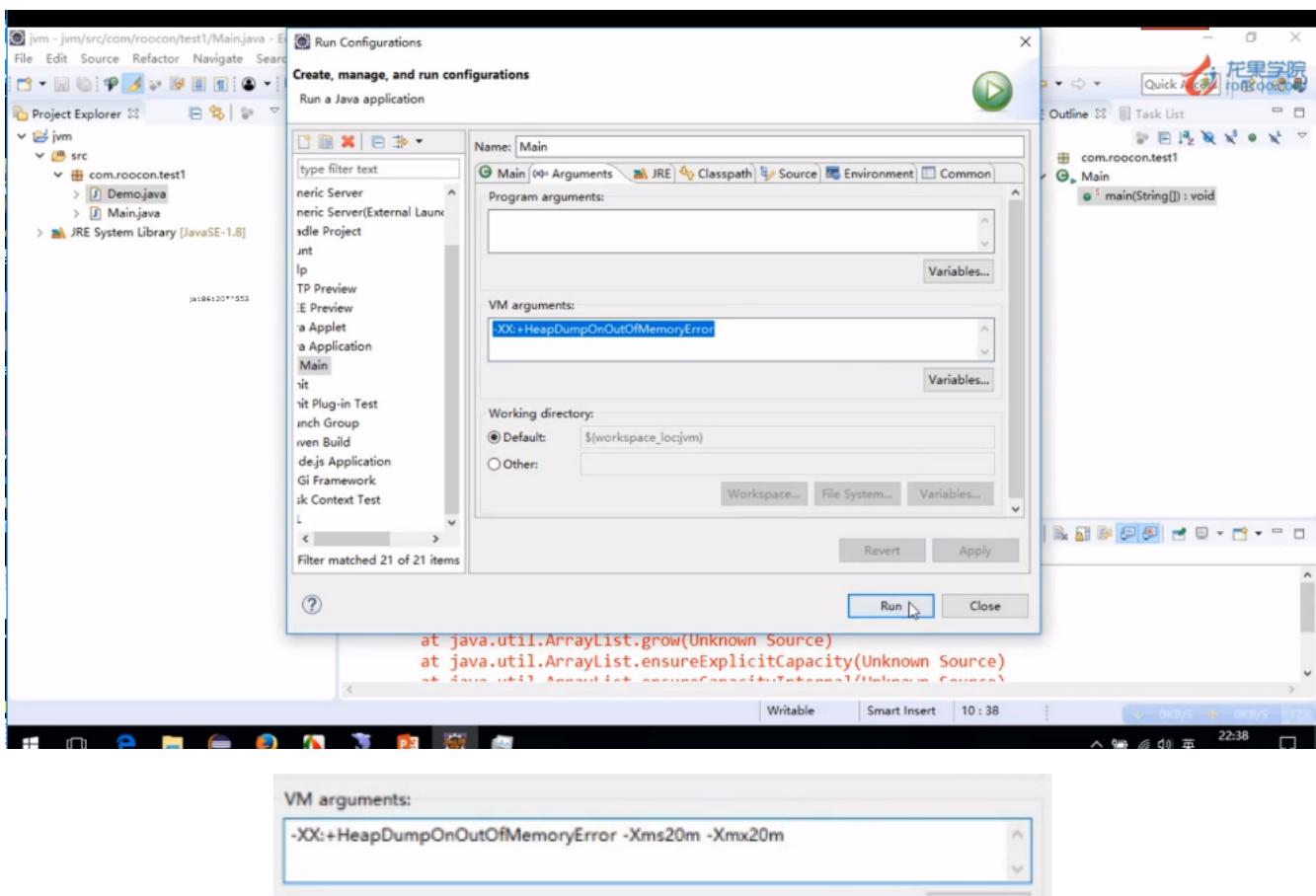


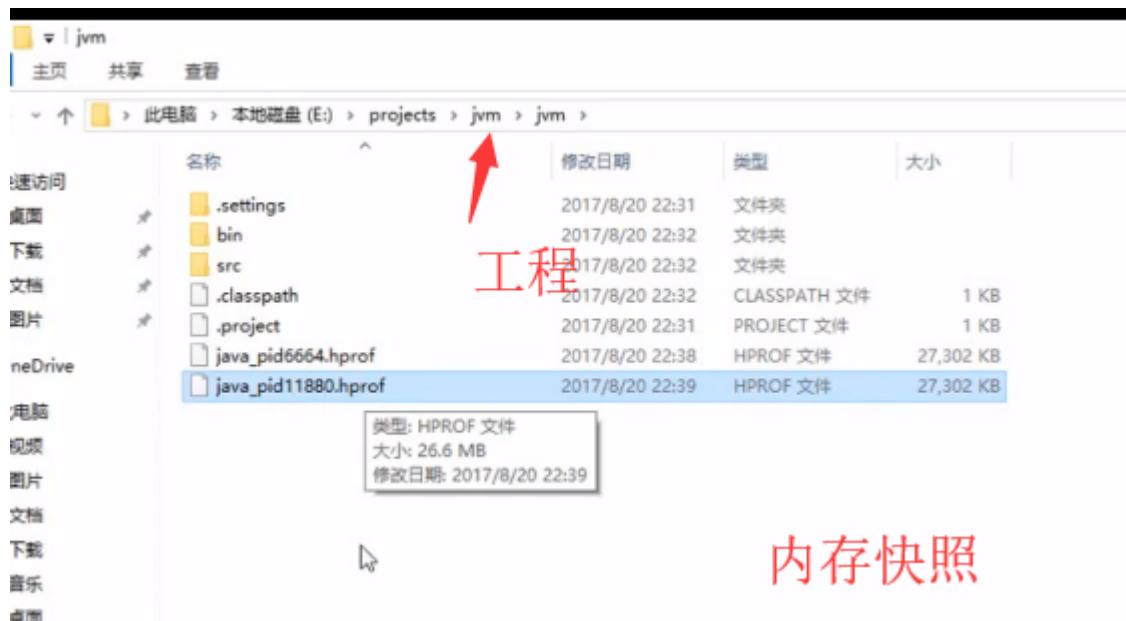
堆内存溢出



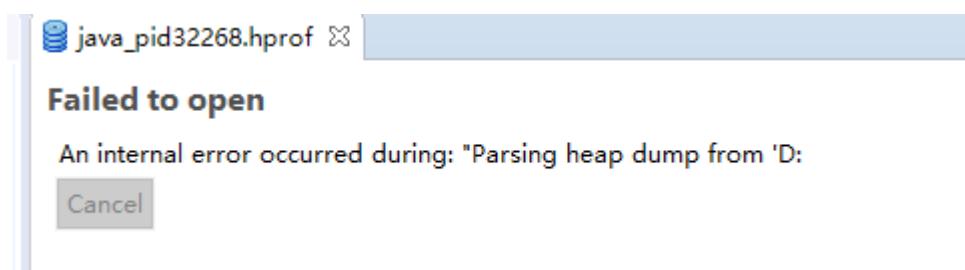
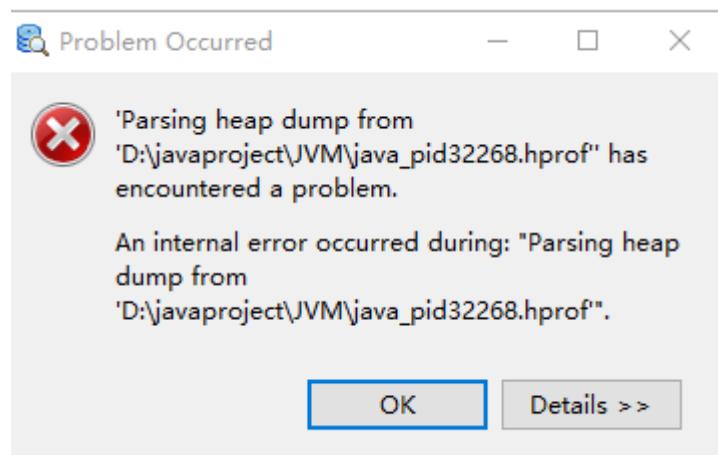
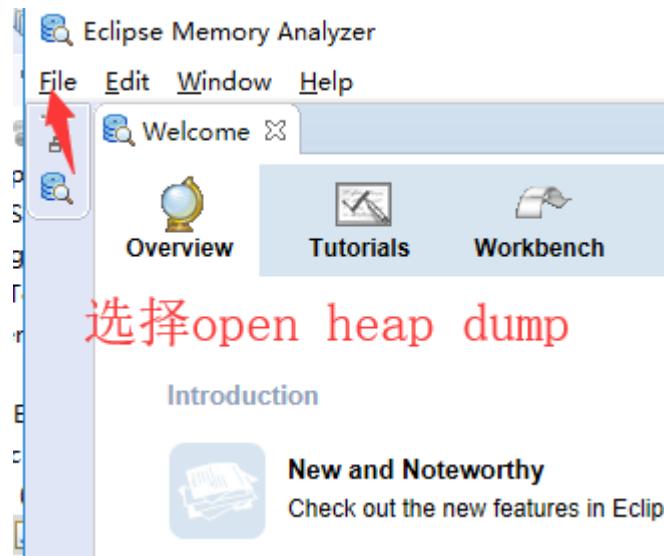
-Xms20m 设置虚拟机内存

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

-XX:+HeapDumpOnOutOfMemoryError -Xms20m -Xmx20m



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

类加载

- 在**Java**代码中，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
- 提供了更大的灵活性，增加了更多的可能性

类加载器深入剖析

- **Java**虚拟机与程序的生命周期
- 在如下几种情况下，**Java**虚拟机将结束生命周期
 - 执行了**System.exit()**方法
 - 程序正常执行结束
 - 程序在执行过程中遇到了异常或错误而异常终止
 - 由于操作系统出现错误而导致**Java**虚拟机进程终止

类的加载、连接与初始化

- 加载：查找并加载类的二进制数据
- 连接
 - -验证：确保被加载的类的正确性
 - -准备：为类的静态变量分配内存，并将其初始化为默认值
- -解析：把类中的符号引用转换为直接引用
- 初始化：为类的静态变量赋予正确的初始值

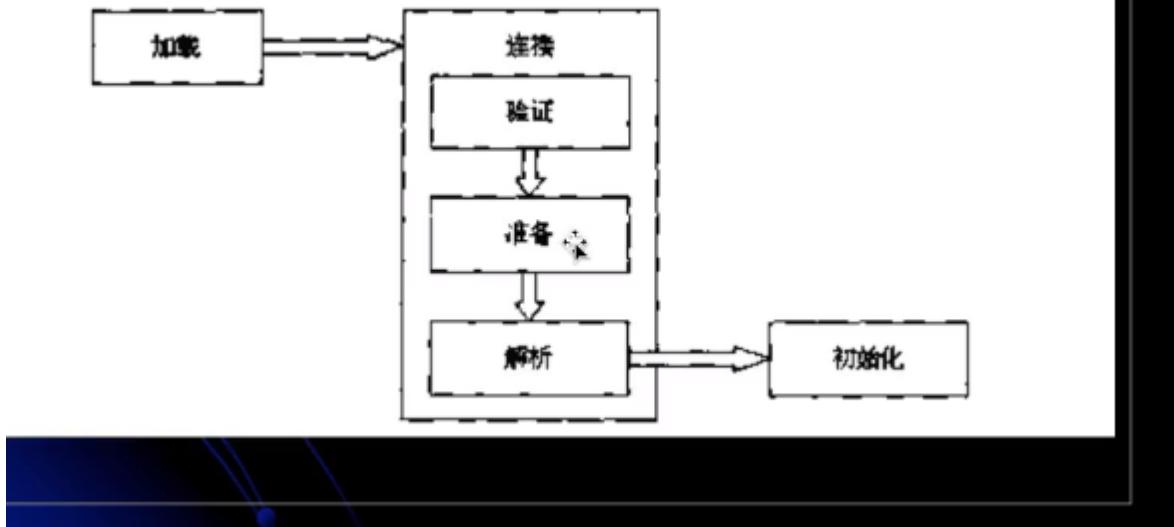
那么成员变量呢？？？？

使用

，卸载

类的使用与卸载

- 使用
- 卸载



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。

```
getstatic  
putstatic  
invokestatic
```

类的加载、连接与初始化

- 除了以上七种情况，其他使用**Java**类的方式都被看作是对类的**被动使用**，都不会导致类的**初始化**

类的加载

- 类的加载指的是将类的**.class**文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在内存中创建一个**java.lang.Class**对象（规范并未说明**Class**对象位于哪里，**HotSpot**虚拟机将其放在了方法区中）用来封装类在方法区内的数据结构

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

- 加载.class文件的方式
 - -从本地系统中直接加载
 - -通过网络下载.class文件
 - -从zip, jar等归档文件中加载.class文件
 - -从专有数据库中提取.class文件
 - -将Java源文件动态编译为.class文件

动态代理, jsp——》servlet

实战演示

例子1

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Structure:** The project is named "jvm_lecture". It contains a ".gradle" folder, an ".idea" folder, an "out" folder, and a "src" folder. The "src/main/java/com.shengsiyuan.jvm.classloader" package contains four files: MyTest1.java, MyChild1.java, MyParent1.java, and MyTest1.java.
- Code Editor:** The code editor displays the following Java code:

```
1 package com.shengsiyuan.jvm.classloader;
2
3 public class MyTest1 {
4
5     public static void main(String[] args) {
6         System.out.println(MyChild1.str);
7     }
8 }
9
10 class MyParent1 {
11
12     public static String str = "hello world";
13
14     static {
15         System.out.println("MyParent1 static block");
16     }
17 }
18
19 class MyChild1 extends MyParent1 {
20
21     static {
22         System.out.println("MyChild1 static block");
23     }
