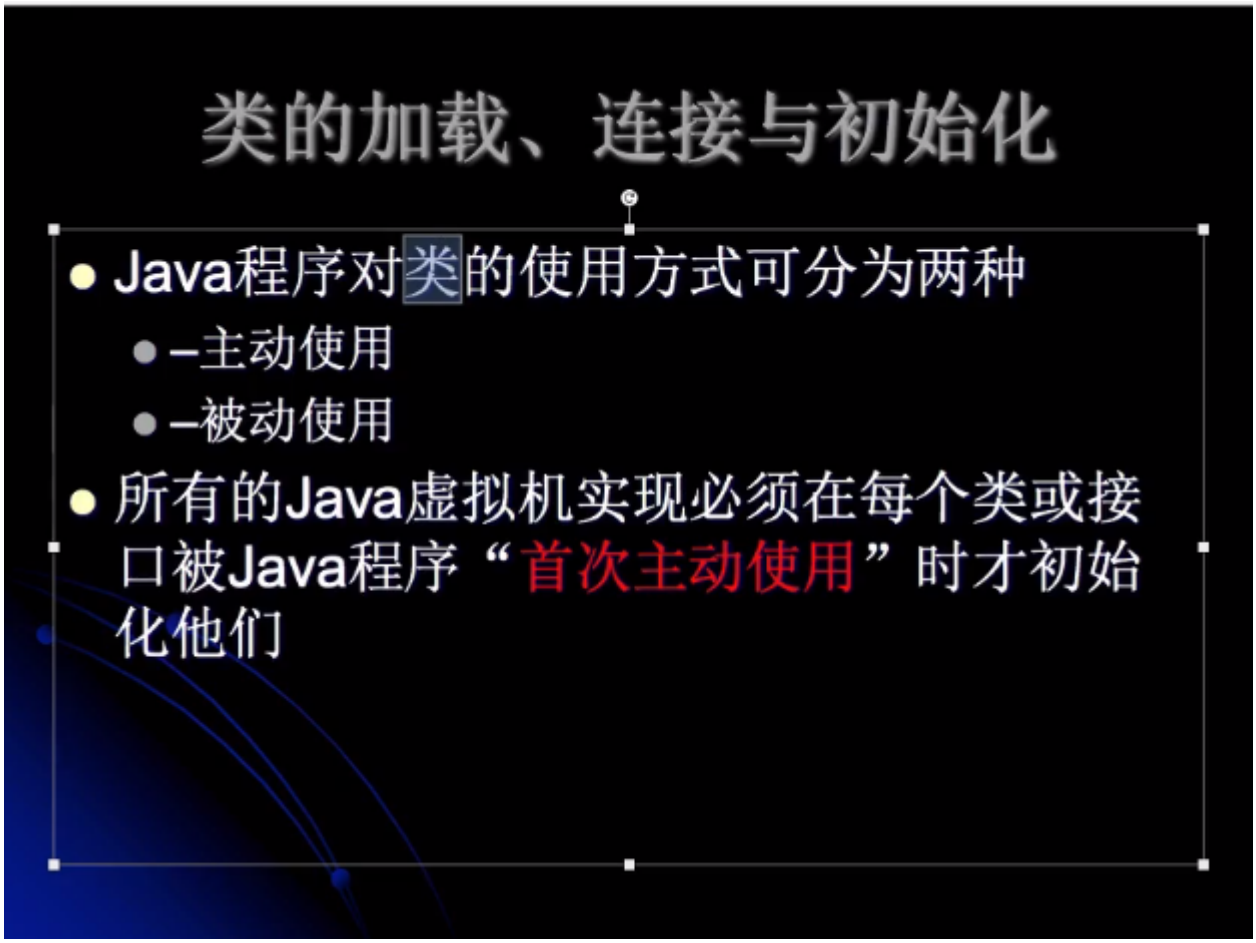
那么成员变量呢？？？？？

使用

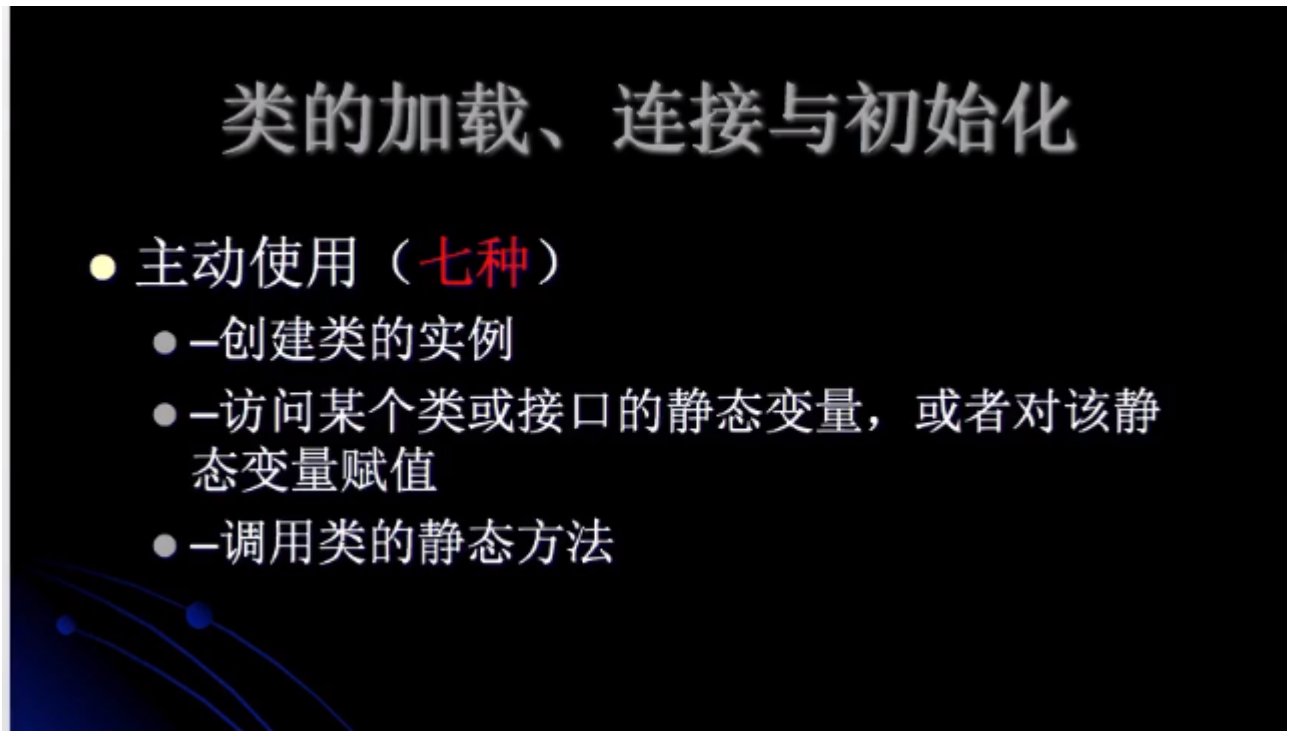
，卸载

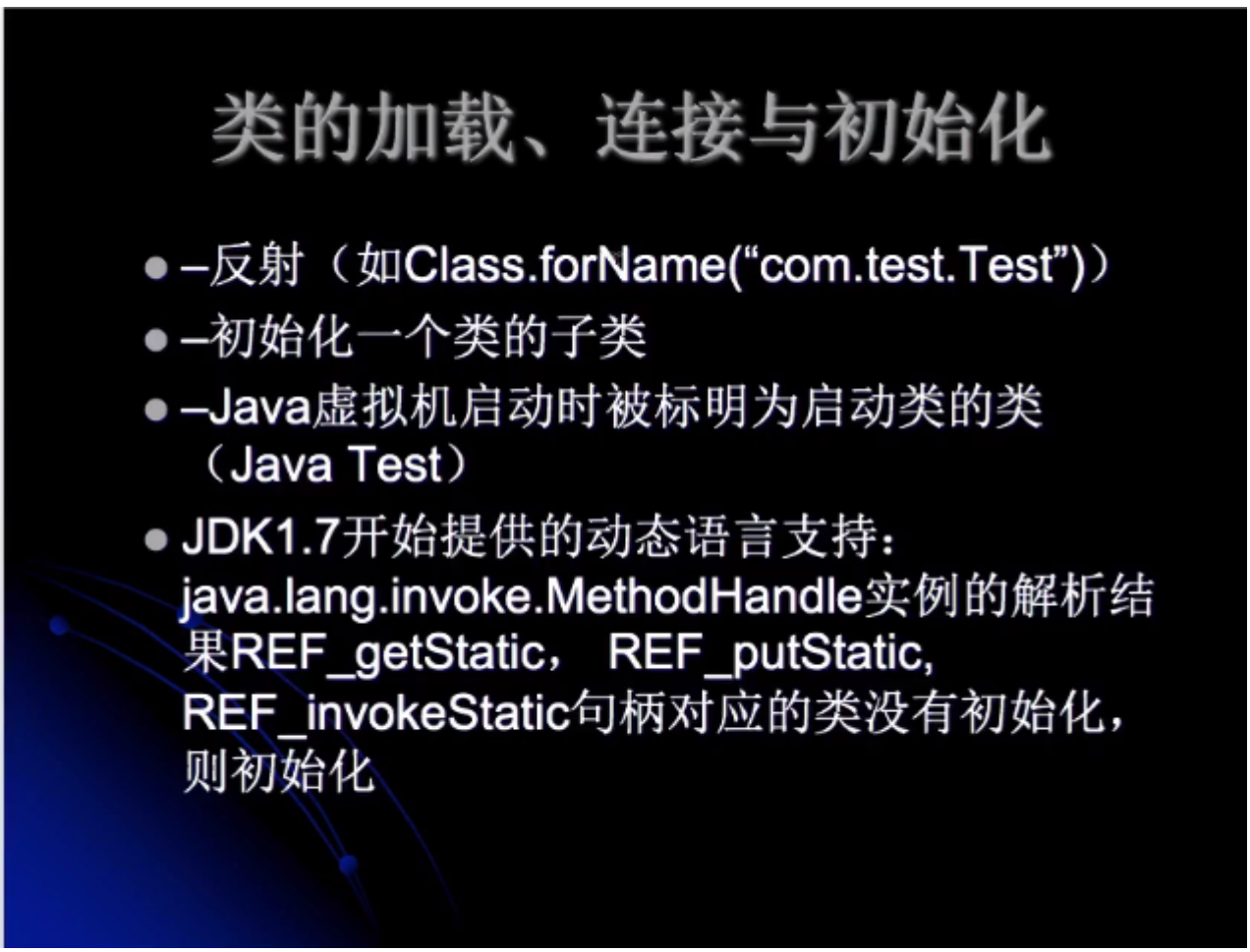



加载
连接
验证
准备
解析
初始化


类的加载、连接与初始化
Java程序对类的使用方式可分为两种
–主动使用
–被动使用
所有的Java虚拟机实现必须在每个类或接口被Java程序“首次主动使用”时才初始化他们

主动使用

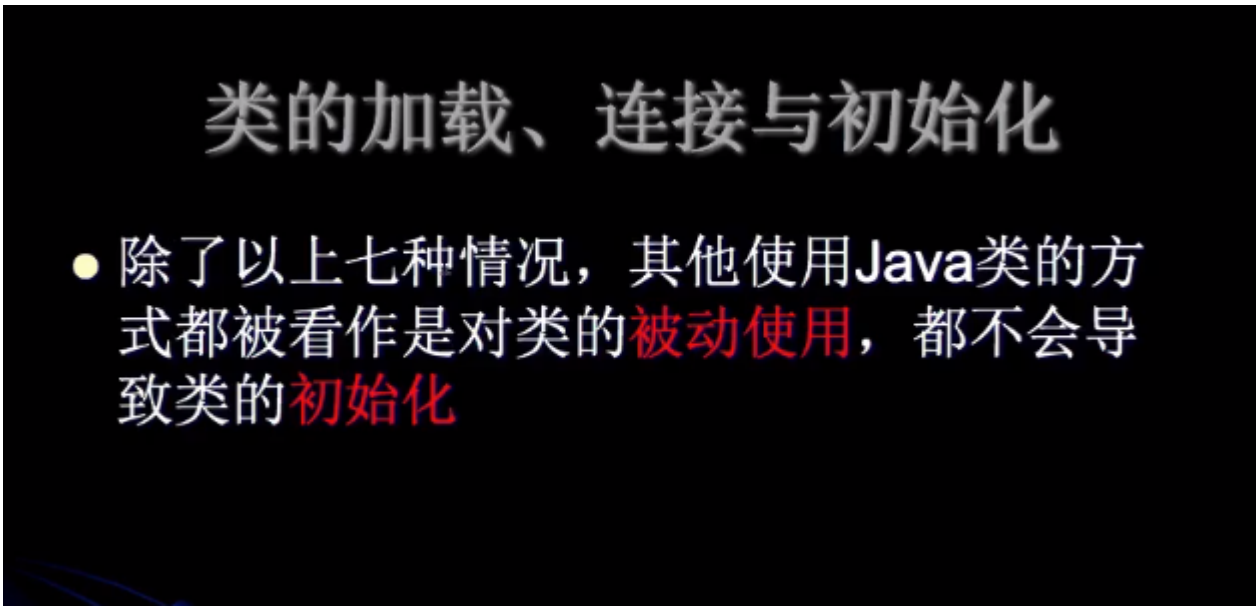

类的加载、连接与初始化
主动使用（七种）
–创建类的实例
–访问某个类或接口的静态变量，或者对该静态变量赋值
–调用类的静态方法

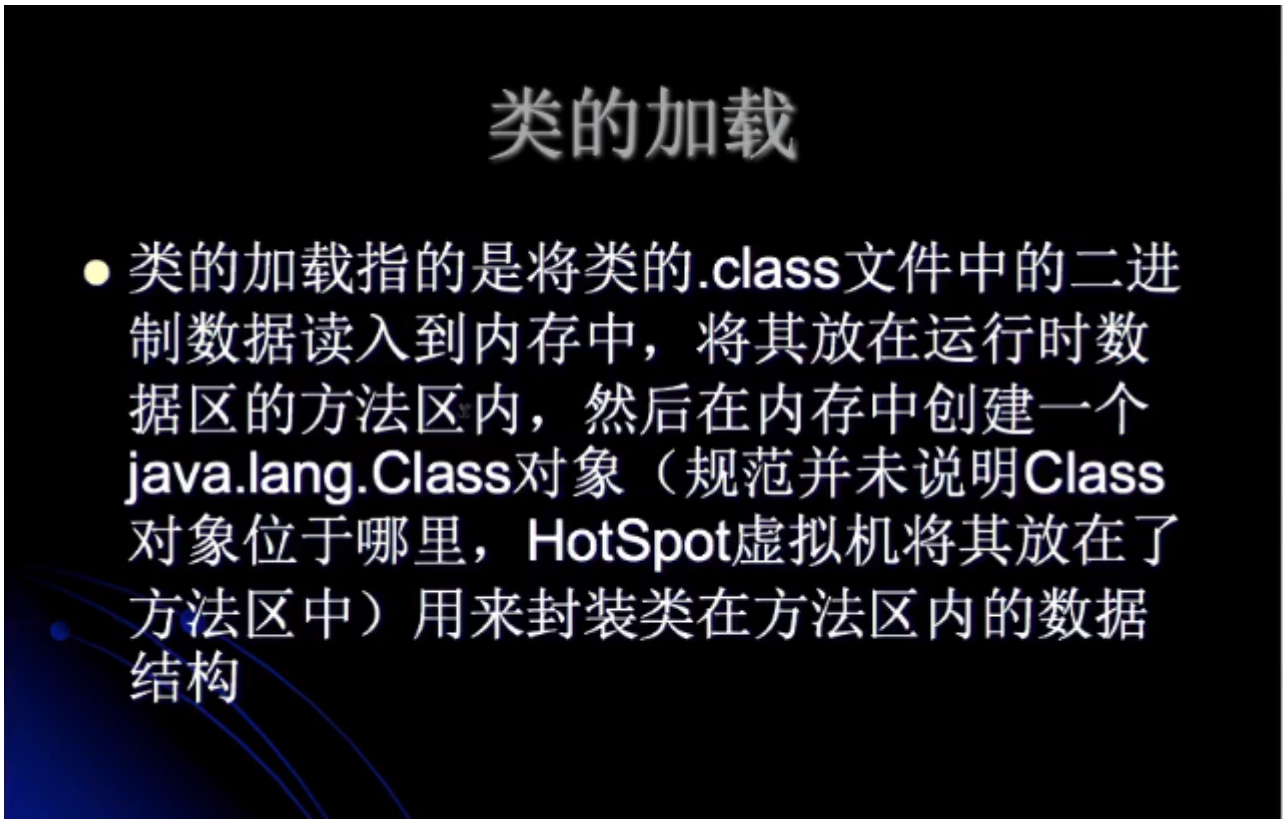

类的加载、连接与初始化
–反射（如Class.forName(“com.test.Test”)）
–初始化一个类的子类
–Java虚拟机启动时被标明为启动类的类（Java Test）
JDK1.7开始提供的动态语言支持：java.lang.invoke.MethodHandle实例的解析结果REF_getStatic， REF_putStatic, REF_invokeStatic句柄对应的类没有初始化，则初始化

助记符

助记符。

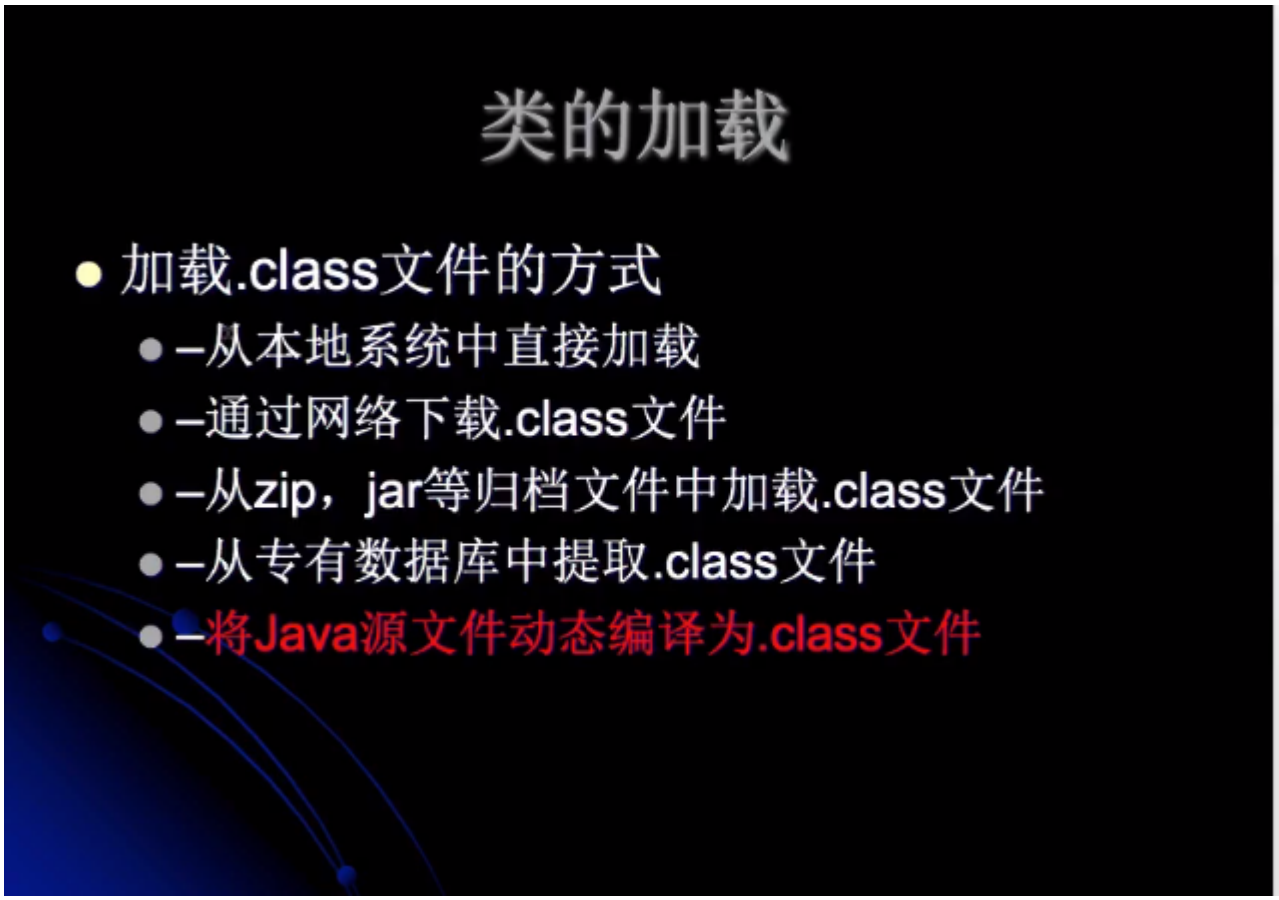
```
getstatic
putstatic
invokestatic
```





一个类，不管生成多少实例，所有实例对应的Class对象只有一份。

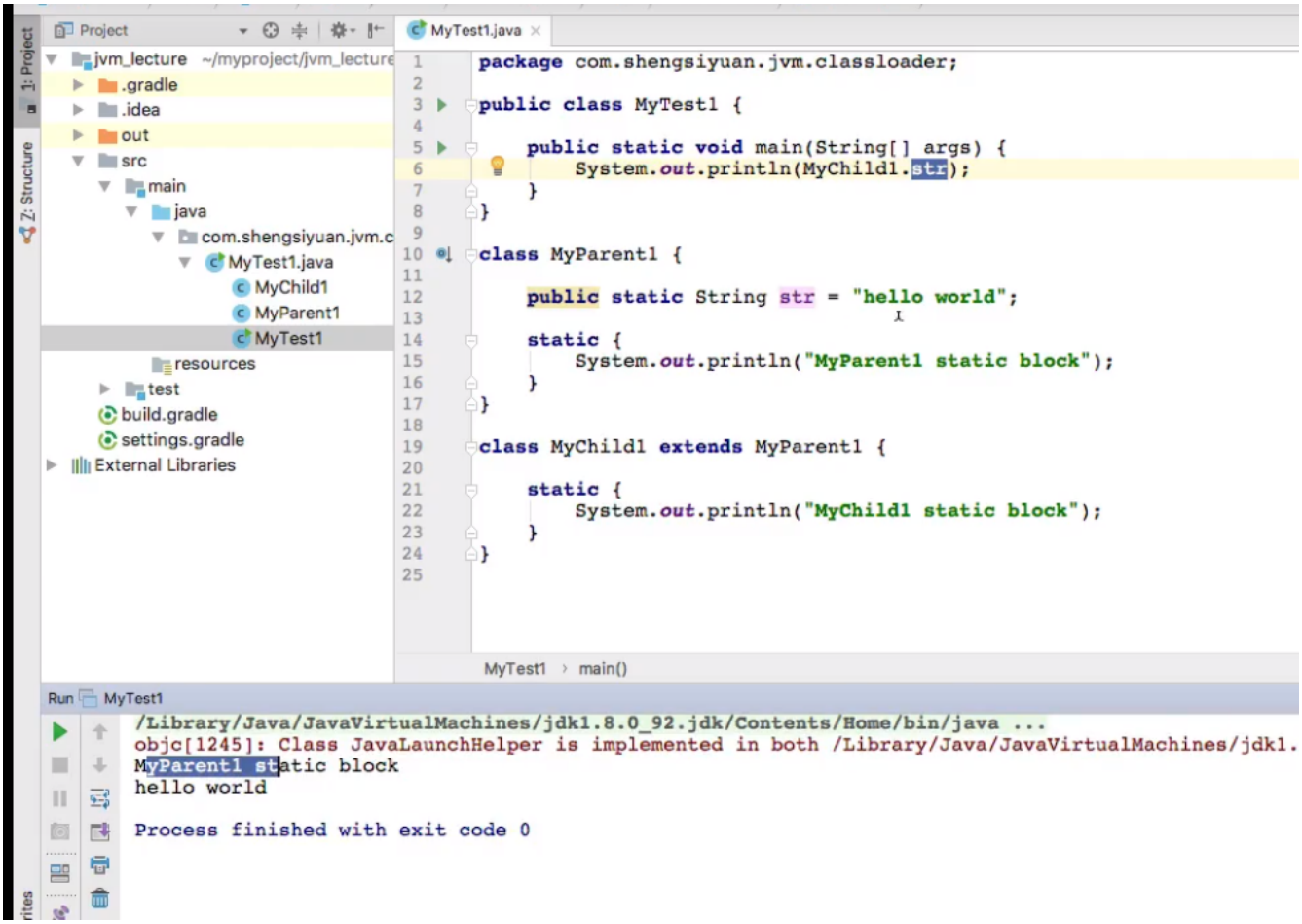
类的加载方式



动态代理，jsp——》servlet

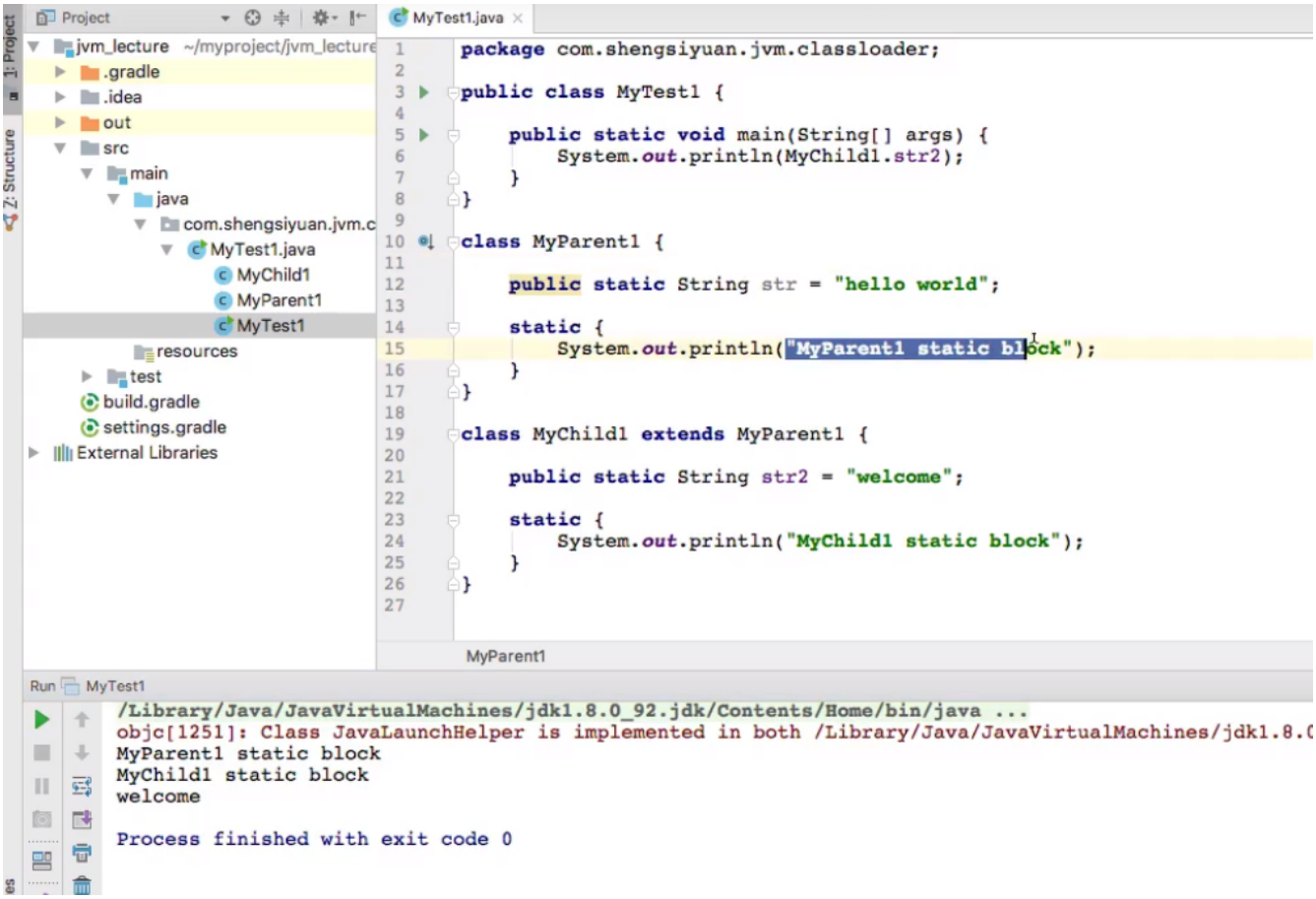
实战演示

例子1



解释：通过子类的名字调用父类的静态变量，并没有主动使用子类，所以子类的静态代码快没有被初始化，并执行。

列子2



解释：初始化父类的子类，对父类来说也是一个主动使用。因此父类的静态代码快被初始化，执行了。

而子类这里调用了子类的静态变量。也是一个主动使用。

第二条：访问某个类或这接口的静态变量，或这对该变量赋值。

第五条：初始化一个类的子类

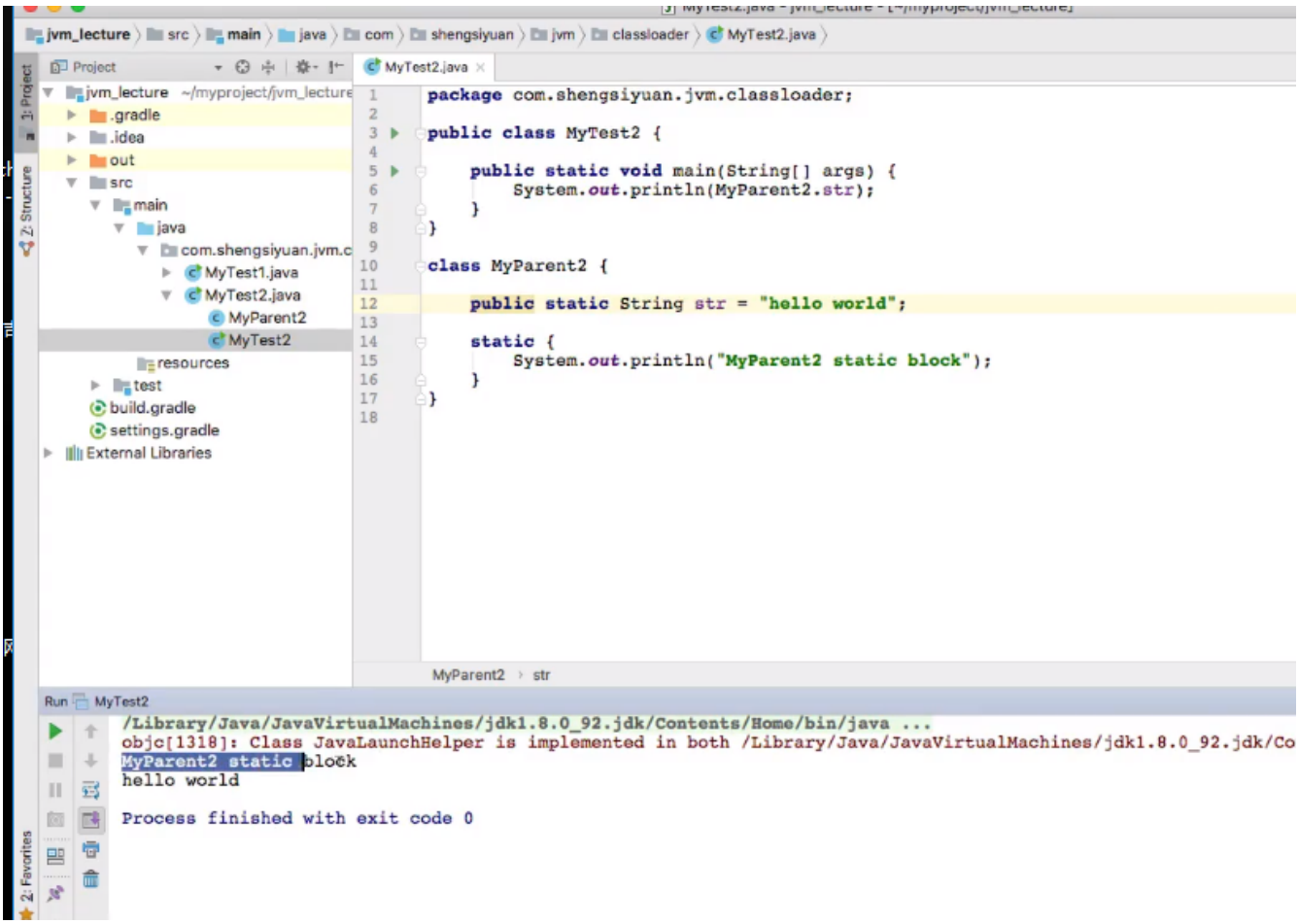
列子3

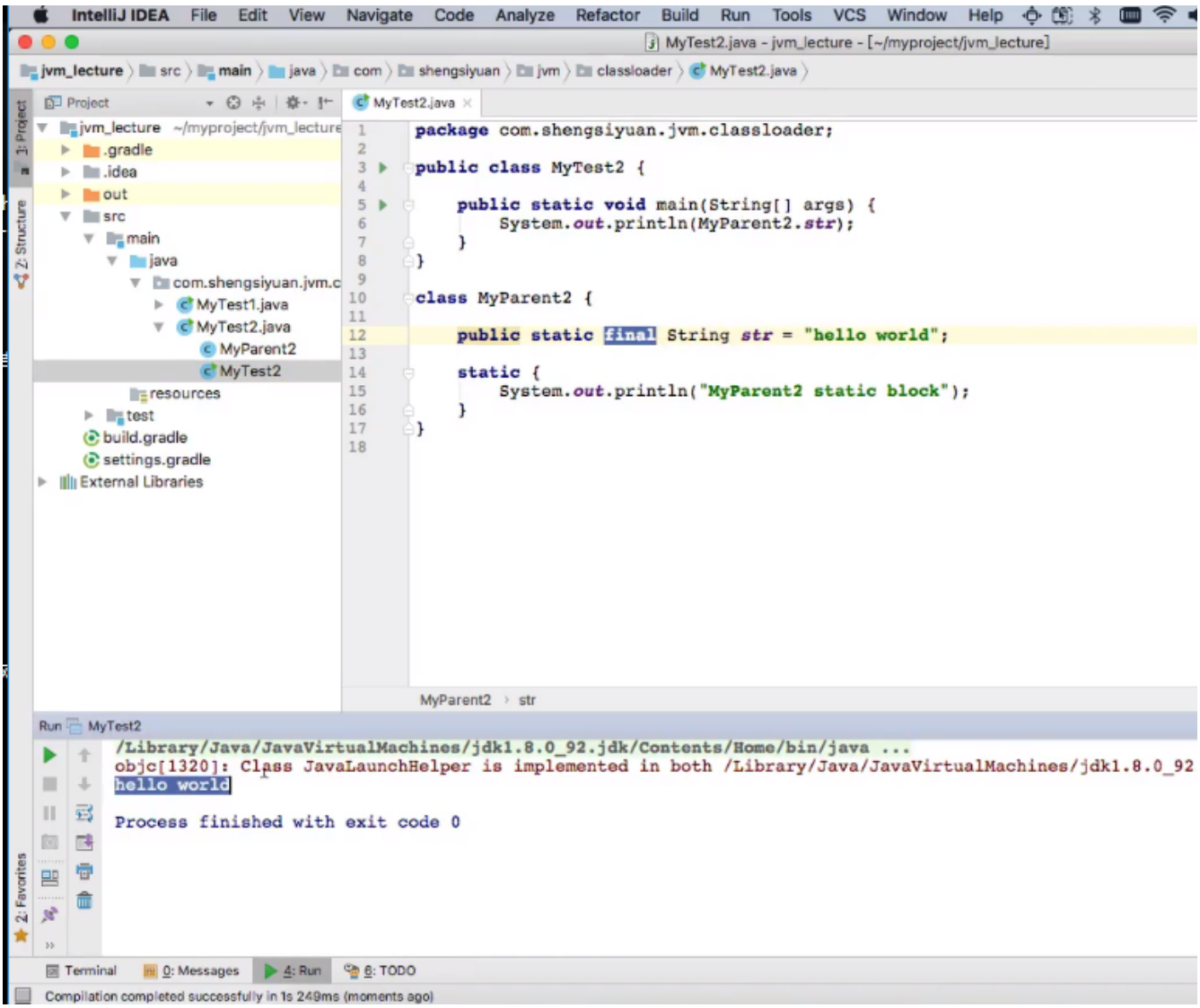
```java
package com.shengsiyuan.jvm.classloader;

/*
    对于静态字段来说，只有直接定义了该字段的类才会被初始化；
    当一个类在初始化时，要求其父类全部都已经初始化完毕了
    -XX:+TraceClassLoading，用于追踪类的加载信息并打印出来

    -XX:+<option>，  表示开启option选项
    -XX:-<option>，  表示关闭option选项
    -XX:<option>=<value>，  表示将option选项的值设置为value

 */
public class MyTest1 {

    public static void main(String[] args) {
        System.out.println(MyChild1.str);
    }
}

class MyParent1 {

    public static String str = "hello world";

    static {
        System.out.println("MyParent1 static block");
    }
}

class MyChild1 extends MyParent1 {

    public static String str2 = "welcome";

    static {
        System.out.println("MyChild1 static block");
    }
}
```

```
[Loaded sun.launcher.LauncherHelper$FXHelper from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
[Loaded java.net.AbstractPlainSocketImpl$1 from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
[Loaded java.lang.Class$MethodArray from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
[Loaded java.lang.Void from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
[Loaded com.shengsiyuan.jvm.classloader.MyParent1 from file:/Users/zhanglong/myproject/jvm_lecture/out/production/classes/]
[Loaded com.shengsiyuan.jvm.classloader.MyChild1 from file:/Users/zhanglong/myproject/jvm_lecture/out/production/classes/]
MyParent1 static block
hello world
[Loaded java.lang.Shutdown from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
[Loaded java.lang.Shutdown$Lock from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
```

即使没有对MyChild初始化，但是完成了对这个类的加载。

列子3


jvm_lecture › src › main › java › com › shengsiyuan › jvm › classloader › MyTest2.java
Project
jvm_lecture ~/myproject/jvm_lecture
.gradle
.idea
out
src
main
java
com.shengsiyuan.jvm.c
MyTest1.java
MyTest2.java
MyParent2
MyTest2
resources
test
build.gradle
settings.gradle
External Libraries
MyTest2.java
1 package com.shengsiyuan.jvm.classloader;
2
3 public class MyTest2 {
4
5 public static void main(String[] args) {
6 System.out.println(MyParent2.str);
7 }
8 }
9
10 class MyParent2 {
11
12 public static String str = "hello world";
13
14 static {
15 System.out.println("MyParent2 static block");
16 }
17 }
18
MyParent2 › str
Run MyTest2
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
objc[1318]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Co
MyParent2 static block
hello world
Process finished with exit code 0
1: Project
2: Structure
2: Favorites



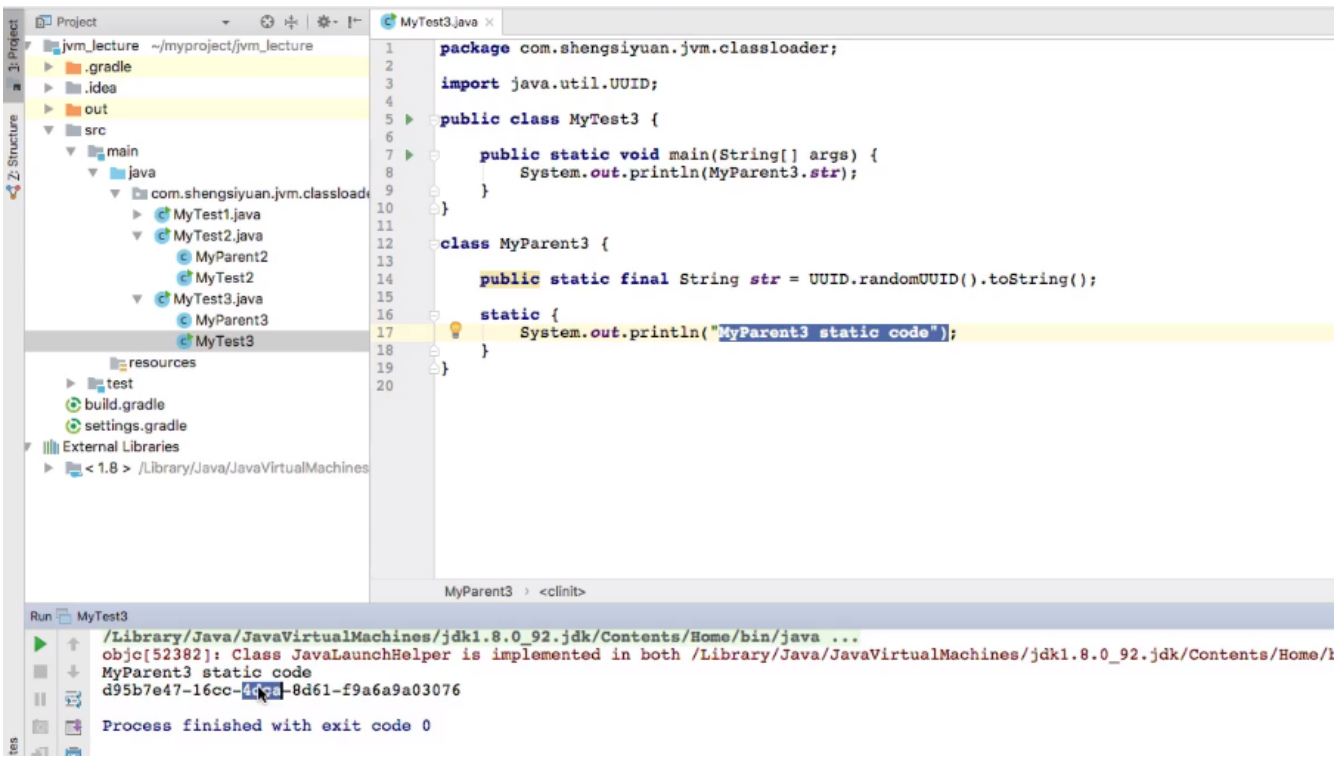
解释：

在编译阶段，把 final定义的常量 放入到 调用这个常量的方法 所在类的常量池当中。

MyTest2.java

```java
package com.shengsiyuan.jvm.classloader;

// 常量在编译阶段会存入到调用这个常量的方法所在的类的常量池中，
// 本质上，调用类并没有直接引用到定义常量的类，因此并不会触发
// 定义常量的类的初始化
// 注意：这里指的是将常量存放到了MyTest2的常量池中，之后MyTest2与MyParent2就没有任何关系了
// 甚至，我们可以将MyParent2的class文件删除

public class MyTest2 {

    public static void main(String[] args) {
        System.out.println(MyParent2.str);
    }
}

class MyParent2 {

    public static final String str = "hello world";

    static {
        System.out.println("MyParent2 static block");
    }
}
```

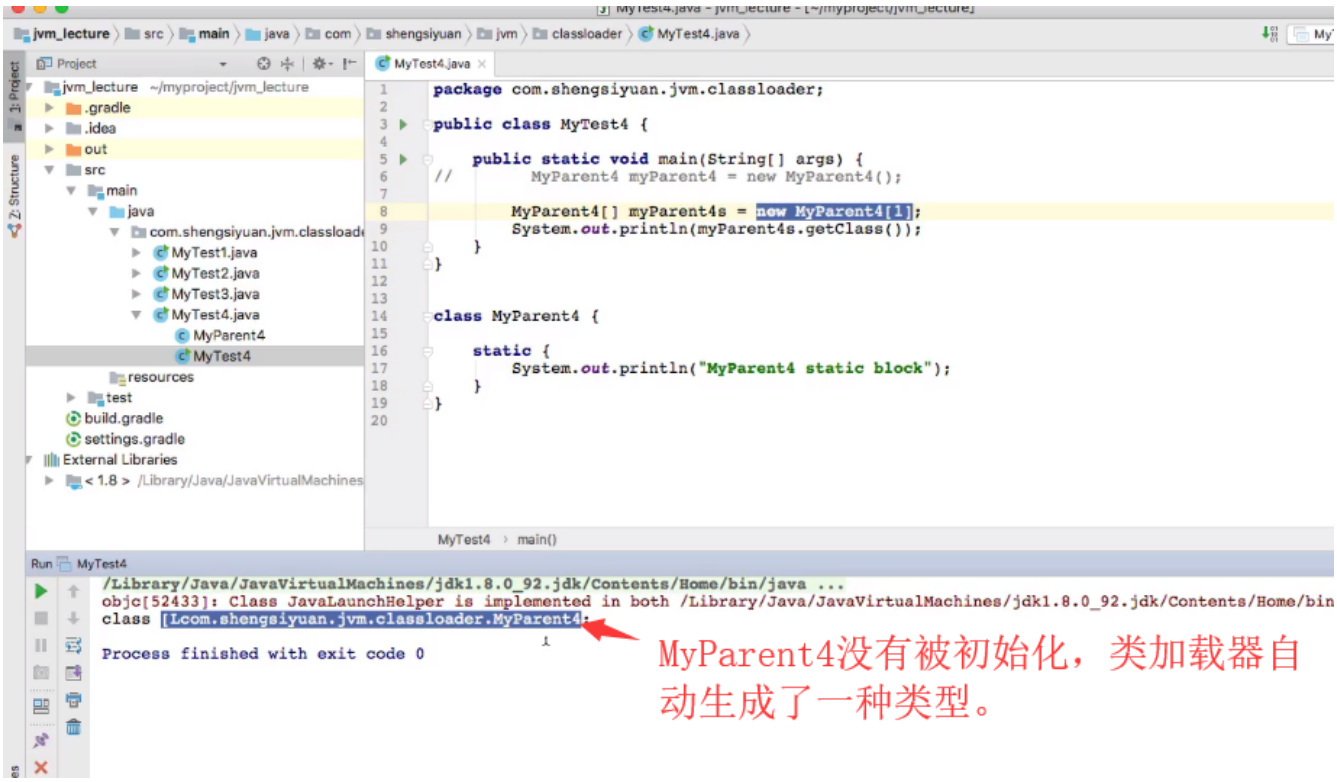
列5

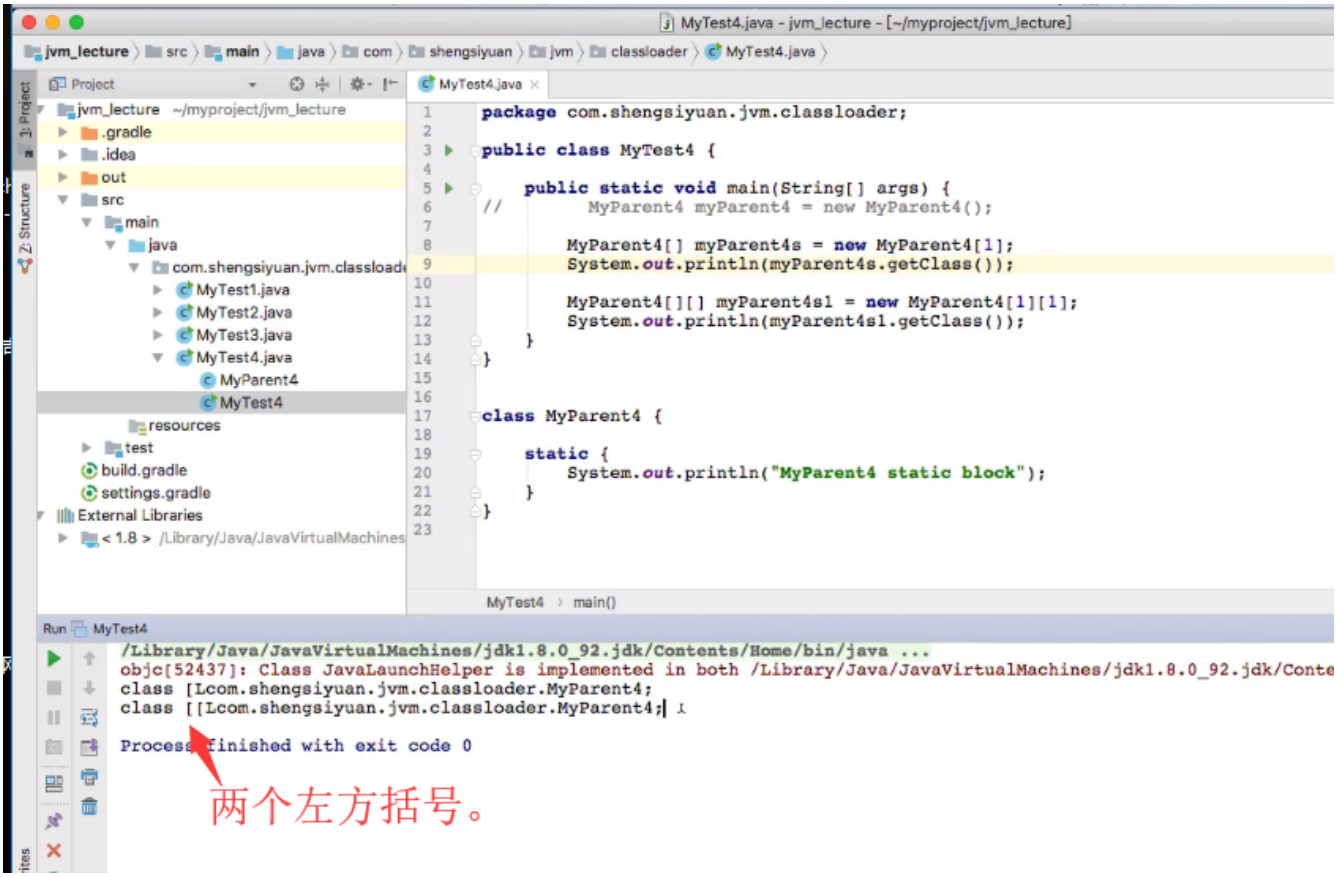


解释：str编译期间不确定，导致MyParent3被初始化，因而静态代码块被执行了。

```java
import java.util.UUID;

/*
    当一个常量的值并非编译期间可以确定的，那么其值就不会被放到调用类的常量池中，
    这时在程序运行时，会导致主动使用这个常量所在的类，显然会导致这个类被初始化。
 */
public class MyTest3 {

    public static void main(String[] args) {
        System.out.println(MyParent3.str);
    }
}

class MyParent3 {

    public static final String str = UUID.randomUUID().toString();

    static {
        System.out.println("MyParent3 static code");
    }
}
```