24 }
25
```

- Run Tab:** The "Run" tab shows the output of the "MyTest1" run configuration. The output window displays:

```
MyTest1 > main()
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
objc[1245]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.
MyParent1 static block
hello world
Process finished with exit code 0
```

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。

例子2

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project View:** Shows the project structure under "jvm_lecture". The "src/main/java/com.shengsiyuan.jvm.classloader" package contains three files: MyTest1.java, MyParent1.java, and MyChild1.java.
- Code Editor (MyTest1.java):**

```
1 package com.shengsiyuan.jvm.classloader;
2
3 public class MyTest1 {
4
5     public static void main(String[] args) {
6         System.out.println(MyChild1.str2);
7     }
8 }
9
10 class MyParent1 {
11
12     public static String str = "hello world";
13
14     static {
15         System.out.println("MyParent1 static block");
16     }
17 }
18
19 class MyChild1 extends MyParent1 {
20
21     public static String str2 = "welcome";
22
23     static {
24         System.out.println("MyChild1 static block");
25     }
26 }
27 }
```
- Run Tab:** Shows the output of running the code:

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
objc[1251]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
MyParent1 static block
MyChild1 static block
welcome

Process finished with exit code 0
```

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

例子3

```
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。

例子3

The screenshot shows the IntelliJ IDEA interface during the execution of a Java application named 'MyTest2'. The code editor displays 'MyTest2.java' with the following content:

```
package com.shengsiyuan.jvm.classloader;
public class MyTest2 {
    public static void main(String[] args) {
        System.out.println(MyParent2.str);
    }
    class MyParent2 {
        public static String str = "hello world";
        static {
            System.out.println("MyParent2 static block");
        }
    }
}
```

The 'Run' tool window at the bottom shows the command line output:

```
/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
```

Below the output, it says 'Process finished with exit code 0'.

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project View:** Shows the project structure under "jvm_lecture". The "src/main/java/com/shengsiyuan/jvm/classloader" package contains four files: MyTest1.java, MyTest2.java, MyParent1.java, and MyParent2.java.
- Code Editor:** The "MyTest2.java" file is open, displaying the following code:

```
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 final String str = "hello world";
13
14     static {
15         System.out.println("MyParent2 static block");
16     }
17 }
18 }
```
- Run Tab:** The "Run" tab is selected, showing the output of the "MyTest2" run. The output window displays:

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
objc[1320]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92
hello world
```

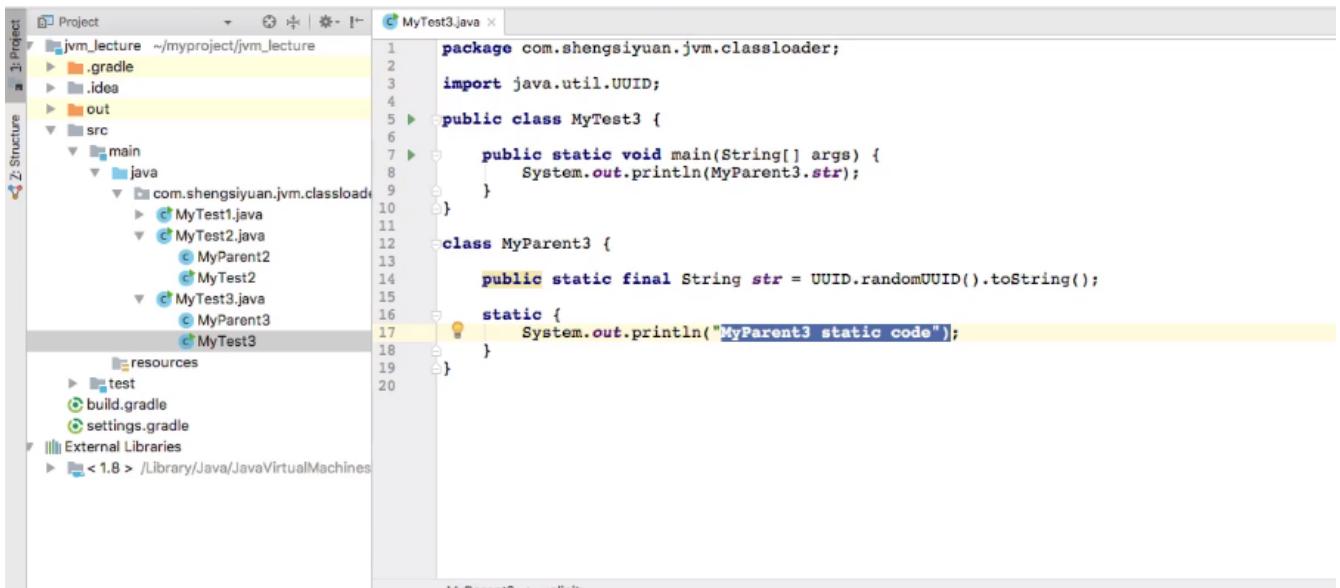
Process finished with exit code 0
- Bottom Status Bar:** Shows "Compilation completed successfully in 1s 249ms (moments ago)".

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
1 package com.shengsiyuan.jvm.classloader;
2
3 // 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,
4 // 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
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MyTest2的常量池中, 之后MyTest2与MyParent2就没有任何关系了
7 // 甚至, 我们可以将MyParent2的class文件删除
8
9 public class MyTest2 {
10
11     public static void main(String[] args) {
12         System.out.println(MyParent2.str);
13     }
14 }
15
16 class MyParent2 {
17
18     public static final String str = "hello world";
19
20     static {
21         System.out.println("MyParent2 static block");
22     }
23 }
24
```

例5



The screenshot shows the IntelliJ IDEA interface with the following details:

- Project View:** Shows a project named "jvm_lecture" with a "src" directory containing "main" and "java". Inside "java", there is a package "com.shengsiyuan.jvm.classloader" which contains four classes: MyTest1, MyTest2, MyParent3, and MyTest3.
- Code Editor:** Displays the file "MyTest3.java" with the following code:

```
package com.shengsiyuan.jvm.classloader;
import java.util.UUID;
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");
        }
    }
}
```
- Run Tab:** Shows the command "java -jar MyTest3.jar" being run, with the output:

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
objc[52382]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/lib
MyParent3 static code
d95b7e47-16cc-448a-8d61-f9a6a9a03076
Process finished with exit code 0
```

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。