列6

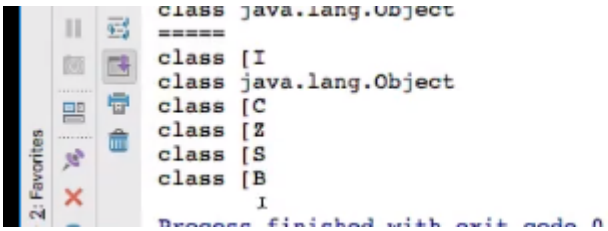




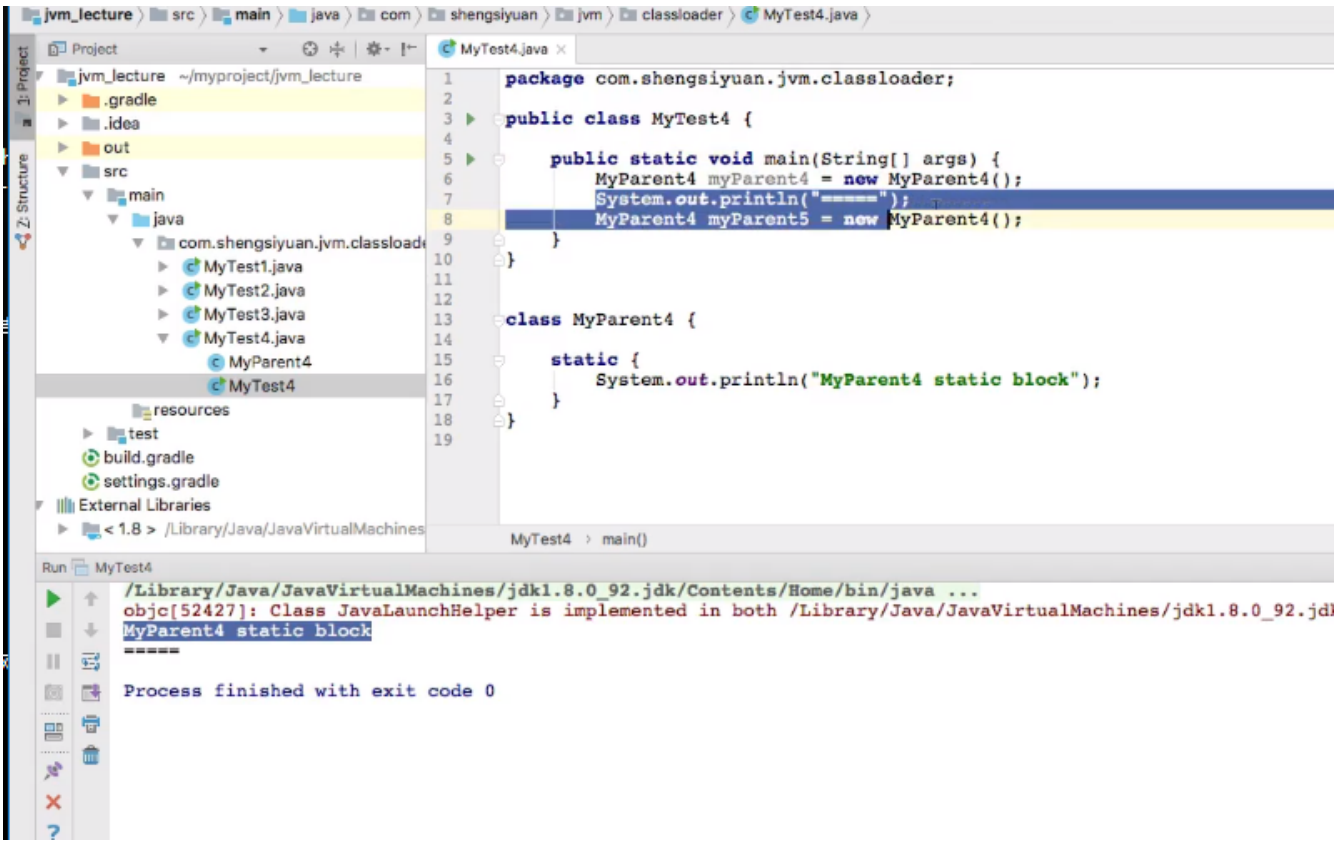
解释：java虚拟机在运行期间生成的一个数组类型的类。

```
/*
    对于数组实例来说，其类型是由JVM在运行期动态生成的，表示为[Lcom.shengsiyuan.jvm.classloader.MyParent4
    这种形式。动态生成的类型，其父类型就是Object。

    对于数组来说，JavaDoc经常将构成数组的元素为Component，实际上就是将数组降低一个维度后的类型。

*/
```

```
        System.out.println("=====");

        int[] ints = new int[1];
        System.out.println(ints.getClass());
        System.out.println(ints.getClass().getSuperclass());

        char[] chars = new char[1];
        System.out.println(chars.getClass());

        boolean[] booleans = new boolean[1];
        System.out.println(booleans.getClass());

        short[] shorts = new short[1];
        System.out.println(shorts.getClass());

        byte[] bytes = new byte[1];
        System.out.println(bytes.getClass());
    }
}
```



列7



两次实例化，但是只有首次实例化算首次主动使用。

接口：

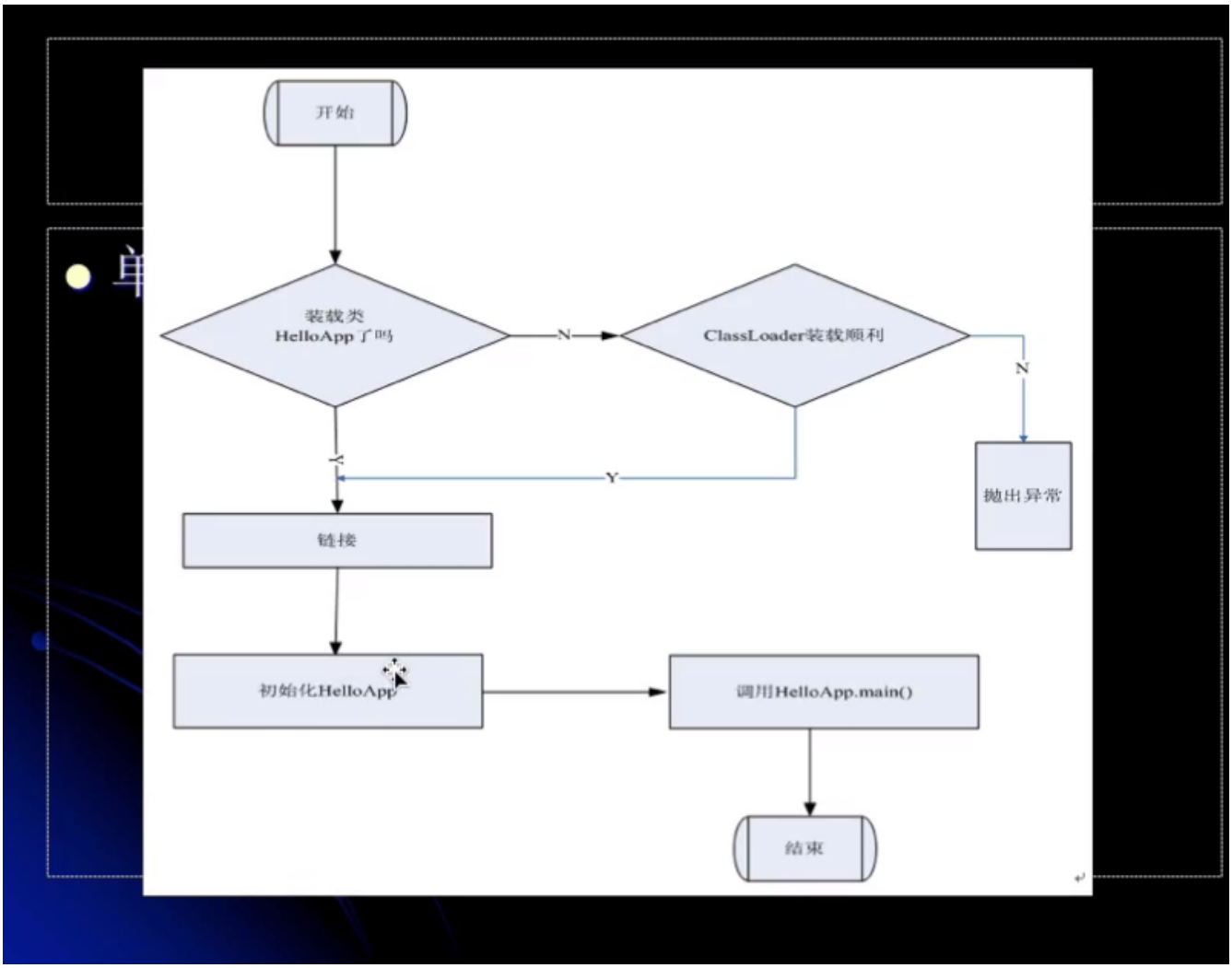
```java
package com.shengsiyuan.jvm.classloader;

/*
    当一个接口在初始化时，并不要求其父接口都完成了初始化
    只有在真正使用到父接口的时候（如引用接口中所定义的常量时），才会初始化
 */

import java.util.Random;

public class MyTest5 {

    public static void main(String[] args) {
        System.out.println(MyChild5.b);
    }
}

interface MyParent5 {

    public static final int a = 5;

}

interface MyChild5 extends MyParent5 {

    public static final int b = new Random().nextInt(4);
}
```

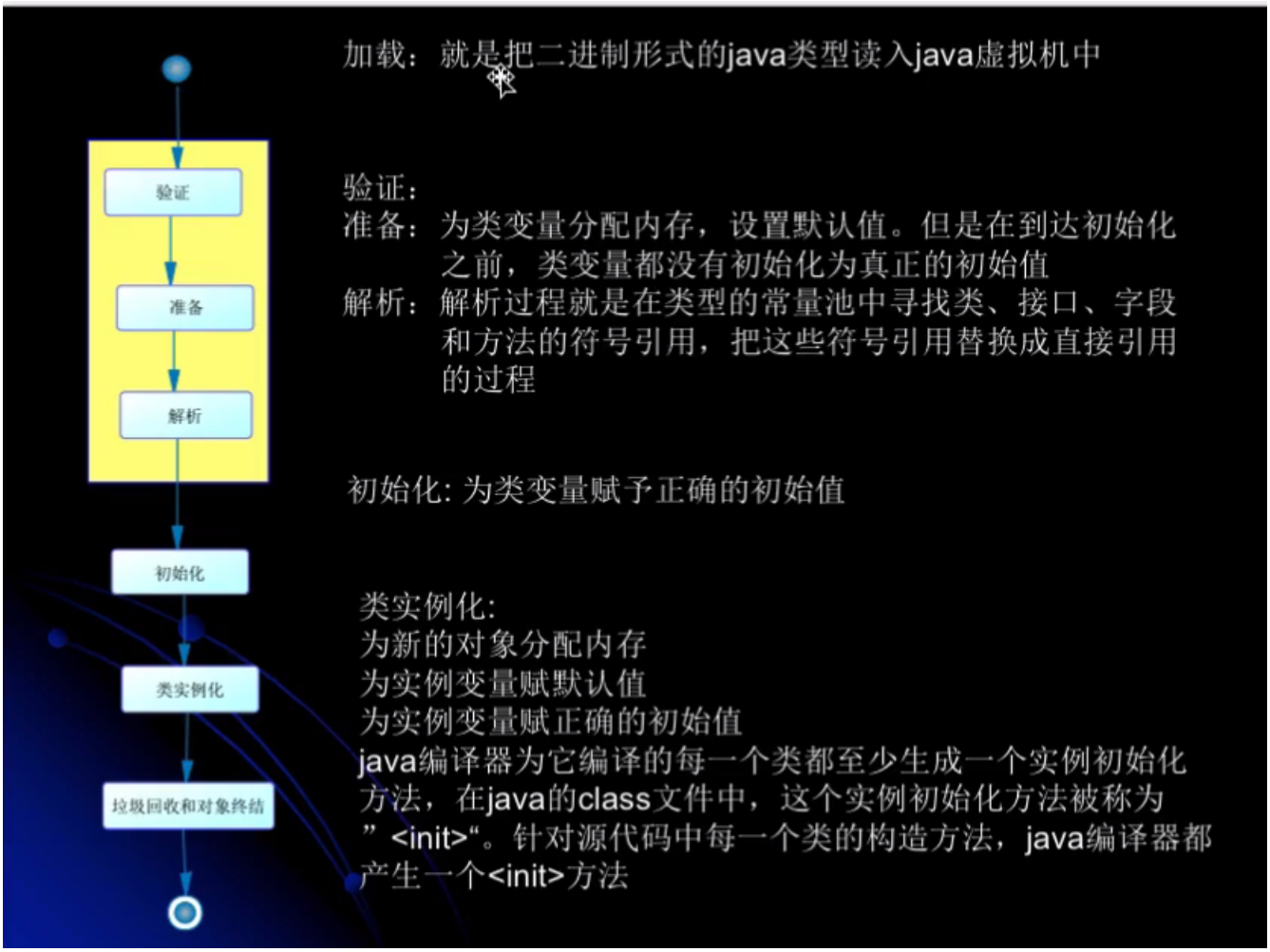
```java
package com.shengsiyuan.jvm.classloader;

public class MyTest6 {

    public static void main(String[] args) {
        Singleton singleton = Singleton.getInstance();

        System.out.println("counter1: " + Singleton.counter1);
        System.out.println("counter2: " + Singleton.counter2);
    }
}

class Singleton {

    public static int counter1;

    private static Singleton singleton = new Singleton();

    private Singleton() {
        counter1++;
        counter2++;

        System.out.println(counter1);
        System.out.println(counter2);
    }

    public static int counter2 = 0;

    public static Singleton getInstance() {
        return singleton;
    }
}
```

```
Run  MyTest6
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Co
objc[52641]: Class JavaLaunchHelper is implemented i
1
1
counter1: 1
counter2: 1

Process finished with exit code 0
```

解释：静态初始化的顺序取决于其代码中的顺序，所以输出的结果是1 0

类加载理论2





成员变量怎么办呢？？？？？在类实例化的时候赋值初始化。

类的加载

- 类的加载的最终产品是位于内存中的Class对象
- Class对象封装了类在方法区内的数据结构，并且向Java程序员提供了访问方法区内的数据结构的接口

类的加载

- 有两种类型的类加载器
  - Java虚拟机自带的加载器
    - 根类加载器（Bootstrap）
    - 扩展类加载器（Extension）
    - 系统（应用）类加载器（System）
  - 用户自定义的类加载器
    - **java.lang.ClassLoader**的子类
    - 用户可以定制类的加载方式

类的加载

- 类加载器并不需要等到某个类被“首次主动使用”时再加载它

类的加载

- JVM规范允许类加载器在预料某个类将要被使用时就预先加载它，如果在预先加载的过程中遇到了.class文件缺失或存在错误，类加载器必须在程序首次主动使用该类时才报告错误（LinkageError错误）
- 如果这个类一直没有被程序主动使用，那么类加载器就不会报告错误

类的验证

- 类被加载后，就进入连接阶段。连接就是将已经读入到内存的类的二进制数据合并到虚拟机的运行时环境中去。

类的准备

在准备阶段，Java 虚拟机为类的静态变量分配内存，并设置默认的初始值。例如对于以下 Sample 类，在准备阶段，将为 int 类型的静态变量 a 分配 4 个字节的内存空间，并且赋予默认值 0，为 long 类型的静态变量 b 分配 8 个字节的内存空间，并且赋予默认值 0。

```
public class Sample{
    private static int a=1;
    public static long b;

    static{
        b=2;
    }
    ...
}
```

类的初始化

- 单击此处添加文本

在初始化阶段，Java 虚拟机执行类的初始化语句，为类的静态变量赋予初始值。在程序中，静态变量的初始化有两种途径：（1）在静态变量的声明处进行初始化；（2）在静态代码块中进行初始化。例如在以下代码中，静态变量 a 和 b 都被显式初始化，而静态变量 c 没有被显式初始化，它将保持默认值 0。

```
public class Sample{
    private static int a=1;          //在静态变量的声明处进行初始化
    public static long b;
    public static long c;

    static{
        b=2;                         //在静态代码块中进行初始化
    }
    ...
}
```

类的初始化

- 单击此处添加文本

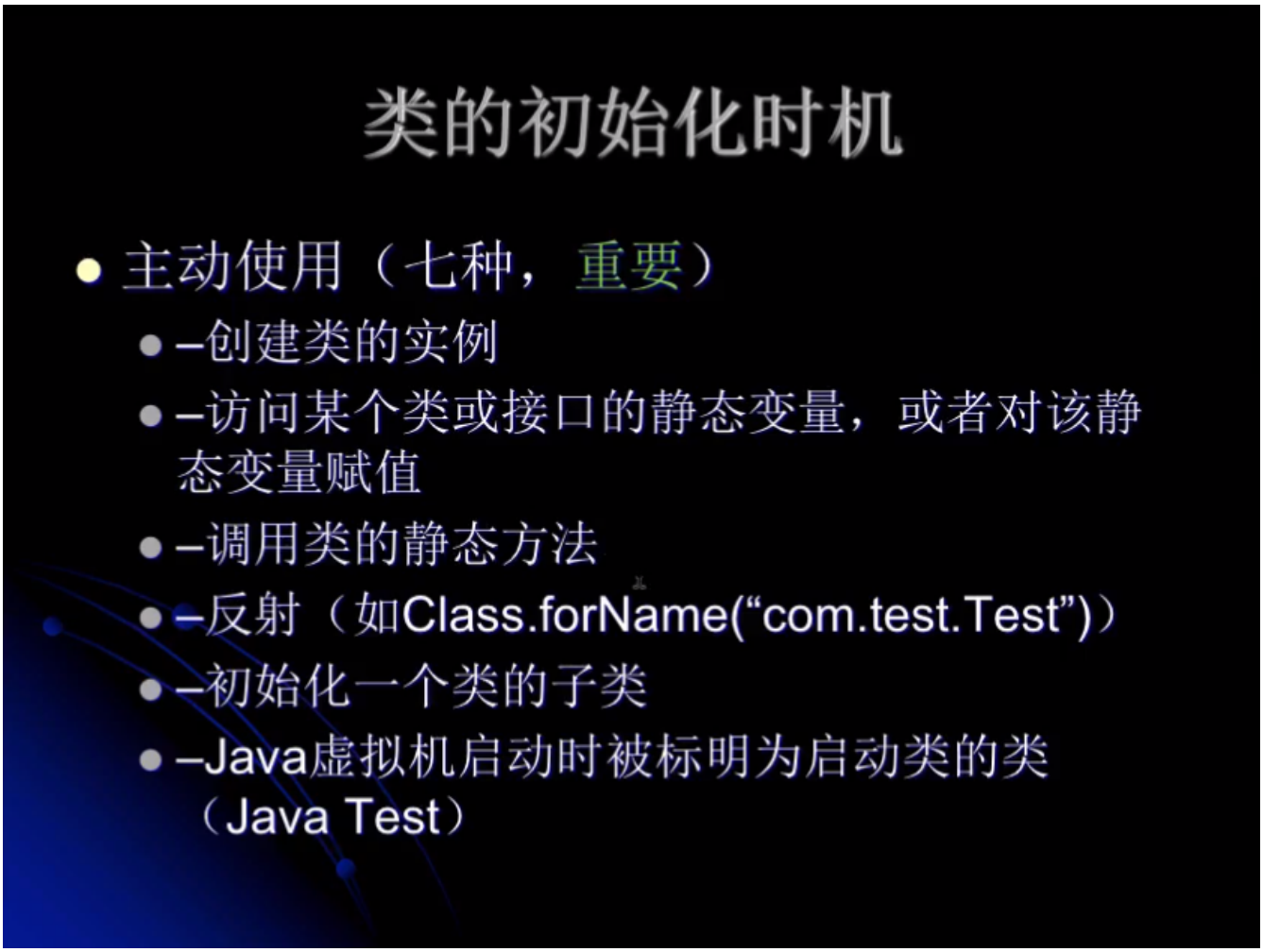
静态变量的声明语句，以及静态代码块都被看做类的初始化语句，Java 虚拟机会按照初始化语句在类文件中的先后顺序来依次执行它们。例如当以下 Sample 类被初始化后，它的静态变量 a 的取值为 4。

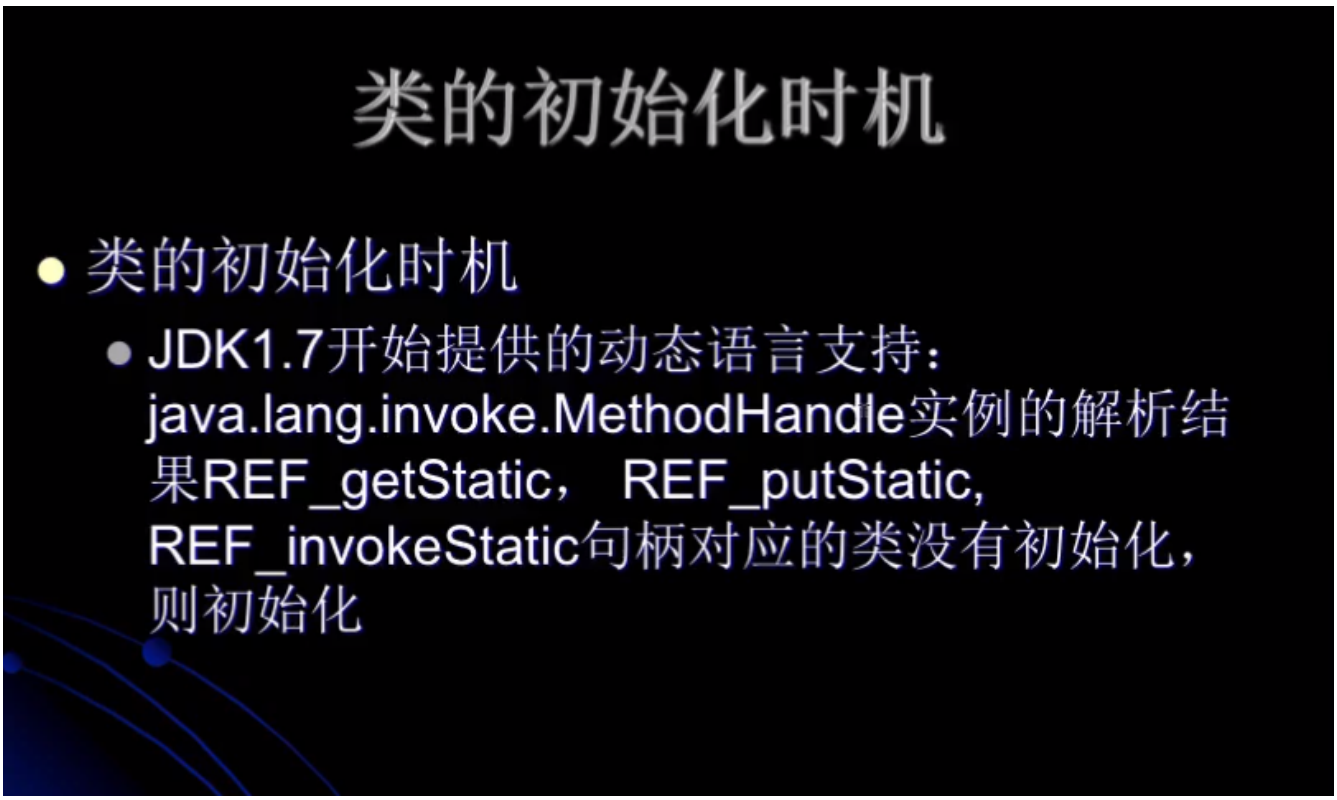
```
public class Sample{
 static int a=1;
 static{ a=2; }
 static{ a=4; }
 public static void main(String args[]){
  System.out.println("a="+a); //打印a=4
 }
}
```

类的初始化

- 类的初始化步骤
  - 假如这个类还没有被加载和连接，那就先进行加载和连接
  - 假如类存在直接父类，并且这个父类还没有被初始化，那就先初始化直接父类
  - 假如类中存在初始化语句，那就依次执行这些初始化语句

这里应该是初始化父类优先。加载也是父类优先。待验证，？？？？？？？？？？？？？？





类的初始化时机

- 除了上述七种情形，其他使用Java类的方式都被看作是被动使用，不会导致类的初始化

类的初始化时机

- 单击此处添加文本

（3）当 Java 虚拟机初始化一个类时，要求它的所有父类都已经被初始化，但是这条规则并不适用于接口。

- 在初始化一个类时，并不会先初始化它所实现的接口。
- 在初始化一个接口时，并不会先初始化它的父接口。

因此，一个父接口并不会因为它的子接口或者实现类的初始化而初始化。只有当程序首次使用特定接口的静态变量时，才会导致该接口的初始化。

类的初始化时机

- 只有当程序访问的静态变量或静态方法确实在当前类或当前接口中定义时，才可以认为是对类或接口的主动使用

类加载器

- 单击此处添加文本

类加载器用来把类加载到 Java 虚拟机中。从 JDK 1.2 版本开始，类的加载过程采用父亲委托机制，这种机制能更好地保证 Java 平台的安全。在此委托机制中，除了 Java 虚拟机自带的根类加载器以外，其余的类加载器都有且只有一个父加载器。当 Java 程序请求加载器 loader1 加载 Sample 类时，loader1 首先委托自己的父加载器去加载 Sample 类，若父加载器能加载，则由父加载器完成加载任务，否则才由加载器 loader1 本身加载 Sample 类。

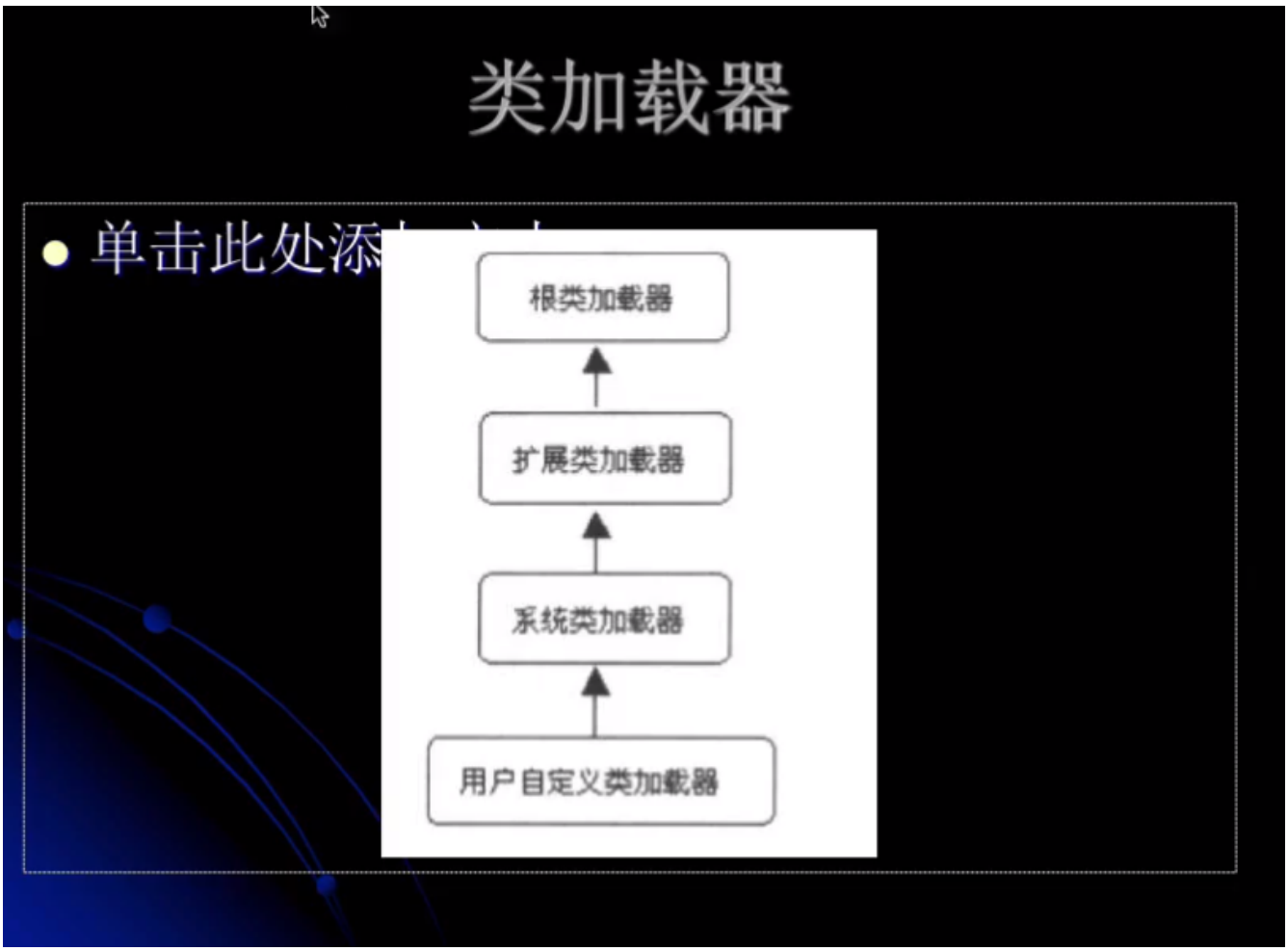
类加载器

Java 虚拟机自带了以下几种加载器。

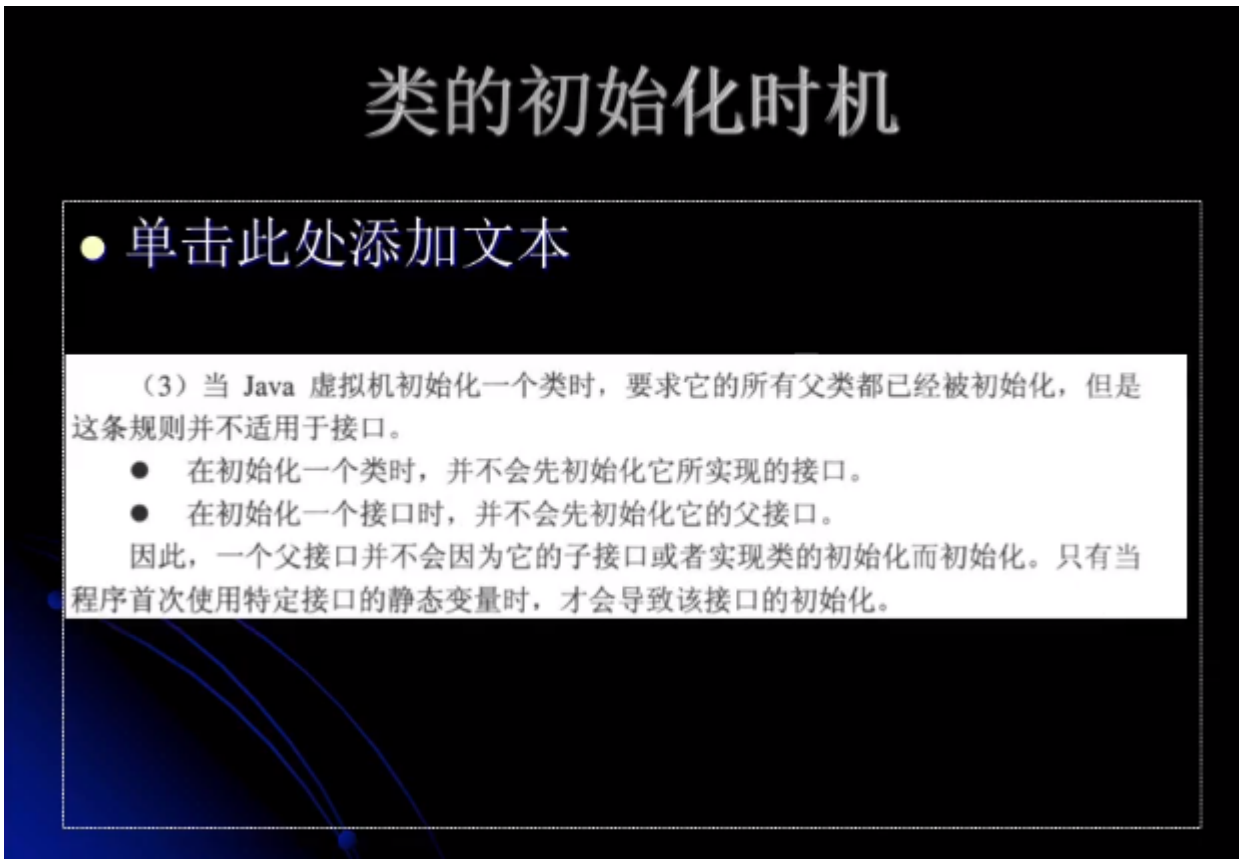
- 根（Bootstrap）类加载器：该加载器没有父加载器。它负责加载虚拟机的核心类库，如 java.lang.*等。例如从例程 10-4（Sample.java）可以看出，java.lang.Object 就是由根类加载器加载的。根类加载器从系统属性 sun.boot.class.path 所指定的目录中加载类库。根类加载器的实现依赖于底层操作系统，属于虚拟机的实现的一部分，它并没有继承 java.lang.ClassLoader 类。
- 扩展（Extension）类加载器：它的父加载器为根类加载器。它从 java.ext.dirs 系统属性所指定的目录中加载类库，或者从 JDK 的安装目录的 jre\lib\ext 子目录（扩展目录）下加载类库，如果把用户创建的 JAR 文件放在这个目录下，也会自动由扩展类加载器加载。扩展类加载器是纯 Java 类，是 java.lang.ClassLoader 类的子类。
- 系统（System）类加载器：也称为应用类加载器，它的父加载器为扩展类加载器。它从环境变量 classpath 或者系统属性 java.class.path 所指定的目录中加载类，它是用户自定义的类加载器的默认父加载器。系统类加载器是纯 Java 类，是 java.lang.ClassLoader 类的子类。

类加载器

- 除了以上虚拟机自带的加载器外，用户还可以定制自己的类加载器。Java提供了抽象类java.lang.ClassLoader，所有用户自定义的类加载器都应该继承ClassLoader类。



10 初始化对于类与接口的异同点深入分析。




```
30      public static int b = 5;
31  }
32
33  class C {
34
35      {
36          System.out.println("Hello");
37      }
38
39      public C() {
40          System.out.println("C");
41      }
42
43  }
```

定义一个非静态的代码块

如果是静态的，那么只会执行一次，否则每次实例化C都会执行一次代码块。

列一

```java
import java.util.Random;

public class MyTest5 {

    public static void main(String[] args) {
//        System.out.println(MyChild5.b);
        new C();
    }
}

interface MyParent5 {

    public static Thread thread = new Thread() {

        {
            System.out.println("MyParent5 invoked");
        }
    };

}

class MyChild5 implements MyParent5 {

    public static int b = 5;
}

class C {

    {
        System.out.println("Hello");
    }

    public C() {
        System.out.println("C");
    }

}
```

```
[Loaded java.net.SocksSocketImpl from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
[Loaded java.lang.Void from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
[Loaded java.net.AbstractPlainSocketImpl$1 from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
[Loaded com.shengsiyuan.jvm.classloader.C from file:/Users/zhanglong/myproject/jvm_lecture/out/production/classes/]
Hello
C
[Loaded java.lang.Shutdown from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
[Loaded java.lang.Shutdown$Lock from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
```

证明：实例化块先执行， 构造函数后执行。

列二：

```java
import java.util.Random;

public class MyTest5 {

    public static void main(String[] args) {
//        System.out.println(MyChild5.b);
        new C();
        new C();
    }
}
```

```
[Loaded sun.launcher.LauncherHelper$FXHelper from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/j
[Loaded java.lang.Class$MethodArray from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt
[Loaded java.lang.Void from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
[Loaded com.shengsiyuan.jvm.classloader.C from file:/Users/zhanglong/myproject/jvm_lecture/out/production/classes/]
Hello
C
Hello
C
[Loaded java.lang.Shutdown from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
```

每次new C都会执行代码块。这里new了两次。

改为static

列三：

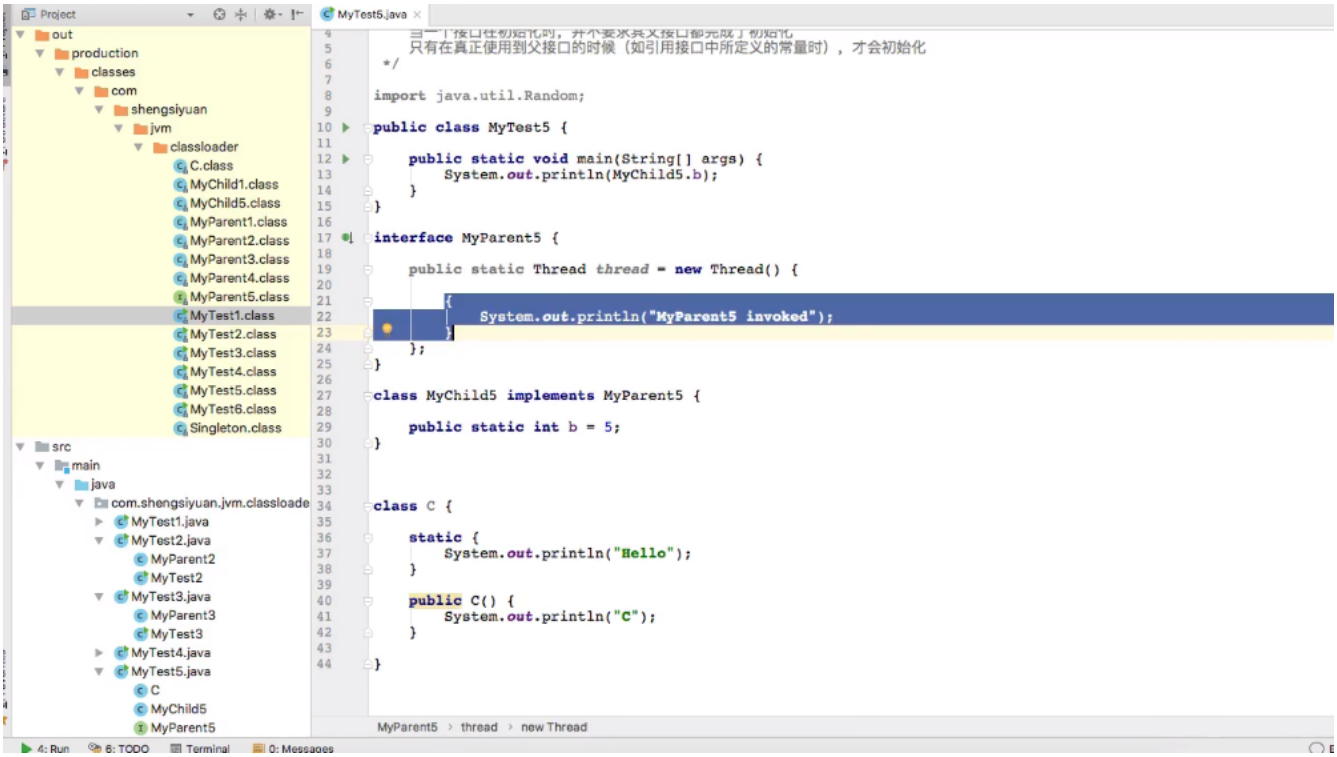
```
7
8   import java.util.Random;
9
10  public class MyTest5 {
11
12      public static void main(String[] args) {
13  //          System.out.println(MyChild5.b);
14          new C();
15          new C();
16      }
17  }
18
19  interface MyParent5 {
20
21      public static Thread thread = new Thread() {
22
23          {
24              System.out.println("MyParent5 invoked");
25          }
26      };
27
28  }
29
30  class MyChild5 implements MyParent5 {
31
32      public static int b = 5;
33  }
34
35  class C {
36
37      static {
38          System.out.println("Hello");
39      }
40
41      public C() {
42          System.out.println("C");
43      }
44
45  }
```

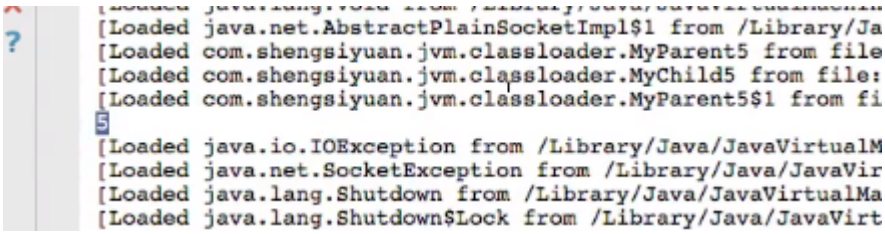
```
[Loaded sun.launcher.LauncherHelper$FXHelper from /L
[Loaded java.lang.Class$MethodArray from /Library/Ja
[Loaded java.net.AbstractPlainSocketImpl$1 from /Lib
[Loaded java.lang.Void from /Library/Java/JavaVirtua
[Loaded com.shengsiyuan.jvm.classloader.C from file:
Hello
C1
C
[Loaded java.lang.Shutdown from /Library/Java/JavaVi
[Loaded java.lang.Shutdown$Lock from /Library/Java/J

Process finished with exit code 0

4: Run   6: TODO   Terminal   0: Messages
Compilation completed successfully in 1s 363ms (moments ago)
```

静态代码块只会执行一次。

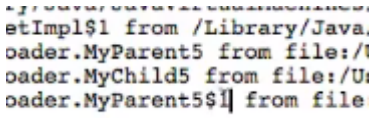
如果对thread初始化，那么里面的代码块一定会被执行一次。这样就能证明MyParent5是否被初始化了。

注意：如果是**准备**阶段，只会分内存，赋默认的值，如null,而**初始化**就是赋正确的初始值。

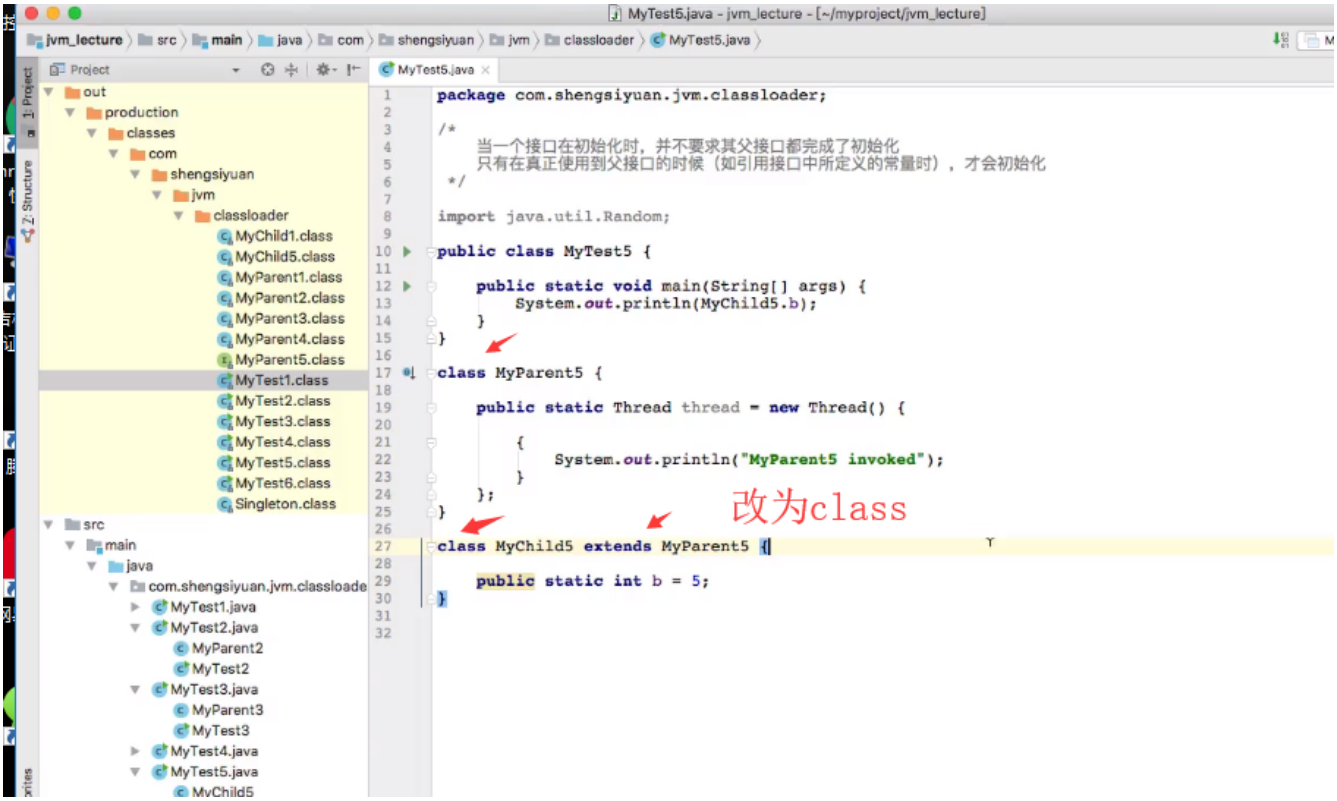


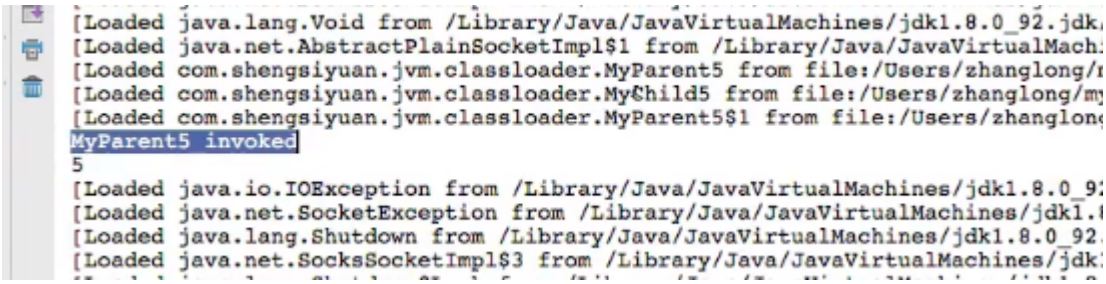


说明：接口并没有被初始化。



但是加载了。





证明：当一个类被初始化的时候，他实现的类是不会被初始化的，而他的父类会被初始化。

```java
/*
    只有在真正使用到父接口的时候（如引用接口中所定义的常量时），才会初始化
 */

import java.util.Random;

public class MyTest5 {

    public static void main(String[] args) {
        System.out.println(MyChild5.b);
    }
}

interface MyGrandpa {

    public static Thread thread = new Thread() {

        {
            System.out.println("MyGrandpa invoked");
        }
    };
}

interface MyParent5 extends MyGrandpa{

    public static Thread thread = new Thread() {

        {
            System.out.println("MyParent5 invoked");
        }
    };
}

class MyChild5 implements MyParent5 {

    public static final int b = 5;
}
```

```
[Loaded java.lang.Class$MethodArray from /Library/
[Loaded java.net.AbstractPlainSocketImpl$1 from /L
[Loaded java.lang.Void from /Library/Java/JavaVirt
5
[Loaded java.lang.Shutdown from /Library/Java/Java
```

证明：当一个类被初始化的时候，他实现的类是不会被初始化的，而他的父类会被初始化。

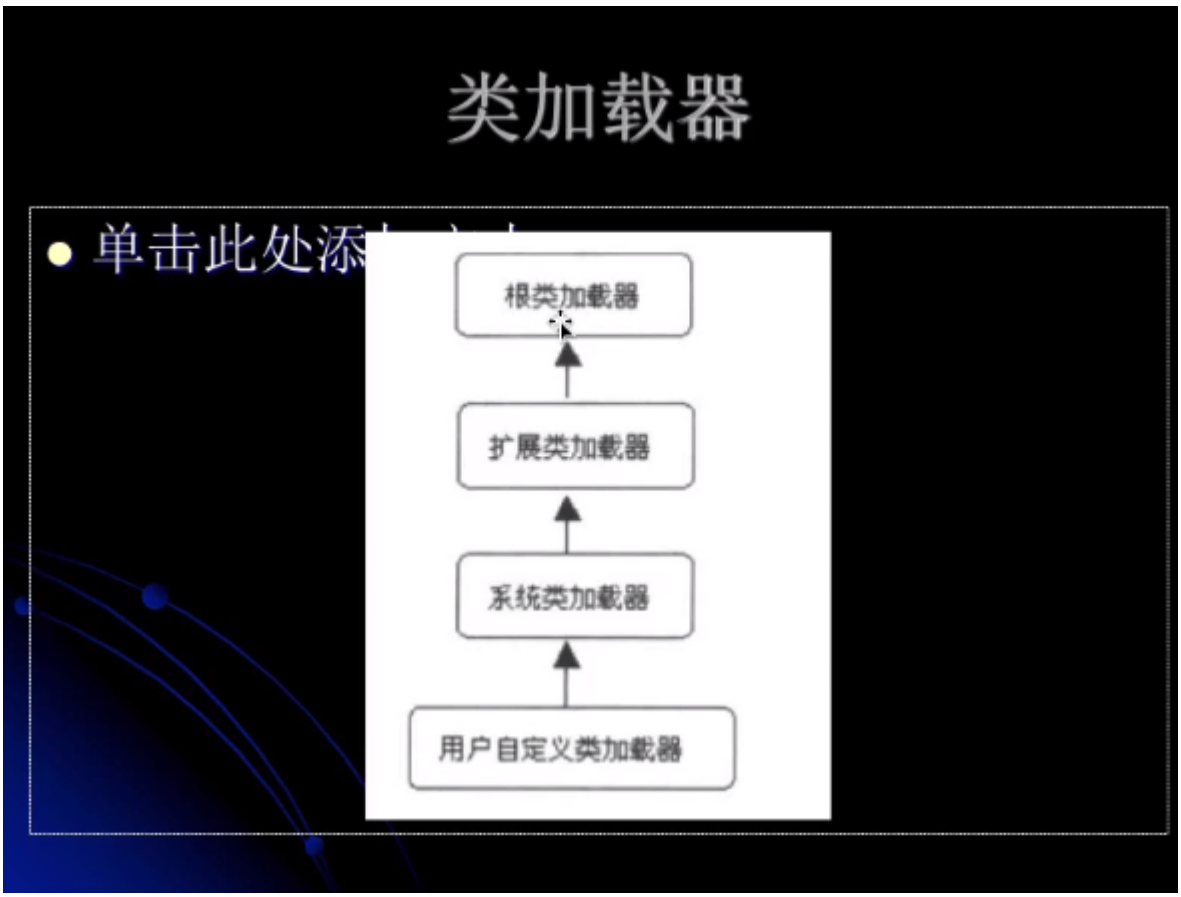
```java
public class MyTest5 {

    public static void main(String[] args) {
//        System.out.println(MyChild5.b);
        System.out.println(MyParent5_1.thread);
    }
}
```

```java
interface MyGrandpa5_1 {

    public static Thread thread = new Thread() {

        {
            System.out.println("MyGrandpa5_1 invoked");
        }
    };
}

interface MyParent5_1 extends MyGrandpa5_1 {

    public static Thread thread = new Thread() {

        {
            System.out.println("MyParent5_1 invoked");
        }
    };
}
```