```

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);
    }
}

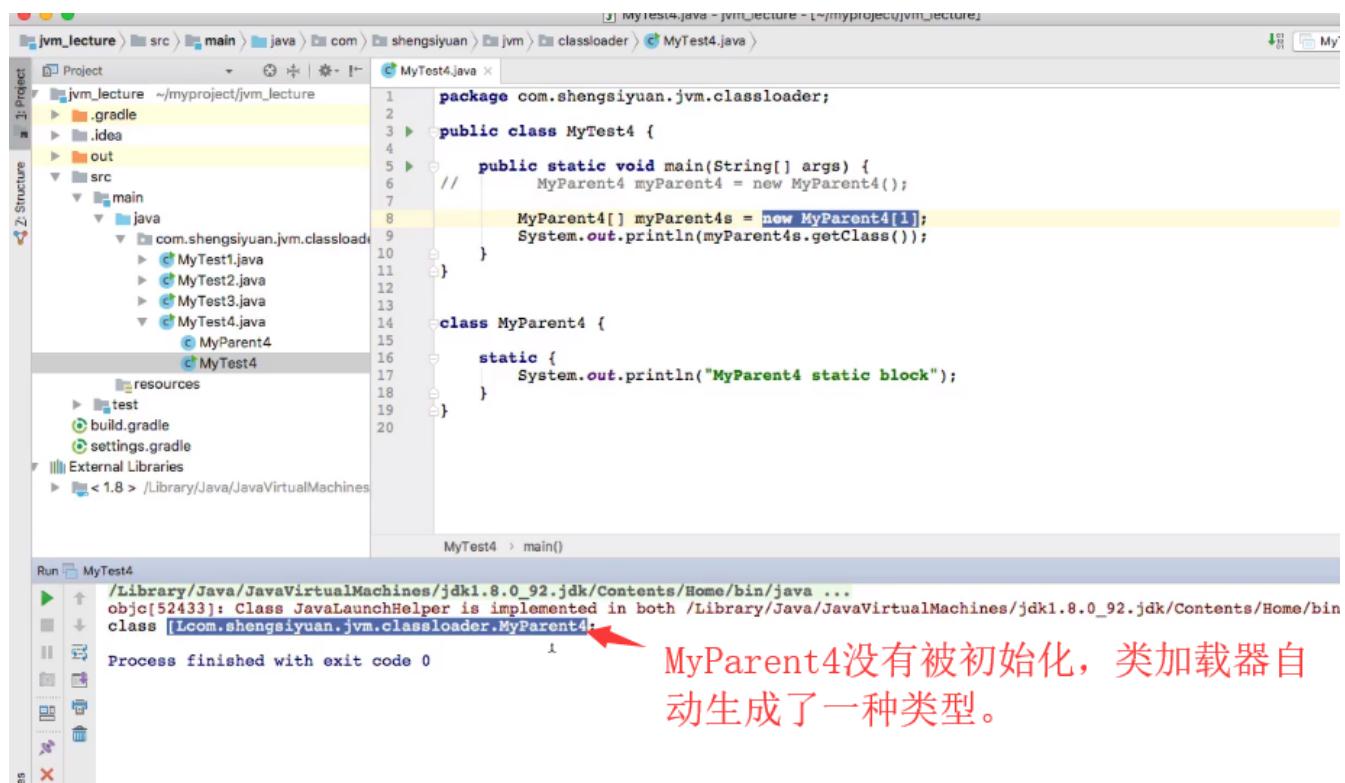
class MyParent3 {

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

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

```

例6



The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Structure:** The project is named "jvm_lecture". The "src/main/java/com/shengsiyuan/jvm/classloader" package contains several files: MyTest1.java, MyTest2.java, MyTest3.java, MyTest4.java, MyParent4.java, and MyTest4.java.
- Code Editor:** The file "MyTest4.java" is open, showing the following code:

```

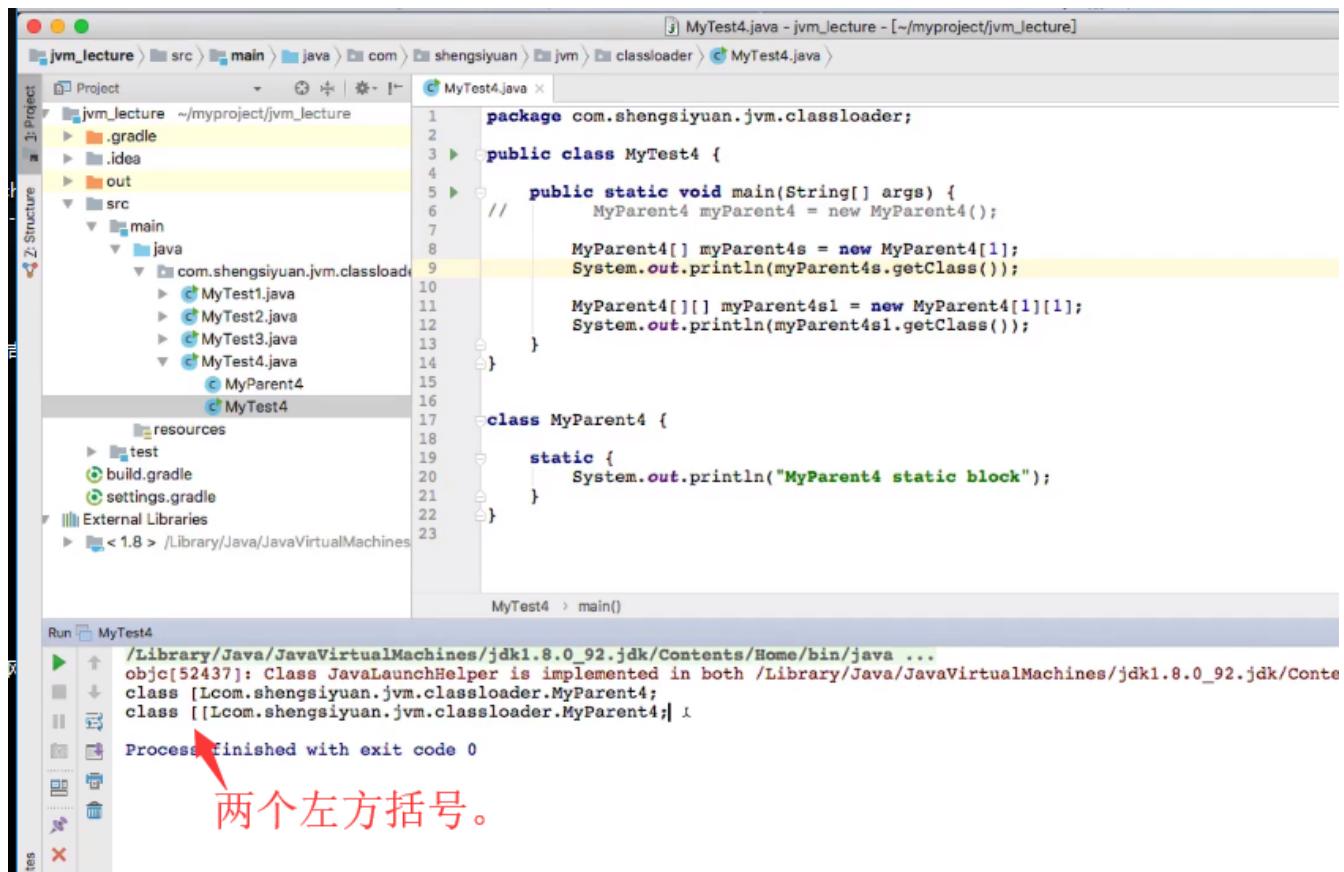
package com.shengsiyuan.jvm.classloader;
public class MyTest4 {
    public static void main(String[] args) {
        // MyParent4 myParent4 = new MyParent4();
        MyParent4[] myParent4s = new MyParent4[1];
        System.out.println(myParent4s.getClass());
    }
    class MyParent4 {
        static {
            System.out.println("MyParent4 static block");
        }
    }
}

```
- Run Tab:** The "Run" tab shows the output of the "MyTest4" run configuration. The output window displays:

```

/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
obj[52433]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin
class [Lcom.shengsiyuan.jvm.classloader.MyParent4;

```
- Annotations:** A red arrow points to the class name "MyParent4" in the run output, with the following red text annotation: "MyParent4没有被初始化，类加载器自动生成了一种类型。"



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。
*/
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());
}
}

```

```
class java.lang.Object
=====
class [I
class java.lang.Object
class [C
class [Z
class [S
class [B
I
Process finished with exit code 0
```

例7

The screenshot shows the IntelliJ IDEA interface. On the left is the Project tool window displaying a project structure with packages like `jvm_lecture` and `com.shengsiyuan.jvm.classloader`. Inside `com.shengsiyuan.jvm.classloader`, there are several Java files: `MyTest1.java`, `MyTest2.java`, `MyTest3.java`, `MyTest4.java`, and `MyParent4.java`. The `MyTest4.java` file is open in the main editor area, showing the following code:

```
package com.shengsiyuan.jvm.classloader;
public class MyTest4 {
    public static void main(String[] args) {
        MyParent4 myParent4 = new MyParent4();
        System.out.println("=====");
        MyParent4 myParent5 = new MyParent4();
    }
}
class MyParent4 {
    static {
        System.out.println("MyParent4 static block");
    }
}
```

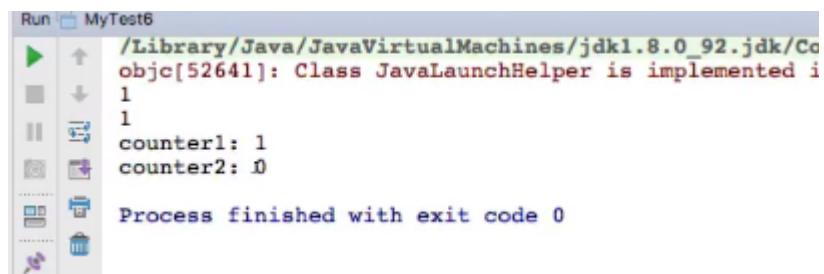
Below the editor is the Run tool window, which shows the command `/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...` and the output: `objc[52427]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...` followed by `MyParent4 static block` and `=====`. The status bar at the bottom indicates `Process finished with exit code 0`.

两次实例化，但是只有首次实例化算首次主动使用。

接口：

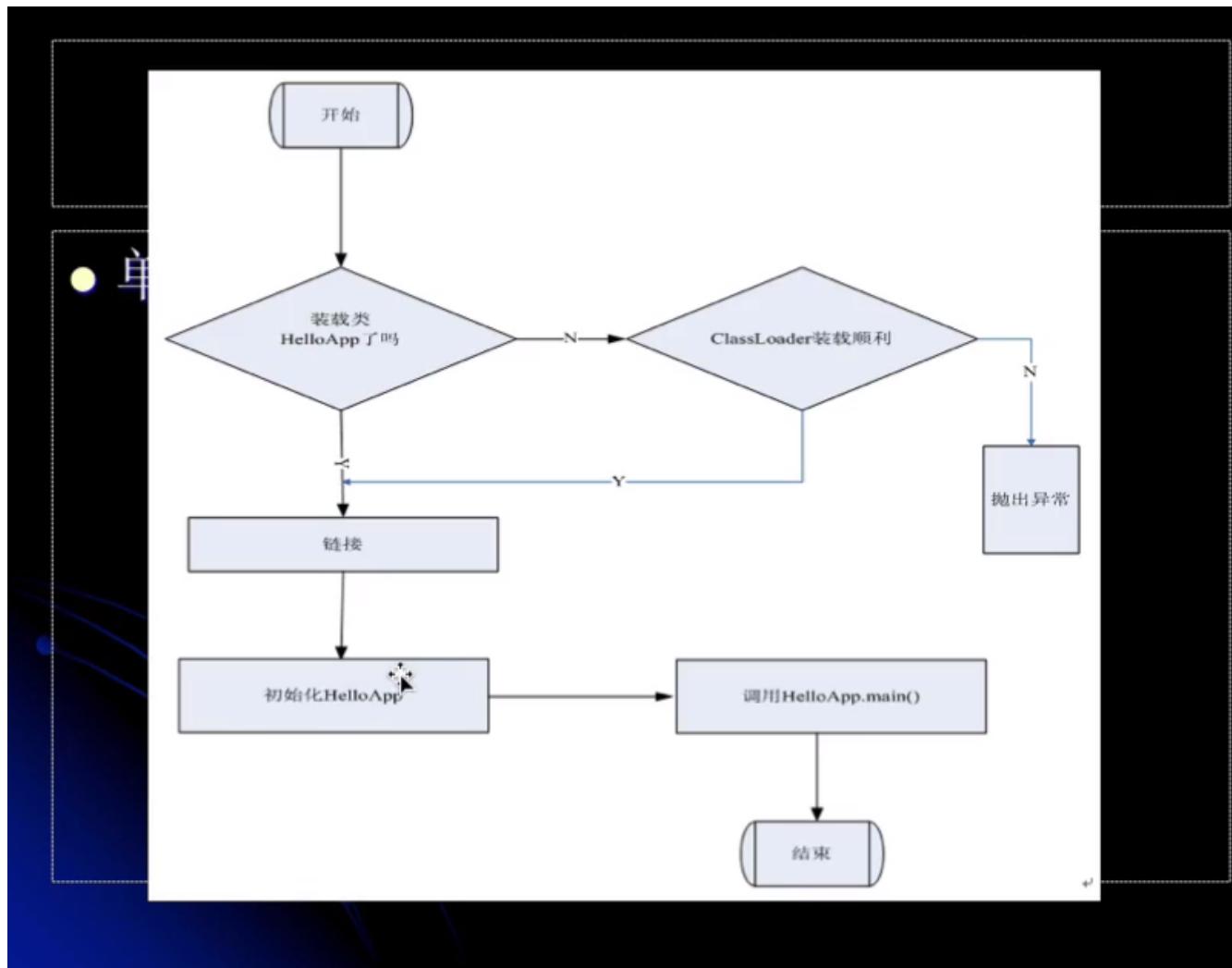
```
1 package com.shengsiyuan.jvm.classloader;
2
3 /**
4     当一个接口在初始化时，并不要求其父接口都完成了初始化
5     只有在真正使用到父接口的时候（如引用接口中所定义的常量时），才会初始化
6 */
7
8 import java.util.Random;
9
10 public class MyTest5 {
11
12     public static void main(String[] args) {
13         System.out.println(MyChild5.b);
14     }
15 }
16
17 interface MyParent5 {
18
19     public static final int a = 5;
20 }
21
22 interface MyChild5 extends MyParent5 {
23
24     public static final int b = new Random().nextInt(4);
25 }
```

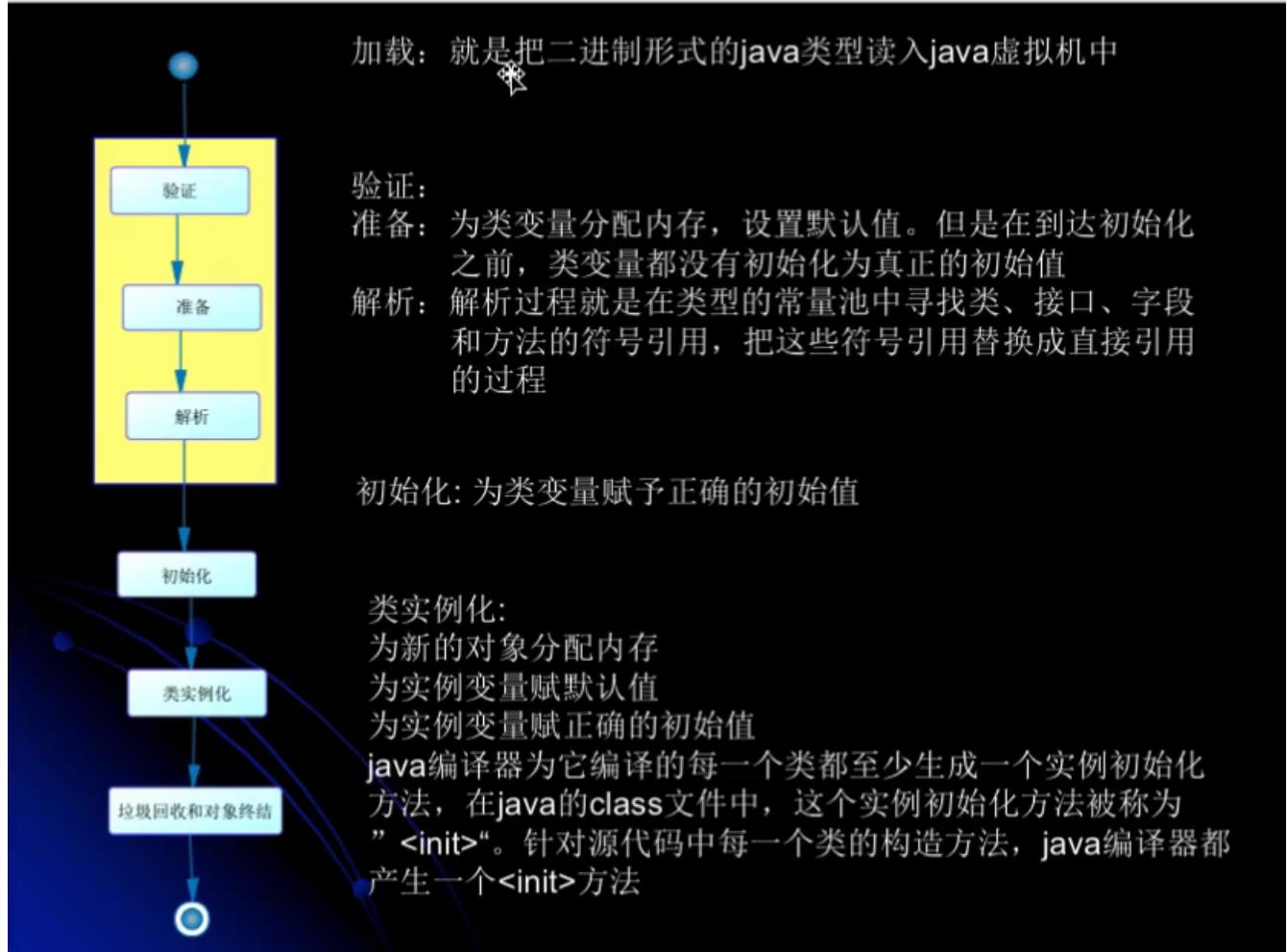
```
1 package com.shengsiyuan.jvm.classloader;
2
3 public class MyTest6 {
4
5     public static void main(String[] args) {
6         Singleton singleton = Singleton.getInstance();
7
8         System.out.println("counter1: " + Singleton.counter1);
9         System.out.println("counter2: " + Singleton.counter2);
10    }
11 }
12
13 class Singleton {
14
15     public static int counter1;
16
17     private static Singleton singleton = new Singleton();
18
19     private Singleton() {
20         counter1++;
21         counter2++;
22
23         System.out.println(counter1);
24         System.out.println(counter2);
25     }
26
27     public static int counter2 = 0;
28
29     public static Singleton getInstance() {
30         return singleton;
31     }
32 }
```