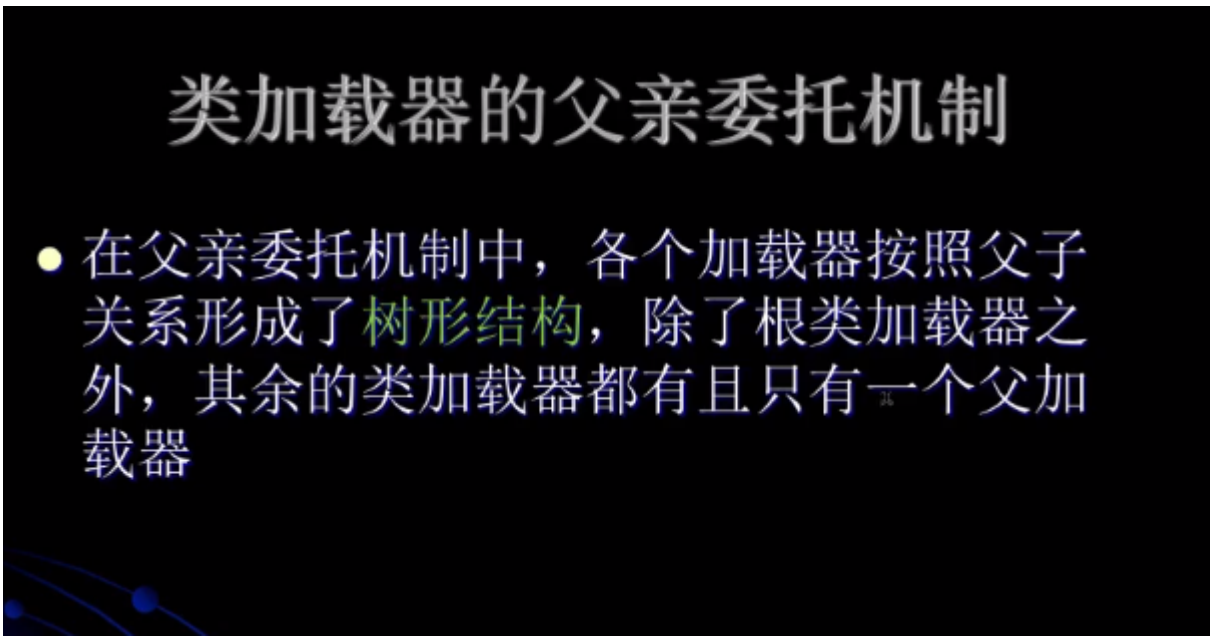
```
[Loaded sun.net.spi.DefaultProxySelector from /Library/Java/JavaVi
[Loaded sun.net.spi.DefaultProxySelector$1 from /Library/Java/Java
[Loaded com.shengsiyuan.jvm.classloader.MyParent5_1$1 from file:/U
[Loaded sun.net.NetProperties from /Library/Java/JavaVirtualMachin
[Loaded sun.net.NetProperties$1 from /Library/Java/JavaVirtualMach
MyParent5_1 invoked
Thread[Thread-0,5,main]
[Loaded java.util.Properties$LineReader from /Library/Java/JavaVir
```

证明：初始化一个接口的时候并不会初始化其父接口。

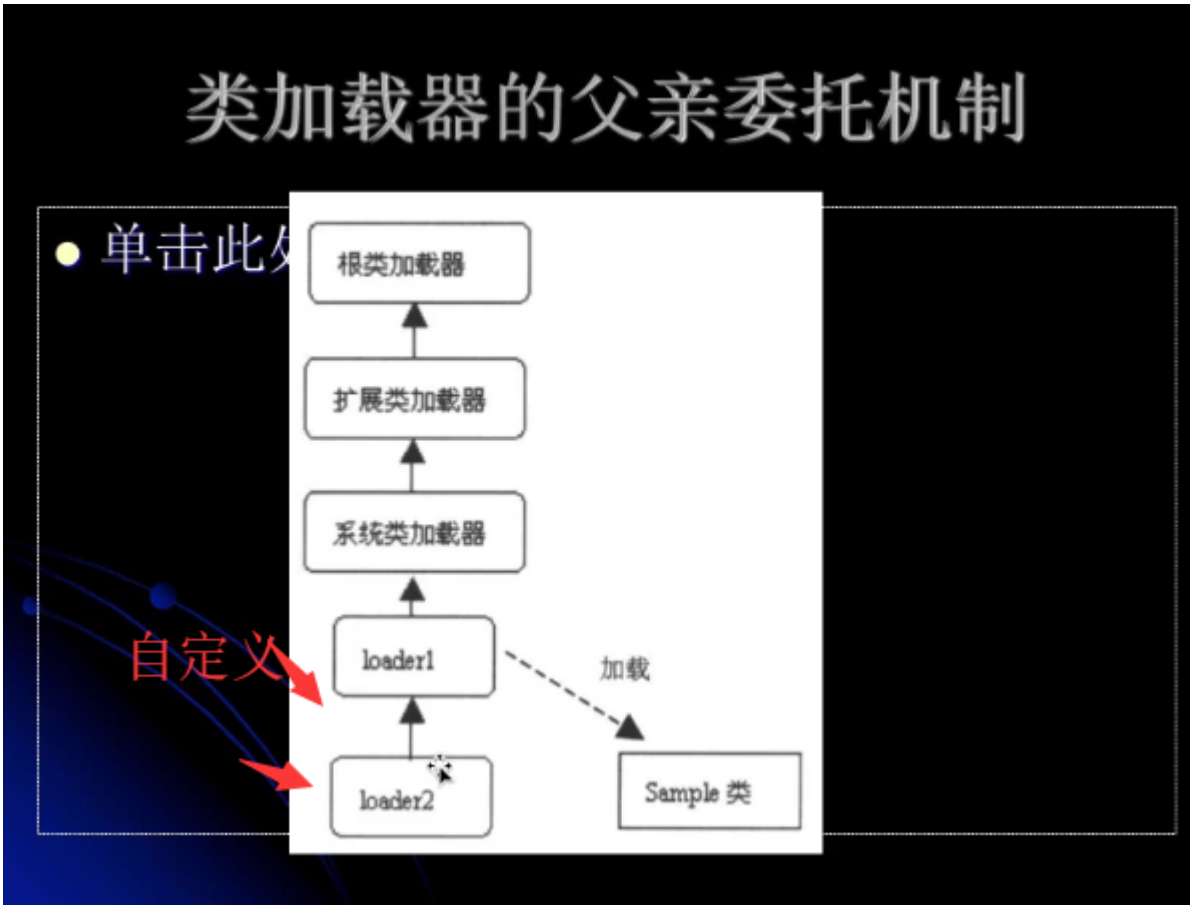
11 类加载器双亲委托机制详解



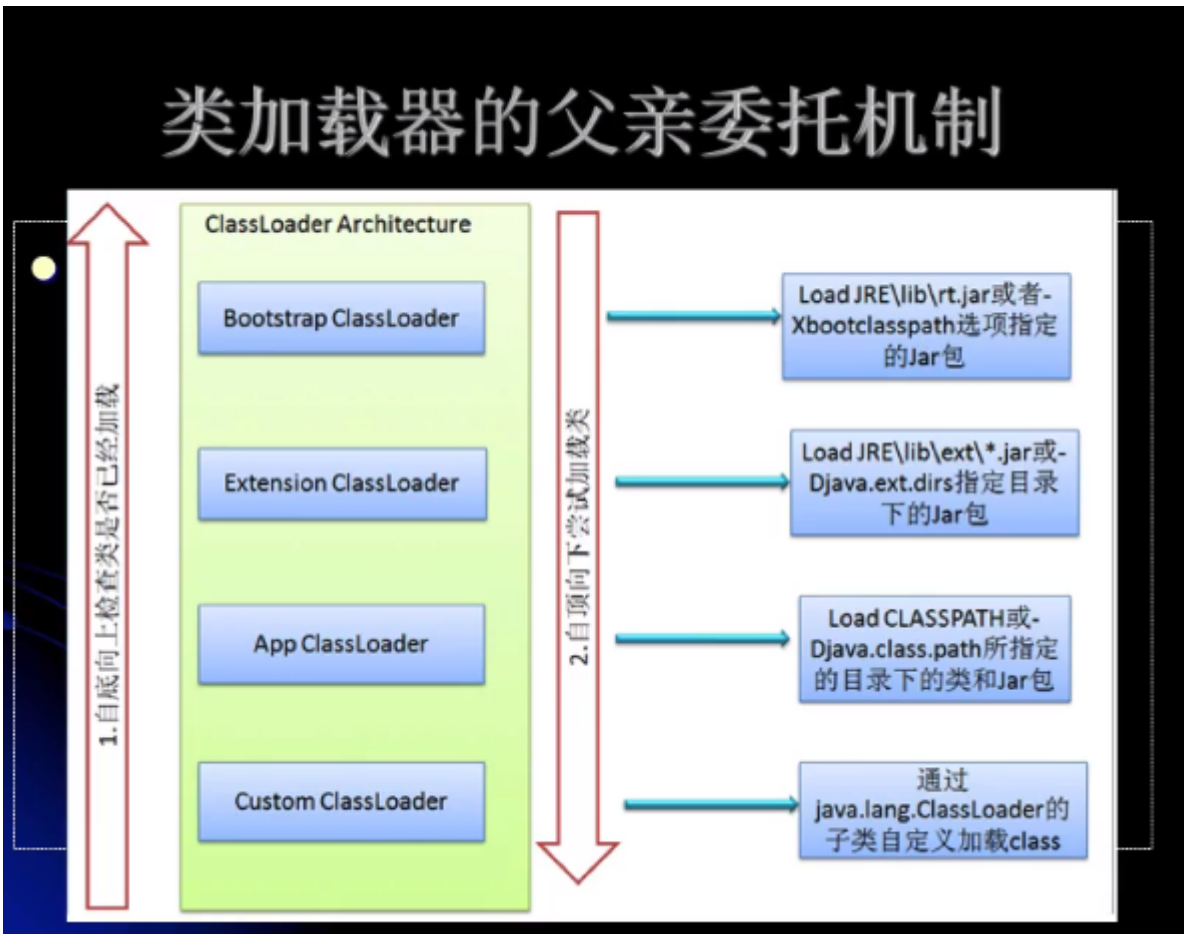
启动类加载器非java实现。



类加载器的父亲委托机制

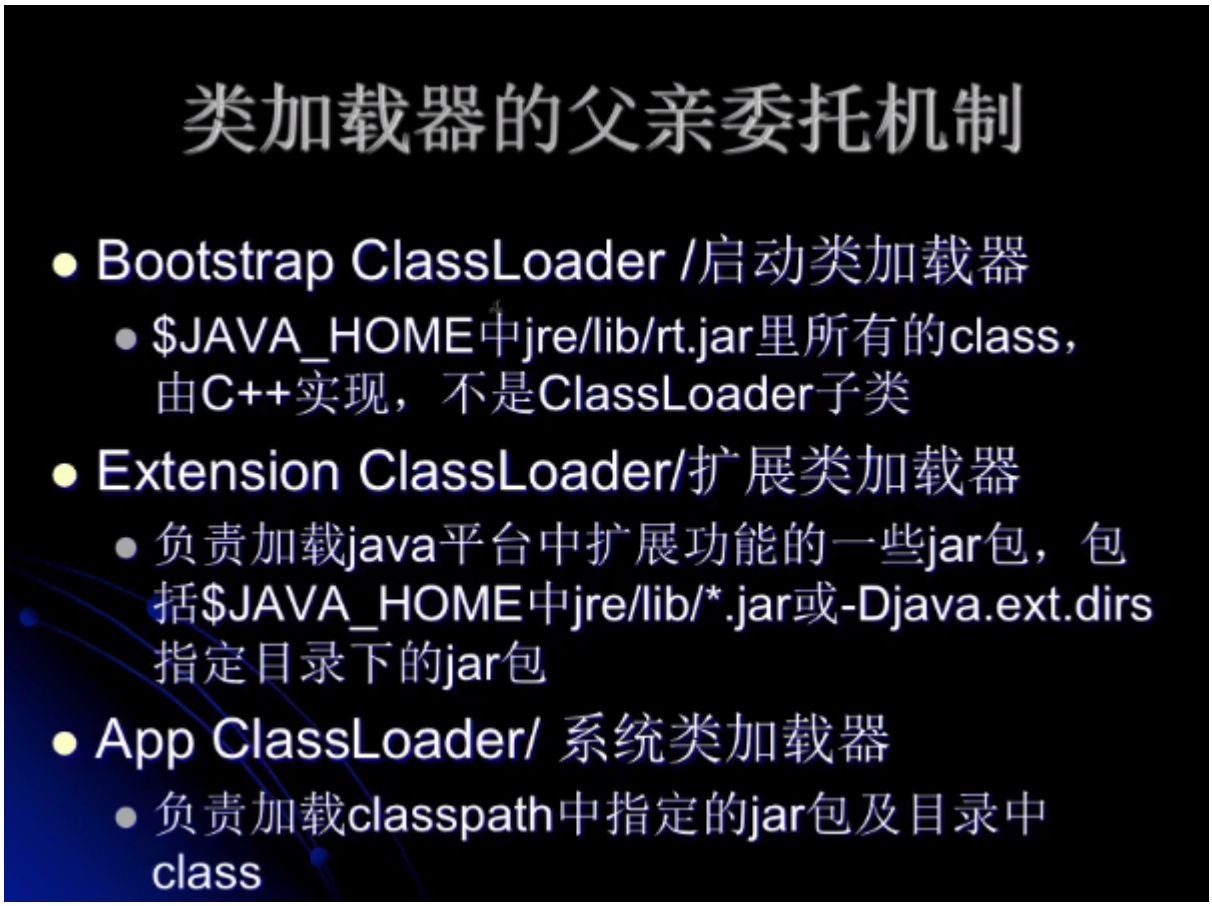


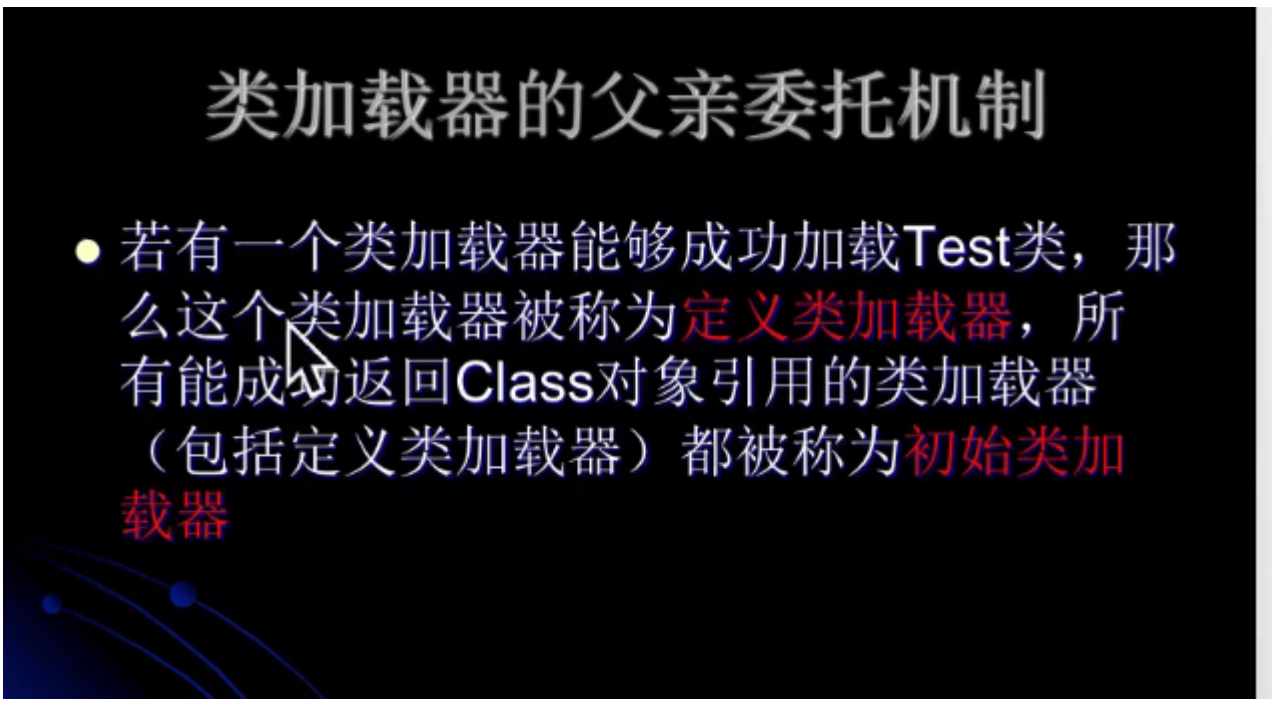
类加载器的父亲委托机制



有一个类需要类加载器去加载，如果有父类，先让父类去加载，如此向上追溯，知道根类加载器，然后根类加载器尝试去加载，加载成功则结束，加载失败，又往下，一层层的尝试去加载，最终如果都没有加载成功，则报错classnotfound；

但是并不是所有的jvm都是这样，hotspot遵循这样规则。



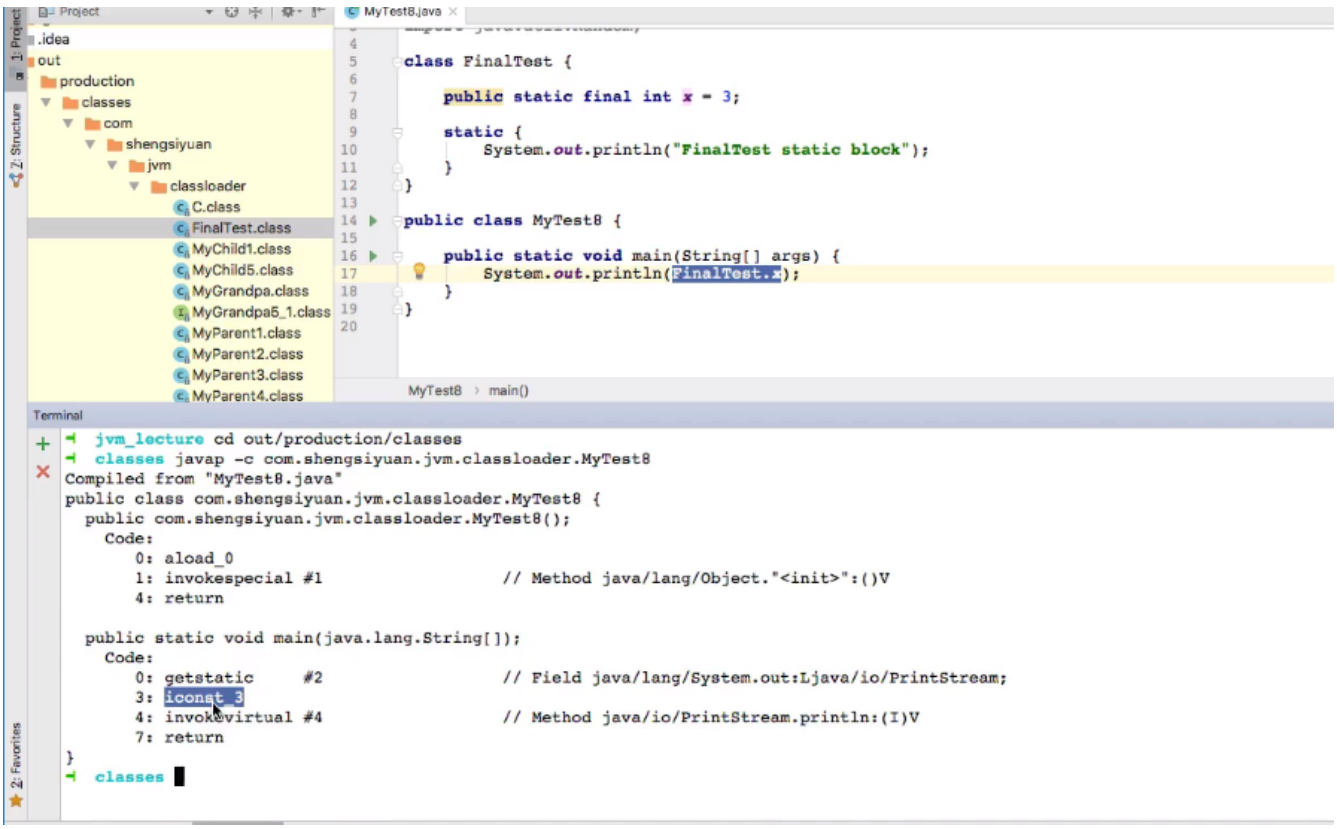


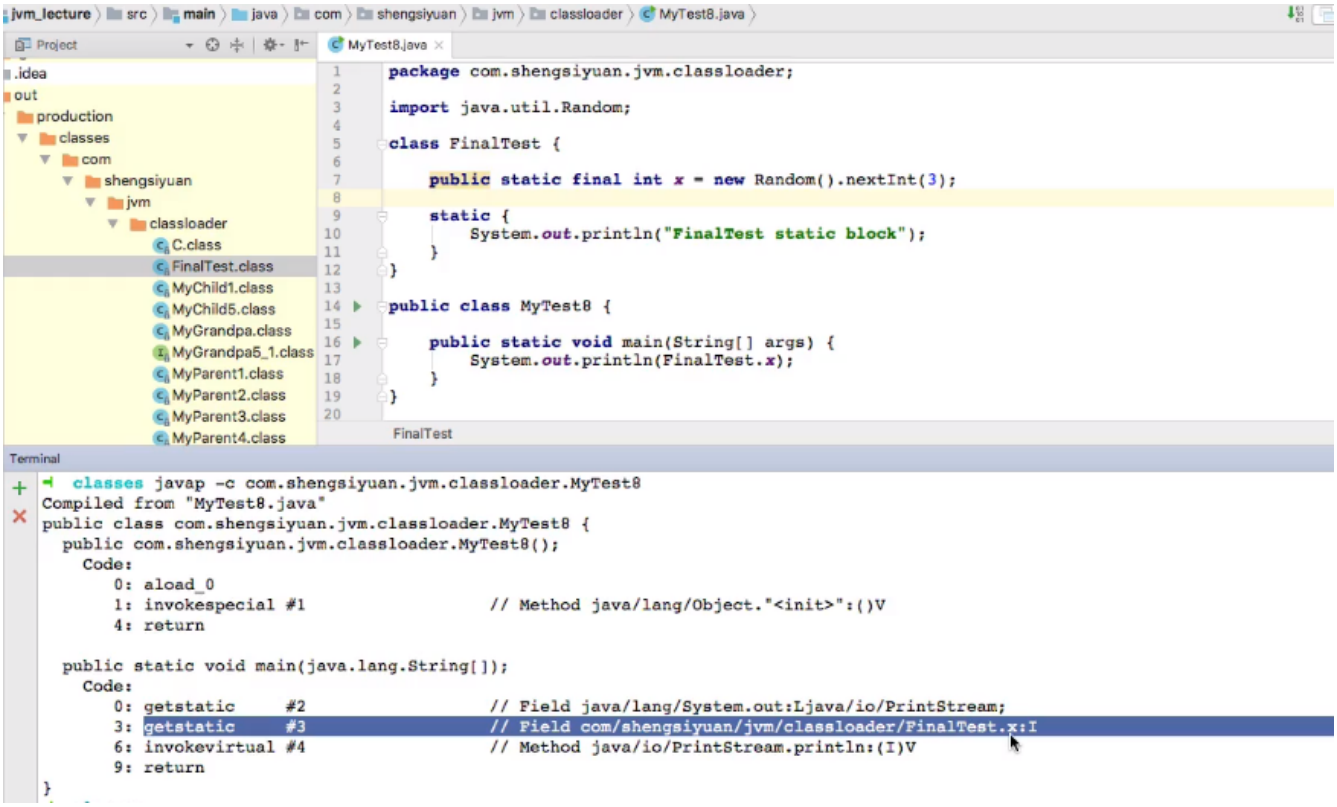
Test类为需要加载的目标类。

例子：

12 类加载器与类初始化的深度剖析

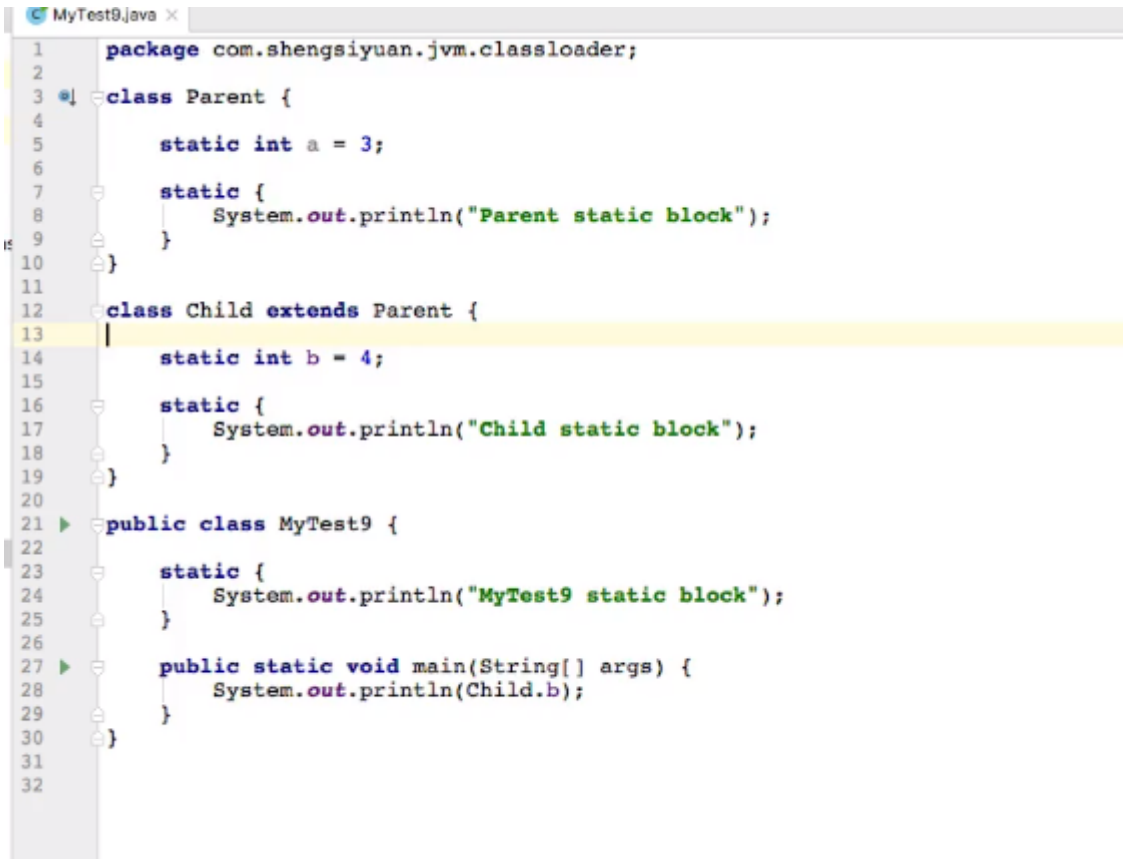
例1：

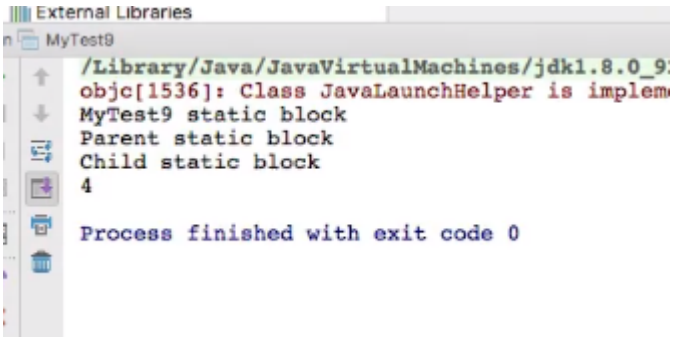




证明：因此有一个静态的引用。

例2：






```
ra.io.FileOutputStream$1 from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
ı.net.util.IPAddressUtil from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
ra.net.Inet4Address from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
ı.launcher.LauncherHelper from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
ı.misc.URLClassPath$FileLoader$1 from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
ra.net.SocksConsts from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
ra.net.SocketOptions from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
ı.shengsiyuan.jvm.classloader.MyTest9 from file:/Users/zhanglong/myproject/jvm_lecture/out/production/classes/]
ra.net.SocketImpl from /Li                    /JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
ra.net.AbstractPlainSocketImpl fro            ibrary/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
ra.net.PlainSocketImpl from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
ra.net.SocksSocketImpl from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
ı.launcher.LauncherHelper$FXHelper from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
ra.lang.Class$MethodArray from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
ra.net.AbstractPlainSocketImpl$1 from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
ra.lang.Void from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
ıtic block
ı.shengsiyuan.jvm.classloader.Parent from file:/Users/zhanglong/myproject/jvm_lecture/out/production/classes/]
ı.shengsiyuan.jvm.classloader.Child from file:/Users/zhanglong/myproject/jvm_lecture/out/production/classes/]
:ic block
:c block

ra.io.IOException from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
ra.net.SocketException from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
ra.lang.Shutdown from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
ra.net.SocksSocketImpl$3 from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
ra.lang.Shutdown$Lock from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
ra.net.ProxySelector from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
ı.net.spi.DefaultProxySelector from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar]
```

初始化顺序：MyTest9 先被初始化。父类再被初始化，最后是子类被初始化。**

加载顺序 MyTest->Parent->Child

例子3：

MyTest10.java

```java
package com.shengsiyuan.jvm.classloader;

class Parent2 {

    static int a = 3;

    static {
        System.out.println("Parent2 static block");
    }
}

class Child2 extends Parent2 {

    static int b = 4;

    static {
        System.out.println("Child 2 static block");
    }
}

public class MyTest10 {

    static {
        System.out.println("MyTest10 static block");
    }

    public static void main(String[] args) {
        Parent2 parent2;

        System.out.println("----------");

        parent2 = new Parent2();

        System.out.println("----------");

        System.out.println(parent2.a);

        System.out.println("----------");

        System.out.println(Child2.b);
    }
}
```

MyTest10

Run MyTest10

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Conten
objc[1541]: Class JavaLaunchHelper is implemented in bot
MyTest10 static block
----------
Parent2 static block
----------
3
----------
Child 2 static block
4

Process finished with exit code 0
```

首先访问main方法：导致MyTest10 static block 输出

声明Parent2不会产生对其的主动使用，因而不会输出Parent2的静态代码块

new Parent2 根据规则1：创建类的实例，产生一个首次主动使用，输出了静态代码块

Child.b: 根据规则2：对类的静态变量的访问或者赋值，产生一个首次主动使用，输出了Child2的静态代码块，但是

但是不会输出Parent2的静态代码块，因为此时他不是首次主动使用了。

例子4：

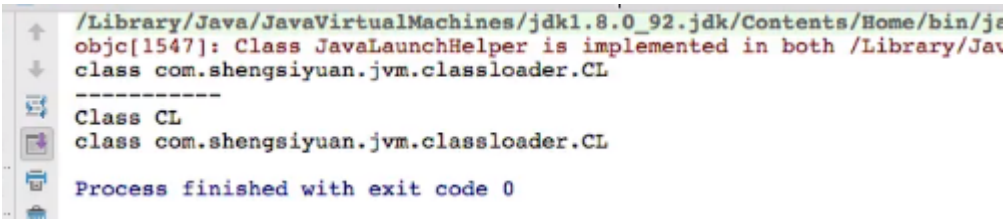
```java
package com.shengsiyuan.jvm.classloader;

class Parent3 {

    static int a = 3;

    static {
        System.out.println("Parent3 static block");
    }

    static void doSomething() {
        System.out.println("do something");
    }
}

class Child3 extends Parent3 {

    static {
        System.out.println("Child3 static block");
    }
}

public class MyTest11 {

    public static void main(String[] args) {
        System.out.println(Child3.a);
        Child3.doSomething();
    }
}
```

```
MyTest11
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Ho
objc[1545]: Class JavaLaunchHelper is implemented in both /Li
Parent3 static block
3
-------
do something

Process finished with exit code 0
```

解释：没有使用到Child的任何数据，没有对其产生一个主动使用。

总结：通过子类的名称，调用父类的静态方法，**本质上都是对父类的主动使用**，而非子类。

例子5：



13不同的类加载器作用与动作分析

1.隐式装载， 程序在运行过程中当碰到通过new 等方式生成对象时，隐式调用类装载器加载对应的类到jvm中。

2.显式装载， 通过class.forname()等方法，显式加载需要的类

类加载的动态性体现:

一个应用程序总是由n多个类组成，Java程序启动时，并不是一次把所有的类全部加载后再运行，它总是先把保证程序运行的基础类一次性加载到jvm中，其它类等到jvm用到的时候再加载，这样的好处是节省了内存的开销，因为java最早就是为嵌入式系统而设计的，内存宝贵，这是一种可以理解的机制，而用到时再加载这也是java动态性的一种体现

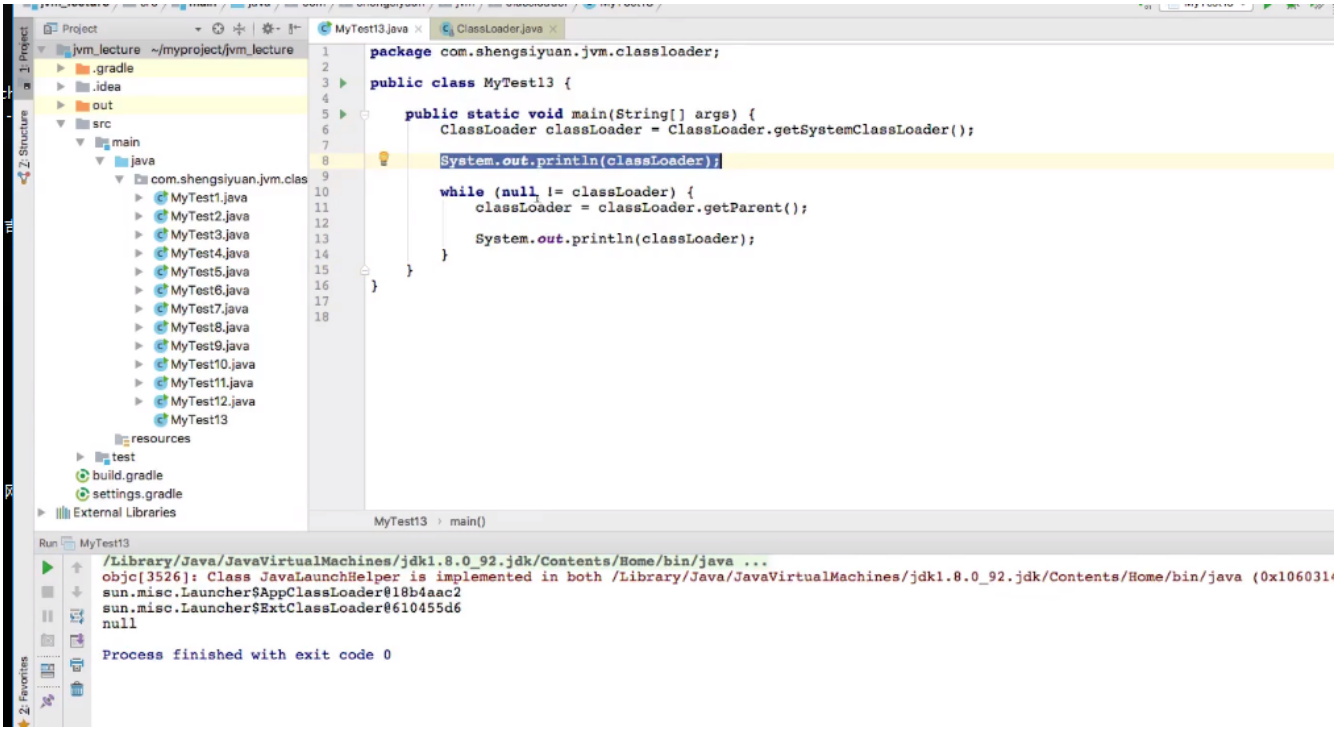
-
  -
    - `public static ClassLoader getSystemClassLoader()`

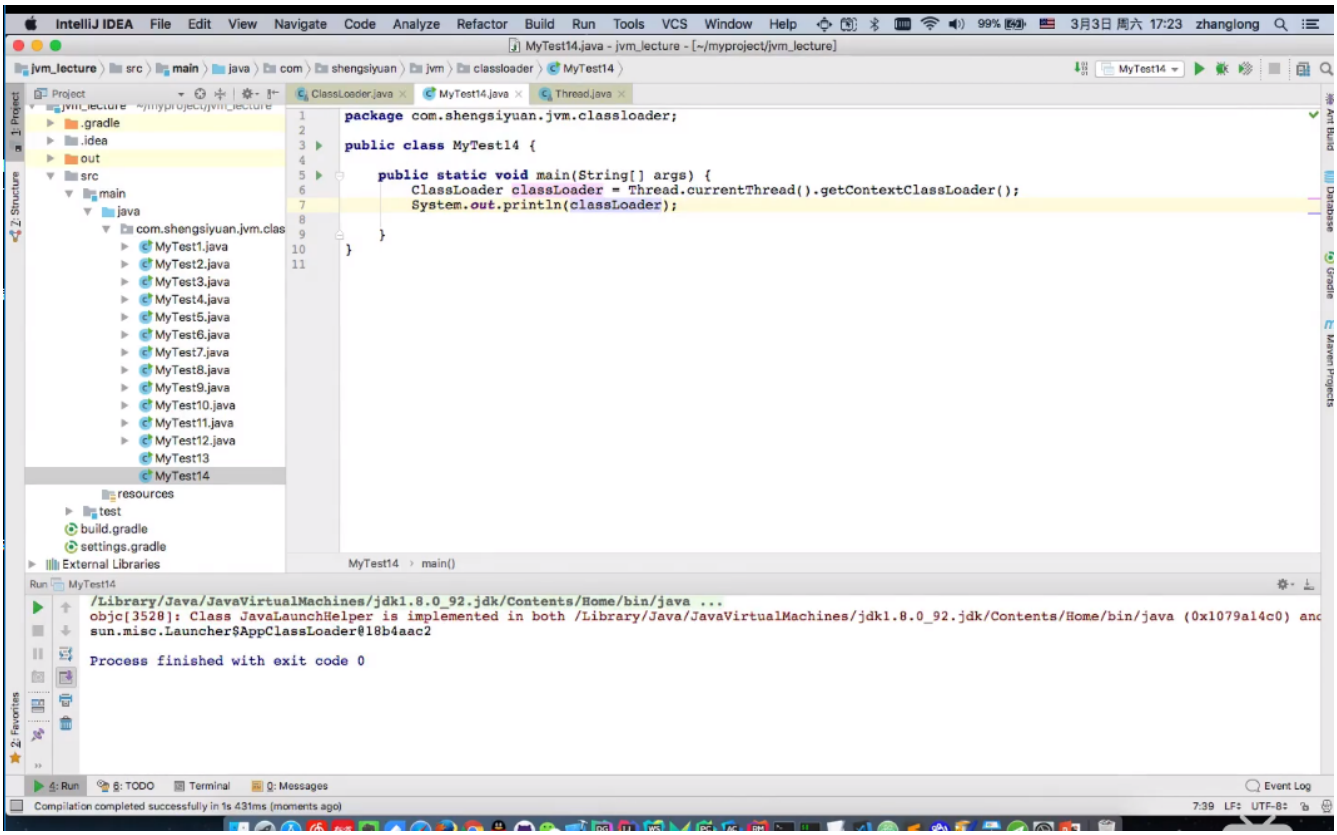
返回用于委派的系统类加载器

-
  - `getParent()` 返回父类加载器进行委派。

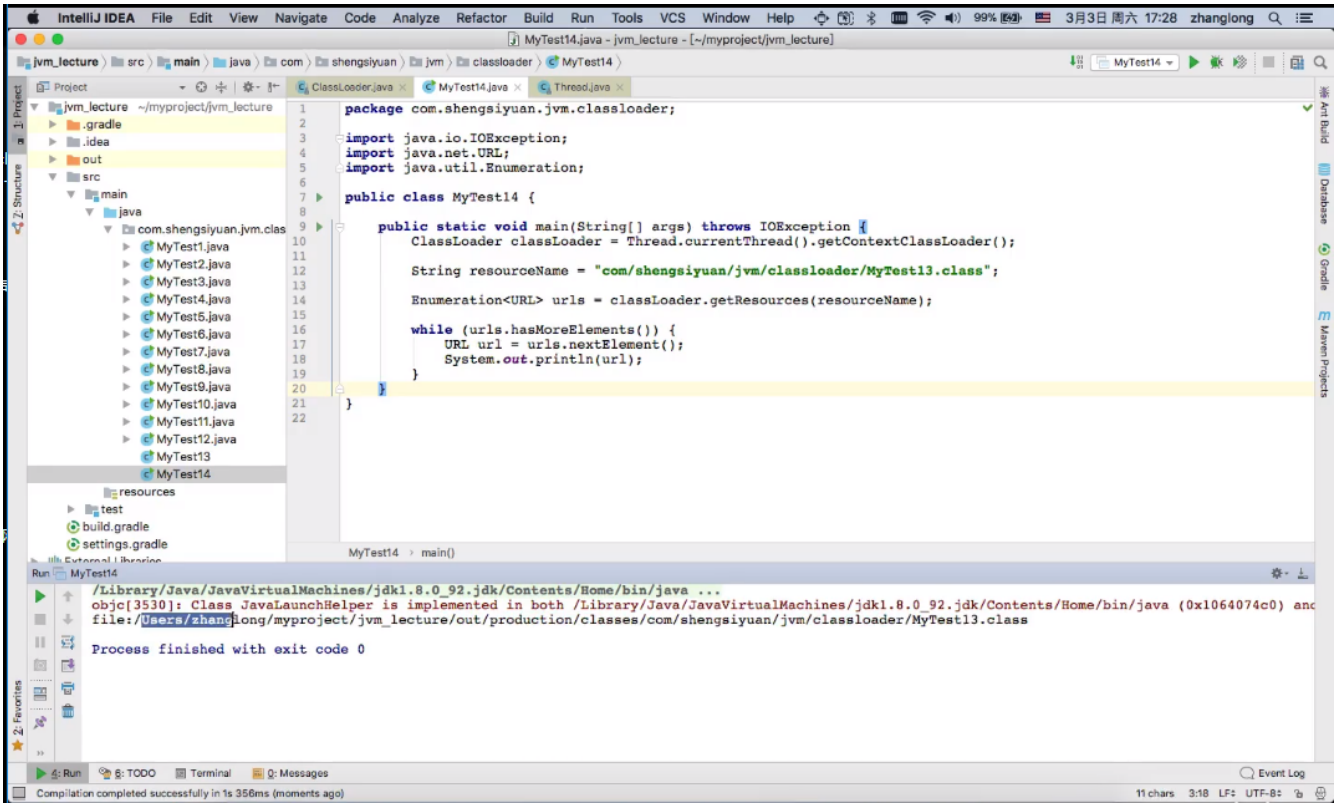
例1

获取类加载器:


MyTest13.java
ClassLoader.java
package com.shengsiyuan.jvm.classloader;
public class MyTest13 {
public static void main(String[] args) {
ClassLoader classLoader = ClassLoader.getSystemClassLoader();
System.out.println(classLoader);
while (null != classLoader) {
classLoader = classLoader.getParent();
System.out.println(classLoader);
}
}
}
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
objc[3526]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (0x106031
sun.misc.Launcher$AppClassLoader@18b4aac2
sun.misc.Launcher$ExtClassLoader@610455d6
null
Process finished with exit code 0


IntelliJ IDEA File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
MyTest14.java - jvm_lecture - [~/myproject/jvm_lecture]
ClassLoader.java
MyTest14.java
Thread.java
package com.shengsiyuan.jvm.classloader;
public class MyTest14 {
public static void main(String[] args) {
ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
System.out.println(classLoader);
}
}
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
objc[3528]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (0x1079a14c0) and
sun.misc.Launcher$AppClassLoader@18b4aac2
Process finished with exit code 0
Compilation completed successfully in 1s 431ms (moments ago)

获取资源：



14 ClassLoader源码分析与实例剖析 待看：

Binary names

Any class name provided as a String parameter to methods in ClassLoader must be a binary name as defined by *The Java™ Language Specification*.

Examples of valid class names include:

```
"java.lang.String"
"javax.swing.JSpinner$DefaultEditor"
"java.security.KeyStore$Builder$FileBuilder$1"
"java.net.URLClassLoader$3$1"
```

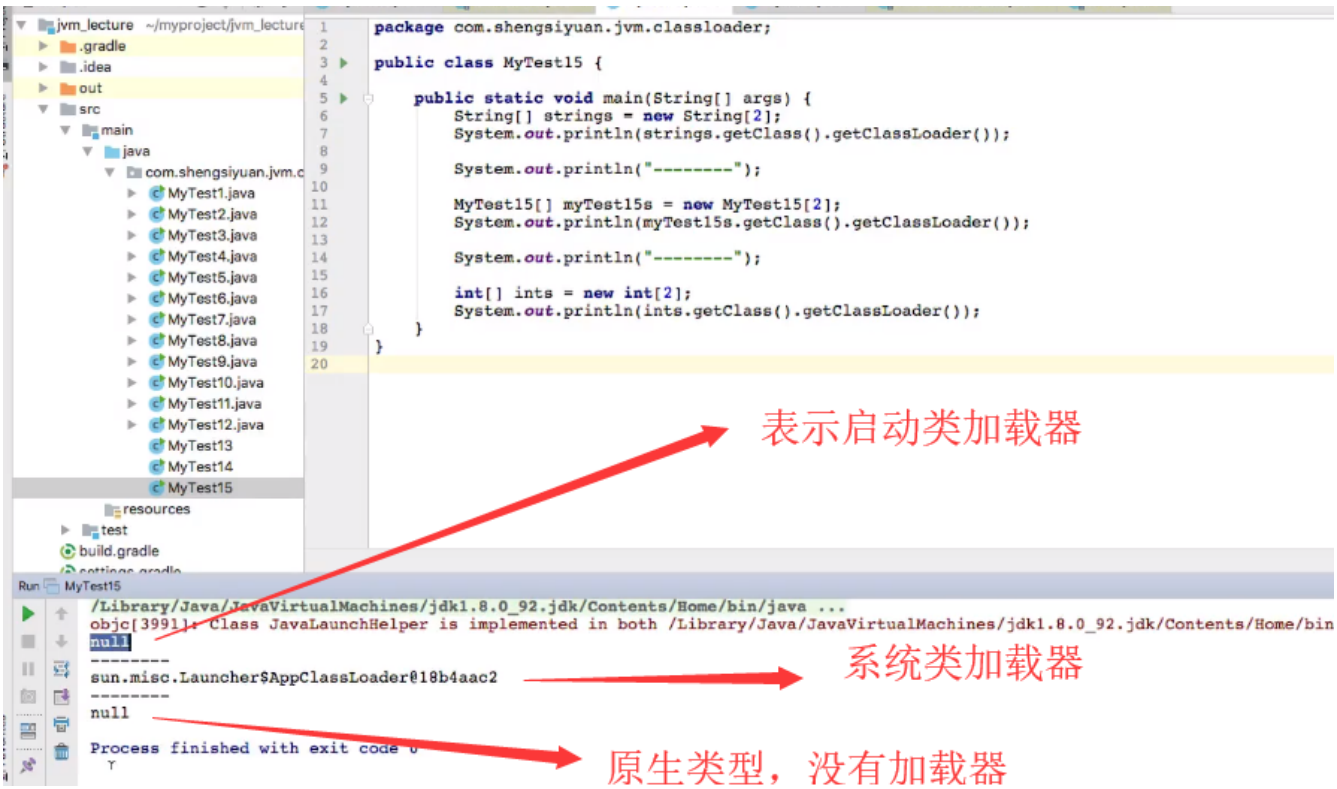
$表示一个内部类，1表示匿名内部类

Since:
1.0

数字标识匿名内部类在代码中的序号，即第几个匿名内部类。

动态代理：类是在运行期间生成的，所以不能用定位的方式，而是generate data；

A class loader is an object that is responsible for loading classes. The class ClassLoader is an abstract class. Given the binary name of a class, a class loader should attempt to locate or generate data that constitutes a definition for the class. A typical strategy is to transform the name into a file name and then read a "class file" of that name from a file system.
Every Class object contains a reference to the ClassLoader that defined it.
Class objects for array classes are not created by class loaders, but are created automatically as required by the Java runtime. The class loader for an array class, as returned by Class.getClassLoader() is the same as the class loader for its element type; if the element type is a primitive type, then the array class has no class loader.



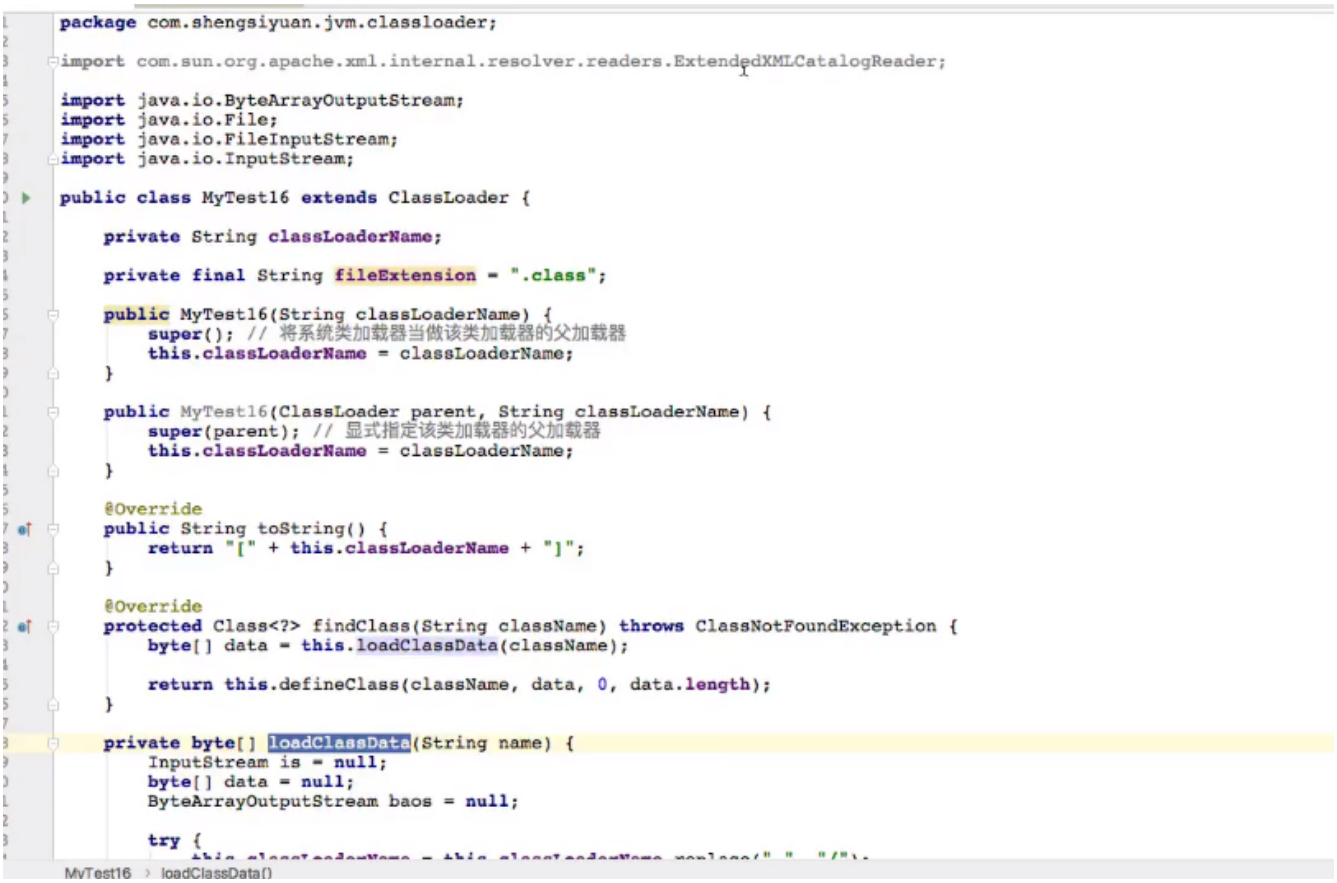
使用intellij显示javadoc文档

https://blog.csdn.net/u013905744/article/details/73162294

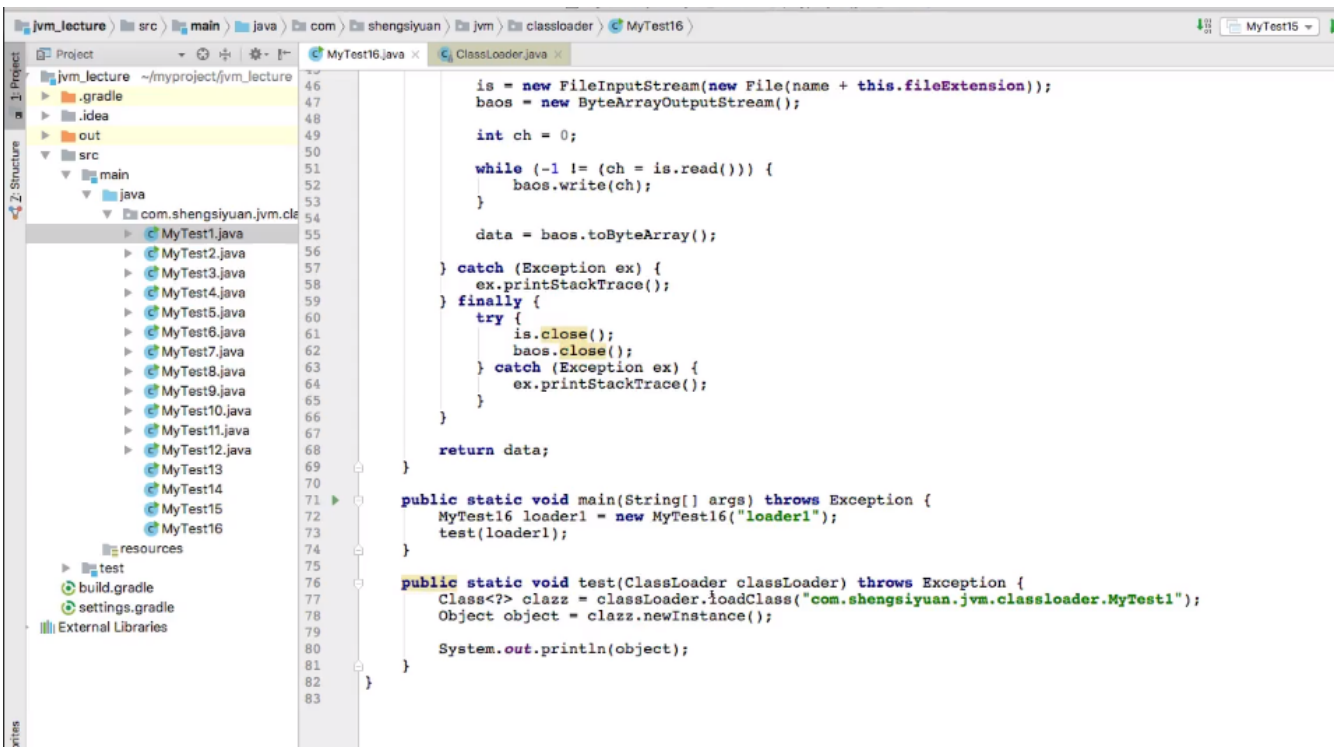
15 自定义类加载器深入详解

The method defineClass converts an array of bytes into an instance of class Class. Instances of this newly defined class can be created using Class.newInstance.

自定义类加载器：



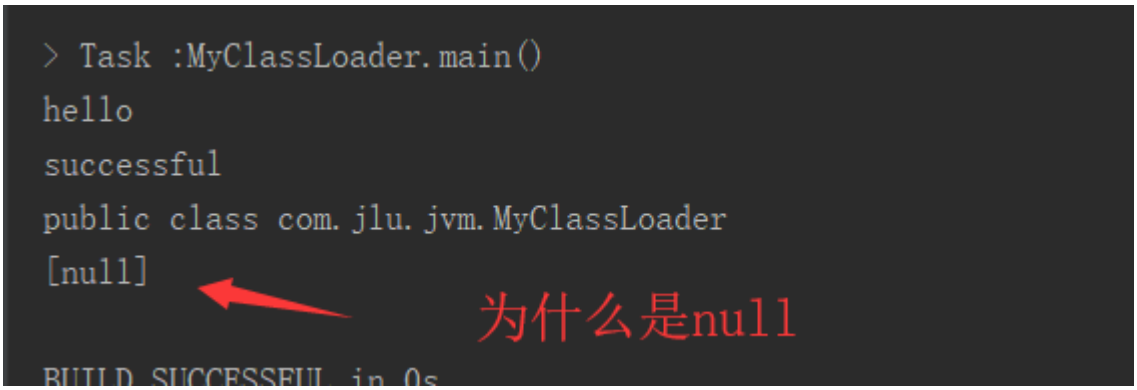
```
    }

    private byte[] loadClassData(String name) {
        InputStream is = null;
        byte[] data = null;
        ByteArrayOutputStream baos = null;

        try {
            this.classLoaderName = this.classLoaderName.replace(".", "/");

            is = new FileInputStream(new File(name + this.fileExtension));
            baos = new ByteArrayOutputStream();

            int ch = 0;

            while (-1 != (ch = is.read())) {
                baos.write(ch);
            }

            data = baos.toByteArray();

        } catch (Exception ex) {
            ex.printStackTrace();
        } finally {
            try {
                is.close();
                baos.close();
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }

        return data;
    }
}
```




```
package com.jlu.jvm;

import java.io.ByteArrayOutputStream;
```

```
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;

public class MyClassLoader extends ClassLoader {
    private String classLoaderName;
    private final String fileExtension=".class";
    public MyClassLoader(){
        super();
    }
    //将系统类加载器作为该类的父类加载器
    public MyClassLoader( String classLoader){
        super();
        this.classLoaderName=classLoaderName;
    }
    //指定一个父类加载器。
    public MyClassLoader(ClassLoader parent,String classLoaderName){
        super(parent);
        this.classLoaderName=classLoaderName;
    }
    @Override
    public String toString(){
        return "["+this.classLoaderName+"]";
    }
    private byte[] loadClassData(String name){
        InputStream is=null;
        byte[] data=null;
        ByteArrayOutputStream baos=null;
        try{
            //转义符号\
            this.classLoaderName=this.classLoaderName.replace(".","\\");
            is=new  FileInputStream(new File(this.classLoaderName+this.fileExtension));
            baos=new ByteArrayOutputStream();
            int ch=0;
            while(-1!=(ch=is.read())){
                baos.write(ch);
            }
            data=baos.toByteArray();
        }catch (Exception e){
            e.printStackTrace();
        }finally{
            try{
                is.close();
                baos.close();
            }catch(Exception e){
                e.printStackTrace();
            }
        }
        return data;
    }
    @Override
    protected  Class<?> findClass(String className)throws ClassNotFoundException{
        byte [] data=this.loadClassData(className);
        return this.defineClass(className,data,0,data.length);
    }
    public static void main(String [] args)throws Exception{
        MyClassLoader loader1=new MyClassLoader("loader1");
        test(loader1);
    }
    public static void test(ClassLoader classLoader)throws Exception{
```

```
        System.out.println("hello");
        Class<?>clazz=classLoader.loadClass("com.jlu.jvm.MyClassLoader");
        System.out.println("successful");
        System.out.println(clazz.toGenericString());
        Object object=clazz.newInstance();
        System.out.println(object);
    }
}
```



解释：因为真正的加载这个类的是其父类所以，没有调用到自己定义的方法loadClassData，

newInstance 需要对应的类存在默认构造方法，但是视频中的确实可以使用？？？？不影响

不对，不是这样，但是好像要类加载器有构造方法。

16 类加载器重要方法详解：

17 类加载器双亲委托机制实例深度剖析

Class 对象有一个方法：getClassLoader();获取类加载器；

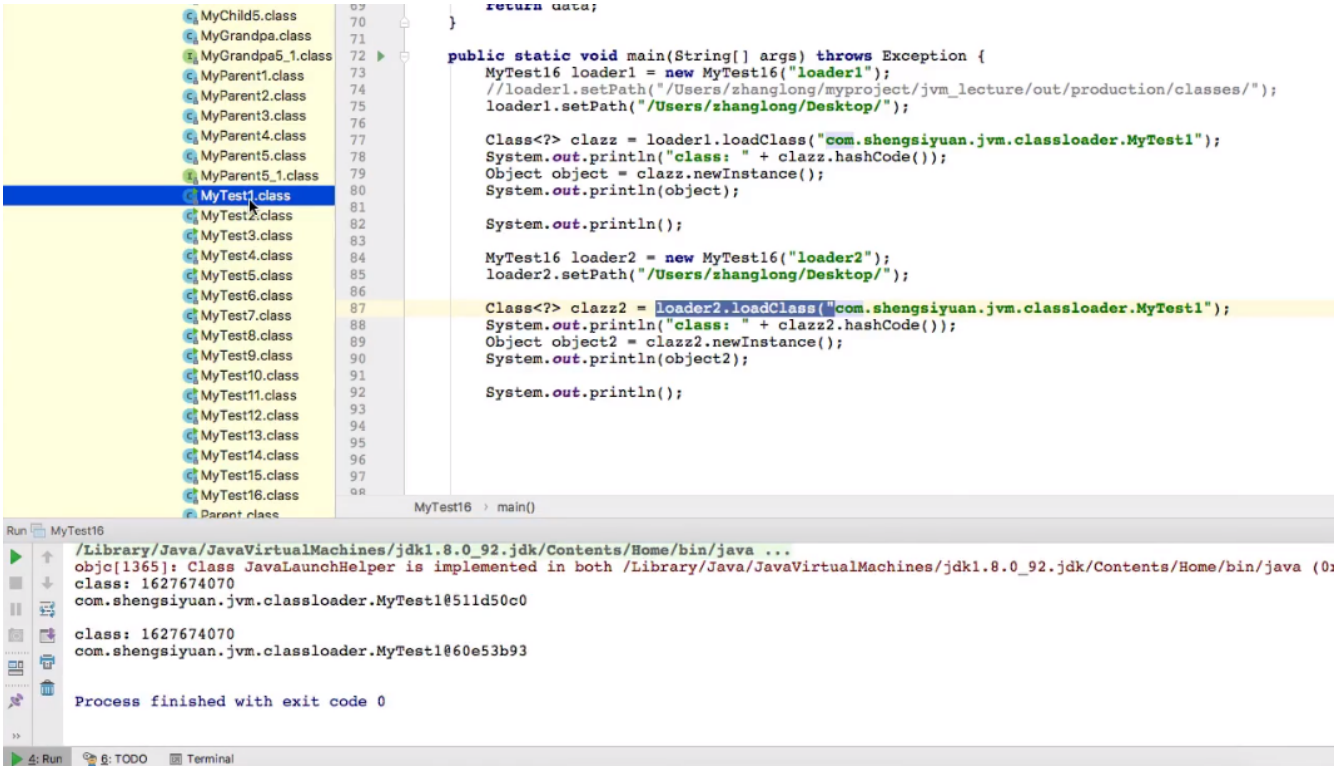
```
package com.jlu.jvm;


import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;

public class MyClassLoader extends ClassLoader {
    private String path;
    private String classLoaderName="MyClassLoader";
    private final String fileExtension=".class";
    public void setPath(String path){
        this.path=path;
    }
    public MyClassLoader(){
        super();
    }
    //将系统类加载器作为该类的父类加载器
    public MyClassLoader( String classLoader){
```

```
        super();
        this.classLoaderName=classLoaderName;
    }
    //指定一个父类加载器。
    public MyClassLoader(ClassLoader parent,String classLoaderName){
        super(parent);
        this.classLoaderName=classLoaderName;
    }
    @Override
    public String toString(){
        return "["+this.classLoaderName+"]";
    }
    private byte[] loadClassData(String name){
        InputStream is=null;
        byte[] data=null;
        ByteArrayOutputStream baos=null;
        try{
            //转义符号\
            name=name.replace(".","\\");
            // 使用绝对路径
            is=new  FileInputStream(new File(this.path+name+this.fileExtension));
            baos=new ByteArrayOutputStream();
            int ch=0;
            while(-1!=(ch=is.read())){
                baos.write(ch);
            }
            data=baos.toByteArray();
        }catch (Exception e){
            e.printStackTrace();
        }finally{
            try{
                is.close();
                baos.close();
            }catch(Exception e){
                e.printStackTrace();
            }
        }
        return data;
    }
    @Override
    protected  Class<?> findClass(String className)throws ClassNotFoundException{
        byte [] data=this.loadClassData(className);
        return this.defineClass(className,data,0,data.length);
    }
    public static void main(String [] args)throws Exception{
        MyClassLoader loader1=new MyClassLoader("loader1");
        String path="C:\\Users\\11375\\Desktop\\";
        loader1.setPath(path);
        Class<?>clazz=loader1.loadClass("com.jlu.jvm.GradleTest1");
        System.out.println(clazz.getClassLoader());
        Object object=clazz.newInstance();
        System.out.println(object);

    }

}
```

```
> Task :MyClassLoader.main()
[MyClassLoader]
com.jlu.jvm.Test@383534aa

BUILD SUCCESSFUL in 0s
```

##

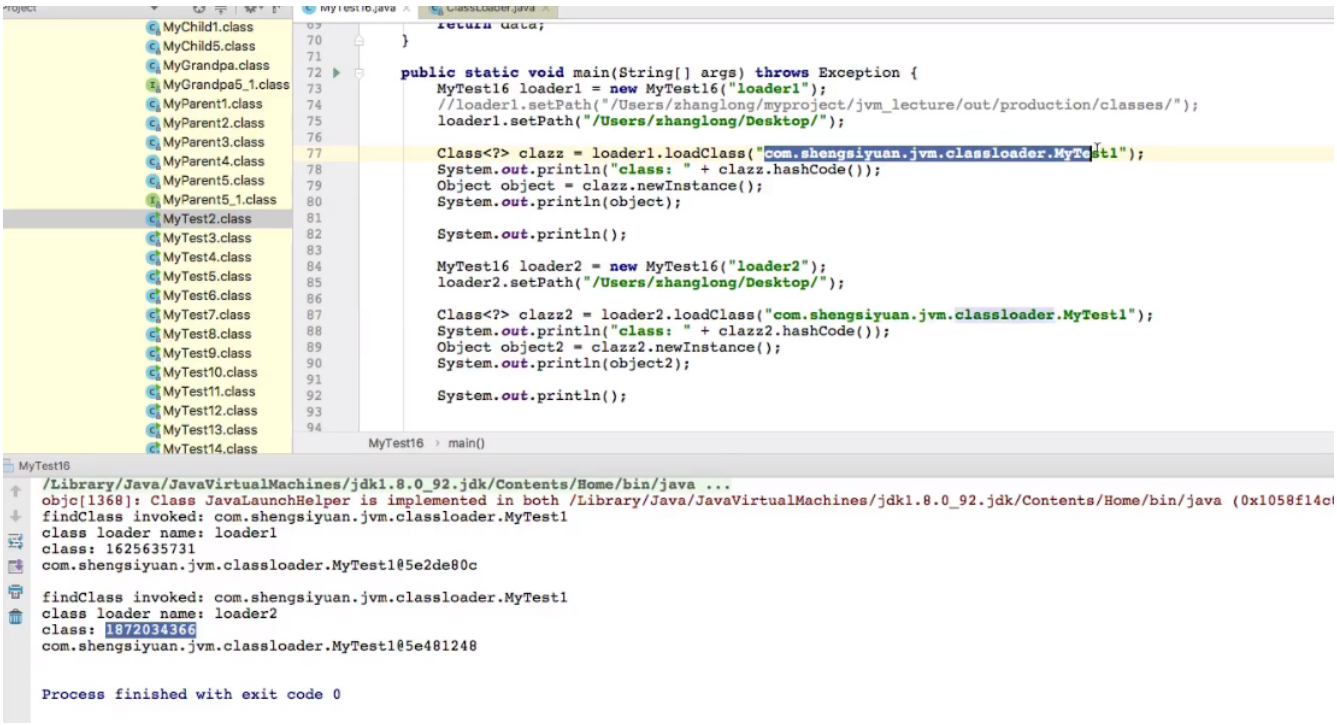
例子：



说明：这是有系统类加载器加载的，两次加载都hash值一样说明只加载一次。

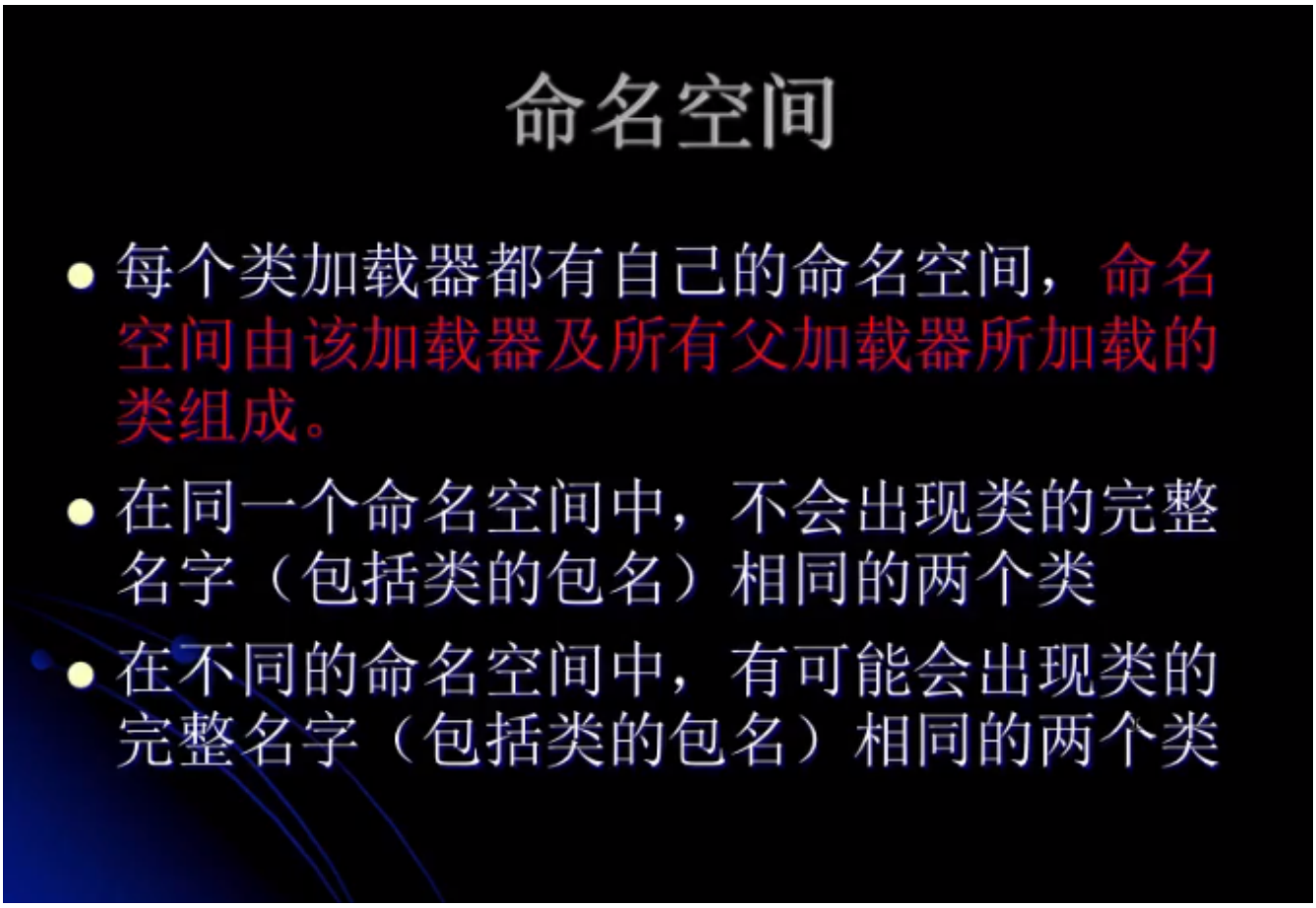
加载两次：

删掉了Mytest1.class

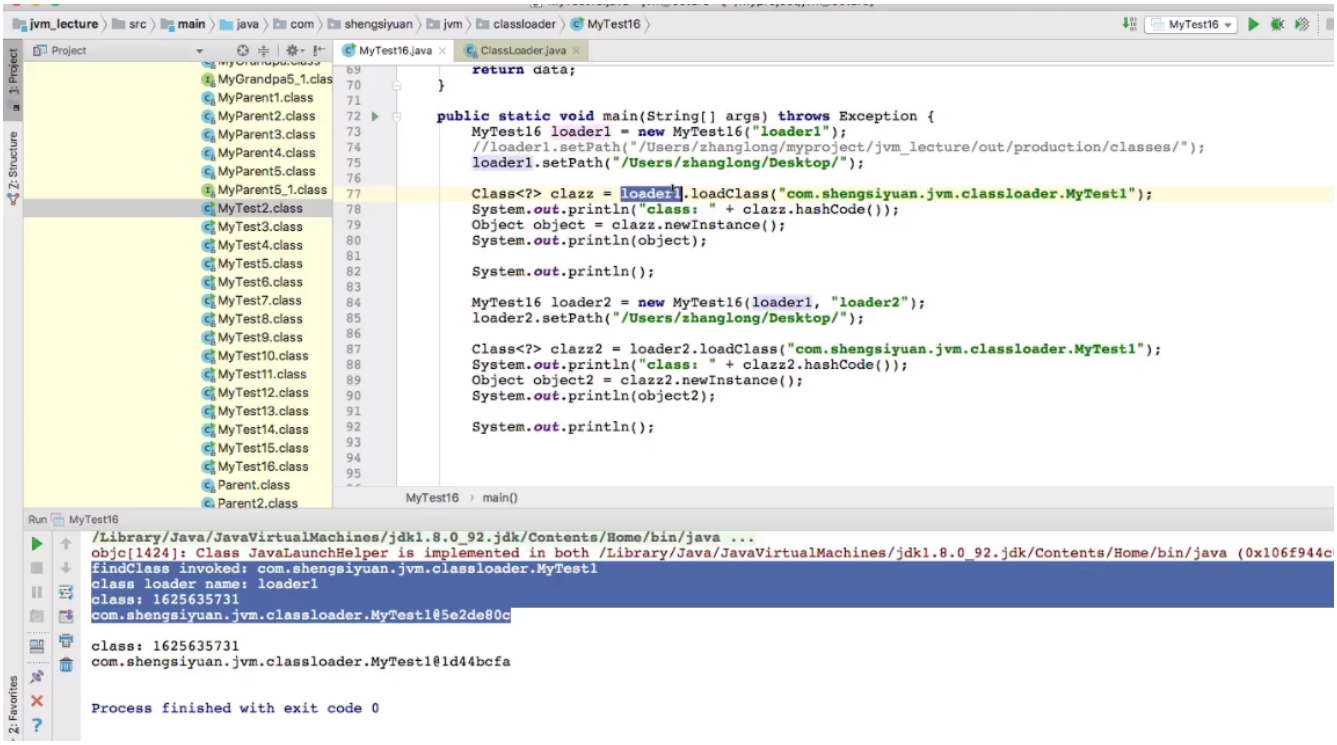


加载了两次。而且都不一样。他们在各自的命名空间里各加载了一次。

解释：

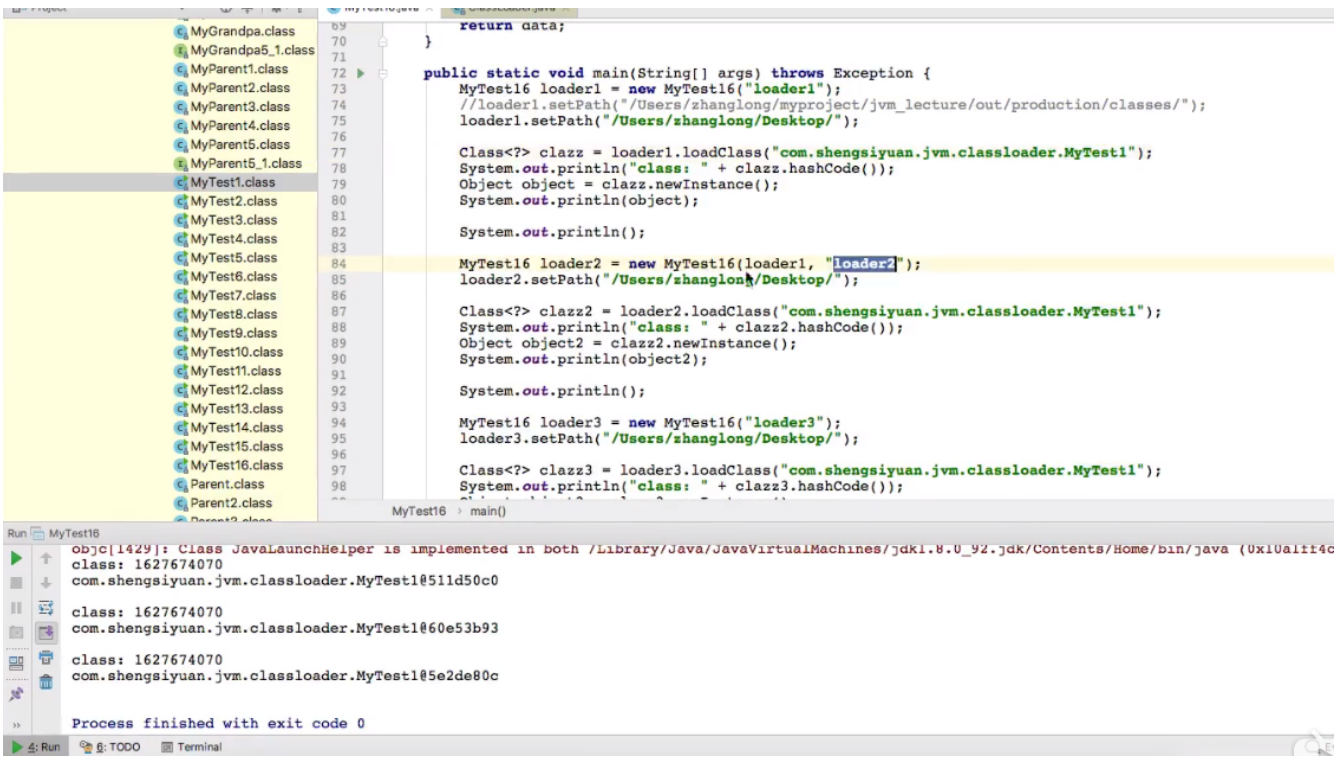


18类的命名空间与卸载详解及jvisualvm的使用



```
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
objc[1424]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (0x106f944c
findClass invoked: com.shengsiyuan.jvm.classloader.MyTest1
class loader name: loader1
class: 1625635731
com.shengsiyuan.jvm.classloader.MyTest1@5e2de80c

class: 1625635731
com.shengsiyuan.jvm.classloader.MyTest1@1d44bcfa

Process finished with exit code 0
```

解释：Loader2是loader1的子加载器。同一个类的在同一个层次之间可以是父子关系。



```
class: 1627674070
com.shengsiyuan.jvm.classloader.MyTest1@511d50c0

class: 1627674070
com.shengsiyuan.jvm.classloader.MyTest1@60e53b93

class: 1627674070
com.shengsiyuan.jvm.classloader.MyTest1@5e2de80c

Process finished with exit code 0
```

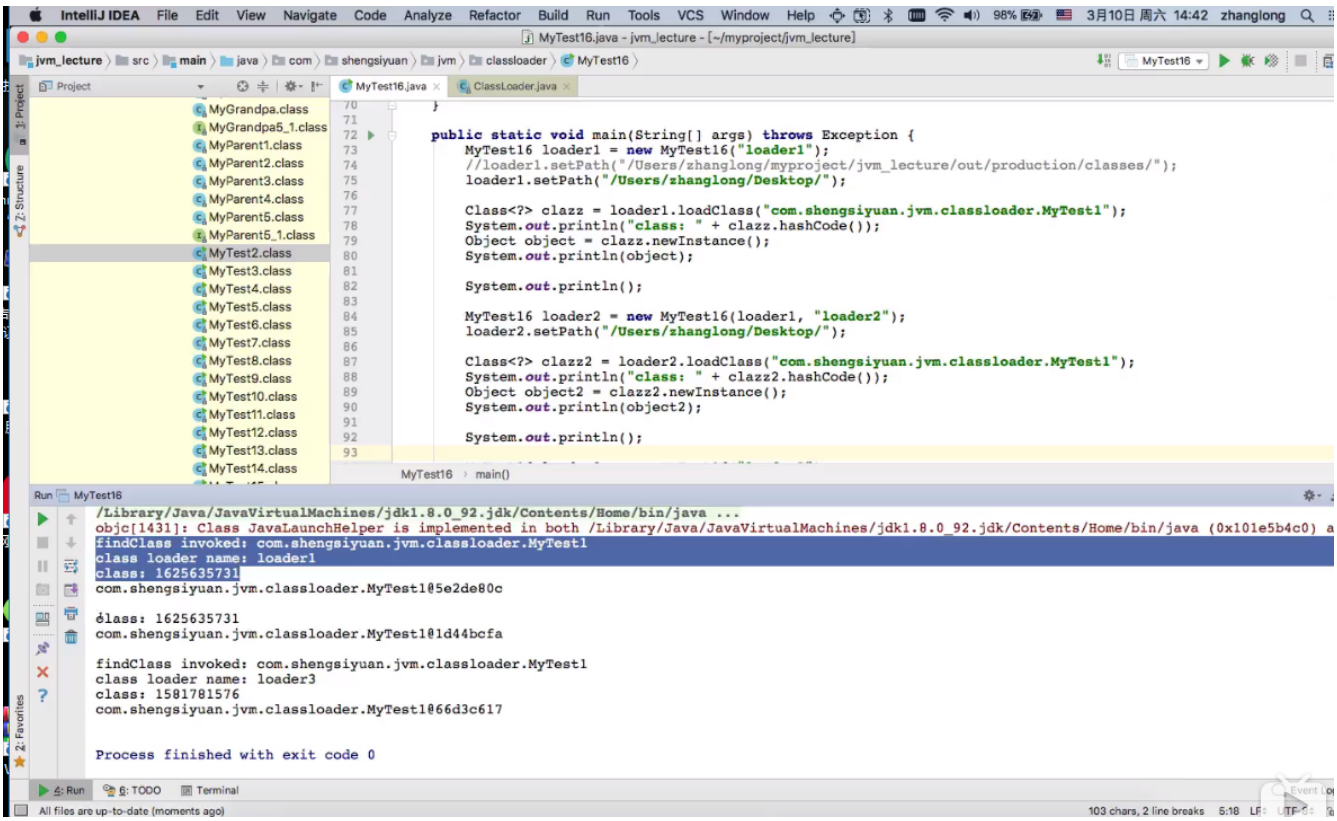
loader1的父加载器是系统类加载器。

Loader2的父加载器是loader1

loader3的父加载器是系统类加载器。

本质都是由系统类加载器加载的。

删掉Mytest的class文件



类的卸载：

类的卸载

- 当MySample类被加载、连接和初始化后，它的生命周期就开始了。当代表MySample类的Class对象不再被引用，即不可触及时，Class对象就会结束生命周期，MySample类在方法区内的数据也会被卸载，从而结束MySample类的生命周期。
- 一个类何时结束生命周期，取决于代表它的Class对象何时结束生命周期

类的卸载

由 Java 虚拟机自带的类加载器所加载的类，在虚拟机的生命周期中，始终不会被卸载。前面已经介绍过，Java 虚拟机自带的类加载器包括根类加载器、扩展类加载器和系统类加载器。Java 虚拟机本身会始终引用这些类加载器，而这些类加载器则会始终引用它们所加载的类的 Class 对象，因此这些 Class 对象始终是可触及的。

- 由用户自定义的类加载器所加载的类是可以被卸载的

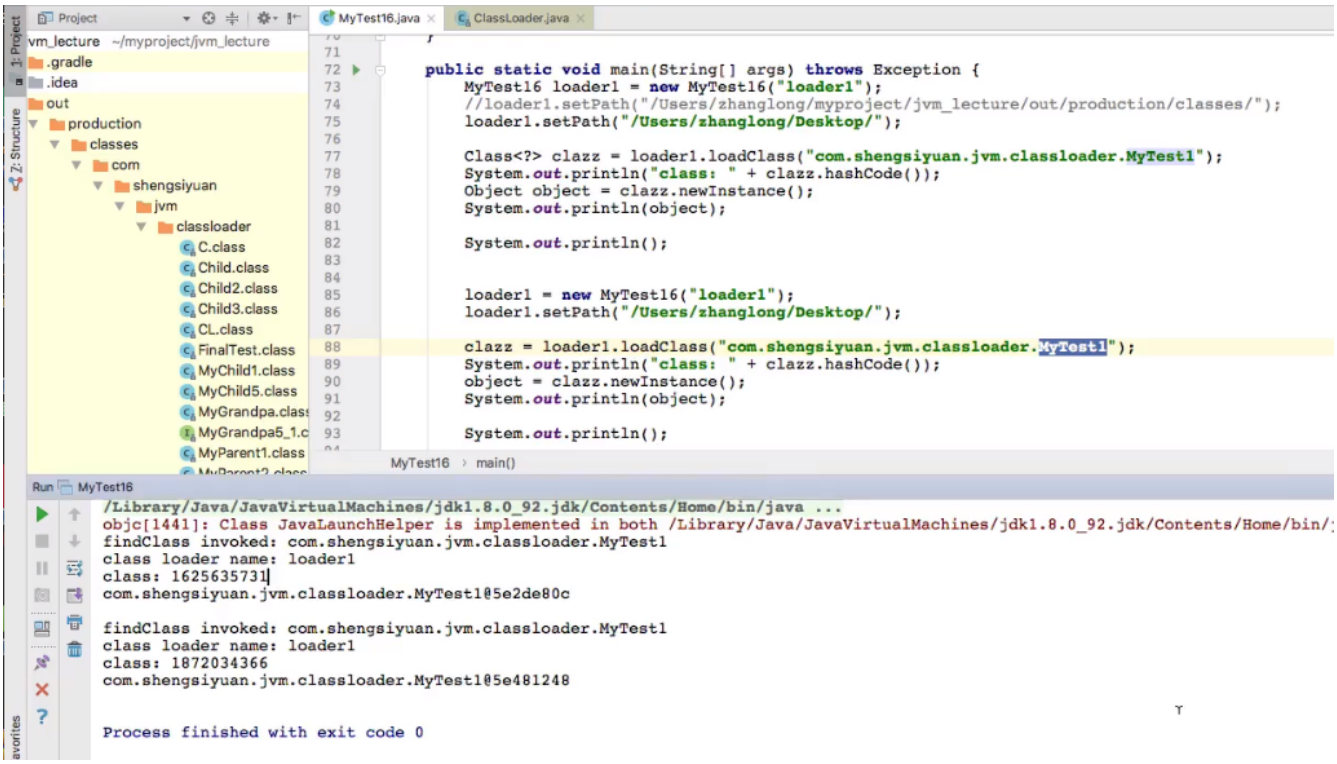
类的卸载

- 单击此处添加文本

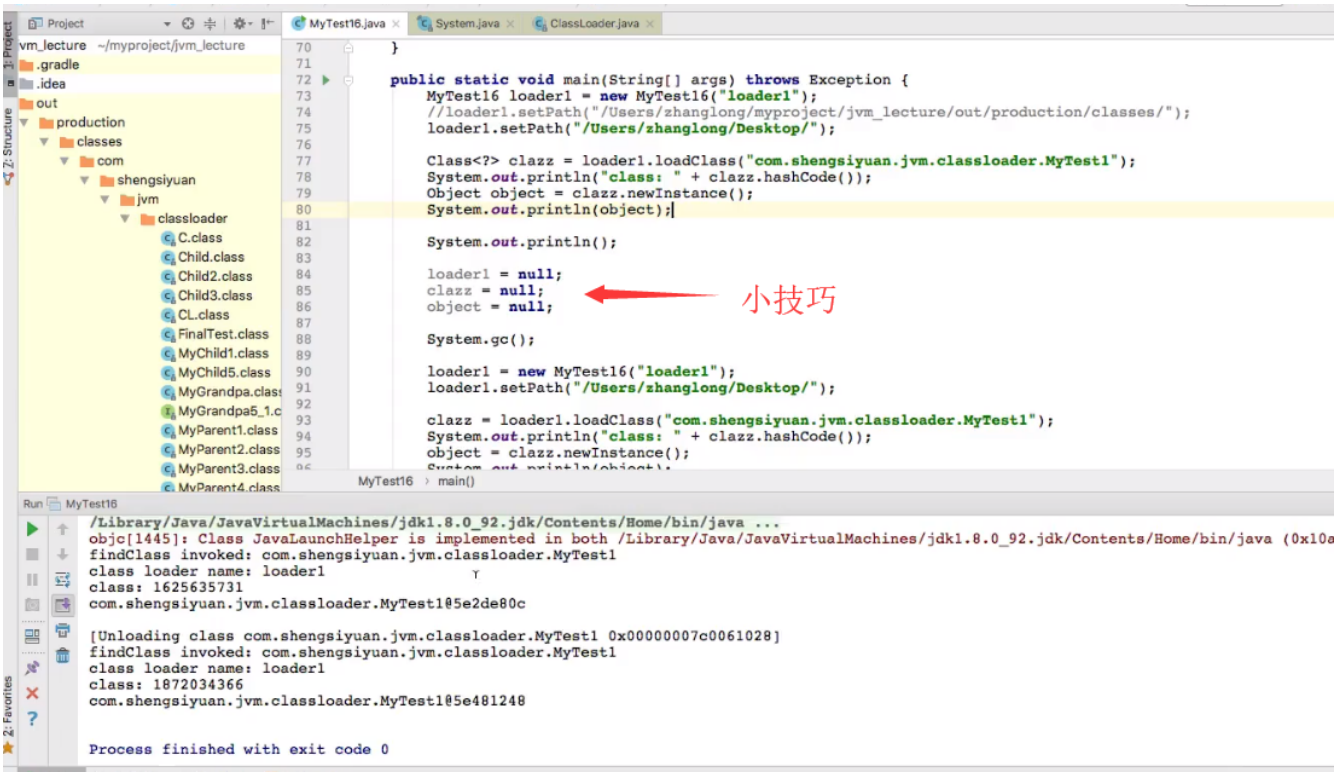
运行以上程序时，Sample 类由 loader1 加载。在类加载器的内部实现中，用一个 Java 集合来存放所加载类的引用。另一方面，一个 Class 对象总是会引用它的类加载器，调用 Class 对象的 getClassLoader()方法，就能获得它的类加载器。由此可见，代表 Sample 类的 Class 实例与 loader1 之间为双向关联关系。

一个类的实例总是引用代表这个类的 Class 对象。在 Object 类中定义了 getClass()方法，这个方法返回代表对象所属类的 Class 对象的引用。此外，所有的 Java 类都有一个静态属性 class，它引用代表这个类的 Class 对象，例如：

例子：

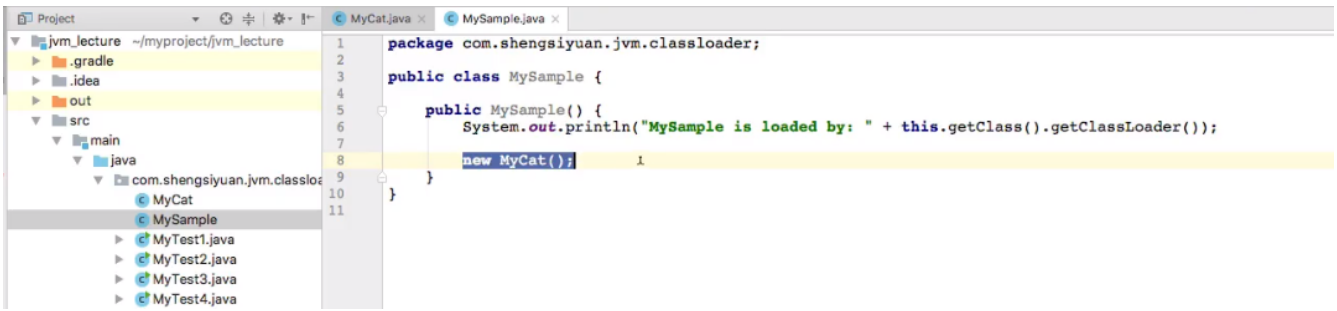


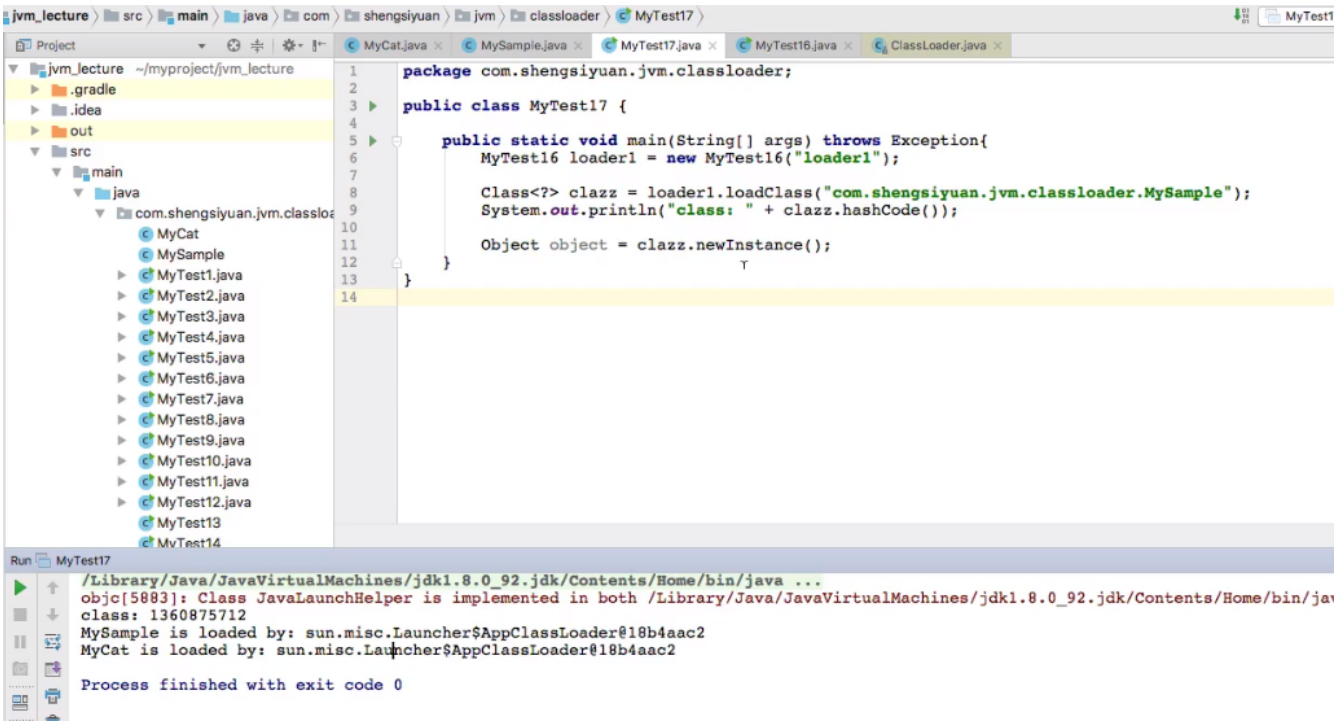
使用参数jvm来查看类的卸载或者使用jvisualvm



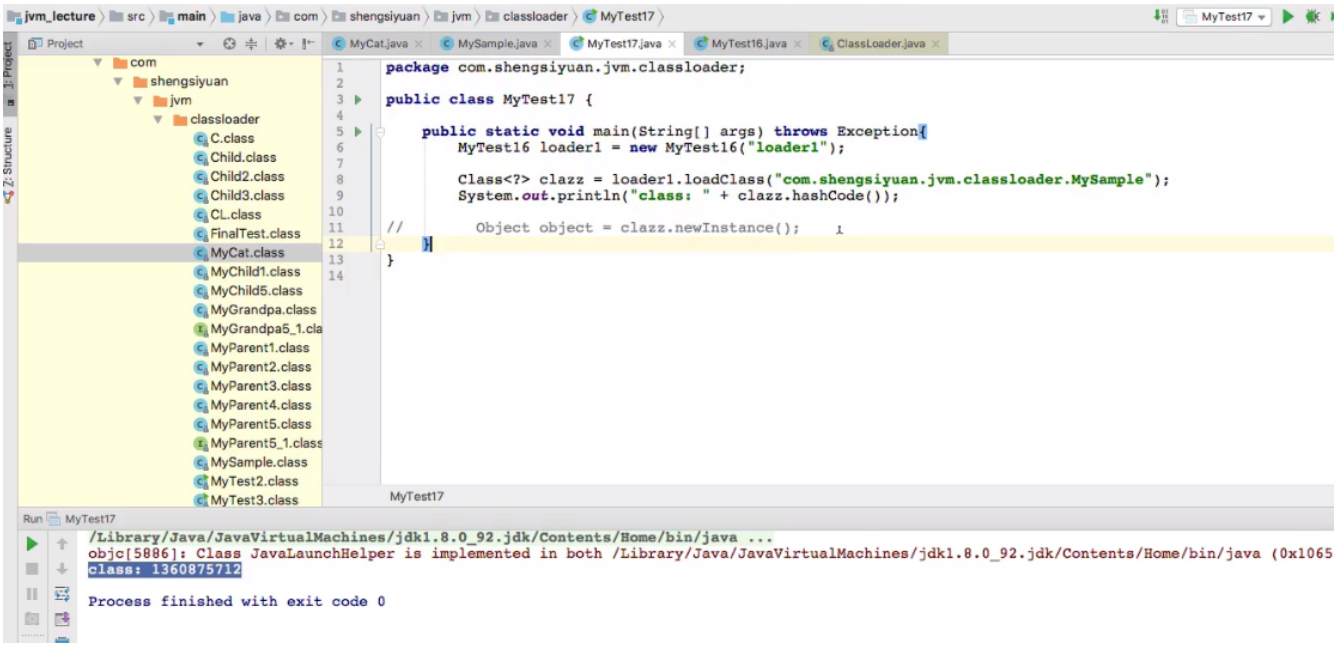
19 自定义类加载器在复杂类加载情况下的分析