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

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;*;  
    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

- 主动使用（七种，重要）
 - - 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
 - - 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）

类的初始化时机

- 类的初始化时机
 - 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

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。

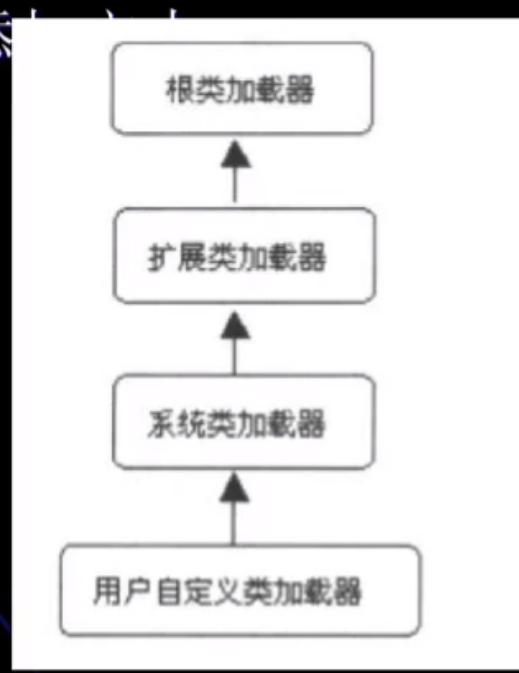
- 根 (Bootstrap) 类加载器：该加载器没有父加载器。它负责加载虚拟机的核心类库，如 `java.lang.*` 等。例如从例程 10-4 (`Sample.java`) 可以看出，`java.lang.Object` 就是由根类加载器加载的。根类加载器从系统属性 `sun.boot.class.path` 所指定的目录中加载类库。根类加载器的实现依赖于底层操作系统，属于虚拟机的实现的一部分，它并没有继承 `java.lang.ClassLoader` 类。
- 扩展 (Extension) 类加载器：它的父加载器为根类加载器。它从 `java.ext.dirs` 系统属性所指定的目录中加载类库，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。

类加载器

- 单击此处添加文字



10 初始化对于类与接口的异同点深入分析。

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。

```
27
28
29     public static int b = 5;
30 }
31
32
33 class C {
34     {
35         System.out.println("Hello");
36     }
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

```

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

```

```

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

例二：

```

import java.util.Random;
public class MyTest5 {
    public static void main(String[] args) {
        // System.out.println(MyChild5.b);
        new C();
        new C();
    }
}

```

```

x [Loaded sun.launcher.LauncherHelper$FXHelper from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/]
? [Loaded java.lang.Class$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

例三：

The screenshot shows a Java code editor with the following code:

```
7 import java.util.Random;
8
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 }
```

An arrow points to the static block in the C class. Below the code editor is a terminal window showing the execution output:

```
[Loaded sun.launcher.LauncherHelper from /Library/Java/JavaVirtualMachine/jdk-11.0.1/lib/jvm/libsunlauncher.dylib]
[Loaded java.lang.Class$MethodArray from /Library/Java/JavaVirtualMachine/jdk-11.0.1/lib/jvm/libjava.lang.dylib]
[Loaded java.net.AbstractPlainSocketImpl$1 from /Library/Java/JavaVirtualMachine/jdk-11.0.1/lib/jvm/libjava.net.dylib]
[Loaded java.lang.Void from /Library/Java/JavaVirtualMachine/jdk-11.0.1/lib/jvm/libjava.lang.dylib]
[Loaded com.shengsiyuan.jvm.classloader.C from file:/Users/shengsiyuan/Desktop/C.jar]
Hello
C
C
[Loaded java.lang.Shutdown from /Library/Java/JavaVirtualMachine/jdk-11.0.1/lib/jvm/libjava.lang.dylib]
[Loaded java.lang.Shutdown$Lock from /Library/Java/JavaVirtualMachine/jdk-11.0.1/lib/jvm/libjava.lang.dylib]

Process finished with exit code 0
```

The terminal also displays the status bar with "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。

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Structure:** The left sidebar shows the project structure. The `out` directory contains `production`, `classes`, and `com.shengsiyuan.jvm.classloader`. The `com.shengsiyuan.jvm.classloader` directory contains `C.java`, `MyChild1.java`, `MyChild5.java`, `MyParent1.java`, `MyParent2.java`, `MyParent3.java`, `MyParent4.java`, `MyParent5.java`, `MyTest1.java`, `MyTest2.java`, `MyTest3.java`, `MyTest4.java`, `MyTest5.java`, `MyTest6.java`, and `Singleton.java`. The `src/main/java` directory also contains these files.
- Code Editor:** The right pane displays the `MyTest5.java` file. The code defines a class `MyTest5` with a static main method and an interface `MyParent5` with a static thread field. It also defines a child class `MyChild5` that implements `MyParent5`. A breakpoint is set at the line `System.out.println("MyParent5 invoked");`. The code editor shows syntax highlighting and line numbers.
- Run Tab:** The bottom left tab bar shows the current configuration: "Run" (selected), "TODO", "Terminal", and "Messages".
- Output Tab:** The bottom right tab bar shows the current configuration: "Run" (selected), "TODO", "Terminal", and "Messages".
- Log:** The bottom pane displays a log of loaded classes. The log output is as follows:

```
[Loaded java.net.AbstractPlainSocketImpl$1 from /Library/Java/JavaVirtualMachine/lib/charsets.jar]
[Loaded com.shengsiyuan.jvm.classloader.MyParent5 from file:/Users/zhengsiyuan/IdeaProjects/Test/out/production/com/shengsiyuan/jvm/classloader/MyParent5.class]
[Loaded com.shengsiyuan.jvm.classloader.MyChild5 from file:/Users/zhengsiyuan/IdeaProjects/Test/out/production/com/shengsiyuan/jvm/classloader/MyChild5.class]
[Loaded com.shengsiyuan.jvm.classloader.MyParent5$1 from file:/Users/zhengsiyuan/IdeaProjects/Test/out/production/com/shengsiyuan/jvm/classloader/MyParent5$1.class]
[Loaded java.io.IOException from /Library/Java/JavaVirtualMachine/lib/charsets.jar]
[Loaded java.net.SocketException from /Library/Java/JavaVirtualMachine/lib/charsets.jar]
[Loaded java.lang.Shutdown from /Library/Java/JavaVirtualMachine/lib/charsets.jar]
[Loaded java.lang.Shutdown$Lock from /Library/Java/JavaVirtualMachine/lib/charsets.jar]
```

说明：接口并没有被初始化。

```
etImpl$1 from /Library/Java/  
oader.MyParent5 from file:/  
oader.MyChild5 from file:/U  
oader.MyParent$5] from file
```

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。

```
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
[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。

```
public class MyTest5 {
    public static void main(String[] args) {
        // System.out.println(MyChild5.b);
        System.out.println(MyParent5_1.thread);
    }
}
```

```
2
3
4 ① interface MyGrandpa5_1 {
5
6      public static Thread thread = new Thread() {
7
8          {
9              System.out.println("MyGrandpa5_1 invoked");
10         }
11     };
12 }
13
14 interface MyParent5_1 extends MyGrandpa5_1 {
15
16     public static Thread thread = new Thread() {
17
18         {
19             System.out.println("MyParent5_1 invoked");
20         }
21     };
22 }
```

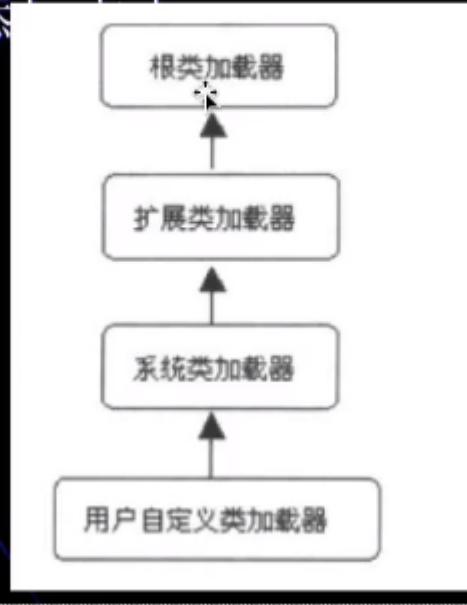
```
? [Loaded sun.net.spi.DefaultProxySelector from /Library/Java/JavaVirtualMachine/lib/sun/net/spi/DefaultProxySelector.class]
[Loaded sun.net.spi.DefaultProxySelector$1 from /Library/Java/JavaVirtualMachine/lib/sun/net/spi/DefaultProxySelector$1.class]
[Loaded com.shengsiyuan.jvm.classloader.MyParent5_1$1 from file:/Users/shengsiyuan/Desktop/MyParent5_1.jar!/com/shengsiyuan/jvm/classloader/MyParent5_1$1.class]
[Loaded sun.net.NetProperties from /Library/Java/JavaVirtualMachine/lib/sun/net/NetProperties.class]
[Loaded sun.net.NetProperties$1 from /Library/Java/JavaVirtualMachine/lib/sun/net/NetProperties$1.class]
MyParent5_1 invoked
Thread[Thread-0,5,main]
[Loaded java.util.Properties$LineReader from /Library/Java/JavaVirtualMachine/lib/java-util-1.8.0_111.jar!/java/util/Properties$LineReader.class]
```