```java
package com.shengsiyuan.jvm.classloader;

public class MyCat {

    public MyCat() {
        System.out.println("MyCat is loaded by: " + this.getClass().getClassLoader());
    }
}
```

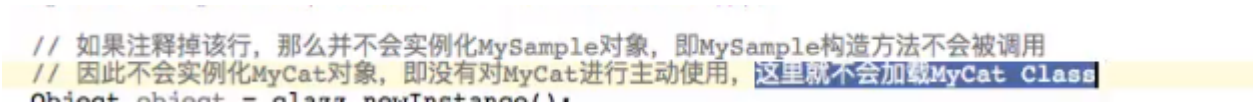




系统类加载器去加载的。

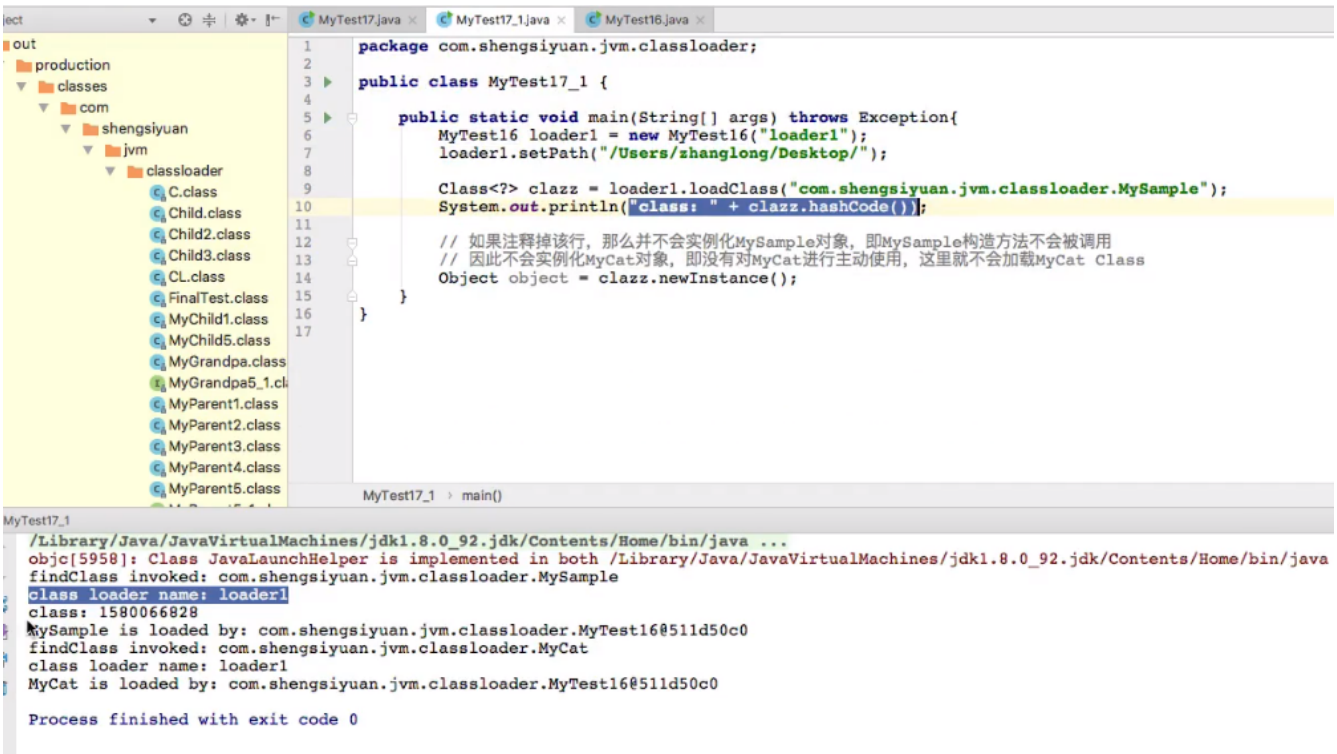


只是加载，但是没有生成实例，因此没有执行到构造方法。**其实不一定加载mycat**

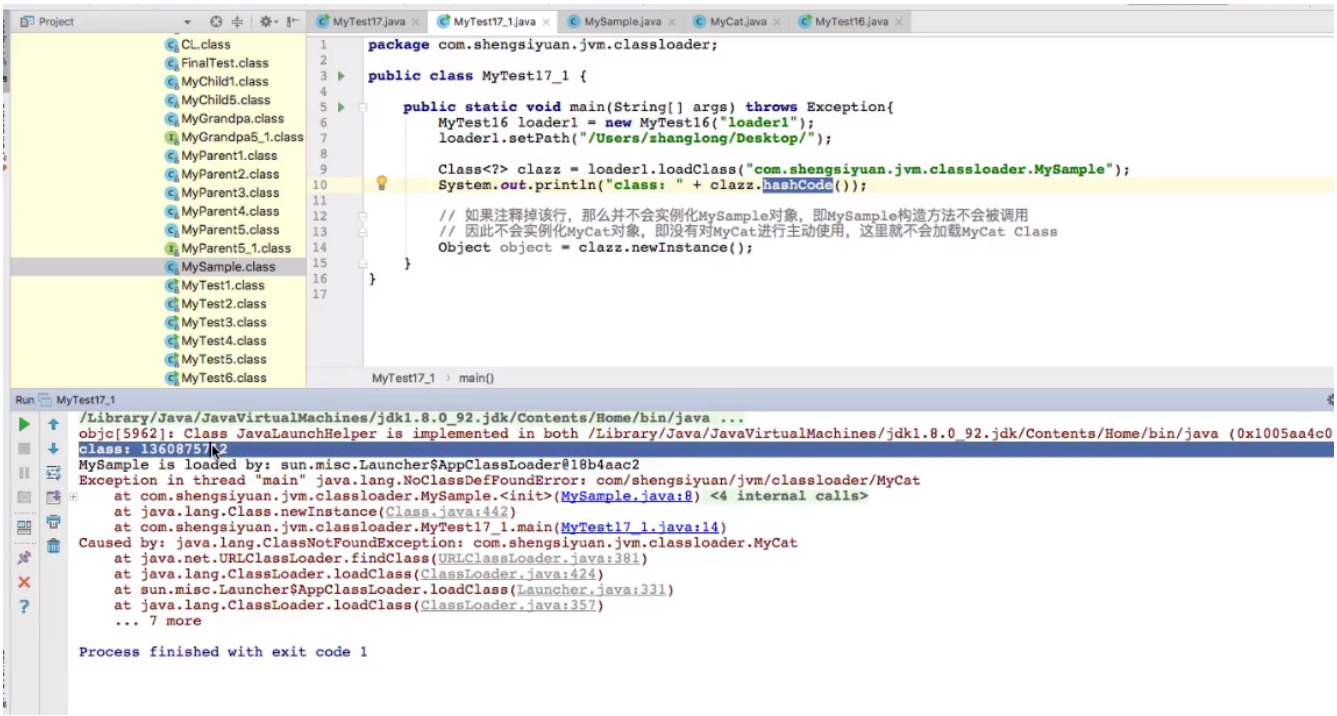


20 类加载器命名空间实战剖析与透彻理解

例子：



删除Mycat的class文件

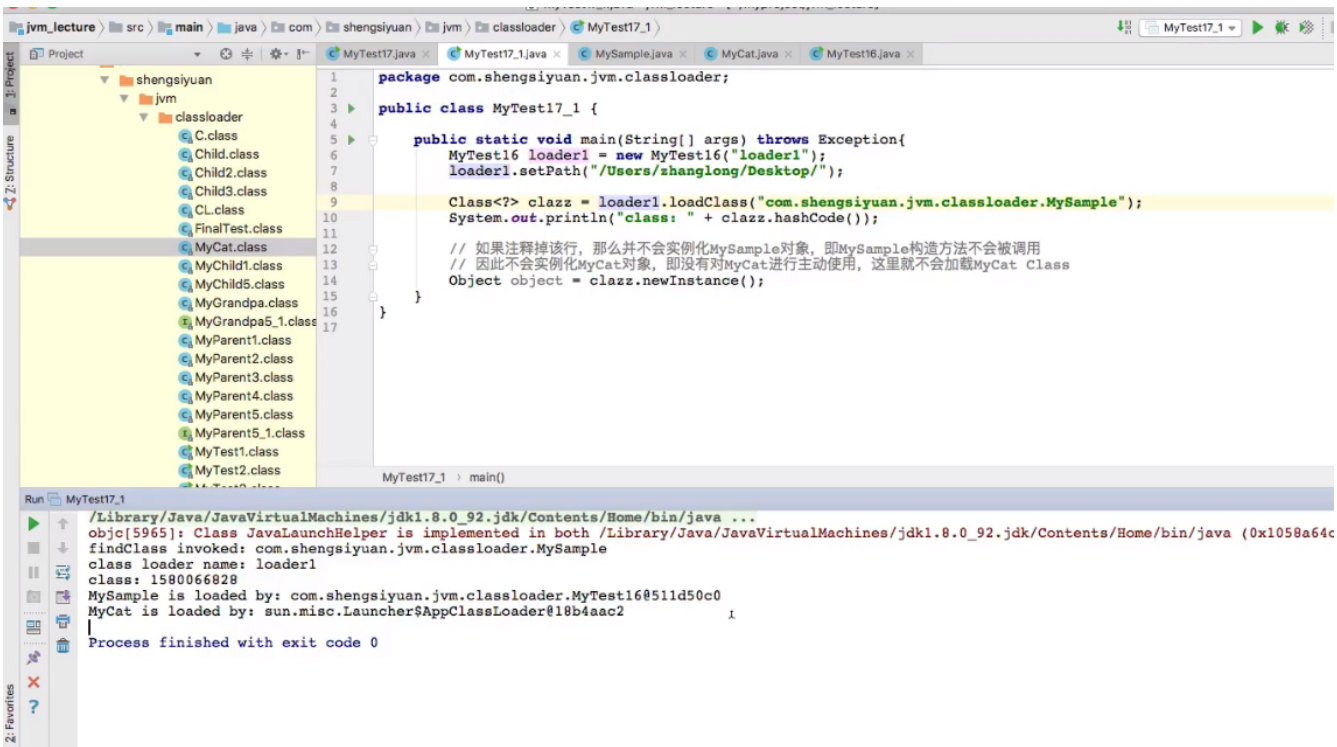


自定义的加载器委派系统类加载器加载MySimple，遇到new MyCat（），系统类加载器尝试去加载这个类，但是加载不了，

主要是这里加载MyCat的发起者是系统类加载器，不会让自定义的加载器去加载。

此时应用类加载器不能加载mycat；

删除mysimple，mysimple放在桌面上

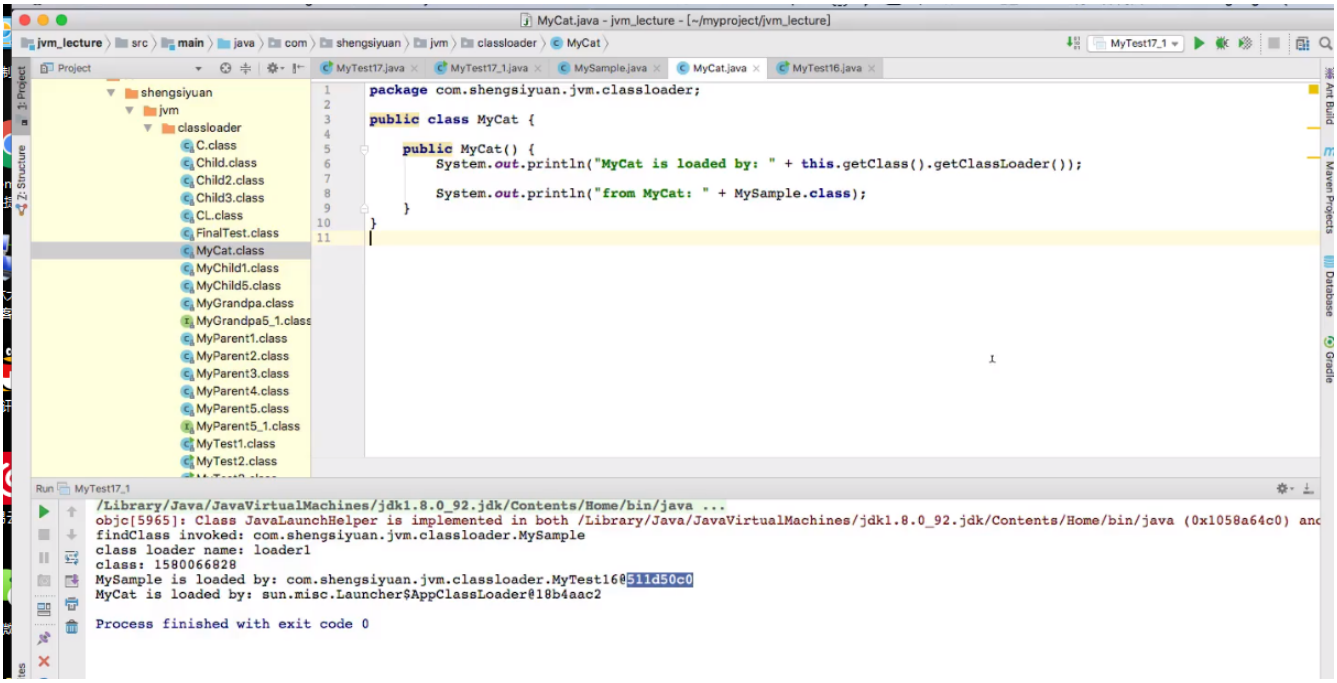


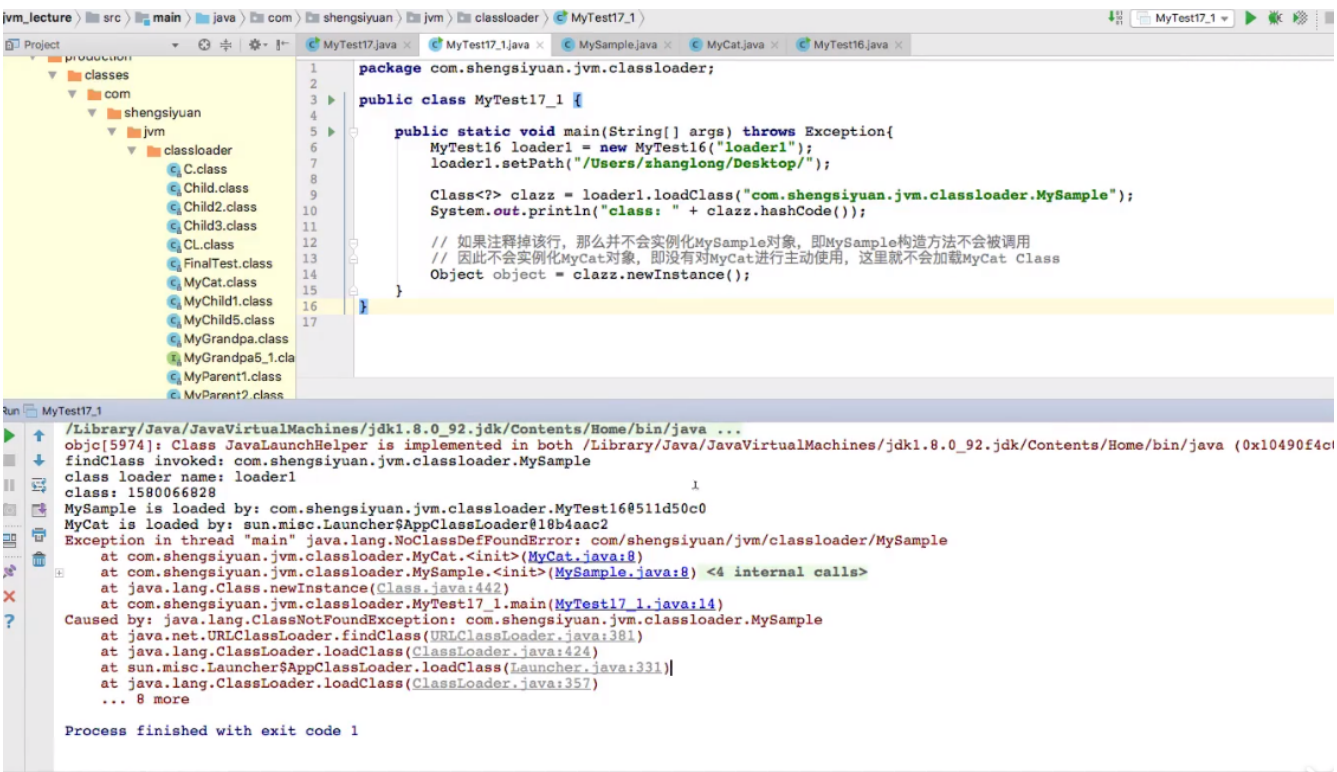
这里的加载MySimple是自定义的类加载器，在加载MySimple是遇到new MyCat（），这个自定义的加载器尝试去加载Mycat，但是MyCat的class问价在，所以是系统类加载器去加载，否则自定义的加载器去加载。

这里是由自定义的加载器去发起加载MyCat的。

只能统一由自定义的类加载器去加载。**使用了两个不同的类加载器去加载**

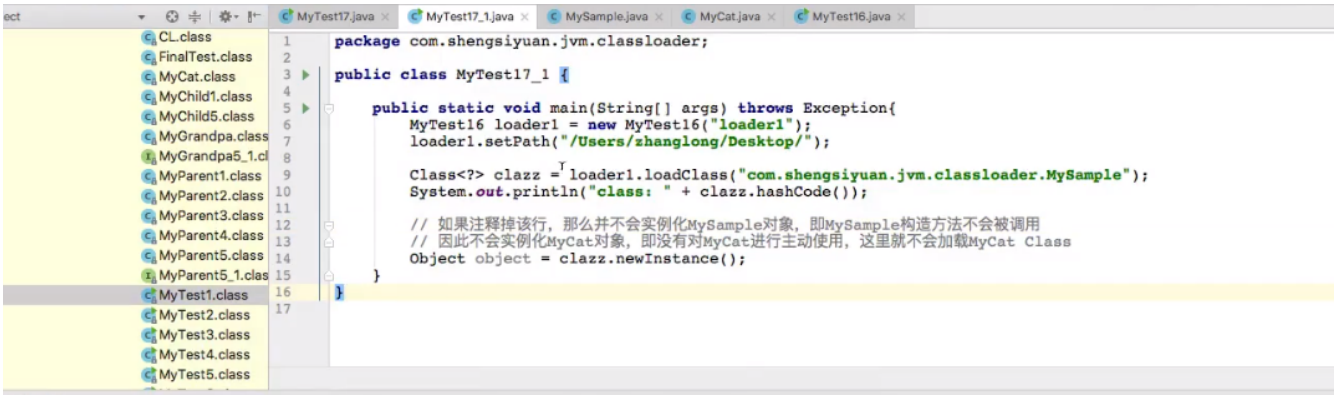
删除mysimple的class文件，在桌面上保留。

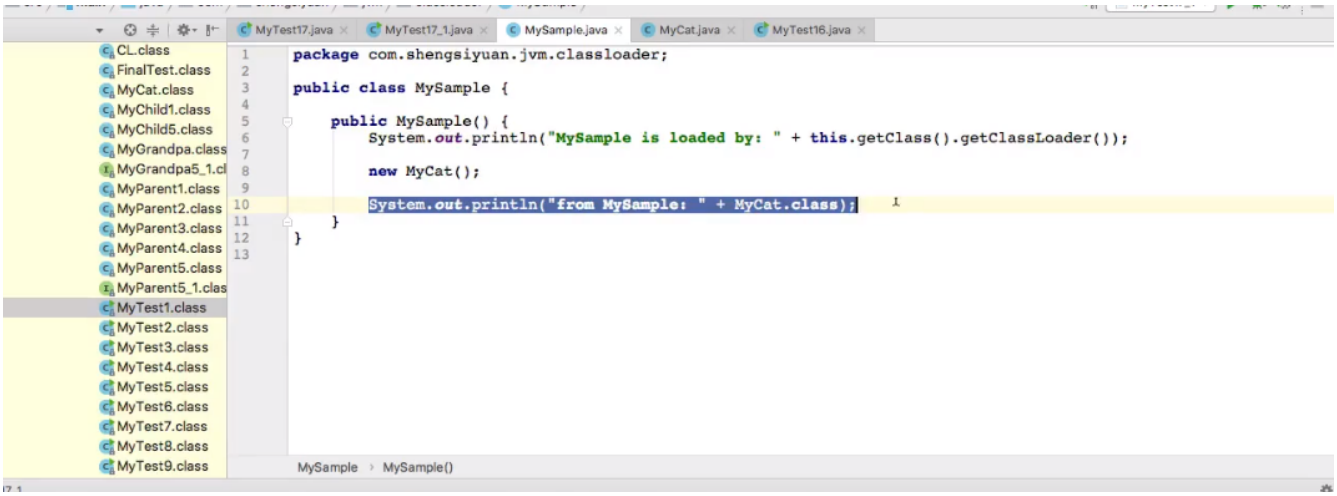


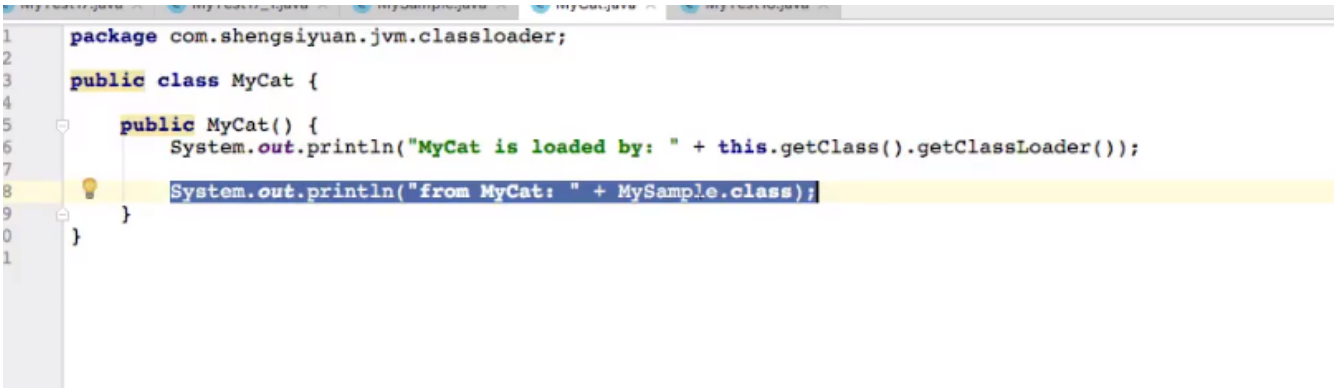


他们是由两个不同的类加载器加载的。他们不在同一个命名。mycat所在的命名空间为系统类加载器的命名空间，看不到子类加载器的命名空间。

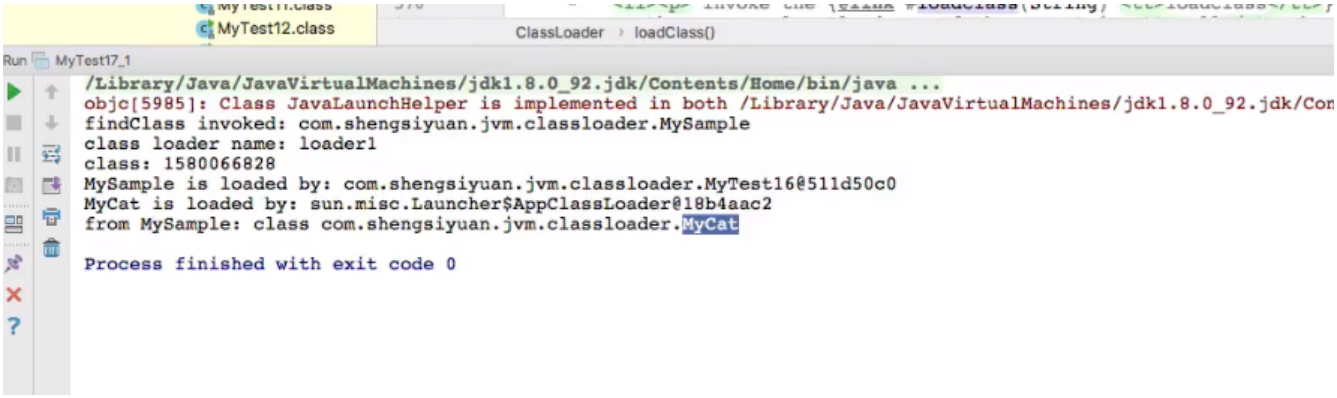
还是删掉mysimple，但是也删掉MyCat对MySimple的引用。



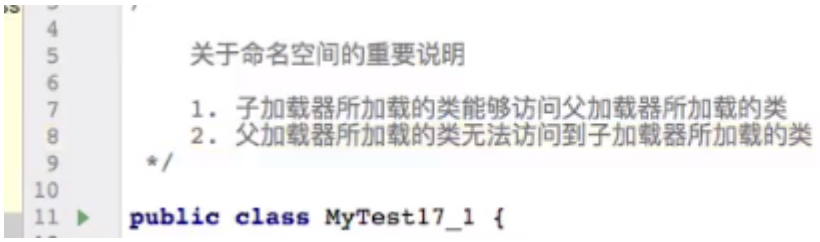




注释掉标记的行。

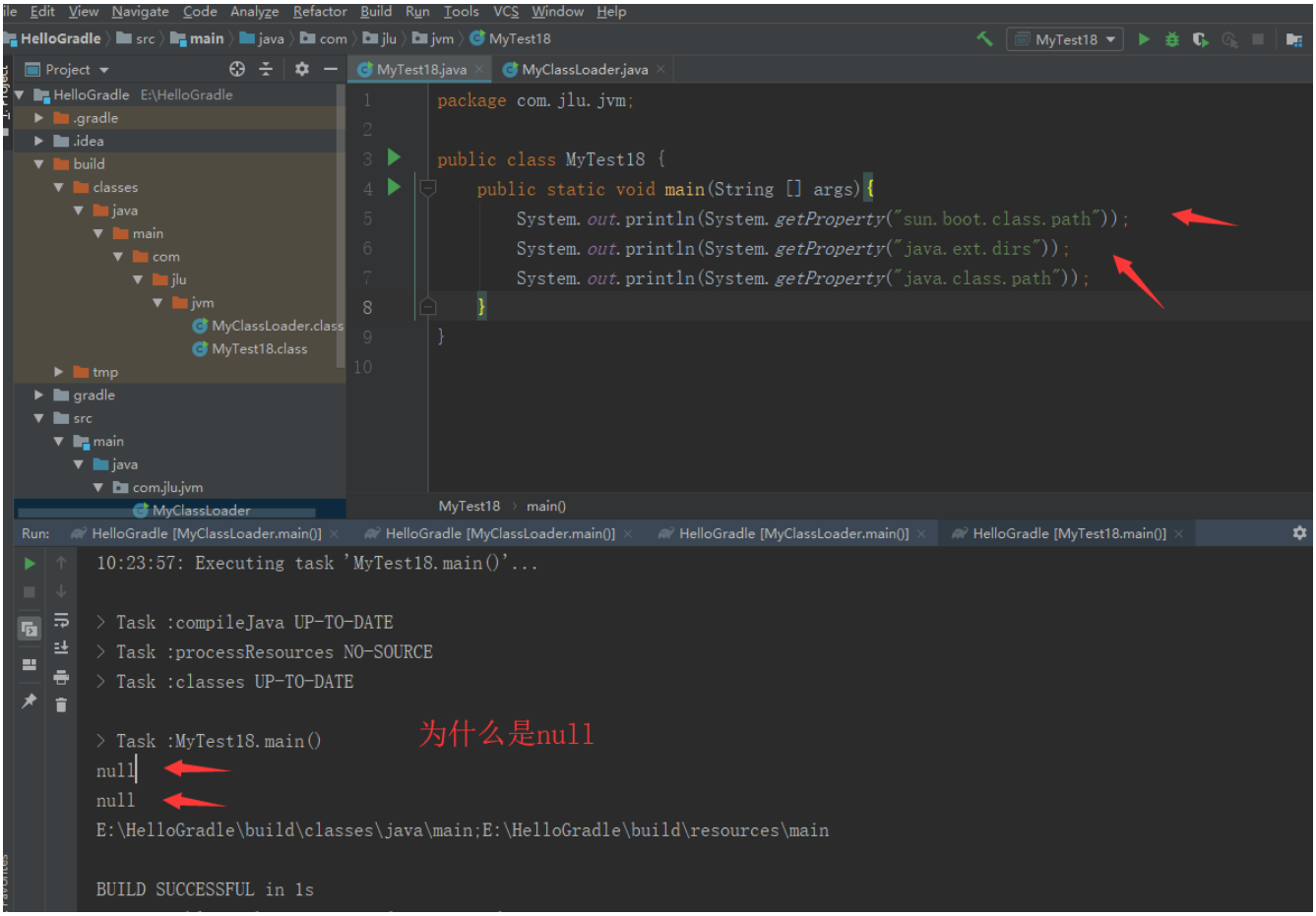


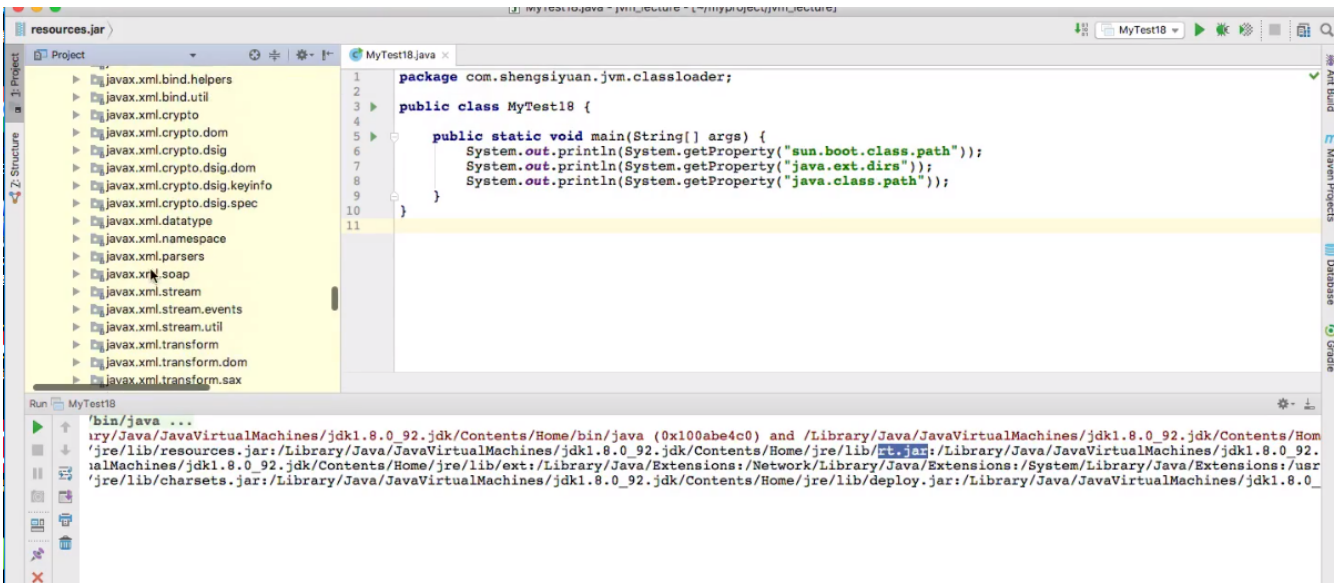
结论：



21 类加载器实例剖析与疑难点解析

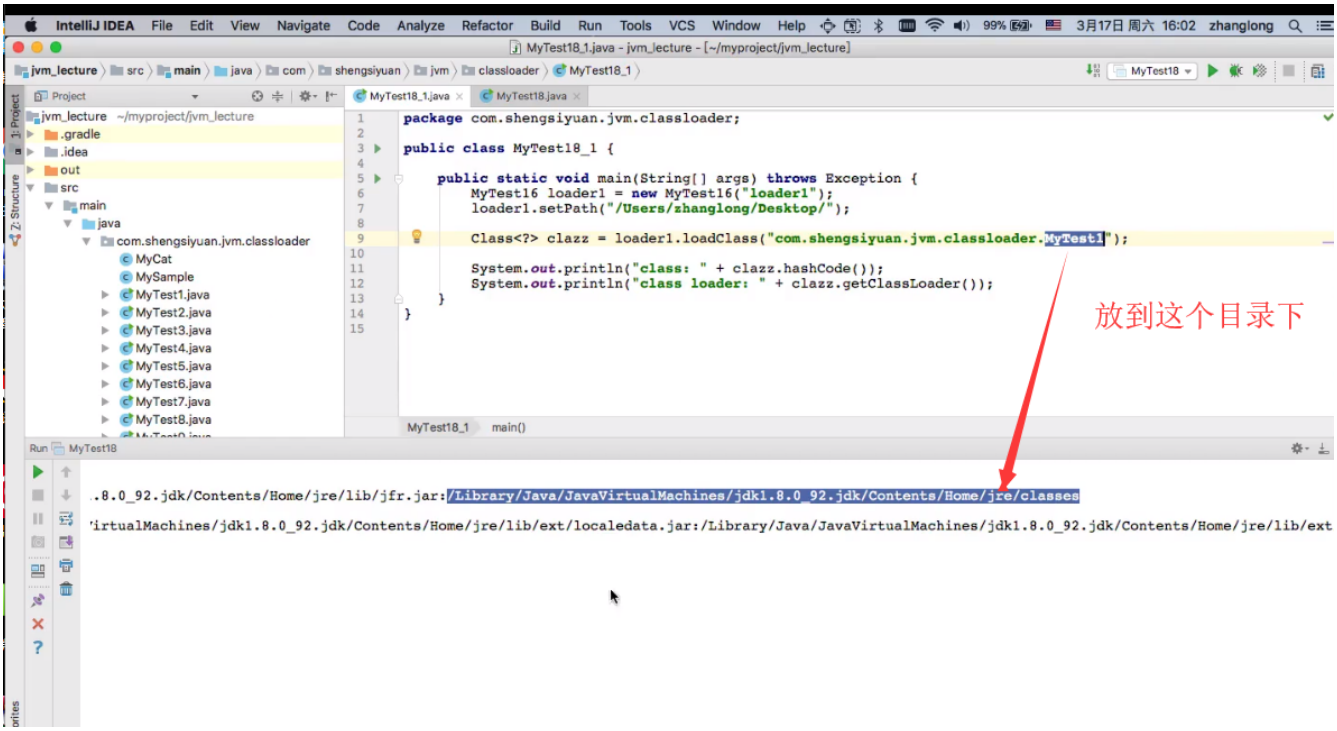
```
package com.jlu.jvm;

public class MyTest18 {
    public static void main(String [] args){
        System.out.println(System.getProperty("sun.boot.class.path"));
        System.out.println(System.getProperty("java.ext.dirs"));
        System.out.println(System.getProperty("java.class.path"));
    }
}
```



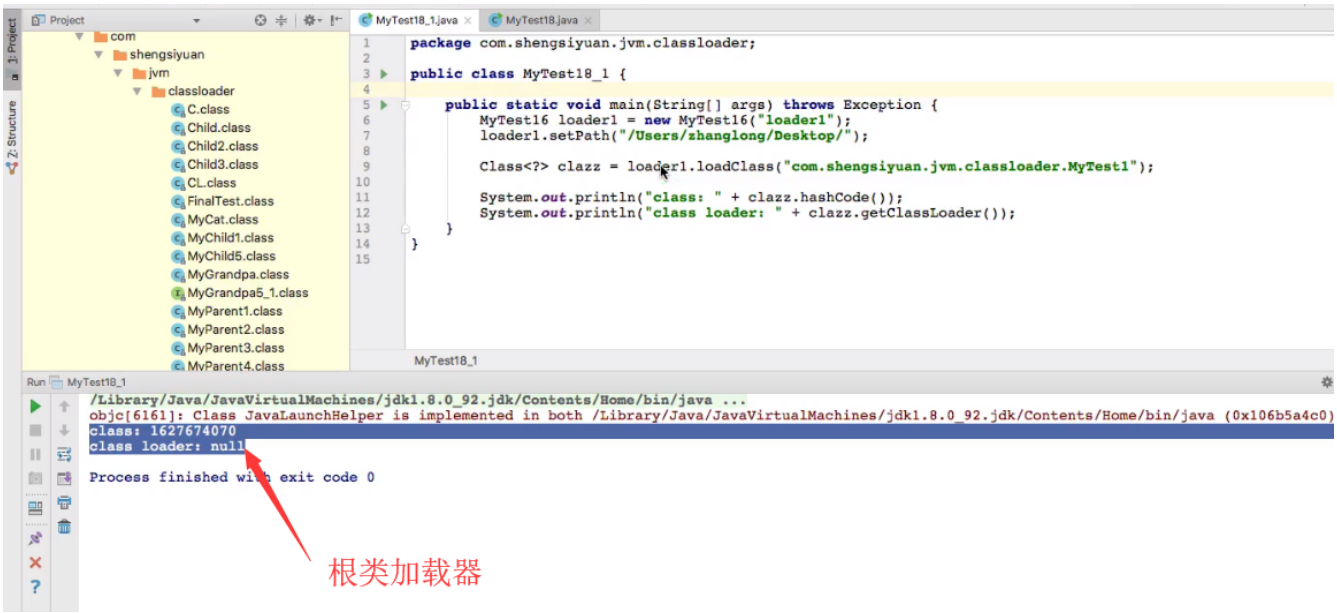


为什么有区别啊？？？？？？？？？mac和windows不同吗

虚拟机的名字：java.vm.name=Java HotSpot(TM) 64-Bit Server VM

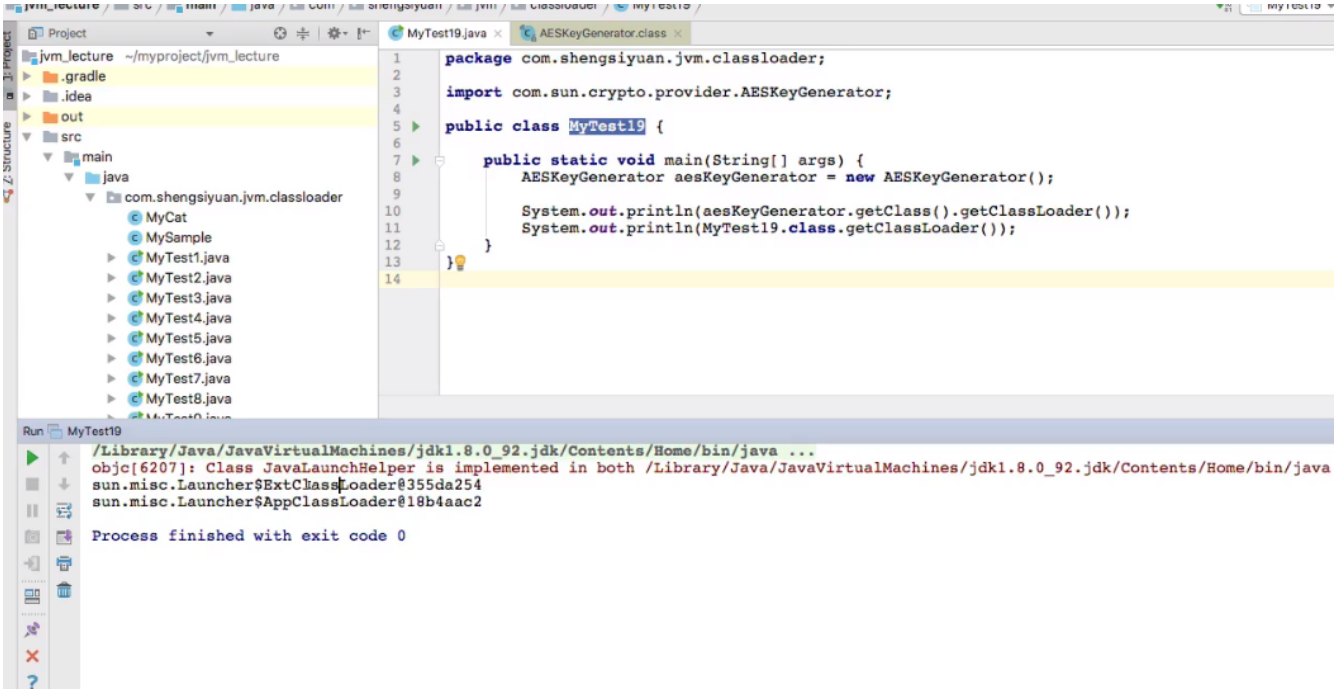


结果：



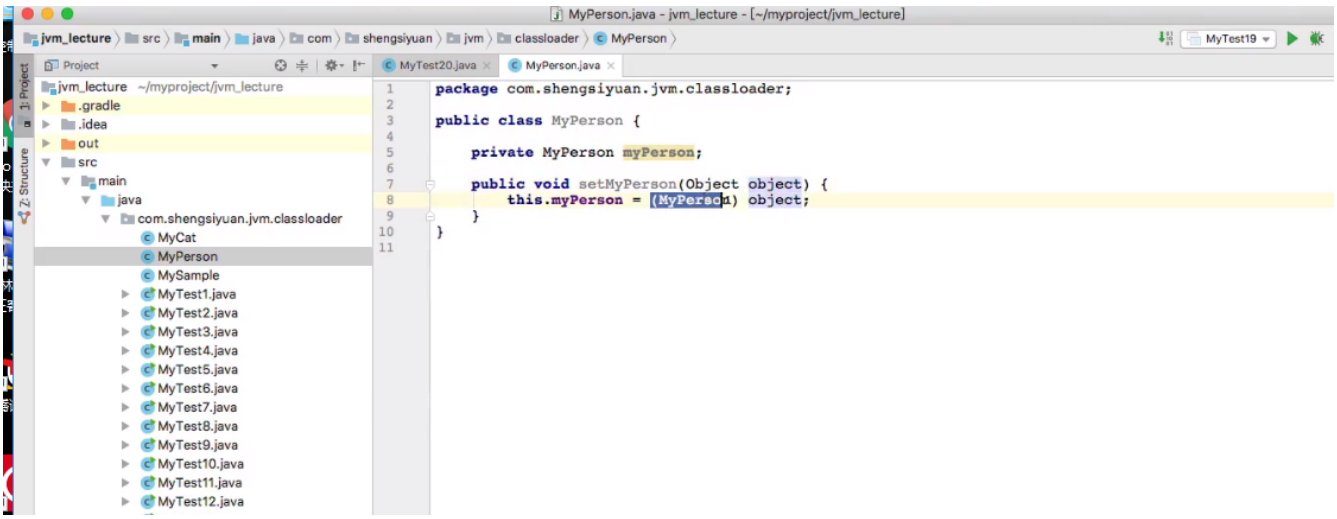
扩展类加载器主要用于加载非jdk核心的类，扩展的jar包里去加载类

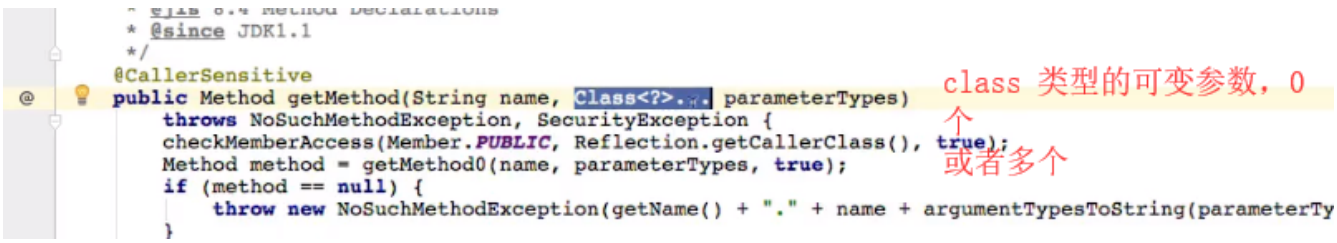
测试扩展类加载器：

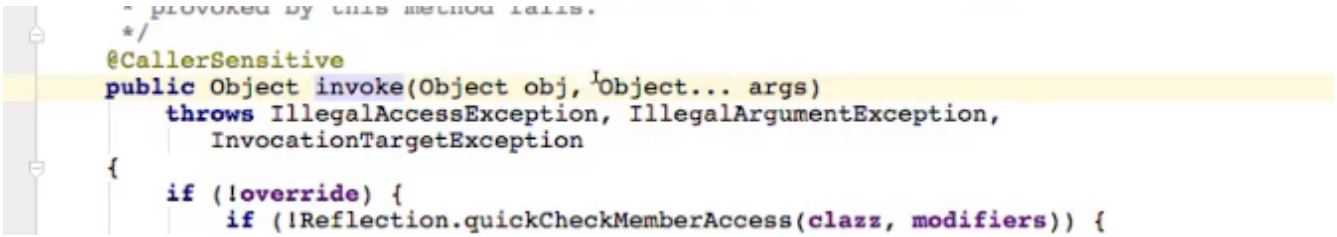

Project
jvm_lecture ~/myproject/jvm_lecture
.gradle
.idea
out
src
main
java
com.shengsiyuan.jvm.classloader
MyCat
MySample
MyTest1.java
MyTest2.java
MyTest3.java
MyTest4.java
MyTest5.java
MyTest6.java
MyTest7.java
MyTest8.java
MyTest19.java
AESKeyGenerator.class
package com.shengsiyuan.jvm.classloader;
import com.sun.crypto.provider.AESKeyGenerator;
public class MyTest19 {
public static void main(String[] args) {
AESKeyGenerator aesKeyGenerator = new AESKeyGenerator();
System.out.println(aesKeyGenerator.getClass().getClassLoader());
System.out.println(MyTest19.class.getClassLoader());
}
}
Run MyTest19
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
objc[6207]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java
sun.misc.Launcher$ExtClassLoader@355da254
sun.misc.Launcher$AppClassLoader@18b4aac2
Process finished with exit code 0


jvm_lecture cd out/production/classes
classes ls
com
classes java -Djava.ext.dirs=./ com.shengsiyuan.jvm.classloader.MyTest19
当前目录
修改属性
Exception in thread "main" java.lang.NoClassDefFoundError: com/sun/crypto/provider/AESKeyGenerator
at com.shengsiyuan.jvm.classloader.MyTest19.main(MyTest19.java:8)
Caused by: java.lang.ClassNotFoundException: com.sun.crypto.provider.AESKeyGenerator
at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:331)
at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
... 1 more
classes

例子2:

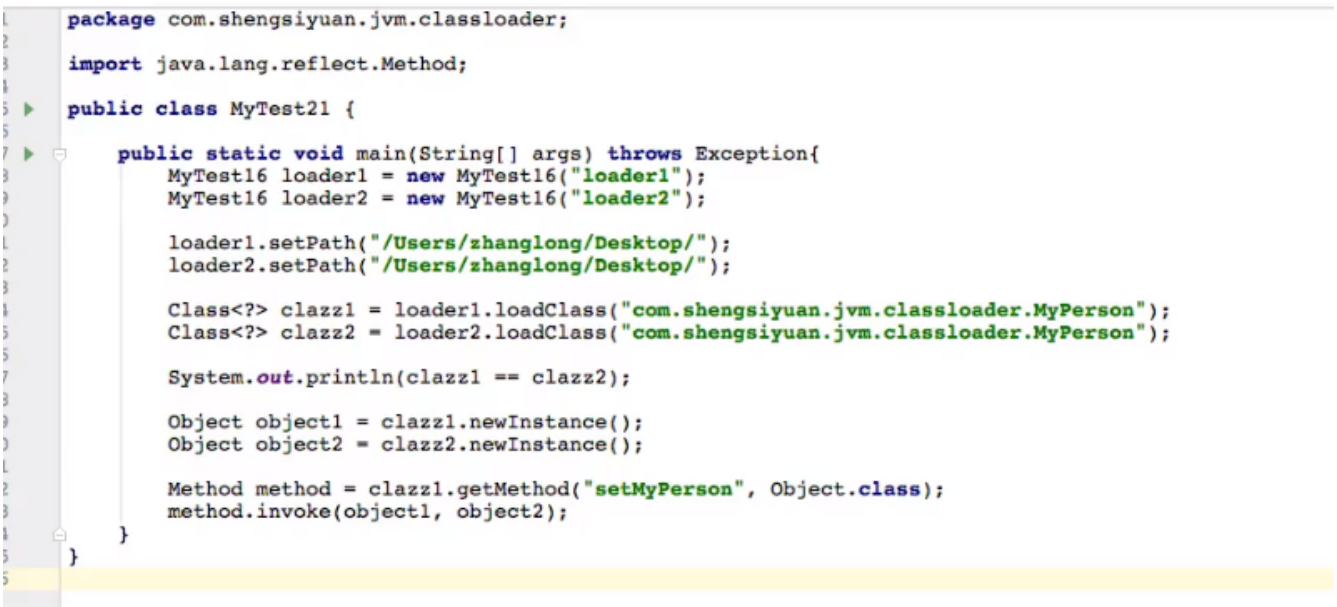

MyPerson.java - jvm_lecture - [~/myproject/jvm_lecture]
jvm_lecture src main java com shengsiyuan jvm classloader MyPerson
MyTest19
Project
jvm_lecture ~/myproject/jvm_lecture
.gradle
.idea
out
src
main
java
com.shengsiyuan.jvm.classloader
MyCat
MyPerson
MySample
MyTest1.java
MyTest2.java
MyTest3.java
MyTest4.java
MyTest5.java
MyTest6.java
MyTest7.java
MyTest8.java
MyTest9.java
MyTest10.java
MyTest11.java
MyTest12.java
MyTest20.java
MyPerson.java
package com.shengsiyuan.jvm.classloader;
public class MyPerson {
private MyPerson myPerson;
public void setMyPerson(Object object) {
this.myPerson = (MyPerson) object;
}
}





例子3：

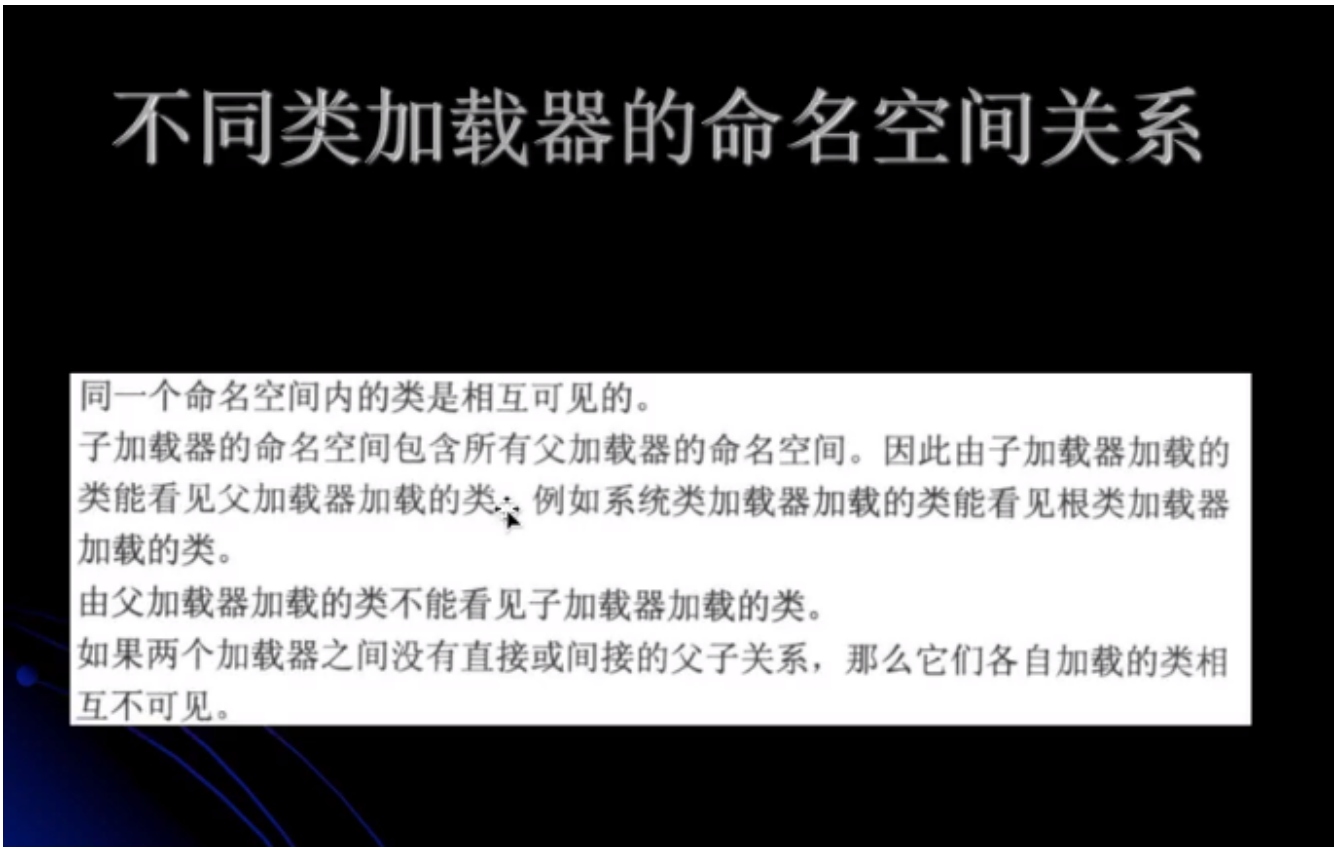
删除 person class文件，将其放到桌面



```
MyTest21
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
objc[982]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (0x10975f4c0) a
findClass invoked: com.shengsiyuan.jvm.classloader.MyPerson
class loader name: loader1
findClass invoked: com.shengsiyuan.jvm.classloader.MyPerson
class loader name: loader2
false
Exception in thread "main" java.lang.reflect.InvocationTargetException
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:498)
    at com.shengsiyuan.jvm.classloader.MyTest21.main(MyTest21.java:23)
Caused by: java.lang.ClassCastException: com.shengsiyuan.jvm.classloader.MyPerson cannot be cast to com.shengsiyuan.jvm.classloader.MyPerson
    at com.shengsiyuan.jvm.classloader.MyPerson.setMyPerson(MyPerson.java:8)
    ... 5 more

Process finished with exit code 1
```

```
    at java.lang.reflect.Method.invoke(Method.java:498)
    at com.shengsiyuan.jvm.classloader.MyTest21.main(MyTest21.java:23)
Caused by: java.lang.ClassCastException: com.shengsiyuan.jvm.classloader.MyPerson cannot be cast to com.shengsiyuan.jvm.classloader.MyPerson
    at com.shengsiyuan.jvm.classloader.MyPerson.setMyPerson(MyPerson.java:8)
    ... 5 more
```

clazz1和clazz2不在同一个类加载器命名空间，他们是相互不可见的，所以实例化的对象也是不可见的。

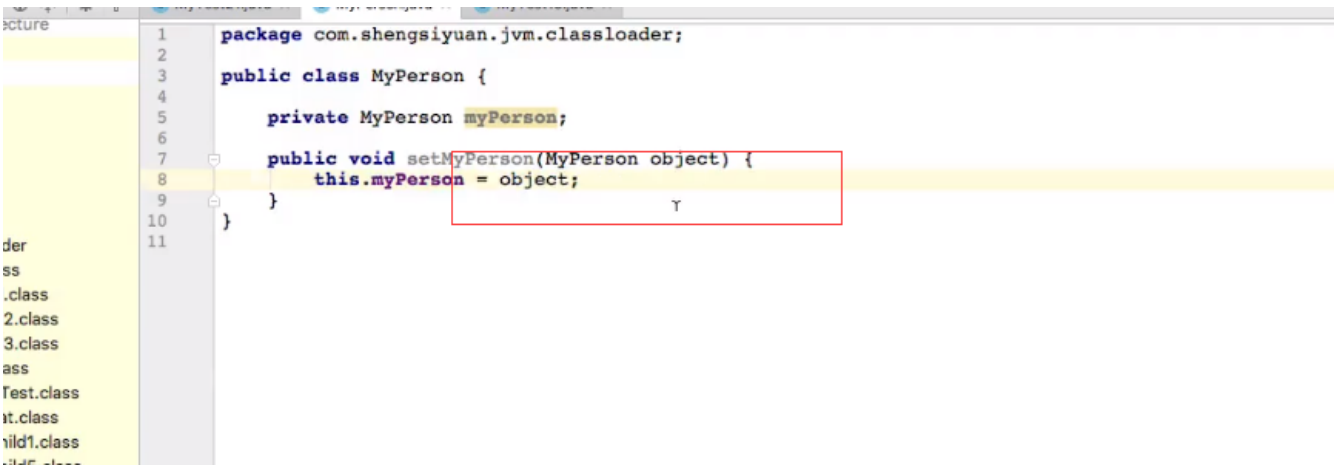


修改：

向下类型转换的原因是因为避免删除MyPerson的classs文件后，MyPerson.class出异常

```
Method method = clazz1.getMethod("setMyPerson", MyPerson.class);
```

```
package com.shengsiyuan.jvm.classloader;

import java.lang.reflect.Method;

public class MyTest21 {

    public static void main(String[] args) throws Exception{
        MyTest16 loader1 = new MyTest16("loader1");
        MyTest16 loader2 = new MyTest16("loader2");

        loader1.setPath("/Users/zhanglong/Desktop/");
        loader2.setPath("/Users/zhanglong/Desktop/");

        Class<?> clazz1 = loader1.loadClass("com.shengsiyuan.jvm.classloader.MyPerson");
        Class<?> clazz2 = loader2.loadClass("com.shengsiyuan.jvm.classloader.MyPerson");

        System.out.println(clazz1 == clazz2);

        Object object1 = clazz1.newInstance();
        Object object2 = clazz2.newInstance();

        Method method = clazz1.getMethod("setMyPerson", MyPerson.class);
        method.invoke(object1, object2);
    }
}
```



```
package com.shengsiyuan.jvm.classloader;

public class MyPerson {

    private MyPerson myPerson;

    public void setMyPerson(MyPerson object) {
        this.myPerson = object;
    }
}
```

23 类加载器命名空间总结，与扩展类加载器分析

面试，必备

```
/*
    类加载器的双亲委托模型的好处：

    1. 可以确保Java核心库的类型安全：所有的Java应用都至少会引用java.lang.Object类，也就是说在运行期，java.lang.Object这个类
       会被加载到Java虚拟机中；如果这个加载过程是由Java应用自己的类加载器所完成的，那么很可能就会在JVM中存在多个版本的
       java.lang.Object类，而且这些类之间还是不兼容的，相互不可见的（正是命名空间在发挥着作用）。
       借助于双亲委托机制，Java核心类库中的类的加载工作都是由启动类加载器来统一完成，从而确保了Java应用所使用的都是同一个版本的
       Java核心类库，他们之间是相互兼容的。
    2. 可以确保Java核心类库所提供的类不会被自定义的类所替代。
    3. 不同的类加载器可以为相同名称（binary name）的类创建额外的命名空间。相同名称的类可以并存在Java虚拟机中，只需要用不同的
       类加载器来加载他们即可。不同类加载器所加载的类之间是不兼容的，这就相当于在Java虚拟机内部创建了一个又一个相互隔离的Java
       类空间，这类技术在很多框架中都得到了实际应用。
 */
```

例子：

```
package com.shengsiyuan.jvm.classloader;

public class MyTest22 {

    static {
        System.out.println("MyTest22 initializer");
    }

    public static void main(String[] args) {
        System.out.println(MyTest22.class.getClassLoader());

        System.out.println(MyTest1.class.getClassLoader());
    }
}
```

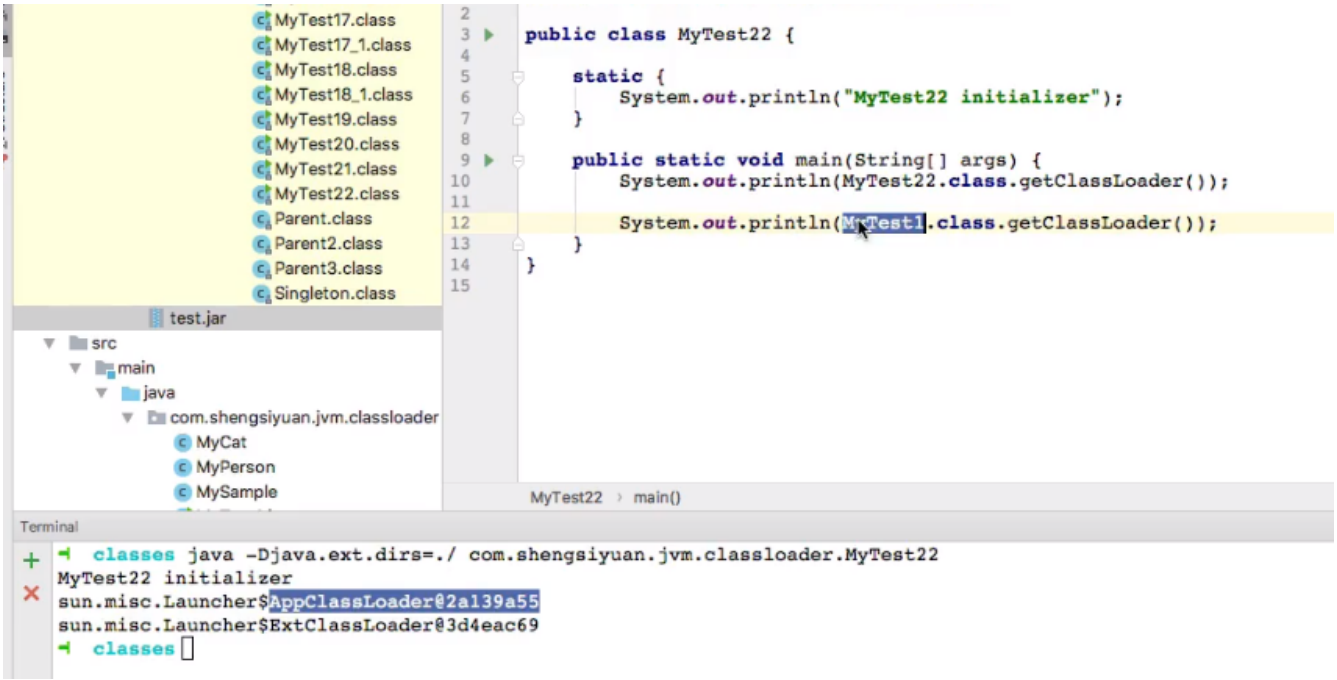
运行期间修改系统属性

```
MyTest22 initializerva.ext.dirs=./ com.shengsiyuan.jvm.classloader.MyTest22
sun.misc.Launcher$AppClassLoader@73d16e93
sun.misc.Launcher$AppClassLoader@73d16e93
➜  classes
```

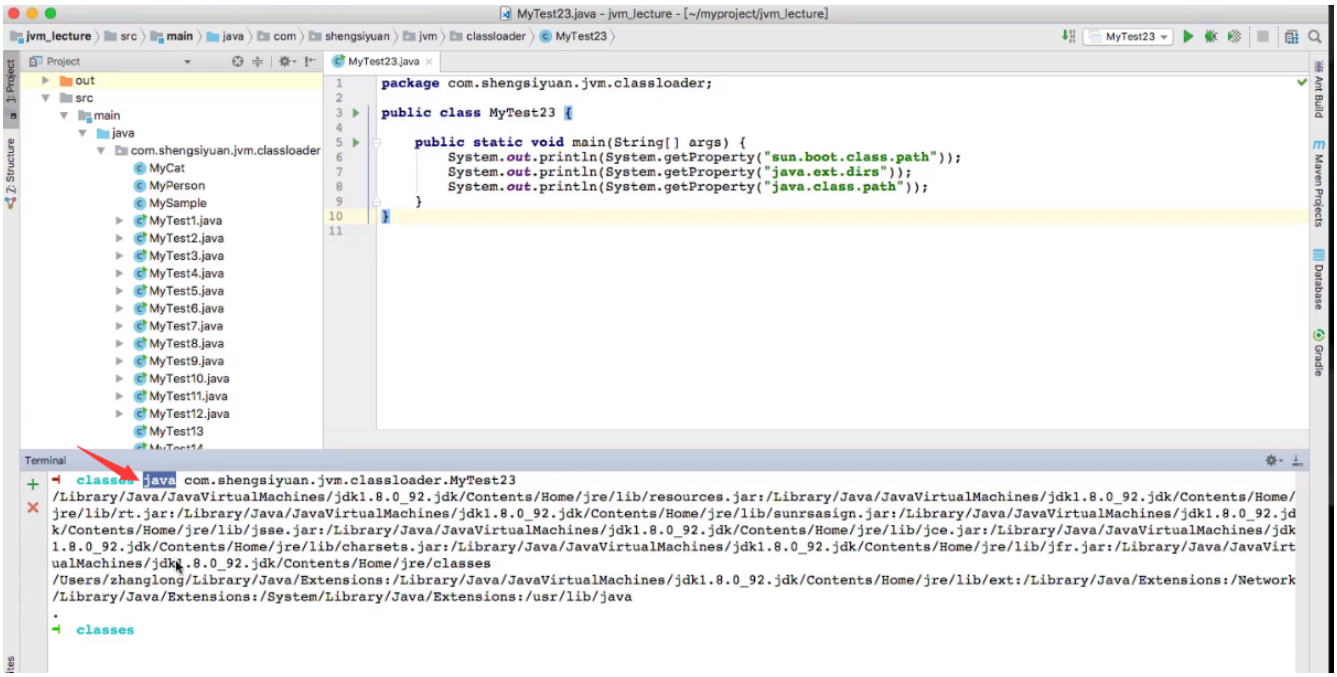
需要打包成jar包才会去加载，扩展类加载器才会加载。

打jar包的命令：

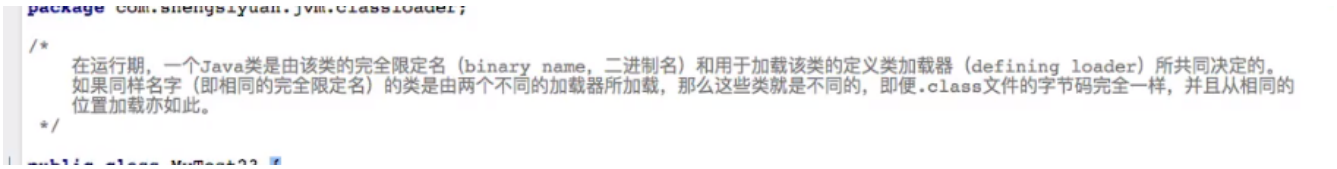
```
➜  classes jar cvf test.jar com/shengsiyuan/jvm/classloader/MyTest1.class
```



24平台特定的启动类加载器深入分析与自定义系统类加载器详解



点 标识当前路径



修改sun.boot.class.path

```
Terminal
classes java -Dsun.boot.class.path=./ com.shengsiyuan.jvm.classloader.MyTest23
Error occurred during initialization of VM
java/lang/NoClassDefFoundError: java/lang/Object
classes
```

```
/*
    在Oracle的Hotspot实现中，系统属性sun.boot.class.path如果修改错了，则运行会出错，提示如下错误信息：

    Error occurred during initialization of VM
    java/lang/NoClassDefFoundError: java/lang/Object
 */
```

特例： 数组不是由类加载器加载或创建的。而是由jvm在运行期间创建的。

类加载器也是类，他们又是又谁加载的。？？

这些类是由启动类加载器加载。

```
/*
    内建于JVM中的启动类加载器会加载java.lang.ClassLoader以及其他的Java平台类，
    当JVM启动时，一块特殊的机器码会运行，它会加载扩展类加载器与系统类加载器，
    这块特殊的机器码叫做启动类加载器（Bootstrap）。

    启动类加载器并不是Java类，而其他的加载器则都是Java类，
    启动类加载器是特定于平台的机器指令，它负责开启整个加载过程。

    所有类加载器（除了启动类加载器）都被实现为Java类。不过，总归要有一个组件来加载第一个Java类加载器，从而让整个加载过程能够顺利进行下去，加载第一个纯Java类加载器就是启动类加载器的职责。

    启动类加载器还会负责加载供JRE正常运行所需要的基本组件，这包括java.util与java.lang包中的类等等。
 */
```

```
15
16      public MyTest16(String classLoaderName) {
17          super(); // 将系统类加载器当做该类加载器的父加载器
18          this.classLoaderName = classLoaderName;
19      }
20
21      public MyTest16(ClassLoader parent, String classLoaderName) {
22          super(parent); // 显式指定该类加载器的父加载器
23          this.classLoaderName = classLoaderName;
24      }
25
26      public MyTest16(ClassLoader parent) {
27          super(parent);
28      }
29
30      public void setPath(String path) {
31          this.path = path;
32      }
```

```
classes java -Djava.system.class.loader=com.shengsiyuan.jvm.classloader.MyTest16 com.shengsiyuan.jvm.classloader.MyTest23
```

指定系统类加载器， 执行的类为后面的一个类。我指定的系统类加载器的父加载器还是appclassloader；

25 Launcher类源码分析

java.lang.ClassLoader public static ClassLoader getSystemClassLoader()

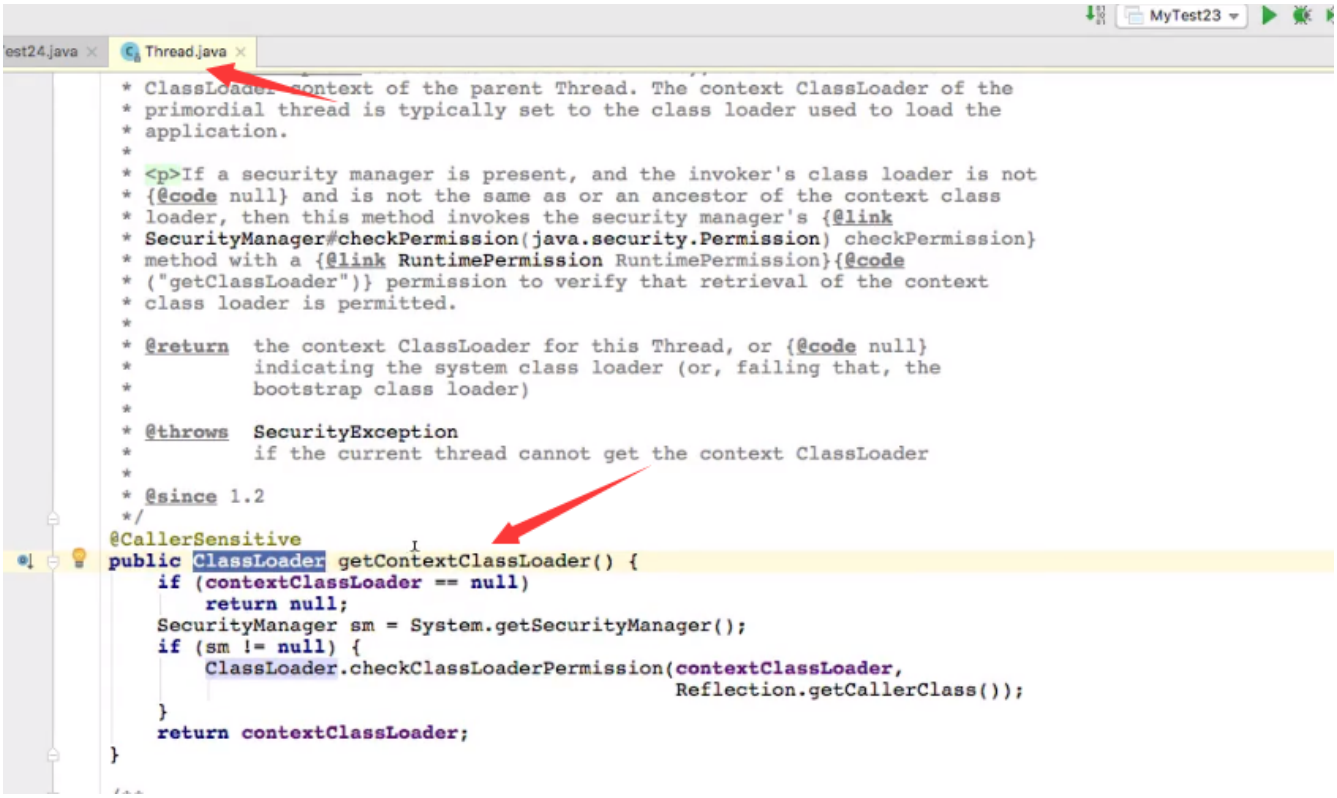
Returns the system class loader. This is the default delegation parent for new ClassLoader instances, and is typically the class loader used to start the application. This method is first invoked early in the runtime's startup sequence, at which point it creates the system class loader. This class loader will be the context class loader for the main application thread (for example, the thread that invokes the main method of the main class). The default system class loader is an implementation-dependent instance of this class. If the system property "java.system.class.loader" is defined when this method is first invoked then the value of that property is taken to be the name of a class that will be returned as the system class loader. The class is loaded using the default system class loader and must define a public constructor that takes a single parameter of type ClassLoader which is used as the delegation parent. An instance is then created using this constructor with the default system class loader as the parameter. The resulting class loader is defined to be the system class loader. During construction, the class loader should take great care to avoid calling getSystemClassLoader(). If circular initialization of the system class loader is detected then an IllegalStateException is thrown.

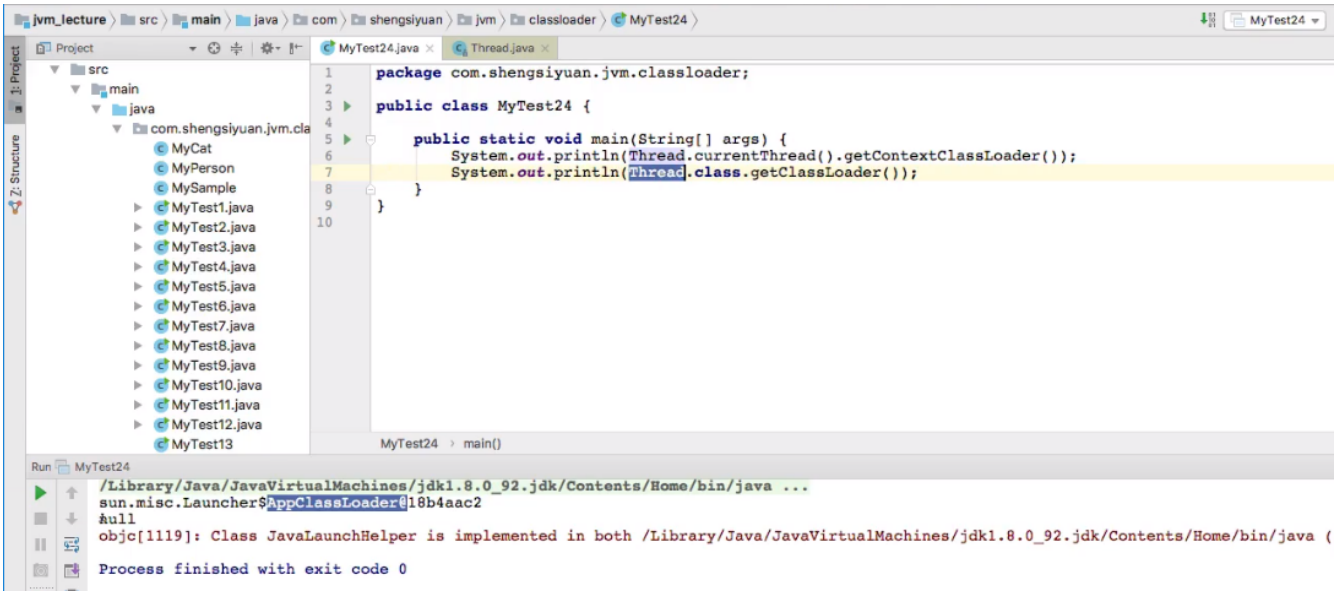
Returns: The system ClassLoader Throws: SecurityException – If a security manager is present, and the caller's class loader is not null and is not the same as or an ancestor of the system class loader, and the caller does not have the RuntimePermission("getClassLoader") IllegalStateException – If invoked recursively during the construction of the class loader specified by the "java.system.class.loader" property. Error – If the system property "java.system.class.loader" is defined but the named class could not be loaded, the provider class does not define the required constructor, or an exception is thrown by that constructor when it is invoked. The underlying cause of the error can be retrieved via the Throwable.getCause() method. implNote: The system property to override the system class loader is not examined until the VM is almost fully initialized. Code that executes this method during startup should take care not to cache the return value until the system is fully initialized. The name of the built-in system class loader is "app". The system property "java.class.path" is read during early initialization of the VM to determine the class path. An empty value of "java.class.path" property is interpreted differently depending on whether the initial module (the module containing the main class) is named or unnamed: If named, the built-in system class loader will have no class path and will search for classes and resources using the application module path; otherwise, if unnamed, it will set the class path to the current working directory.

26 自定义系统类加载器源码分析与classForName方法底层剖析。

```
    }

    public ClassLoader run() throws Exception {
        String cls = System.getProperty("java.system.class.loader");
        if (cls == null) {
            return parent;
        }

        Constructor<?> ctor = Class.forName(cls, true, parent)
            .getDeclaredConstructor(new Class<?>[] { ClassLoader.class });
        ClassLoader sys = (ClassLoader) ctor.newInstance(
            new Object[] { parent });
        Thread.currentThread().setContextClassLoader(sys);
        return sys;
    }
}
```

27 线程上下文类加载器分析与实现。





解释：线程上下文类加载器重要性：

```
/*
    当前类加载器（Current Classloader）

    每个类都会使用自己的类加载器（即加载自身的类加载器）来去加载其他类（指的是所依赖的类），
    如果ClassX引用了ClassY，那么ClassX的类加载器就会去加载ClassY（前提是ClassY尚未被加载）

    线程上下文类加载器（Context Classloader）

    线程上下文类加载器是从JDK 1.2开始引入的，类Thread中的getContextClassLoader()与setContextClassLoader(ClassLoader cl
    分别用来获取和设置上下文类加载器。

    如果没有通过setContextClassLoader(ClassLoader cl)进行设置的话，线程将继承其父线程的上下文类加载器。
    Java应用运行时的初始线程的上下文类加载器是系统类加载器。在线程中运行的代码可以通过该类加载器来加载类与资源。

    线程上下文类加载器的重要性：

    SPI (Service Provider Interface)

    父ClassLoader可以使用当前线程Thread.currentThread().getContextClassLoader()所指定的classloader加载的类。
    这就改变了父ClassLoader不能使用子ClassLoader或是其他没有直接父子关系的ClassLoader加载的类的情况，即改变了
    双亲委托模型。

    线程上下文类加载器就是当前线程的Current Classloader。

    在双亲委托模型下，类加载是由下至上的，即下层的类加载器会委托上层进行加载。但是对于SPI来说，有些接口是Java核心库所提供的，
    而Java核心库是由启动类加载器来加载的，而这些接口的实现却来自于不同的jar包（厂商提供），Java的启动类加载器是不会加载其他
    来源的jar包，这样传统的双亲委托模型就无法满足SPI的要求。而通过给当前线程设置上下文类加载器，就可以由设置的上下文类加载器
    来实现对于接口实现类的加载。

*/
```

28 线程类加载器本质剖析与实战

```java
package com.shengsiyuan.jvm.classloader;

public class MyTest25 implements Runnable{

    private Thread thread;

    public MyTest25() {
        thread = new Thread(this);
        thread.start();
    }

    @Override
    public void run() {
        ClassLoader classLoader = this.thread.getContextClassLoader();

        this.thread.setContextClassLoader(classLoader);

        System.out.println("Class: " + classLoader.getClass());
        System.out.println("Parent: " + classLoader.getParent().getClass());
    }

    public static void main(String[] args) {
        new MyTest25();
    }
}
```

```
MyTest14                                MyTest25 › main()
Run  MyTest25
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/j
Class: class sun.misc.Launcher$AppClassLoader
objc[1065]: Class JavaLaunchHelper is implemented in both /Library/Ja
Parent: class sun.misc.Launcher$ExtClassLoader

Process finished with exit code 0
```

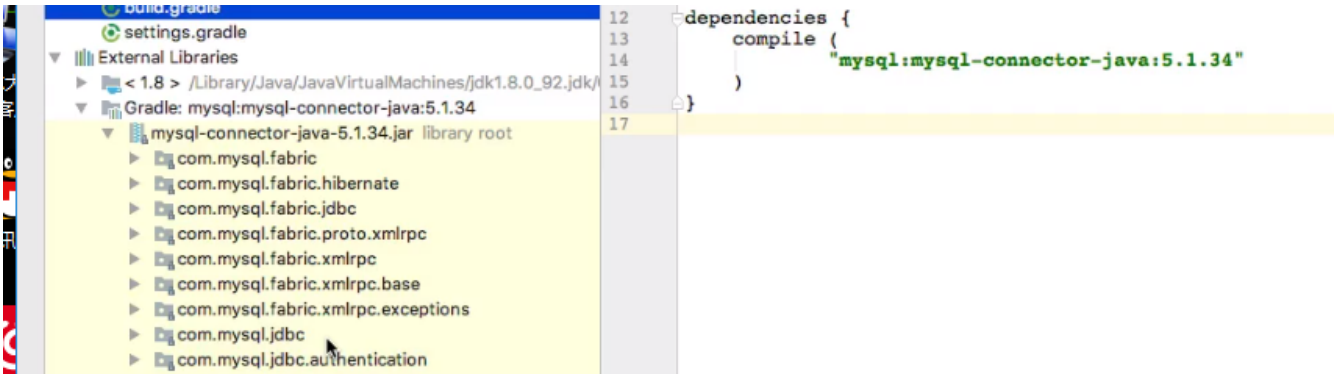
线程上下文类加载器的一般使用模式：获取-使用-还原

```
/*
    线程上下文类加载器的一般使用模式（获取 - 使用 - 还原）

    ClassLoader classLoader = Thread.currentThread().getContextClassLoader();

    try {
        Thread.currentThread().setContextClassLoader(targetTccl);
        myMethod();
    } finally {
        Thread.currentThread().setContextClassLoader(classLoader);
    }

    myMethod里面则调用了Thread.currentThread().getContextClassLoader()，获取当前线程的上下文类加载器做某些事情。

    如果一个类由类加载器A加载，那么这个类的依赖类也是由相同的类加载器加载的（如果该依赖类之前没有被加载过的话）

    ContextClassLoader的作用就是为了破坏Java的类加载委托机制。

    当高层提供了统一的接口让低层去实现，同时又要在高层加载（或实例化）低层的类时，就必须要通过线程上下文类加载器来帮助高层的ClassLoader
找到并加载该类。
*/
```

29 ServiceLoader在SPI中的重要作用分析(有时间再看一遍)

```
sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile (
            "mysql:mysql-connector-java:5.1.34"
    )
}
```

Gradle拉取jar包



serviceLoader 服务类加载器:

A service provider is identified by placing a *provider-configuration file* in the resource directory `META-INF/services`. The file's name is the fully-qualified binary name of the service's type. The file contains a list of fully-qualified binary names of concrete provider classes, one per line. Space and tab characters surrounding each name, as well as blank lines, are ignored. The comment character is `'#'` (`'\u0023'`, NUMBER SIGN); on each line all characters following the first comment character are ignored. The file must be encoded in UTF-8.

通过在资源目录META-INF/services中放置提供程序配置文件来标识服务提供程序。文件名是服务类型的完全限定二进制名称。该文件包含具体提供程序类的完全限定的二进制名称列表，每行一个。忽略每个名称周围的空格和制表符以及空行。注释字符是"（'\u0023'，数字符号）；在每行中，第一个注释字符后面的所有字符都将被忽略。文件必须以UTF-8编码。

< 1.8 >

java.util

public final class **ServiceLoader**<S>
extends Object
implements Iterable<S>

A simple service-provider loading facility.

A *service* is a well-known set of interfaces and (usually abstract) classes. A *service provider* is a specific implementation of a service. The classes in a provider typical implement the interfaces and subclass the classes defined in the service itself. Service providers can be installed in an implementation of the Java platform in the form of extensions, that is, jar files placed into any of the usual extension directories. Providers can also be made available by adding them to the application's class path o by some other platform-specific means.

For the purpose of loading, a service is represented by a single type, that is, a single interface or abstract class. (A concrete class can be used, but this is not recommended.) A provider of a given service contains one or more concrete classes that extend this *service type* with data and code specific to the provider. The *provider class* is typically not the entire provider itself but rather a proxy which contains enough information to decide whether the provider is able to satisfy a particul request together with code that can create the actual provider on demand. The details of provider classes tend to be highly service-specific; no single class or interface could possibly unify them, so no such type is defined here. The only requirement enforced by this facility is that provider classes must have a zero-argument constructor so that they can be instantiated during loading.

A service provider is identified by placing a *provider-configuration file* in the resource directory `META-INF/services`. The file's name is the fully-qualified binary name of the service's type. The file contains a list of fully-qualified binary names of concrete provider classes, one per line. Space and tab characters surrounding each name, as well as blank lines, are ignored. The comment character is `'#'` (`'\u0023'`, NUMBER SIGN); on each line all characters following the first comment character ar ignored. The file must be encoded in UTF-8.

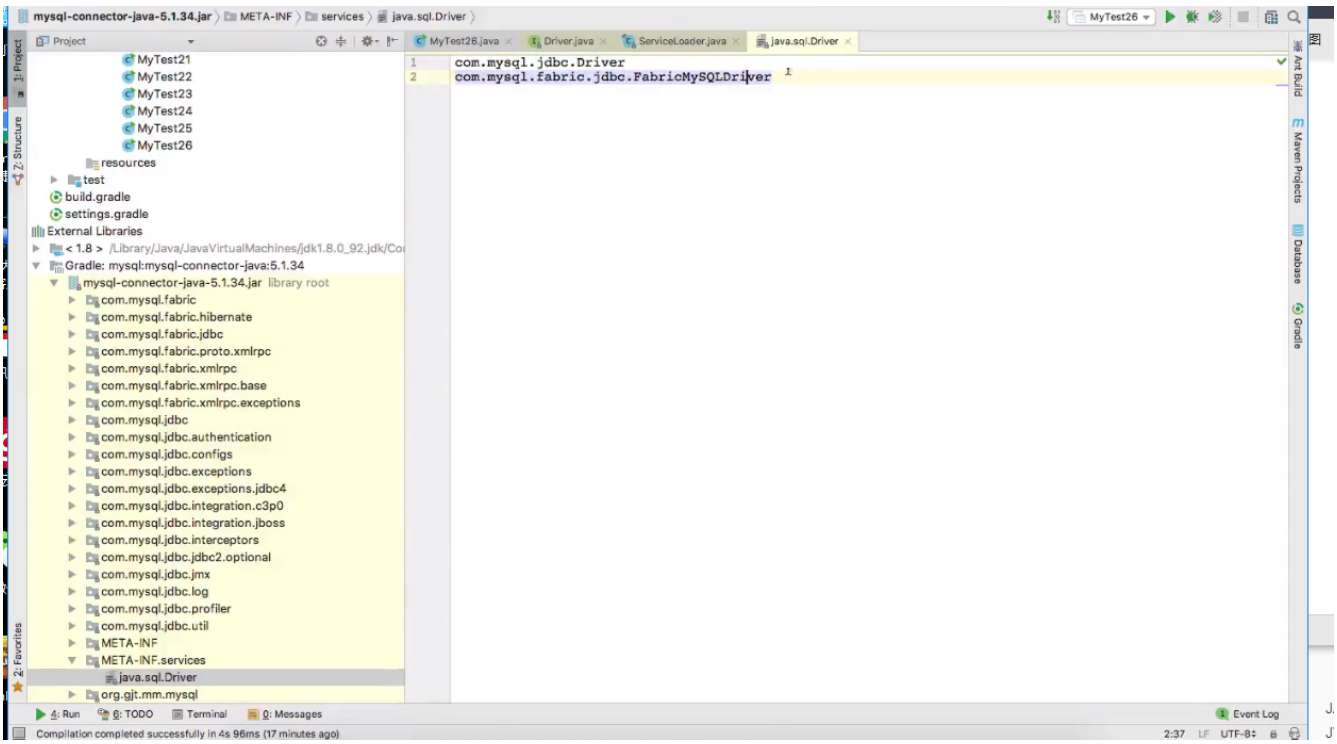
If a particular concrete provider class is named in more than one configuration file, or is named in the same configuration file more than once, then the duplicates are ignored. The configuration file naming a particular provider need not be in the same jar file or other distribution unit as the provider itself. The provider must be accessible from the same class loader that was initially queried to locate the configuration file; note that this is not necessarily the class loader from which the file was actually loaded.

Providers are located and instantiated lazily, that is, on demand. A service loader maintains a cache of the providers that have been loaded so far. Each invocation of the iterator method returns an iterator that first yields all of the elements of the cache, in instantiation order, and then lazily locates and instantiates any remaining providers, adding each one to the cache in turn. The cache can be cleared via the reload method.

Service loaders always execute in the security context of the caller. Trusted system code should typically invoke the methods in this class, and the methods of the iterators which they return, from within a privileged security context.

Instances of this class are not safe for use by multiple concurrent threads.

Unless otherwise specified, passing a `null` argument to any method in this class will cause a NullPointerException to be thrown.



29 第二遍观看：

javadoc

serviceLoader 服务类加载器

加载具体 的服务的实现的如jdbc，

A simple service-provider loading facility.

一个简单的服务提供商加载设施。

- A *service* is a well-known set of interfaces and (usually abstract) classes

服务提供者：service的具体实现 如jdbc

- A *service provider* is a specific implementation of a service
- **The only requirement enforced by this facility is that provider classes must have a zero-argument constructor so that they can be instantiated during loading.**

- A service provider is identified by placing a *provider-configuration file* in the resource directory `META-INF/services`. The file's name is the fully-qualified binary name of the service's type. The file contains a list of fully-qualified binary names of concrete provider classes, one per line. Space and tab characters surrounding each name, as well as blank lines, are ignored. The comment character is `'#'` (`'\u0023'`, NUMBER SIGN); on each line all characters following the first comment character are ignored. The file must be encoded in UTF-8.

通过在资源目录META-INF/services中放置提供程序配置文件来标识服务提供程序。文件名是服务类型的完全限定二进制名称。该文件包含具体提供程序类的完全限定的二进制名称列表，每行一个。忽略每个名称周围的空格和制表符以及空行。注释字符是"（'\u0023'，数字符号）；在每行中，第一个注释字符后面的所有字符都将被忽略。文件必须以UTF-8编码。

- Providers are located and instantiated lazily, that is, on demand. A service loader maintains a cache of the providers that have been loaded so far. Each invocation of the `iterator` method returns an iterator that first yields all of the elements of the cache, in instantiation order, and then lazily locates and instantiates any remaining providers, adding each one to the cache in turn. The cache can be cleared via the `reload` method.

private final List~~ instantiatedProviders = new ArrayList<>();~~

-

-

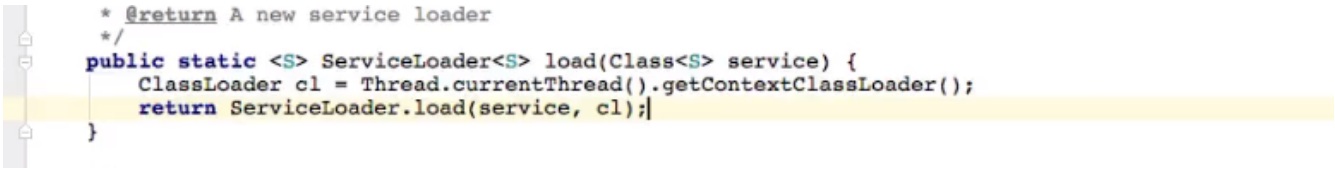
<

-

https://javadoc.allimant.org/>

30 线程上下文类加载器实战与分析

核心代码：



```
     * @return A new service loader
     */
    public static <S> ServiceLoader<S> load(Class<S> service) {
        ClassLoader cl = Thread.currentThread().getContextClassLoader();
        return ServiceLoader.load(service, cl);
    }
```

-

-

@CallerSensitive public static ServiceLoader load(Class service) {

//获取当前线程的上下文类加载器

ClassLoader cl = Thread.currentThread().getContextClassLoader();

//将要加载的服务程序和加载的类加载器作为参数

return new ServiceLoader<>(Reflection.getCallerClass(), service, cl); }

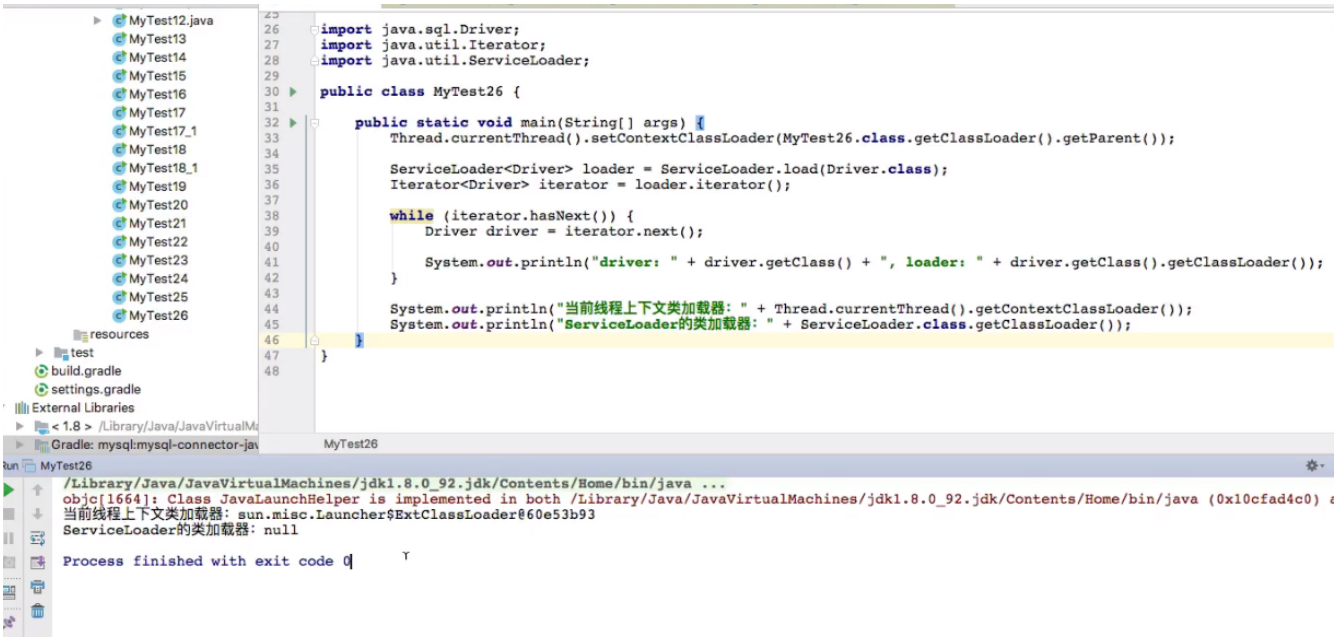
ServiceLoader loader=ServiceLoader.load(Driver.class);

因为ServiceLoader时jdk核心类，是由**启动类加载器**加载的，然后**启动类加载器**会去尝试加载Driver.class,但是这个类是在**classpath**下的，

所以加载不了，因此通过当前**线程的上下文类加载器**去加载，就可以加载，然后因为线程的上下文类加载器**不要求双亲委派机制**，

因而由**线程上下文类加载器**加载的类，对于启动类加载器来说也是可见的。

classforname 第二个参数boolean 表示是否去初始化。



~~31通过jdbc驱动加载深刻理解上下文类加载机制~~

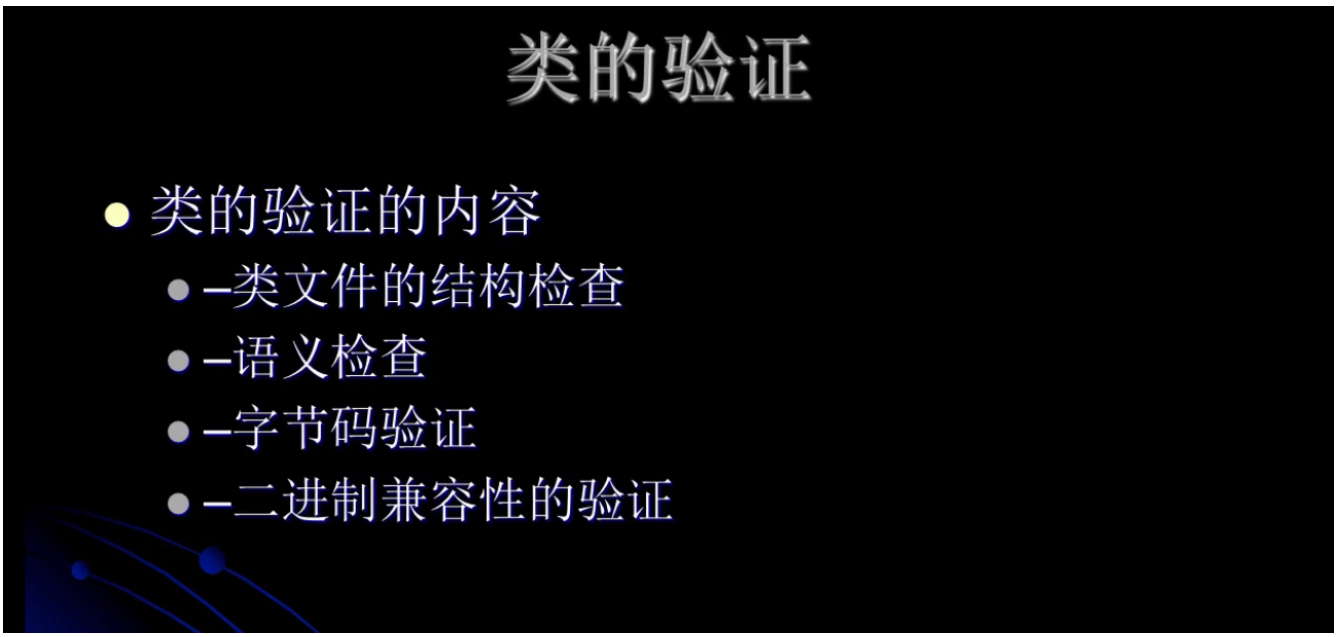
~~32 jvm类加载器总结与学习方式升级；~~

~~34类加载器系统回顾与内~~

-

~~35类加载精华：~~

-



-

-

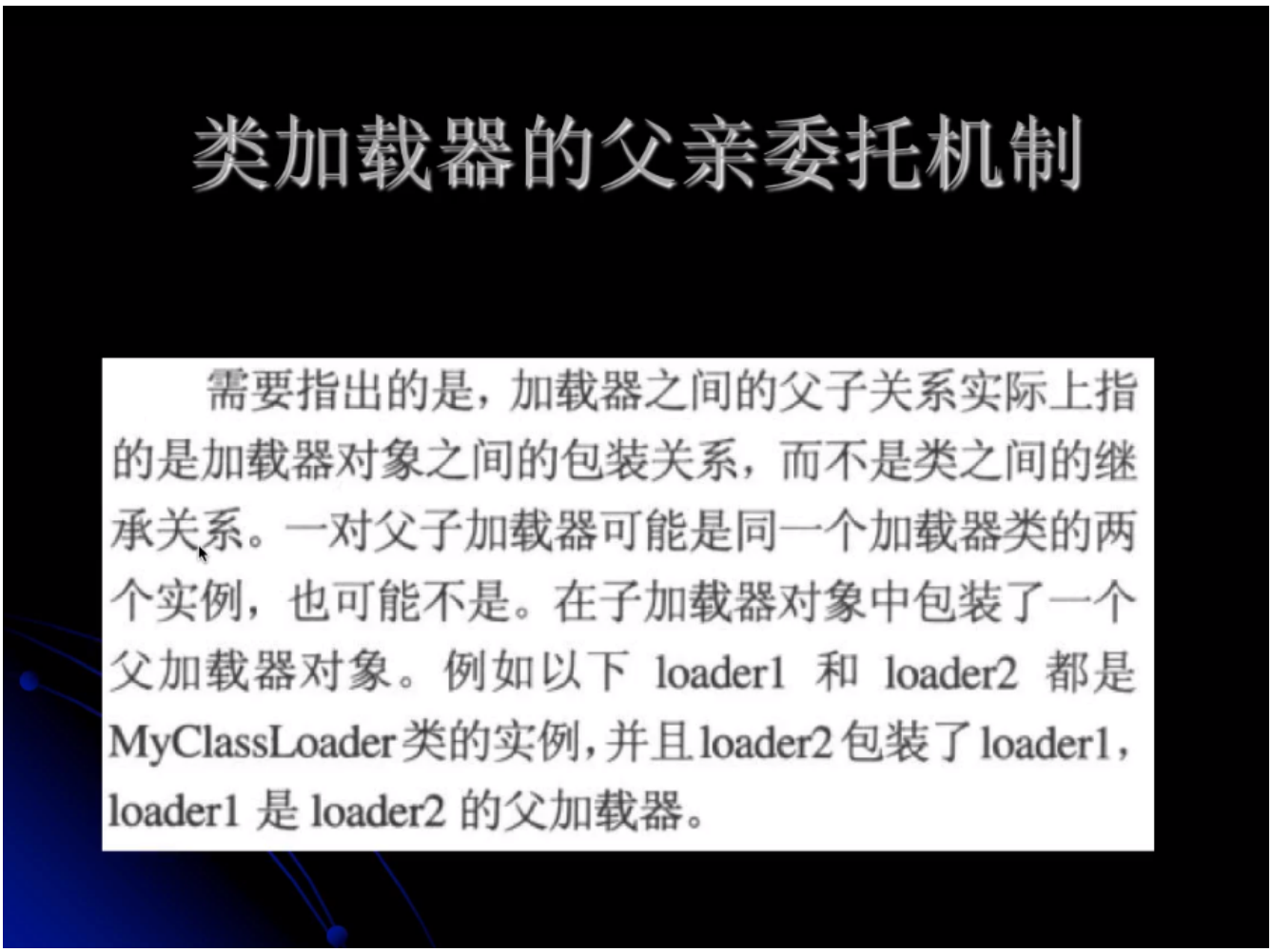
类的初始化时机

- 调用ClassLoader类的loadClass方法加载一个类，并不是对类的主动使用，不会导致类的初始化。

Jar hell问题以及解决办法

- 当一个类或者一个资源文件存在多个jar中，就会存在jar hell问题。
- 可以通过以下代码来诊断问题：

```
ClassLoader classLoader = Thread.currentThread()
        .getContextClassLoader();
String reosurceName = "java/lang/String.class";

Enumeration<URL> urls = classLoader.getResources(reosurceName);
while (urls.hasMoreElements())
{
    URL url = urls.nextElement();
    System.out.println(url);
```



父亲委托机制的优点是能够提高软件系统的安全性。因为在此机制下，用户自定义的类加载器不可能加载应该由父加载器加载的可靠类，从而防止不可靠甚至恶意的代码代替由父加载器加载的可靠代码。例如，java.lang.Object 类总是由根类加载器加载，其他任何用户自定义的类加载器都不可能加载含有恶意代码的 java.lang.Object 类

~~重点必备：~~

父亲委托机制的优点是能够提高软件系统的安全性。因为在此机制下，用户自定义的类加载器不可能加载应该由父加载器加载的可靠类，从而防止不可靠甚至恶意的代码代替由父加载器加载的可靠代码。例如，java.lang.Object 类总是由根类加载器加载，其他任何用户自定义的类加载器都不可能加载含有恶意代码的 java.lang.Object 类

-

类的卸载

- 当MySample类被加载、连接和初始化后，它的生命周期就开始了。当代表MySample类的Class对象不再被引用，即不可触及时，Class对象就会结束生命周期，MySample类在方法区内的数据也会被卸载，从而结束MySample类的生命周期。
- 一个类何时结束生命周期，取决于代表它的Class对象何时结束生命周期

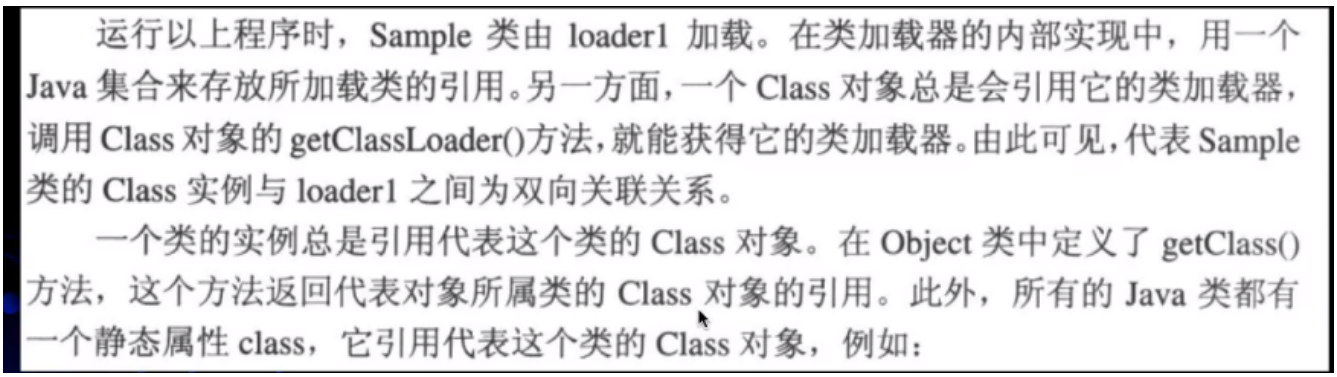
-

~~重点:~~

类的卸载

由 Java 虚拟机自带的类加载器所加载的类，在虚拟机的生命周期中，始终不会被卸载。前面已经介绍过，Java 虚拟机自带的类加载器包括根类加载器、扩展类加载器和系统类加载器。Java 虚拟机本身会始终引用这些类加载器，而这些类加载器则会始终引用它们所加载的类的 Class 对象，因此这些 Class 对象始终是可触及的。

- 由用户自定义的类加载器所加载的类是可以被卸载的



运行以上程序时，Sample 类由 loader1 加载。在类加载器的内部实现中，用一个 Java 集合来存放所加载类的引用。另一方面，一个 Class 对象总是会引用它的类加载器，调用 Class 对象的 getClassLoader()方法，就能获得它的类加载器。由此可见，代表 Sample 类的 Class 实例与 loader1 之间为双向关联关系。

一个类的实例总是引用代表这个类的 Class 对象。在 Object 类中定义了 getClass()方法，这个方法返回代表对象所属类的 Class 对象的引用。此外，所有的 Java 类都有一个静态属性 class，它引用代表这个类的 Class 对象，例如：

~~37 java字节码结构剖析~~

```
package com.jlu.bytecode;

public class MyTest1 {
    private int a=1;

    public int getA() {
        return a;
    }

    public void setA(int a) {
        this.a = a;
    }
}
```

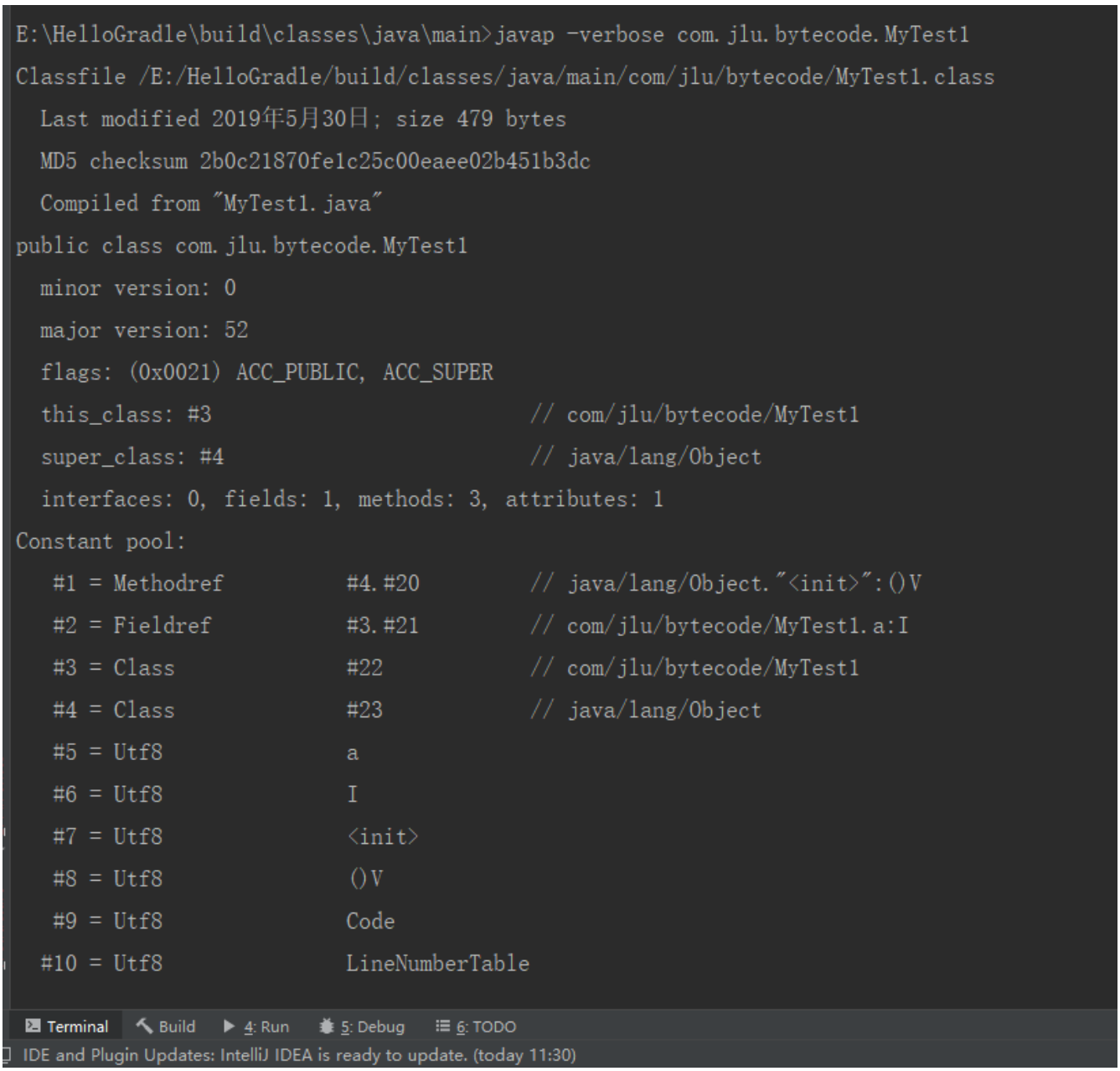
~~39 透彻分析常量池 常量池结构与描述符~~

```
E:\HelloGradle\build\classes\java\main>javap -c com.jlu.bytecode.MyTest1
Compiled from "MyTest1.java"
public class com.jlu.bytecode.MyTest1 {
  public com.jlu.bytecode.MyTest1();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: aload_0
       5: iconst_1
       6: putfield      #2                  // Field a:I
       9: return

  public int getA();
    Code:
       0: aload_0
       1: getfield      #2                  // Field a:I
       4: ireturn

  public void setA(int);
    Code:
       0: aload_0
       1: iload_1
       2: putfield      #2                  // Field a:I
```

-

~~javap -verbose com.jlu.bytecode.MyTest1~~

-



-

~~常量池也可以有变量的。~~

~~字面量~~

Class文件结构中常量池中11种数据类型的结构总表

常量	项目	类型	描述
CONSTANT_Utf8_info	tag	U1	值为1
	length	U2	UTF-8编码的字符串长度
	bytes	U1	长度为length的UTF-8编码的字符串
CONSTANT_Integer_info	tag	U1	值为3
	bytes	U4	按照高位在前存储的int值
CONSTANT_Float_info	tag	U1	值为4
	bytes	U4	按照高位在前存储的float值
CONSTANT_Long_info	tag	U1	值为5
	bytes	U8	按照高位在前存储的long值
CONSTANT_Double_info	tag	U1	值为6
	bytes	U8	按照高位在前存储的double值
CONSTANT_Class_info	tag	U1	值为7
	index	U2	指向全限定名常量项的索引
CONSTANT_String_info	tag	U1	值为8
	index	U2	指向字符串字面量的索引
CONSTANT_Fieldref_info	tag	U1	值为9
	index	U2	指向声明字段的类或者接口描述符CONSTANT_Class_info的索引项
	index	U2	指向字段描述符CONSTANT_NameAndType_info的索引项
CONSTANT_Methodref_info	tag	U1	值为10
	index	U2	指向声明方法的类描述符CONSTANT_Class_info的索引项
	index	U2	指向名称及类型描述符CONSTANT_NameAndType_info的索引项
CONSTANT_InterfaceMethodref_info	tag	U1	值为11
	index	U2	指向声明方法的接口描述符CONSTANT_Class_info的索引项
	index	U2	指向名称及类型描述符CONSTANT_NameAndType_info的索引项
CONSTANT_NameAndType_info	tag	U1	值为12
	index	U2	指向该字段或方法名称常量项的索引
	index	U2	指向该字段或方法描述符常量项的索引

-

~~u1 一个字节~~

~~u2 两个字节~~

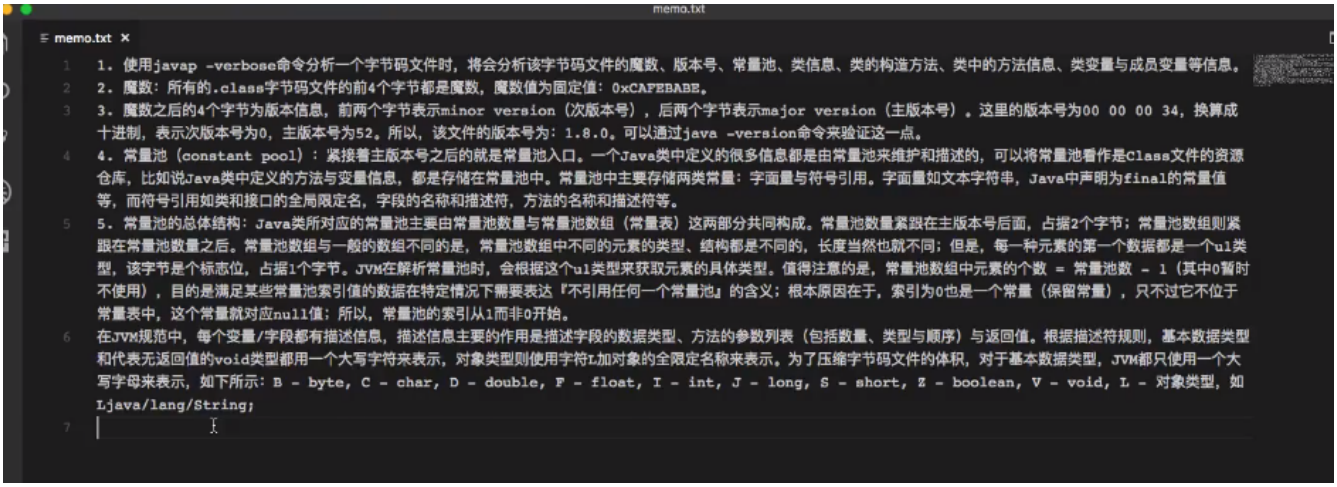
~~u4 四个字节~~

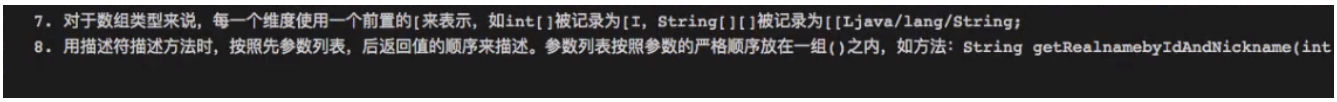
~~u8 8个字节~~

~~字面量~~

~~全限定名常量~~

~~字段描述符 nameAndType 变量的名字及其类型，或者方法的参数列表及其返回类型~~





-

-

~~#1 = Methodref #4.#20 // java/lang/Object."":()V #2 = Fieldref #3.#21 // com/jlu/bytecode/MyTest1.a:I #3 = Class #22 // com/jlu/bytecode/MyTest1 #4 = Class #23 // java/lang/Object #5 = Utf8 a #6 = Utf8 I #7 = Utf8 //表示类的构造方法 #8 = Utf8 ()V //无参数返回值为void #9 = Utf8 Code #10 = Utf8 LineNumberTable #11 = Utf8 LocalVariableTable #12 = Utf8 this #13 = Utf8 Lcom/jlu/bytecode/MyTest1; #14 = Utf8 getA #15 = Utf8 ()I #16 = Utf8 setA #17 = Utf8 (I)V #18 = Utf8 SourceFile #19 = Utf8 MyTest1.java #20 = NameAndType #7:#8 // "":()V #21 = NameAndType #5:#6 // a:I #22 = Utf8 com/jlu/bytecode/MyTest1 #23 = Utf8 java/lang/Object~~

~~////// #表示引用 或者 索引项~~

-

~~java字节码结构~~

~~this class name 表示一个常量池的引用，这个引用表示该类名~~

~~Super Class Name 表示一个常量池的引用，这个引用表示该类的父类名~~

~~他们使用的都是索引。~~

字段表集合

- 字段表用于描述类和接口中声明的变量。这里的字段包含了类级别变量以及实例变量，但是不包括方法内部声明的局部变量

静态变量

-

变量表是由自己的结构的。

字段表集合

- fields_count：u2

字段表结构

类型	名称	数量
u2	access_flags	1
u2	name_index	1
u2	descriptor_index	1
u2	attributes_count	1
attribute_info	attributes	attributes_count

-

-

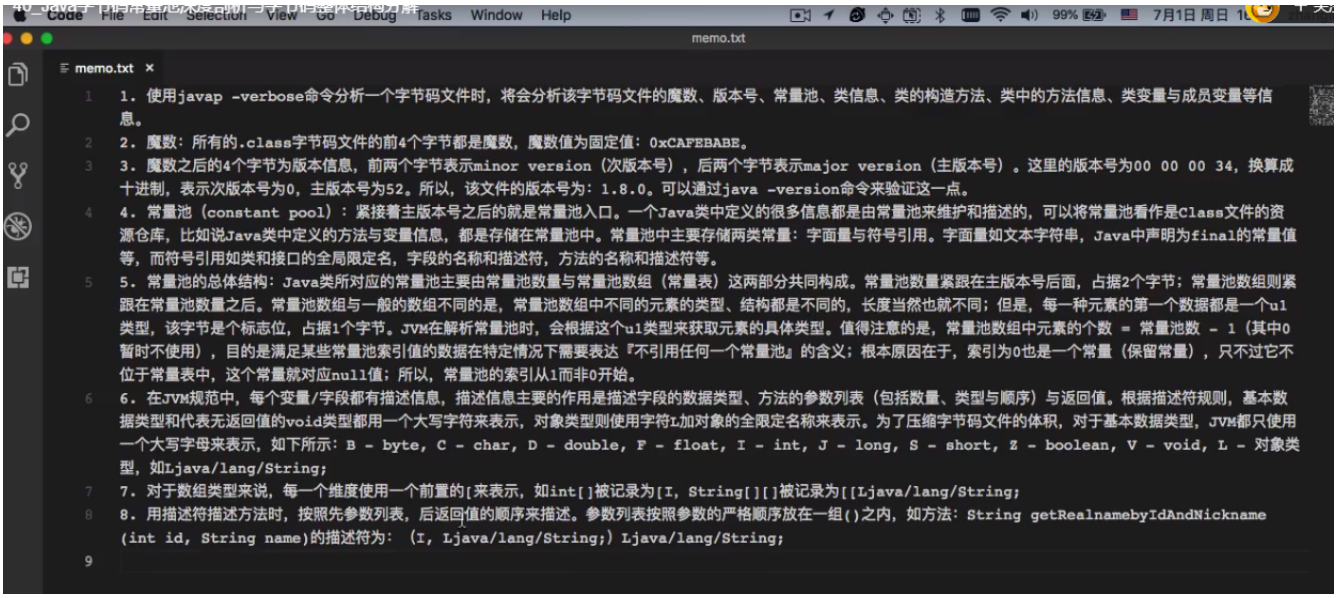
-

-

-

~~40 java字节码常量池与字节码整体结构~~

-



-

~~41 字节码详解~~

-

Java字节码整体结构

类型	名称	数量
u4	magic(魔术)	1
u2	minor_version(次版本号)	1
u2	major_version(主版本号)	1
u2	constant_pool_count(常量个数)	1
cp_info	constant_pool(常量池表)	constant_pool_count-1
u2	access_flags(类的访问控制权限)	1
u2	this_class(类名)	1
u2	super_class(父类名)	1
u2	interfaces_count(接口个数)	1
u2	interfaces(接口名)	interfaces_count
u2	fields_count(域个数)	1
field_info	fields(域的表)	fields_count
u2	methods_count(方法的个数)	1
method_info	methods(方法表)	methods_count
u2	attributes_count(附加属性的个数)	1
attribute_info	attributes(附加属性的表)	attributes_count

-

-

完整的Java字节码结构

```
ClassFile {
    u4              magic;
    u2              minor_version;
    u2              major_version;
    u2              constant_pool_count;
    cp_info         contant_pool[constant_pool_count - 1];
    u2              access_flags;
    u2              this_class;
    u2              super_class;
    u2              interfaces_count;
    u2              interfaces[interfaces_count];
    u2              fields_count;
    field_info      fields[fields_count];
    u2              methods_count;
    method_info     methods[methods_count];
    u2              attributes_count;
    attribute_info  attributes[attributes_count];
}
```

-

Java字节码结构

- Class字节码中有两种数据类型
  - 字节数据直接量：这是基本的数据类型。共细分为u1、u2、u4、u8四种，分别代表连续的1个字节、2个字节、4个字节、8个字节组成的整体数据。
  - 表（数组）：表是由多个基本数据或其他表，按照既定顺序组成的大的数据集合。表是有结构的，它的结构体现在：组成表的成分所在的位置和顺序都是已经严格定义好的。

-

Access_Flag 访问标志

- 访问标志信息包括该Class文件是类还是接口，是否被定义成public，是否是abstract，如果是类，是否被声明成final。通过上面的源代码，我们知道该文件是类并且是public。

-

Access_Flag 访问标志

Table 4.1. Class access and property modifiers

Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared `public`; may be accessed from outside its package.
ACC_FINAL	0x0010	Declared `final`; no subclasses allowed.
ACC_SUPER	0x0020	Treat superclass methods specially when invoked by the *invokespecial* instruction.
ACC_INTERFACE	0x0200	Is an interface, not a class.
ACC_ABSTRACT	0x0400	Declared `abstract`; must not be instantiated.
ACC_SYNTHETIC	0x1000	Declared synthetic; not present in the source code.
ACC_ANNOTATION	0x2000	Declared as an annotation type.
ACC_ENUM	0x4000	Declared as an `enum` type.

~~acc_private value 0x0002~~

Access_Flag 访问标志

- 0x 00 21：是0x 0020和0x 0001的并集，表示ACC_PUBLIC与ACC_SUPER

-

-

字段表集合

- fields_count：u2

字段表结构

类型	名称	数量
u2	access_flags	1
u2	name_index	1
u2	descriptor_index	1
u2	attributes_count	1
attribute_info	attributes	attributes_count

-

字段表结构

- 单击此处添加文本

```
field_info {
        u2 access_flags; 0002
        u2 name_index; 0005
        u2 descriptor_index; 0006
        u2 attributes_count; 0000
        attribute_info attributes[attributes_count];
}
```

-

-

方法表

- methods_count：u2

方法表结构

类型	名称	数量
u2	access_flags	1
u2	name_index	1
u2	descriptor_index	1
u2	attributes_count	1
attribute_info	attributes	attributes_count

-

方法的属性结构

- 方法中的每个属性都是一个attribute_info结构

```
attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
}
```

-

~~Code:方法的执行代码~~

-

-

方法的属性结构

- JVM预定义了部分attribute，但是编译器自己也可以实现自己的attribute写入class文件里，供运行时使用。
- 不同的attribute通过attribute_name_index来区分

-

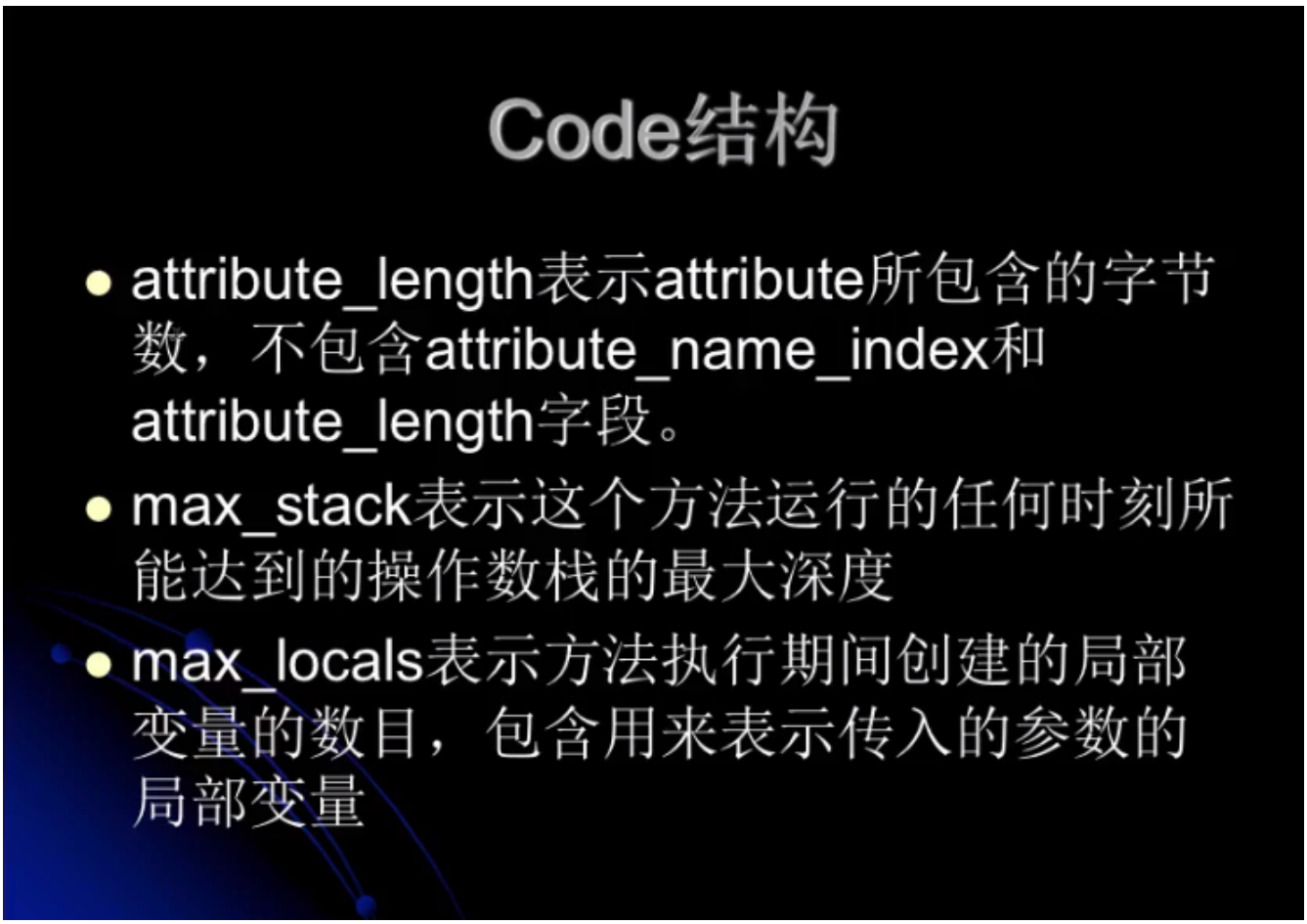
JVM规范预定义的attribute

Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared public; may be accessed from outside its package.
ACC_FINAL	0x0010	Declared final; no subclasses allowed.
ACC_SUPER	0x0020	Treat superclass methods specially when invoked by the *invokespecial* instruction.
ACC_INTERFACE	0x0200	Is an interface, not a class.
ACC_ABSTRACT	0x0400	Declared abstract; must not be instantiated.
ACC_SYNTHETIC	0x1000	Declared synthetic; not present in the source code.
ACC_ANNOTATION	0x2000	Declared as an annotation type.
ACC_ENUM	0x4000	Declared as an enum type.

Code结构

- Code attribute的作用是保存该方法的结构，如所对应的字节码

```
Code_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 max_stack;
    u2 max_locals;
    u4 code_length;
    u1 code[code_length];
    u2 exception_table_length;
    {
        u2 start_pc;
        u2 end_pc;
        u2 handler_pc;
        u2 catch_type;
    } exception_table[exception_table_length];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```



-

-

~~字节码查看工具 classlib~~

~~https://github.com/ingokegel/jclasslib/~~

-

~~46 sychronized 关键字详解~~

~~只要类中有静态变量，就会有静态代码块，对其处理。~~

~~静态代码快中是不能存在this的，实例上下文的信息。~~

~~javap -verbose -p~~

~~显示包括私有信息。~~

~~加了synchronized~~

```
未命名.txt — 已编辑
private void setX(int);
    descriptor: (I)V
    flags: ACC_PRIVATE
    Code:
      stack=2, locals=2, args_size=2
         0: aload_0
         1: iload_1
         2: putfield      #4                  // Field x:I
         5: return
      LineNumberTable:
        line 20: 0
        line 21: 5
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       6     0  this   Lcom/shengsiyuan/jvm/bytecode/MyTest2;
            0       6     1     x   I


private synchronized void setX(int);
    descriptor: (I)V
    flags: ACC_PRIVATE, ACC_SYNCHRONIZED
    Code:
      stack=2, locals=2, args_size=2
         0: aload_0
         1: iload_1
         2: putfield      #4                  // Field x:I
         5: return
      LineNumberTable:
        line 20: 0
        line 21: 5
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       6     0  this   Lcom/shengsiyuan/jvm/bytecode/MyTest2;
            0       6     1     x   I
```

-

```
moniterenter
moniterexit
```

~~虽然在这个方法中没有看到monitorenter等，但是实际上这个方法还是被上锁了的。~~

~~加锁和释放锁。~~

~~synchronized的使用方式~~

```
private synchronized void setX(int x) {
    this.x = x;
}

private void test(String str) {
    synchronized (str) {
        System.out.println("hello world");
    }
}
```

放在方法外面

放一个对象

~~this.object~~

```
private synchronized static void test2() {

}
```

对静态方法上锁，实际上是对该类的class对象上锁，

~~而其他的是对该对象实例上锁。~~

```
private void test(java.lang.String);
    descriptor: (Ljava/lang/String;)V
    flags: ACC_PRIVATE
    Code:
      stack=2, locals=4, args_size=2
         0: aload_1
         1: dup
         2: astore_2
         3: monitorenter
         4: getstatic     #10                 // Field java/lang/System.out:Ljava/io/PrintStream;
         7: ldc           #11                 // String hello world
         9: invokevirtual #12                 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
        12: aload_2
        13: monitorexit
        14: goto          22
        17: astore_3
        18: aload_2
        19: monitorexit
        20: aload_3
        21: athrow
        22: return
      Exception table:
         from    to  target type
             4    14    17   any
```

~~不止一个出口，是因为程序的退出有很多情况，但是都要释放锁。~~

-

~~47 复杂字节码的分析过程，~~

~~48 构造方法与静态代码块~~

-

~~成员变量或实例变量的赋值是在构造方法里面去执行的（对象实例化期间去执行），~~

~~静态变量是在静态方法中去赋值的。（我们赋予的值，而非准备阶段jvm给予的对应类型的默认值）（所有的静态代码块都会被合并到一个方法中去初始化）~~

~~如果自己提供了默认构造方法的时候：？？？~~

~~他依然会将赋值放（赋值的语句添加到）在我们的所有的构造方法中，因而无论用哪个方法初始化，都会对成员变量赋值到。~~

-

-

~~* monitorenter*~~

Operation

Enter monitor for object

Format

monitorenter

Forms

monitorenter = 194 (0xc2)

Operand Stack

..., *objectref* →

...

Description

The *objectref* must be of type `reference`.

Each object is associated with a monitor. A monitor is locked if and only if it has an owner. The thread that executes *monitorenter* attempts to gain ownership of the monitor associated with *objectref*, as follows:

- If the entry count of the monitor associated with *objectref* is zero, the thread enters the monitor and sets its entry count to one. The thread is then the owner of the monitor.
- If the thread already owns the monitor associated with *objectref*, it reenters the monitor, incrementing its entry count.
- If another thread already owns the monitor associated with *objectref*, the thread blocks until the monitor's entry count is zero, then tries again to gain ownership.

49 通过字节码分析this关键字以及异常表的作用

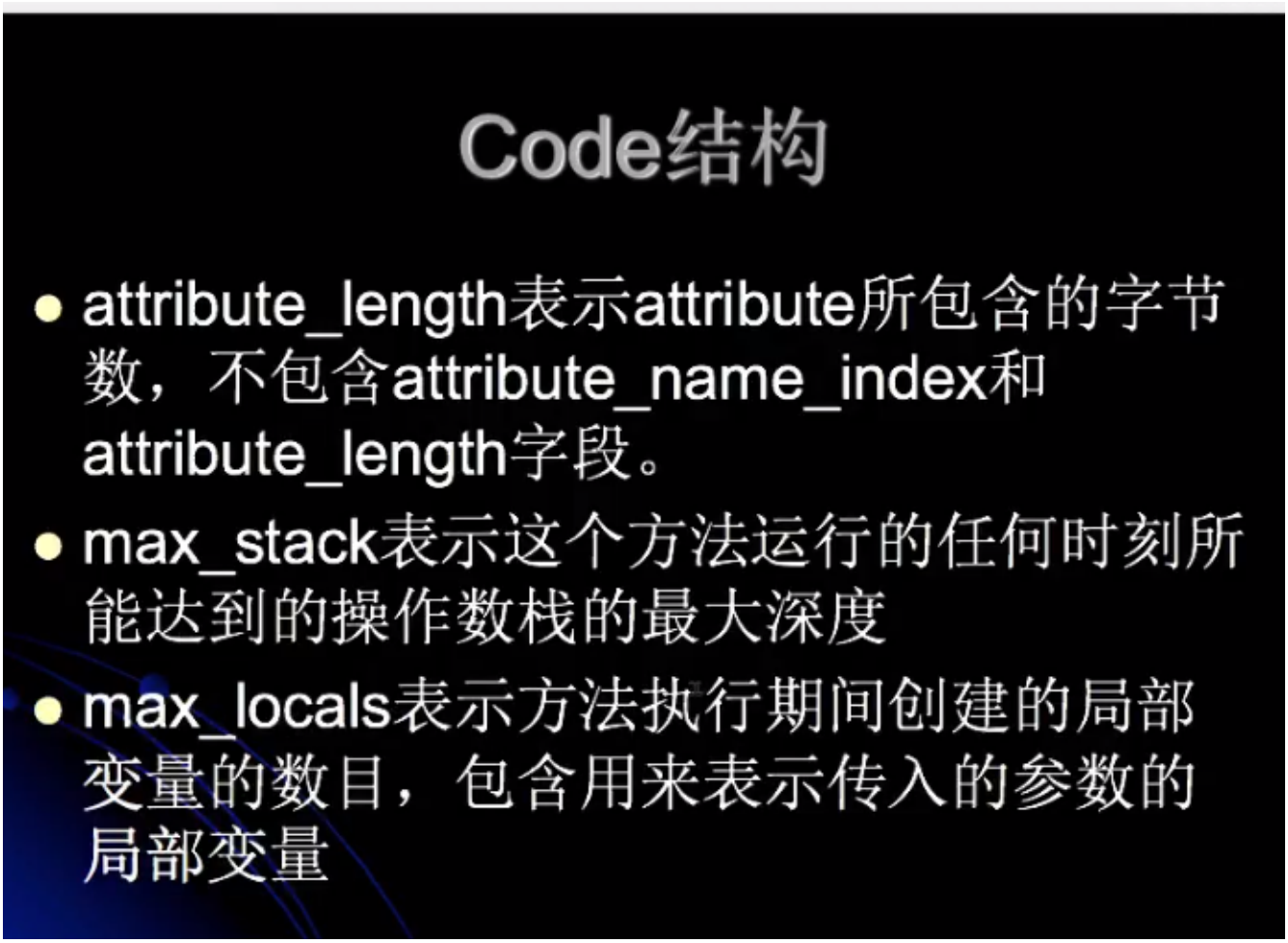
对于java中的每一个非静态的实例方法来说，都会有隐含的一个参数this 指向当前对象。在编译的时候，在生成class文件的时候就会完成这个操作。

而静态方法没有的原因是 1 用不到，2静态方法是属于该类所对应的对象

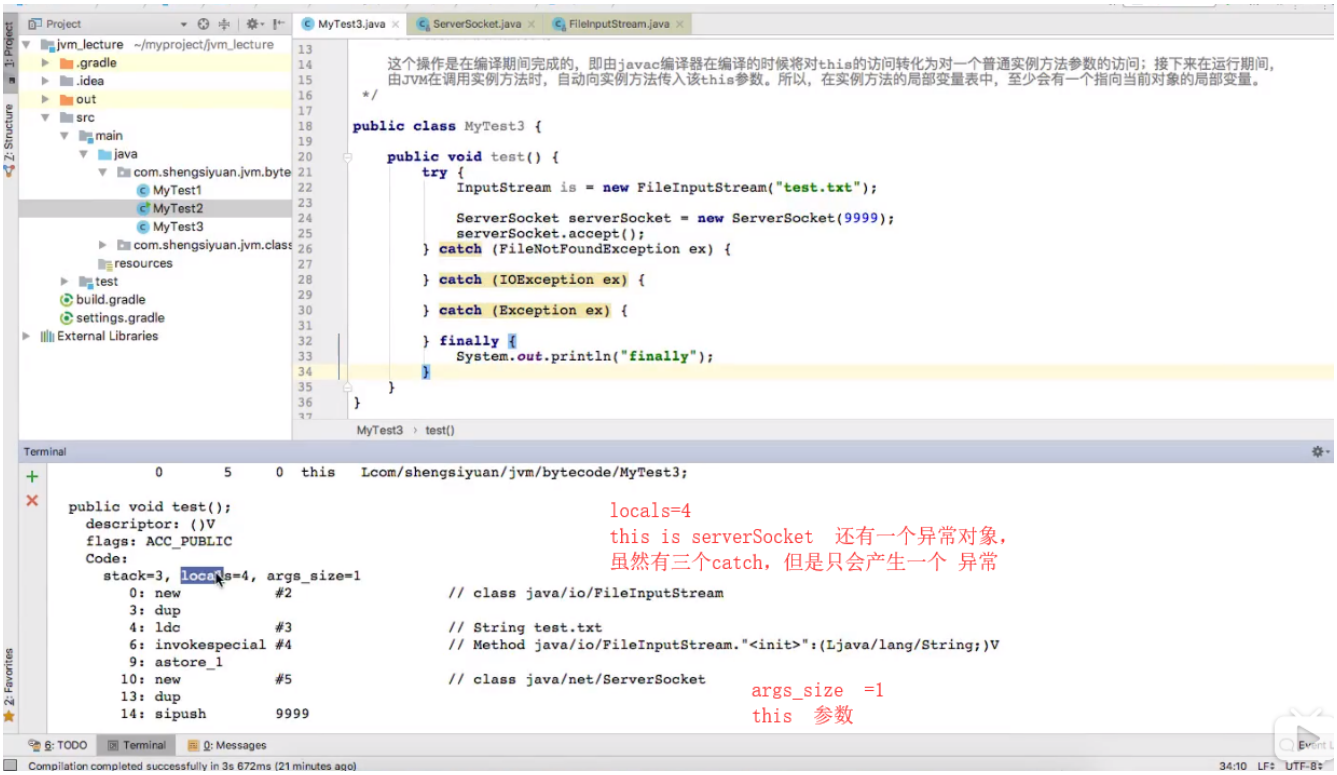
```
/*
    对于Java类中的每一个实例方法（非static方法），其在编译后所生成的字节码当中，方法参数的数量总是会比源代码中方法
    参数的数量多一个（this），它位于方法的第一个参数位置处；这样，我们就可以在Java的实例方法中使用this来去访问当前
    对象的属性以及其他方法。

    这个操作是在编译期间完成的，即由javac编译器在编译的时候将对this的访问转化为对一个普通实例方法参数的访问，接下来在运行期间，
    由JVM在调用实例方法时，自动向实例方法传入该this参数。所以，在实例方法的局部变量表中，至少会有一个指向当前对象的局部变量。
*/
```

局部变量：~~方法的参数加上方法内定义的变量。~~

-



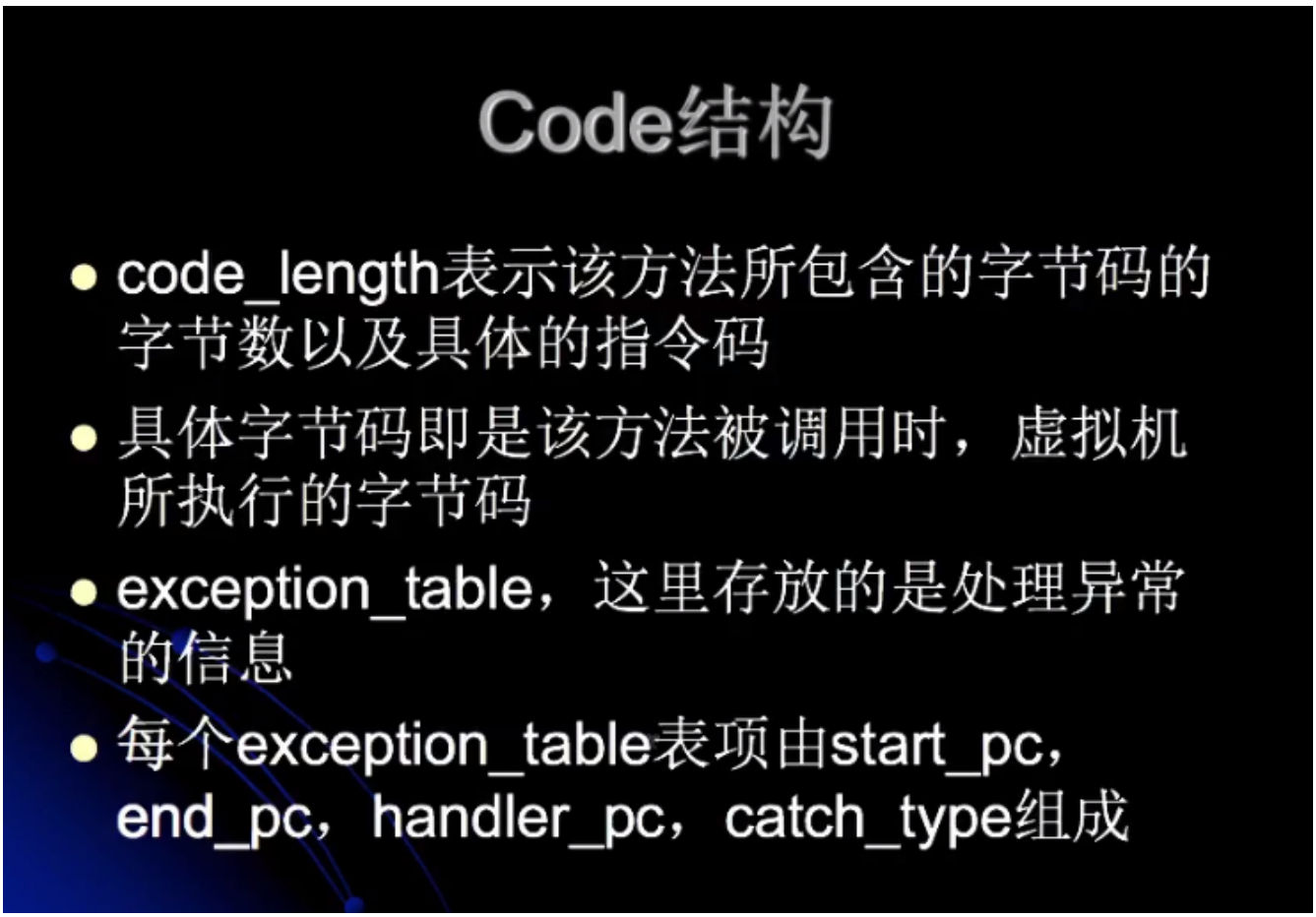
-



-

~~stack=3，对应locals的个数。减去this~~

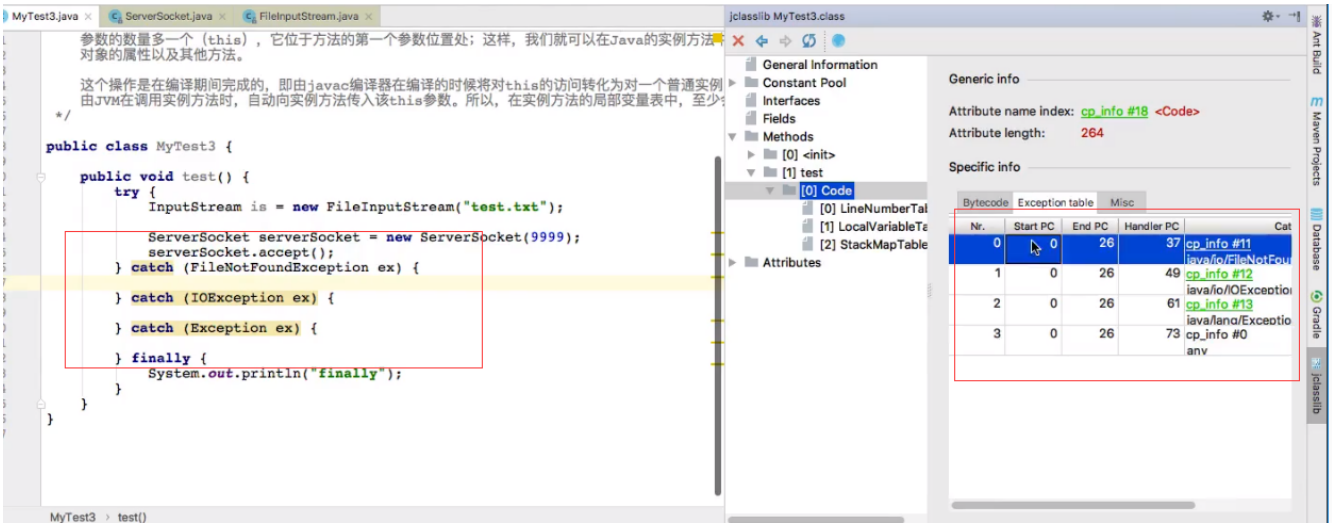
-

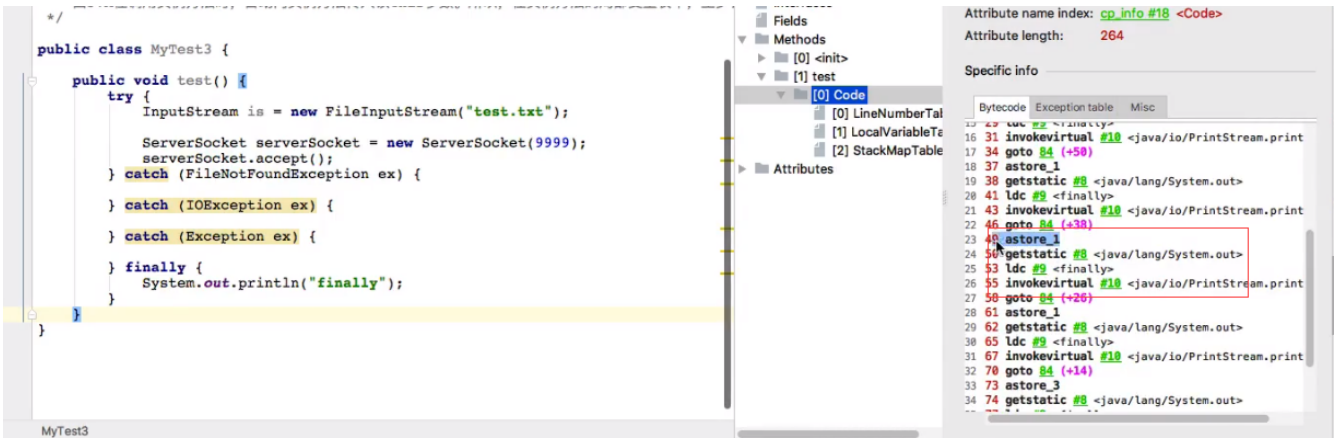


-

-

~~50 通过字节码分析java异常的处理机制~~





-

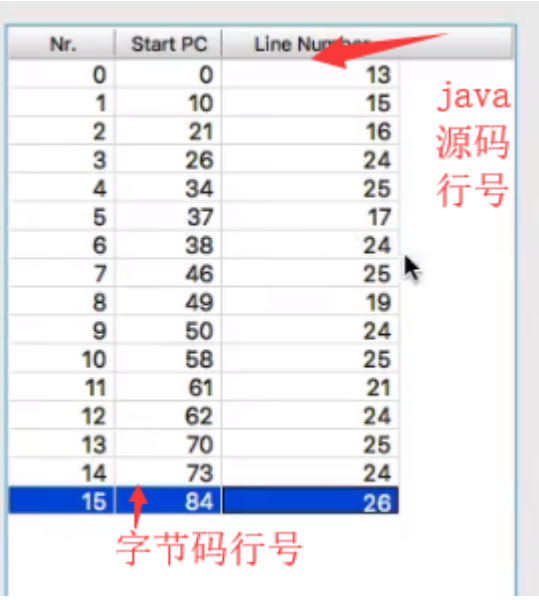
-

-

```
/*
    Java字节码对于异常的处理方式：

    1. 统一采用异常表的方式来对异常进行处理。
    2. 在jdk 1.4.2之前的版本中，并不是使用异常表的方式来对异常进行处理的，而是采用特定的指令方式。
    3. 当异常处理存在finally语句块时，现代化的JVM采取的处理方式是将finally语句块的字节码拼接到每一个catch块后面，
       换句话说，程序中存在多少个catch块，就会在每一个catch块后面重复多少个finally语句块的字节码。
 */
```

-

~~LineNumberTable：对应字节码和源代码在行号上的对应关系。~~



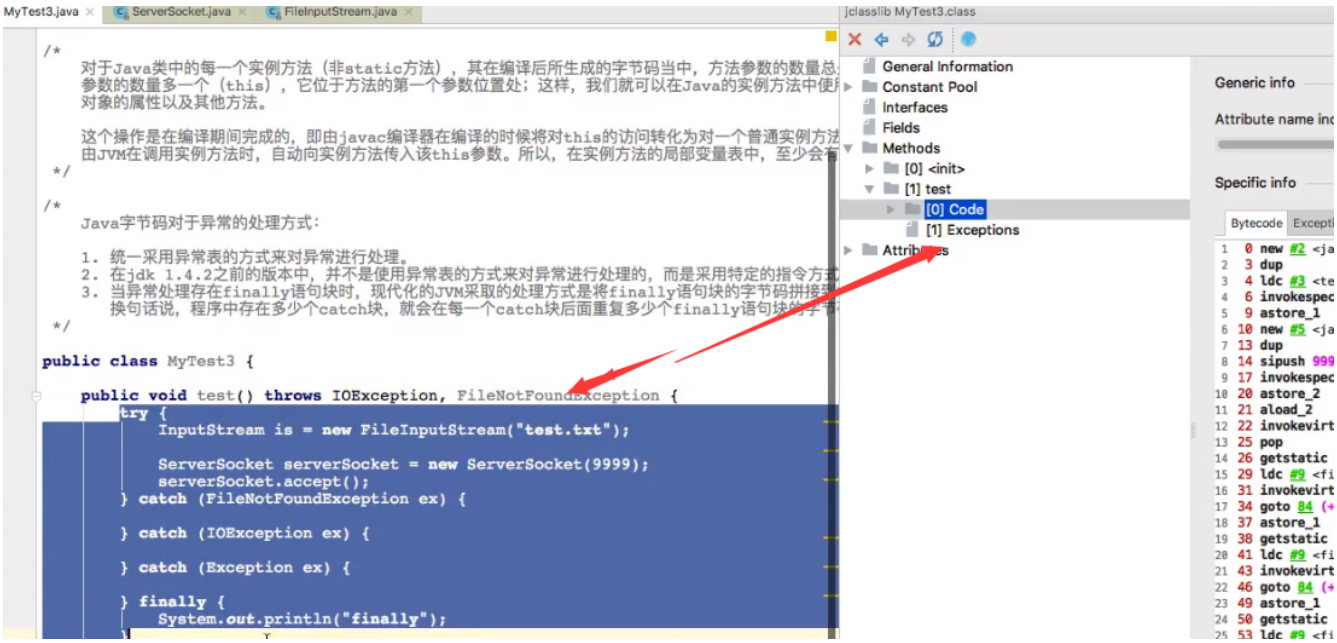
~~字节码的代码的行号。~~

-

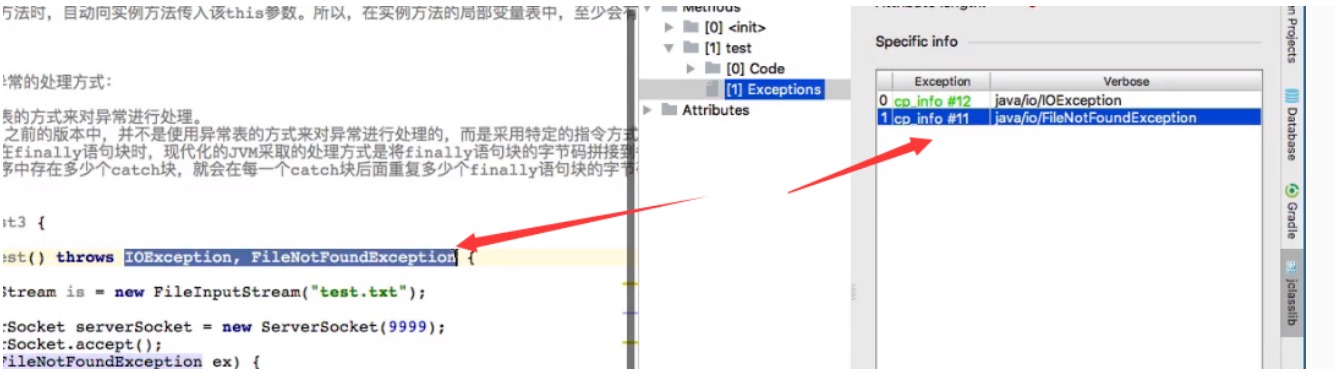
~~LocalVariableTable 局部变量表，用于描述局部变量表~~

-

~~50 栈帧与操作数剖析及符号引用与直接引用的转换~~



-



-

~~栈帧（stack frame）用于帮助虚拟机执行方法执行的数据结构，只会归属于单个特定的线程，不能公用。~~

~~本身是一种数据结构，~~

-

-

```
/*
    栈帧（stack frame）

    栈帧是一种用于帮助虚拟机执行方法调用与方法执行的数据结构。

    栈帧本身是一种数据结构，封装了方法的局部变量表、动态链接信息、方法的返回地址以及操作数栈等信息。

*/
```

~~A类对B类的调用~~

~~编译期间是不知道的。~~

~~加载的时候或者运行时真正的调用才会知道。~~

-

~~符号引用：A中会维护B的全限定名字。~~

~~直接引用：当A用到了B的方法等，就会将符号引用转换为直接应用，指向其对应的地址。（相当于指针，指向调用的目标）~~

-

~~slot 局部变量表：一个数据，对应一个slot（但是取决于数据的大小，如果slot是32位，但是数据是64位，则由两个slot维护这个数据）~~

~~（可复用的）~~

```
 */
public class MyTest4 {

    public void test() {

        int a = 3;

        if (a < 4) {
            int b = 4;
            int c = 5;
        }

        int d = 7;
        int e = 8;



    }
}
```

-

~~b c 生命周期结束后其对应的slot可能会被 d e占用，这就是复用。~~

~~符号应用如何转换为一个直接引用：~~

```
    符号引用，直接引用

    有些符号引用是在类加载阶段或是第一次使用时就会转换为直接引用，这种转换叫做静态解析；另外一些符号引用则是在每次运行期转换为直接引用，这种转换叫做动态链接，这体现为Java的多态性。

*/
```

-

-

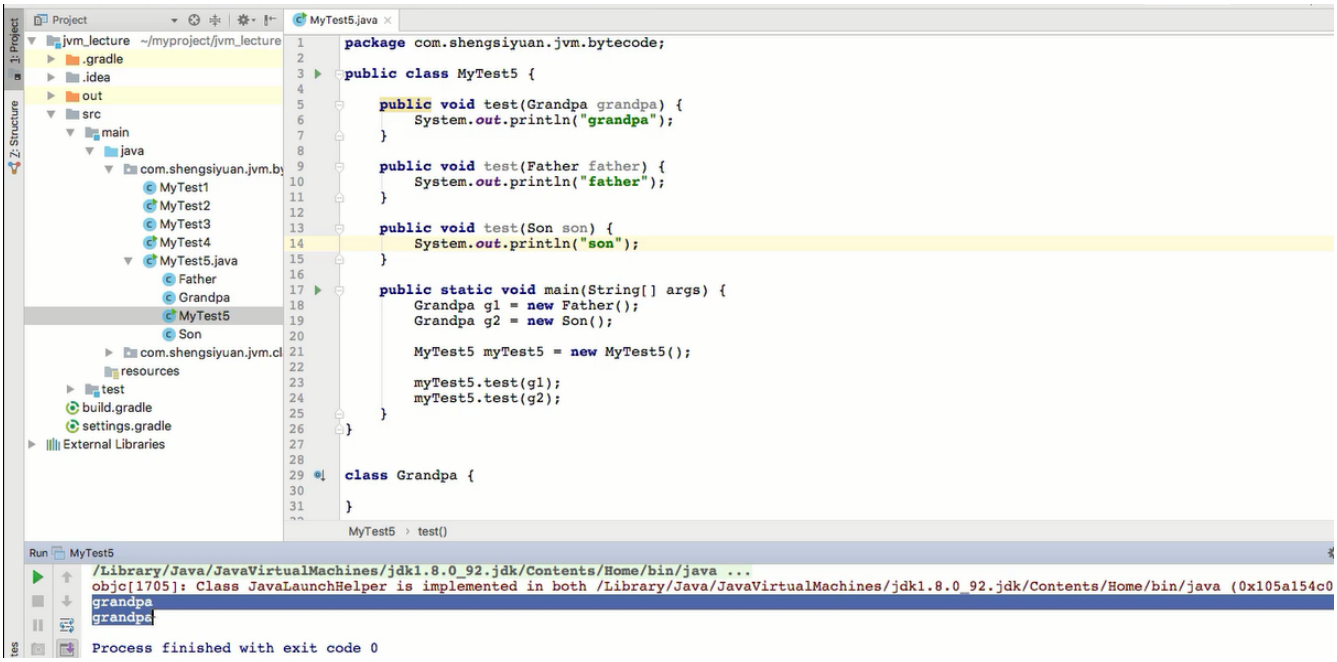
~~52 方法重载与invokevirtual 字节码指令的关系~~

```
/*
    1. invokeinterface：调用接口中的方法，实际上是在运行期决定的，决定到底调用实现该接口的哪个对象的特定方法。
    2. invokestatic：调用静态方法。
    3. invokespecial：调用自己的私有方法、构造方法（<init>）以及父类的方法。
    4. invokevirtual：调用虚方法，运行期动态查找的过程。
    5. invokedynamic：动态调用方法。
 */
```

~~2 3 在类加载阶段就能确定的。~~

```
/*
    静态解析的4种情形：

    1. 静态方法
    2. 父类方法
    3. 构造方法
    4. 私有方法

    以上4类方法称作非虚方法，他们是在类加载阶段就可以将符号引用转换为直接引用的。
 */
```

-




```
/*
    方法的静态分派。

    Grandpa g1 = new Father();

    以上代码，g1的静态类型是Grandpa，而g1的实际类型（真正指向的类型）是Father。

    我们可以得出这样一个结论：变量的静态类型是不会发生变化的，而变量的实际类型则是可以发生变化的（多态的一种体现），实际类型是在
    运行期方可

 */
```

-

```java
public class MyTest5 {

    // 方法重载，是一种静态的行为

    public void test(Grandpa grandpa) {
        System.out.println("grandpa");
    }

    public void test(Father father) {
        System.out.println("father");
    }

    public void test(Son son) {
        System.out.println("son");
    }

    public static void main(String[] args) {
        Grandpa g1 = new Father();
        Grandpa g2 = new Son();

        MyTest5 myTest5 = new MyTest5();

        myTest5.test(g1);
        myTest5.test(g2);
    }
}


class Grandpa {

}

class Father extends Grandpa {
```

~~根据g1的静态类型进行匹配~~

~~重载是静态的概念，而重写是动态的概念。~~

-

-

~~53 通过字节码分析java方法的静态分派与动态分派机制~~

MyTest6.java

```java
package com.shengsiyuan.jvm.bytecode;

public class MyTest6 {

    public static void main(String[] args) {
        Fruit apple = new Apple();
        Fruit orange = new Orange();

        apple.test();
        orange.test();

        apple = new Orange();
        apple.test();
    }
}

class Fruit {

    public void test() {
        System.out.println("Fruit");
    }
}

class Apple extends Fruit {

    @Override
    public void test() {
        System.out.println("Apple");
    }
}

class Orange extends Fruit {

    @Override
    public void test() {
        System.out.println("Orange");
    }
}
```

-

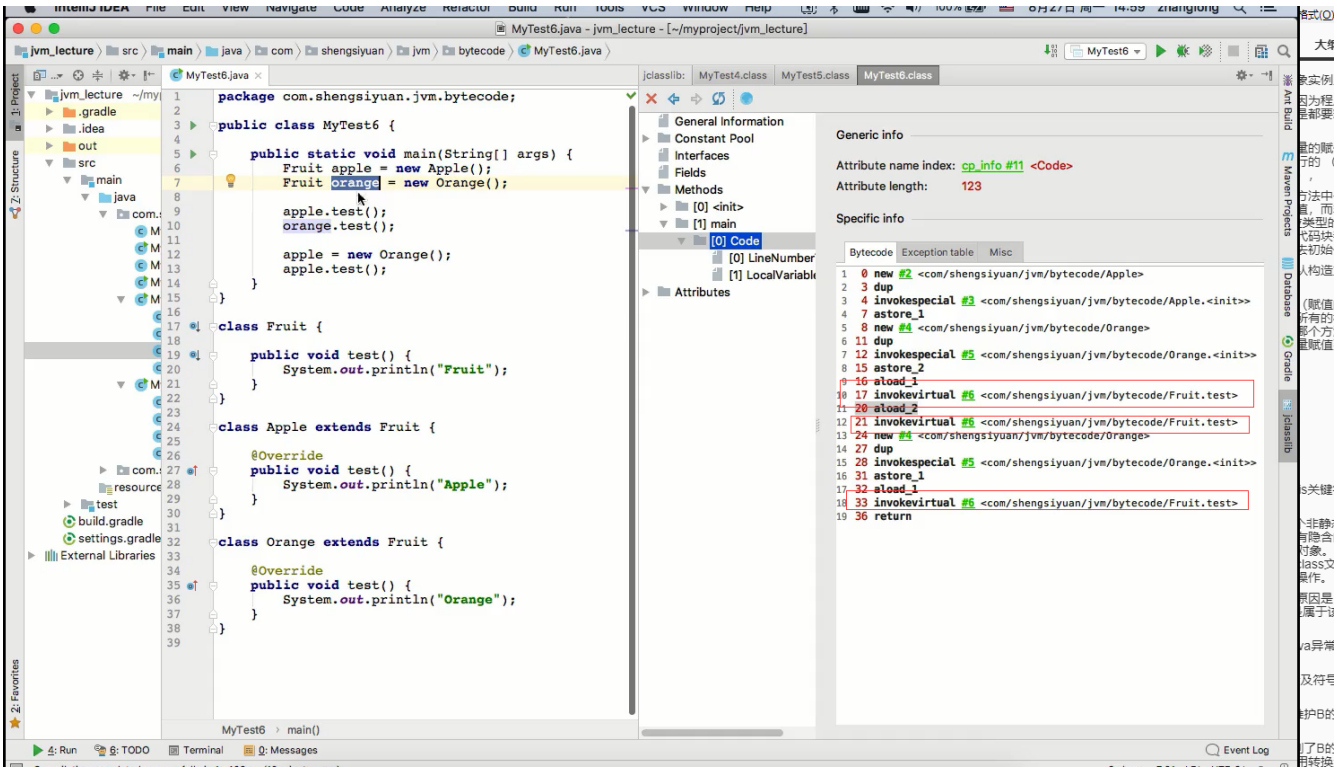
~~new 的作用：~~

~~1 开辟内存空间~~

~~2 执行构造方法，~~

~~3 返回该对象的引用~~

-



-

~~invokeVurtual 执行流程~~

~~找到操作数栈顶第一个元素 对象指向的实际类型： apple~~

~~在apple类中找到与test方法完全一致的方法，那么就去执行（确定一个方法的方式 方法名+方法的描述符即方法的参数和返回值）~~

~~否则，就去找父类的方法，~~

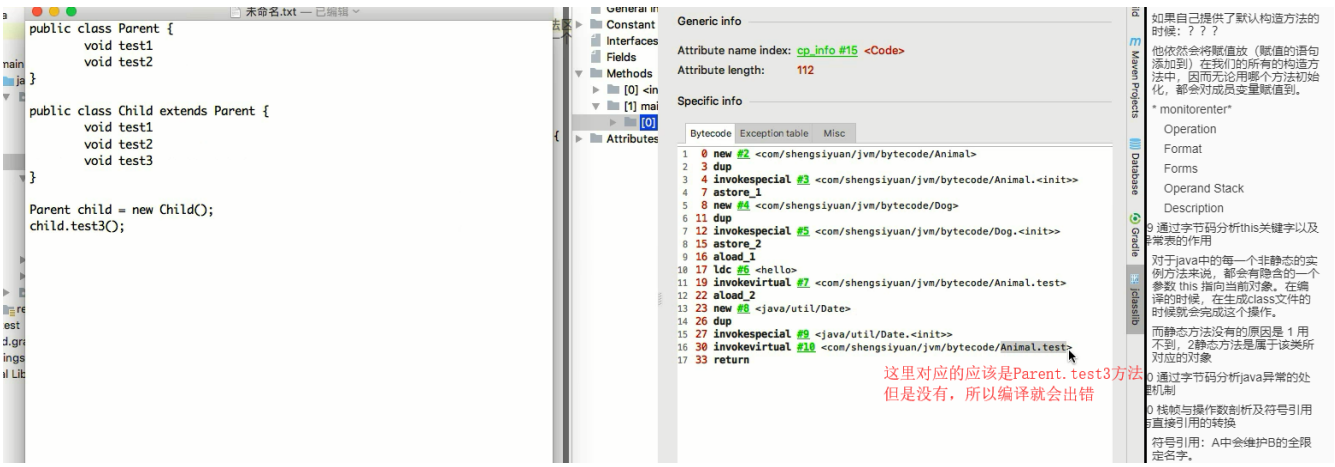
~~54 虚方法表与动态分派机制详解~~

```
/*
    针对于方法调用动态分派的过程，虚拟机会在类的方法区建立一个虚方法表的数据结构（virtual method table，vtable），
    针对于invokeinterface指令来说，虚拟机会建立一个叫做接口方法表的数据结构（interface method table，itable），
*/
```

-

```
public class Parent {
        void test1
        void test2
}

public class Child extends Parent {
        void test1
        void test2
        void test3
}

Parent child = new Child();
child.test3();
```

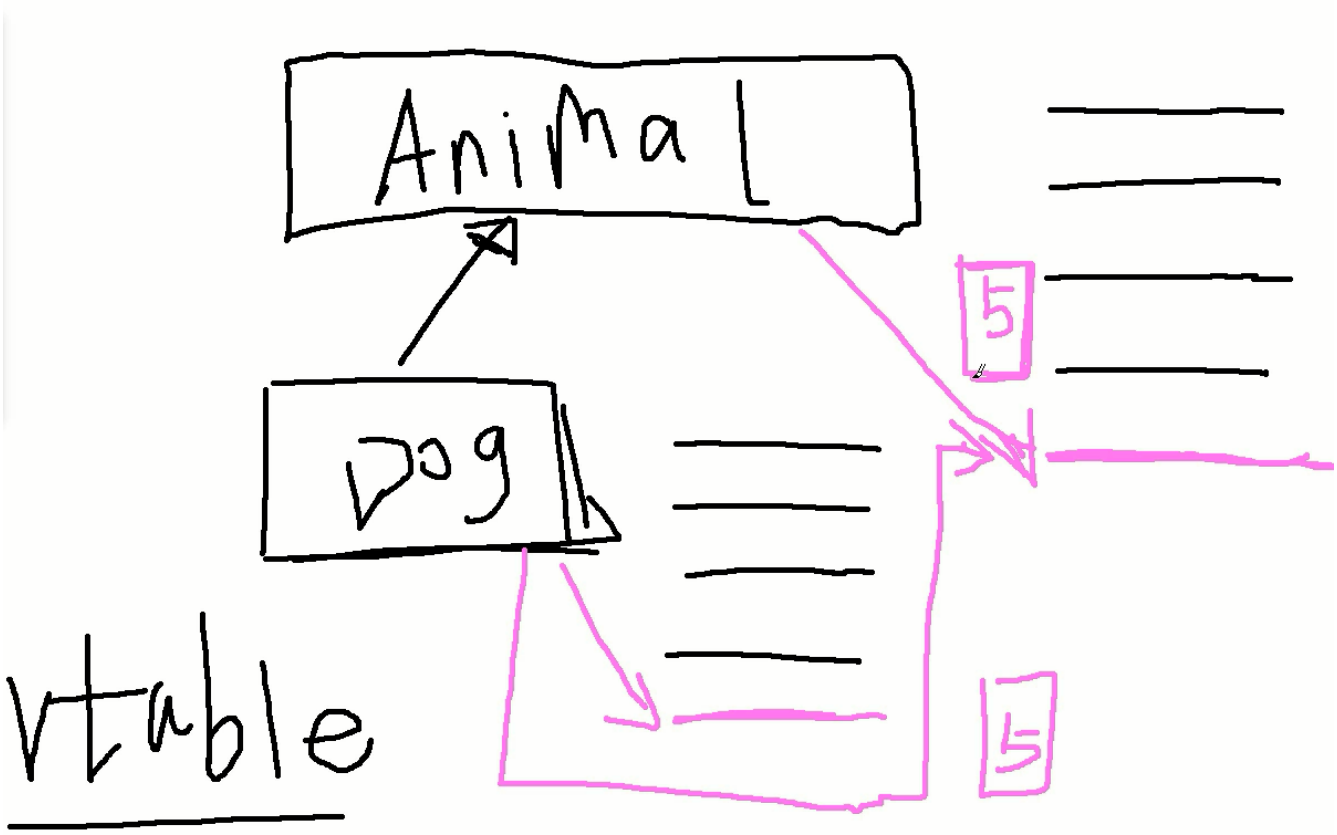
-



-

-

-



-

-

~~虚方法表存的是每一个方法的入口的调用地址~~

~~55 基于栈的指令集于基于寄存器的指令集的详细对比~~

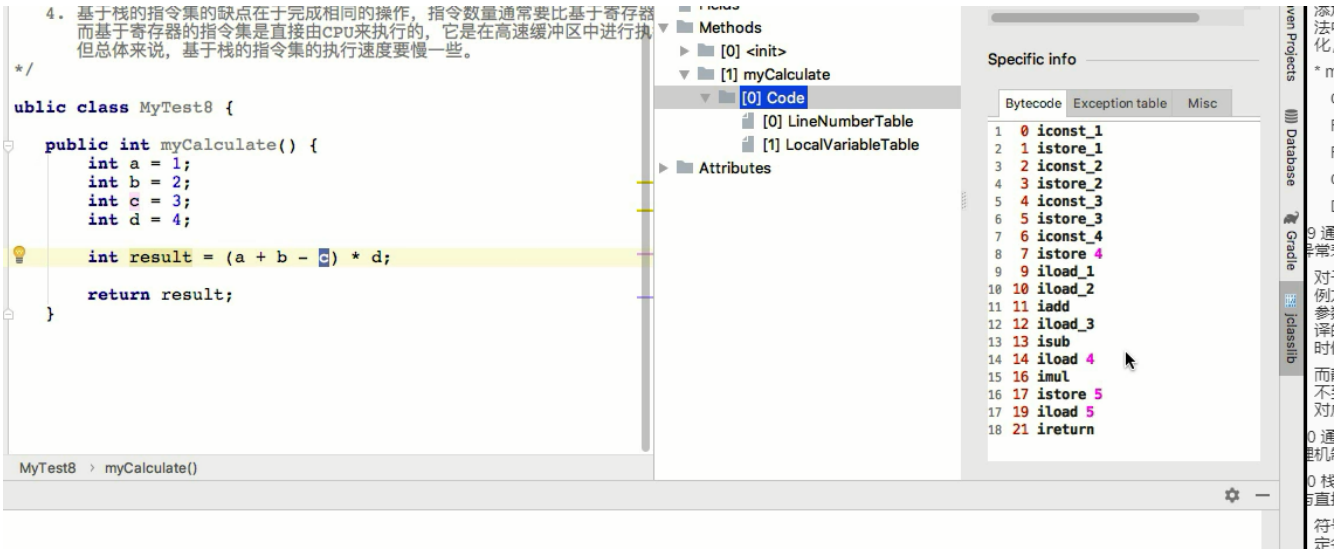
```
/*
    现代JVM在执行Java代码的时候，通常都会将解释执行与编译执行二者结合起来进行。

    所谓解释执行，就是通过解释器来读取字节码，遇到相应的指令就去执行该指令。
    所谓编译执行，就是通过即时编译器（Just In Time，JIT）将字节码转换为本地机器码来执行；现代JVM会根据代码热点来生成相应的本地机器码。

 */

  基于栈的指令集与基于寄存器的指令集之间的关系：

  1. JVM执行指令时所采取的方式是基于栈的指令集。
  2. 基于栈的指令集主要的操作有入栈与出栈两种。
  3. 基于栈的指令集的优势在于它可以在不同平台之间移植，而基于寄存器的指令集是与硬件架构紧密关联的，无法做到可移植。
  4.
/
```

-

```
  基于栈的指令集与基于寄存器的指令集之间的关系：

  1. JVM执行指令时所采取的方式是基于栈的指令集。
  2. 基于栈的指令集主要的操作有入栈与出栈两种。
  3. 基于栈的指令集的优势在于它可以在不同平台之间移植，而基于寄存器的指令集是与硬件架构紧密关联的，无法做到可移植。
  4. 基于栈的指令集的缺点在于完成相同的操作，指令数量通常要比基于寄存器的指令集数量要多；基于栈的指令集是在内存中完成操作的，
     而基于寄存器的指令集是直接由CPU来执行的，它是在高速缓冲区中进行执行的，速度要快很多。虽然虚拟机可以采用一些优化手段，
     但总体来说，基于栈的指令集的执行速度要慢一些。
/
```

-

~~56 jvm 执行栈指令集实例剖析~~



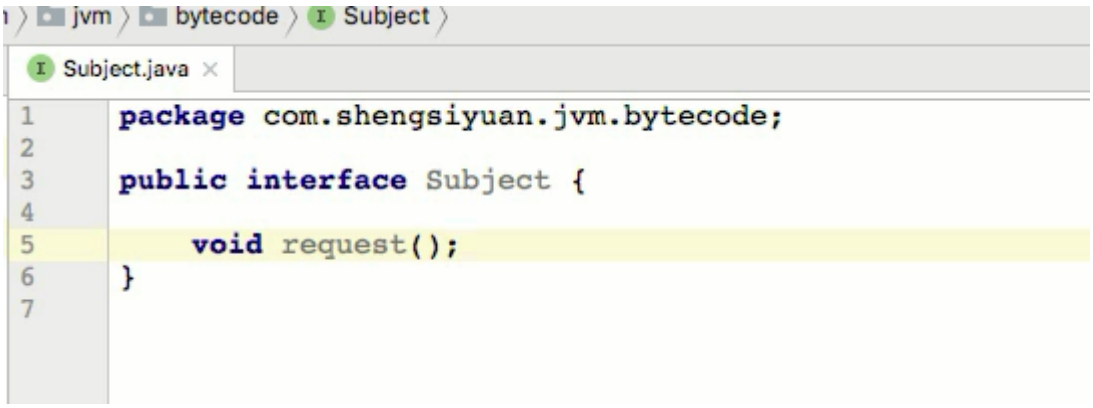
~~57 透过字节码生成审视java动态代理运作机制~~

-

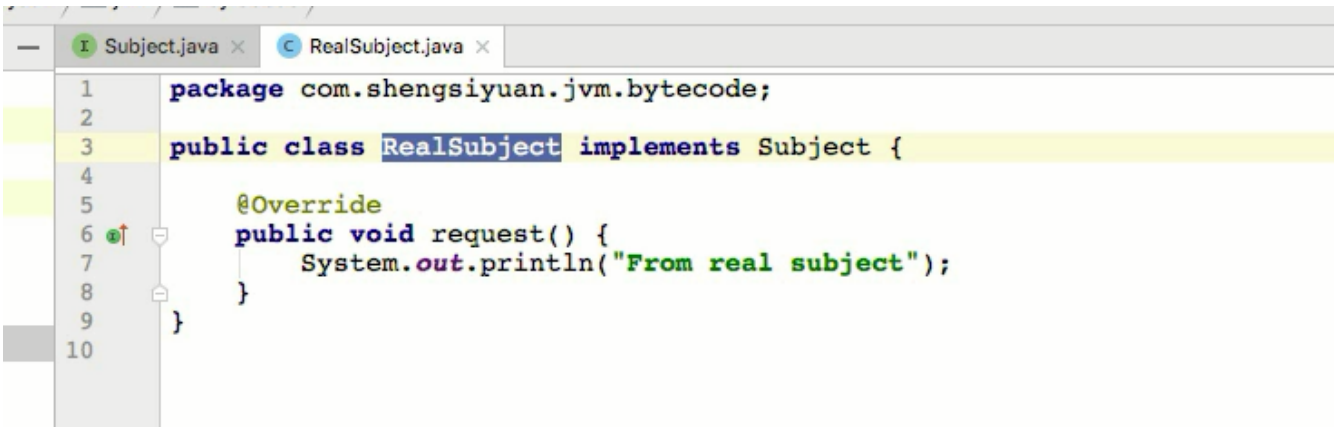
-

-

-



-



-

-

Subject.java | RealSubject.java | DynamicSubject.java

```java
package com.shengsiyuan.jvm.bytecode;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

public class DynamicSubject implements InvocationHandler {

    private Object sub;

    public DynamicSubject(Object obj) {
        this.sub = obj;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("before calling: " + method);

        method.invoke(this.sub, args);

        System.out.println("after calling: " + method);

        return null;
    }
}
```

动态代理

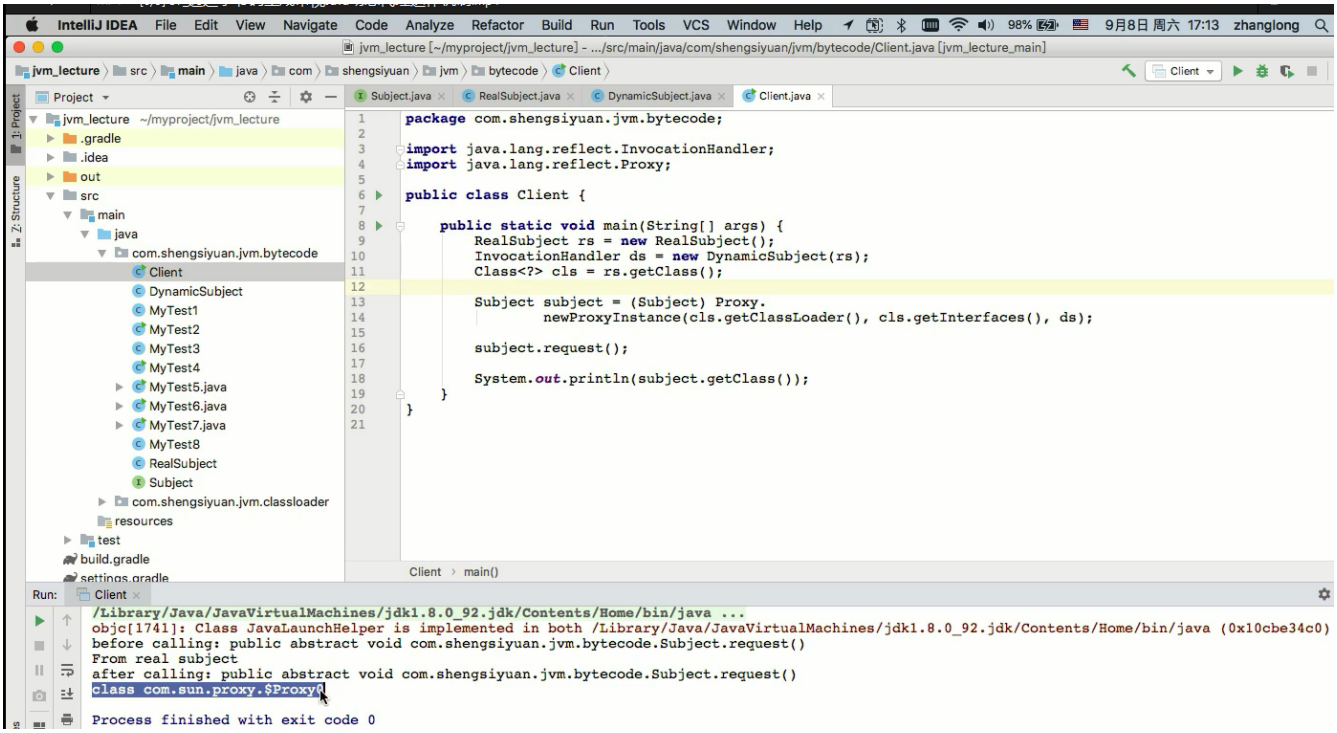
代理的对象

-

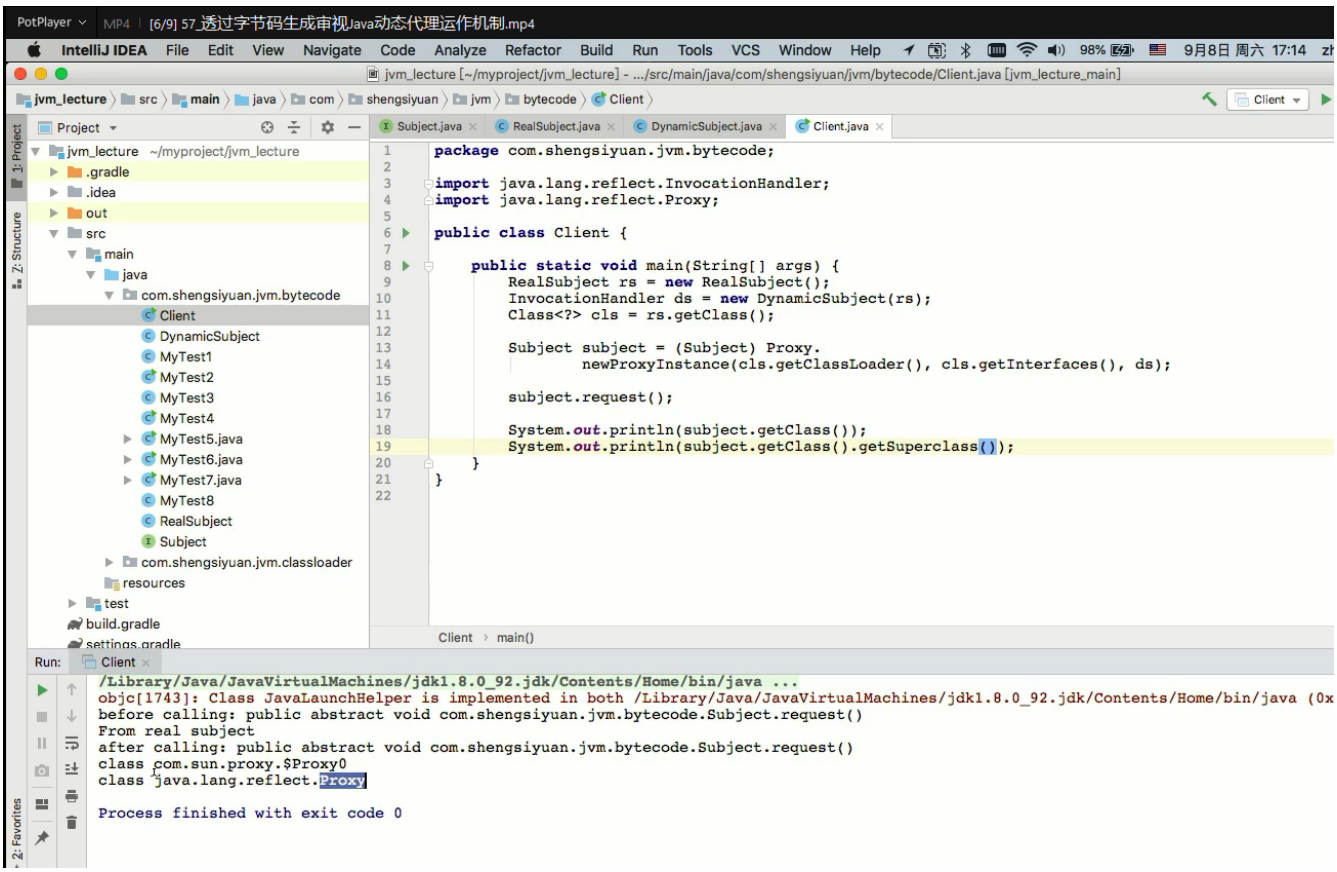
Subject.java | RealSubject.java | DynamicSubject.java | Client.java

```java
package com.shengsiyuan.jvm.bytecode;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Proxy;

public class Client {

    public static void main(String[] args) {
        RealSubject rs = new RealSubject();
        InvocationHandler ds = new DynamicSubject(rs);
        Class<?> cls = rs.getClass();

        Subject subject = (Subject) Proxy.
                newProxyInstance(cls.getClassLoader(), cls.getInterfaces(), ds);

        subject.request();
    }
}
```

-



这是动态生成的



~~父类~~

```
public static void main(String[] args) {
    System.getProperties().put("sun.misc.ProxyGenerator.saveGeneratedFiles", "true");
```

~~设置这个属性，生成class文件~~

-

59 JVM字节整体回顾与总结：

java class文件中可以没有任何一个方法。这是与java语言规范不同的。jvm规范。

-

60 JVM内存空间划分与作用详解：

-

每个方法执行的时候都会创建一个栈帧：

其包含的有 操作数栈，局部变量表，方法的返回地址，还有一个记不得

-

-

虚拟机栈：stack frame 栈帧

程序计数器：（program counter）

本地方法栈：主要用于执行本地方法。

堆（Heap）：存放实例，（线程共享）

Method Area：存储元信息，永久代，（permanent generation）从JDK1.8以后就没有了。换作了元空间。（meta space）

运行时常量池：方法区的一部分的内容

直接内存：

-

-

-

-

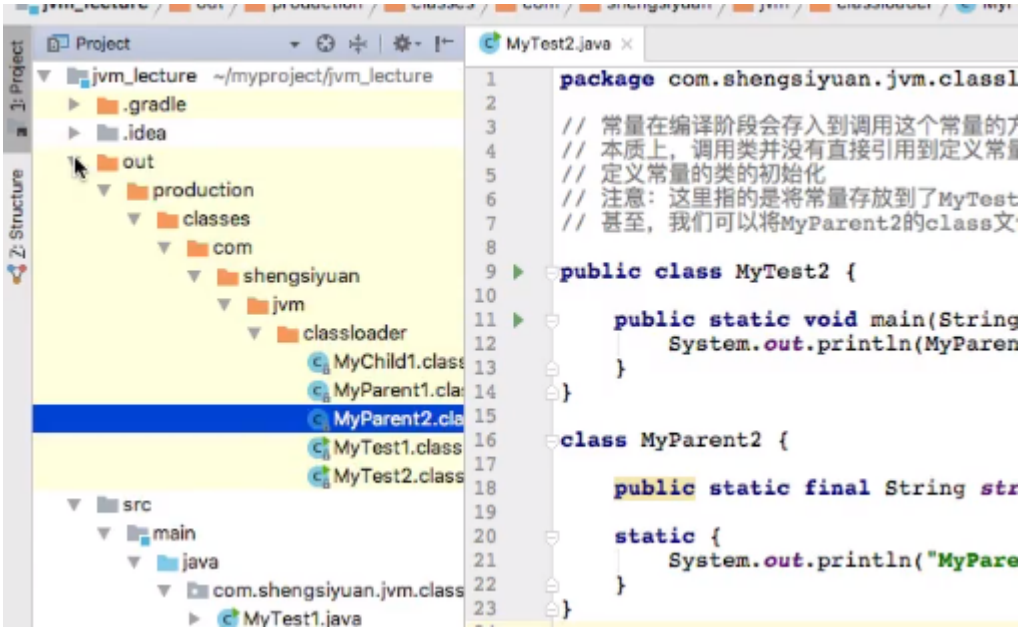
-

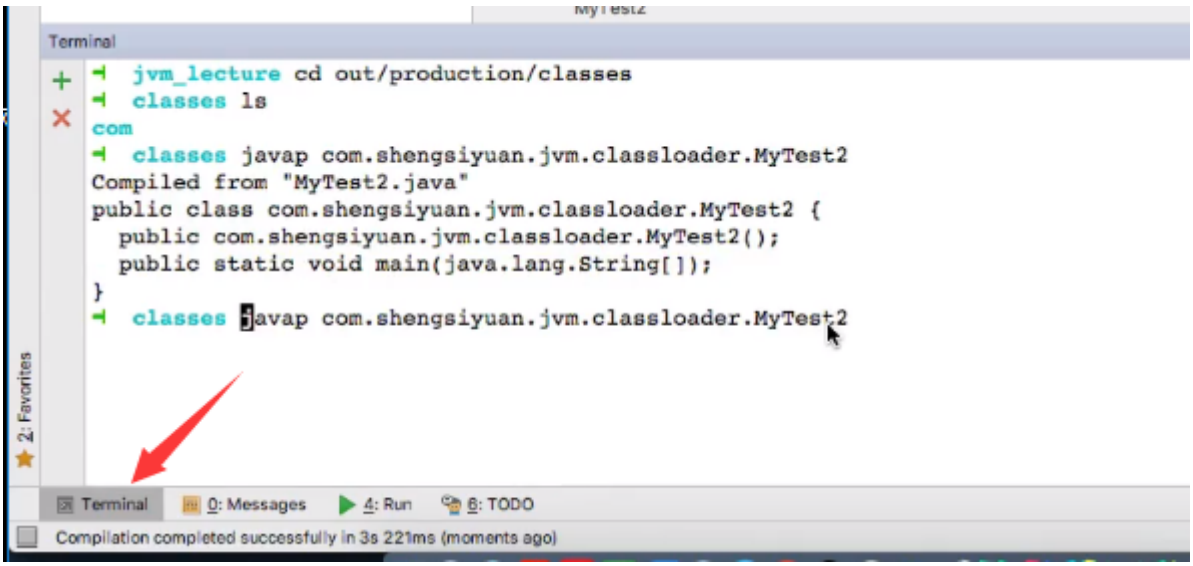
-

//

当一个类在初始化的时候，要求其父类全都初始化完毕。

使用反编译对class文件


Project
jvm_lecture ~/myproject/jvm_lecture
.gradle
.idea
out
production
classes
com
shengsiyuan
jvm
classloader
MyChild1.class
MyParent1.cla
MyParent2.cla
MyTest1.class
MyTest2.class
src
main
java
com.shengsiyuan.jvm.class
MyTest1.java
MyTest2.java
1 package com.shengsiyuan.jvm.classl
2
3 // 常量在编译阶段会存入到调用这个常量的
4 // 本质上，调用类并没有直接引用到定义常
5 // 定义常量的类的初始化
6 // 注意：这里指的是将常量存放到了MyTest
7 // 甚至，我们可以将MyParent2的class文
8
9 public class MyTest2 {
10
11 public static void main(String
12 System.out.println(MyParen
13 }
14 }
15
16 class MyParent2 {
17
18 public static final String str
19
20 static {
21 System.out.println("MyPare
22 }
23 }


Terminal
jvm_lecture cd out/production/classes
classes ls
com
classes javap com.shengsiyuan.jvm.classloader.MyTest2
Compiled from "MyTest2.java"
public class com.shengsiyuan.jvm.classloader.MyTest2 {
public com.shengsiyuan.jvm.classloader.MyTest2();
public static void main(java.lang.String[]);
}
classes javap com.shengsiyuan.jvm.classloader.MyTest2
2: Favorites
Terminal
0: Messages
4: Run
6: TODO
Compilation completed successfully in 3s 221ms (moments ago)


Terminal
public class com.shengsiyuan.jvm.classloader.MyTest2 {
public com.shengsiyuan.jvm.classloader.MyTest2();
public static void main(java.lang.String[]);
}
classes javap -c com.shengsiyuan.jvm.classloader.MyTest2
Compiled from "MyTest2.java"
public class com.shengsiyuan.jvm.classloader.MyTest2 {
public com.shengsiyuan.jvm.classloader.MyTest2();
Code:
0: aload_0
1: invokespecial #1 // Method java/lang/Object."<init>":()V
4: return
public static void main(java.lang.String[]);
Code:
0: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
3: ldc #4 // String hello world
5: invokevirtual #5 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
8: return
}
classes
助记符
2: Favorites
Terminal
0: Messages
4: Run
6: TODO

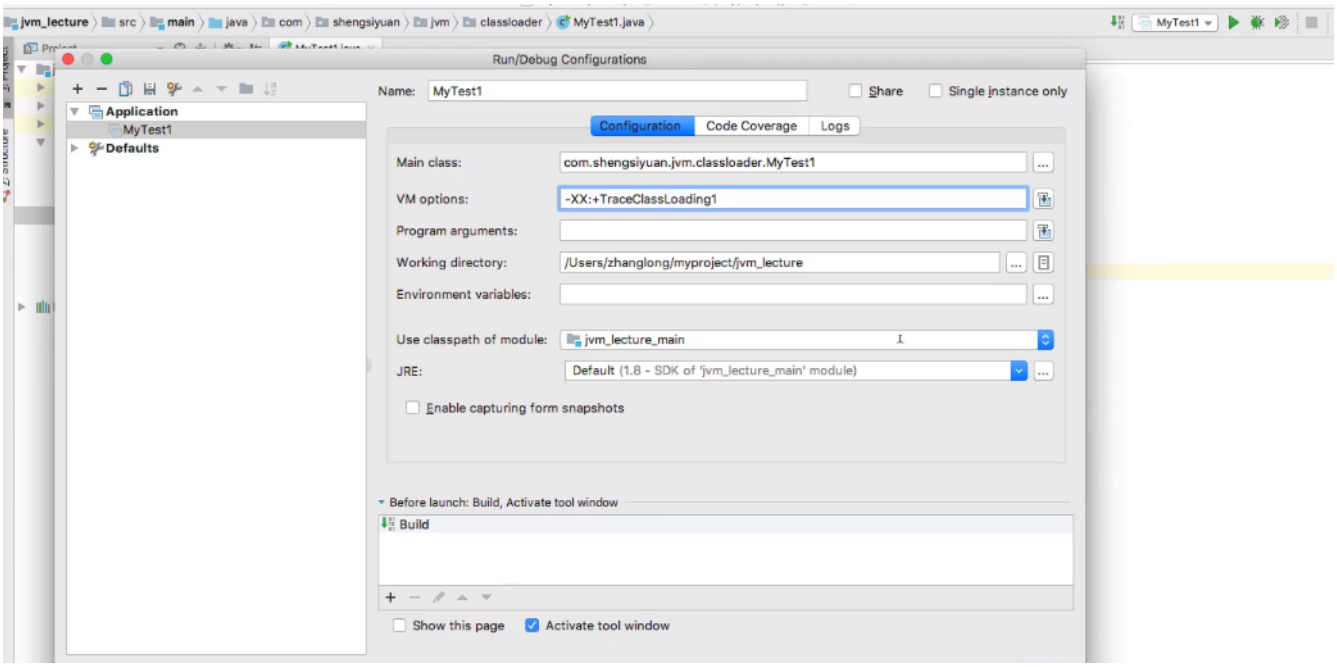
-

-

~~使用intellij构建Gradle项目~~

~~https://blog.csdn.net/csdnlijingran/article/details/80610429~~

~~intellij设置jvm参数~~



-

~~JVM工具~~

~~1.jconsole~~

~~2.jvisualvm~~

~~3.jmap~~

~~JAVA助记符~~

```
/*
    助记符:
    ldc表示将int, float或是String类型的常量值从常量池中推送至栈顶
    bipush表示将单字节（-128 ~ 127）的常量值推送至栈顶
    sipush表示将一个短整型常量值（-32768 ~ 32767）推送至栈顶
    iconst_1表示将int类型1推送至栈顶  (iconst_1 ~ iconst_5)

 */
```

```
助记符:
anewarray：表示创建一个引用类型的（如类、接口、数组）数组，并将其引用值压入栈顶
newarray：表示创建一个指定的原始类型（如int、float、char等）的数组，并将其引用值压入栈顶
*/
```

-

~~助记符的包~~

```
/.../
package com.sun.org.apache.bcel.internal.generic;
```

-

-

-

~~JVM参数~~

~~JVM参考：~~

数据类型的大小

java的基本数据类型占据的内存大小

这是8中基本类型的内存中占用字节数（取值范围是2的（字节数X8-1）次方）

1.整型

类型	存储需求	bit数	取值范围
byte	1字节	1*8	（-2的31次方到2的31次方-1）
short	2字节	2*8	－32768～32767
int	4字节	4*8	（-2的63次方到2的63次方-1）
long	8字节	8*8	－128～127

2.浮点型

类型	存储需求	bit数	备注
float	4字节	4*8	float类型的数值有一个后缀F(例如：3.14F)
double	8字节	8*8	没有后缀F的浮点数值(如3.14)默认为double类型

3.char类型

类型	存储需求	bit数
char	2字节	2*8

4.boolean类型

类型	存储需求	bit数	取值范围
boolean	1字节	1*8	false、true

~~5.方法区：~~

~~方法区在JVM也是一个非常重要的一块内存区域，和堆一样，可以被多个线程多共享。~~

~~主要存放每一个加载class的信息。class信息主要包含魔数（确定是否是一个class文件），常量~~

池，访问标志（当前的类是普通类还是接口，是否是抽象类，是否被public修饰，是否使用了final修饰等描述信息.....），字段表集合信息（使用什么访问修饰符，是实例变量还是静态变量，是否使用了final修饰等描述信息.....），方法表集合信息（使用什么访问修饰符，是否静态方法，是否使用了final修饰，是否使用了synchronized修饰，是否是native方法.....）等内容。当一个类加载器加载了一个类的时候，会根据这个class文件创建一个class对象，class对象就包含了上述的信息。后续要创建这个类的实例，都根据这个class对象创建出来的。

System.gc()

其实这个gc()函数的作用只是提醒虚拟机：程序员希望进行一次垃圾回收。但是它不能保证垃圾回收一定会进行，而且具体什么时候进行是取决于具体的虚拟机的，不同的虚拟机有不同的对策。

finalize的作用

- finalize()是Object的protected方法，子类可以覆盖该方法以实现资源清理工作，GC在回收对象之前调用该方法。
- finalize()与C++中的析构函数不是对应的。C++中的析构函数调用的时机是确定的（对象离开作用域或delete掉），但Java中的finalize的调用具有不确定性
- 不建议用finalize方法完成“非内存资源”的清理工作，但建议用于：① 清理本地对象(通过JNI创建的对象)；② 作为确保某些非内存资源(如Socket、文件等)释放的一个补充：在finalize方法中显式调用其他资源释放方法。其原因可见下文[finalize的问题]

finalize的执行过程(生命周期)

(1) 首先，大致描述一下finalize流程：当对象变成(GC Roots)不可达时，GC会判断该对象是否覆盖了finalize方法，若未覆盖，则直接将其回收。否则，若对象未执行过finalize方法，将其放入F-Queue队列，由一低优先级线程执行该队列中对象的finalize方法。执行finalize方法完毕后，GC会再次判断该对象是否可达，若不可达，则进行回收，否则，对象“复活”。