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

类加载器

- 单击此处添加文本

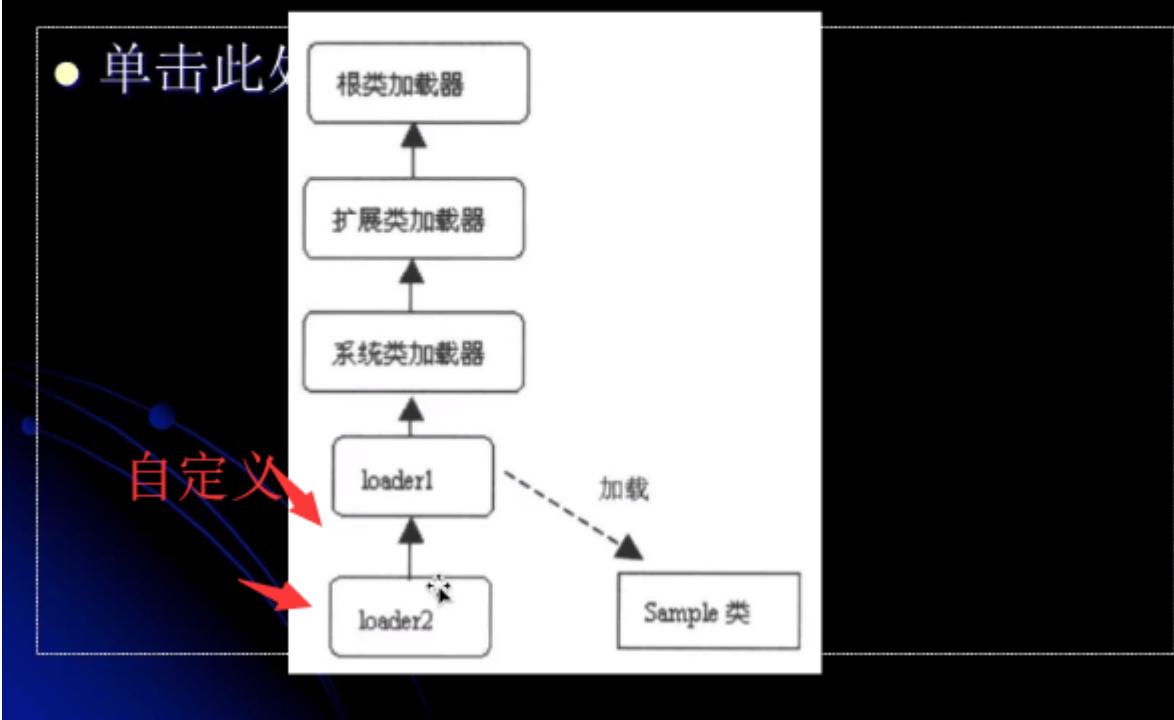


启动类加载器非java实现。

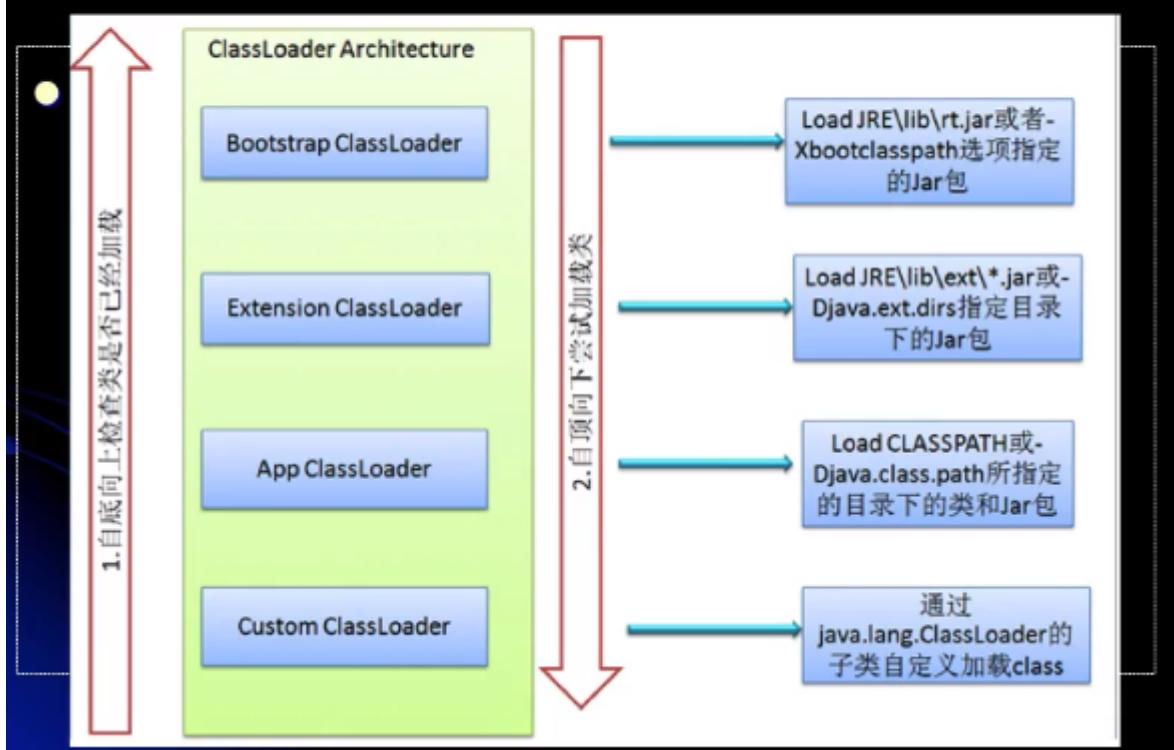
类加载器的父亲委托机制

- 在父亲委托机制中，各个加载器按照父子关系形成了树形结构，除了根类加载器之外，其余的类加载器都有且只有一个父加载器

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。

类加载器的父亲委托机制

- Bootstrap ClassLoader /启动类加载器
 - \$JAVA_HOME中jre/lib/rt.jar里所有的class，由C++实现，不是ClassLoader子类
- Extension ClassLoader/扩展类加载器
 - 负责加载java平台中扩展功能的一些jar包，包括\$JAVA_HOME中jre/lib/*.jar或-Djava.ext.dirs指定目录下的jar包
- App ClassLoader/ 系统类加载器
 - 负责加载classpath中指定的jar包及目录中class

类加载器的父亲委托机制

- 若有一个类加载器能够成功加载**Test**类，那么这个类加载器被称为**定义类加载器**，所有能成功返回**Class**对象引用的类加载器（包括**定义类加载器**）都被称为**初始类加载器**

Test类为需要加载的目标类。

例子：

```

1 package com.shengsiyuan.jvm.classloader;
2
3 public class MyTest7 {
4
5     public static void main(String[] args) throws Exception {
6         Class<?> clazz = Class.forName("java.lang.String");
7         System.out.println(clazz.getClassLoader());
8
9         Class<?> clazz2 = Class.forName("com.shengsiyuan.jvm.classloader.C");
10        System.out.println(clazz2.getClassLoader());
11    }
12
13    class C {
14    }
15
16 }
17

```

如果是根加载器加载的类
这里返回为null

系统类加载器

```

/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
objc[132]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (0x10982c4c0)
null
sun.misc.Launcher$AppClassLoader@18b4aac2
Process finished with exit code 0

```

12 类加载器与类初始化的深度剖析

例1：

```

1 package com.shengsiyuan.jvm.classloader;
2
3 class FinalTest {
4
5     public static final int x = 3;
6
7     static {
8         System.out.println("FinalTest static block");
9     }
10
11     public class MyTest8 {
12
13         public static void main(String[] args) {
14             System.out.println(FinalTest.x);
15         }
16     }
17 }
18

```

如果是根加载器加载的类
这里返回为null

FinalTest static block

```

+jvm_lecture cd out/production/classes
+ classes javap -c com.shengsiyuan.jvm.classloader.MyTest8
Compiled from "MyTest8.java"
public class com.shengsiyuan.jvm.classloader.MyTest8 {
    public com.shengsiyuan.jvm.classloader.MyTest8();
        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: iconst_3
            4: invokevirtual #4                  // Method java/io/PrintStream.println:(I)V
            7: return
}
classes

```

The screenshot shows the IntelliJ IDEA interface with the project structure and code editor. The code editor displays `MyTest8.java` containing a static final field `x` and a static block that prints "FinalTest static block". The terminal window shows the assembly output for the main method, which includes a call to `System.out.println(FinalTest.x)`. The assembly code shows the instruction `getstatic #3` which corresponds to the static field `x`.

```

package com.shengsiyuan.jvm.classloader;
import java.util.Random;
class FinalTest {
    public static final int x = new Random().nextInt(3);
    static {
        System.out.println("FinalTest static block");
    }
}
public class MyTest8 {
    public static void main(String[] args) {
        System.out.println(FinalTest.x);
    }
}

```

```

+ classes javap -c com.shengsiyuan.jvm.classloader.MyTest8
Compiled from "MyTest8.java"
public class com.shengsiyuan.jvm.classloader.MyTest8 {
    public com.shengsiyuan.jvm.classloader.MyTest8();
        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: getstatic     #3                  // Field com/shengsiyuan/jvm/classloader/FinalTest.x:I
            6: invokevirtual #4                  // Method java/io/PrintStream.println:(I)V
            9: return
}

```

证明：因此有一个静态的引用。

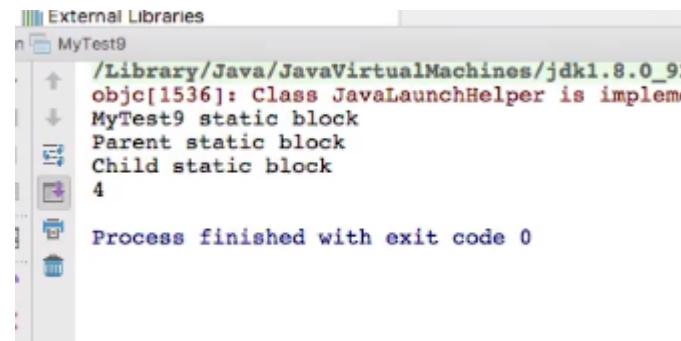
例2：

The screenshot shows the IntelliJ IDEA interface with the project structure and code editor. The code editor displays `MyTest9.java` containing two static blocks: one in `Parent` and one in `Child`, both printing their respective static blocks. The `main` method prints the value of `Child.b`. The assembly output shows the instruction `getstatic #3` which corresponds to the static field `b` in the `Child` class.

```

package com.shengsiyuan.jvm.classloader;
class Parent {
    static int a = 3;
    static {
        System.out.println("Parent static block");
    }
}
class Child extends Parent {
    static int b = 4;
    static {
        System.out.println("Child static block");
    }
}
public class MyTest9 {
    static {
        System.out.println("MyTest9 static block");
    }
}
public static void main(String[] args) {
    System.out.println(Child.b);
}

```



```

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

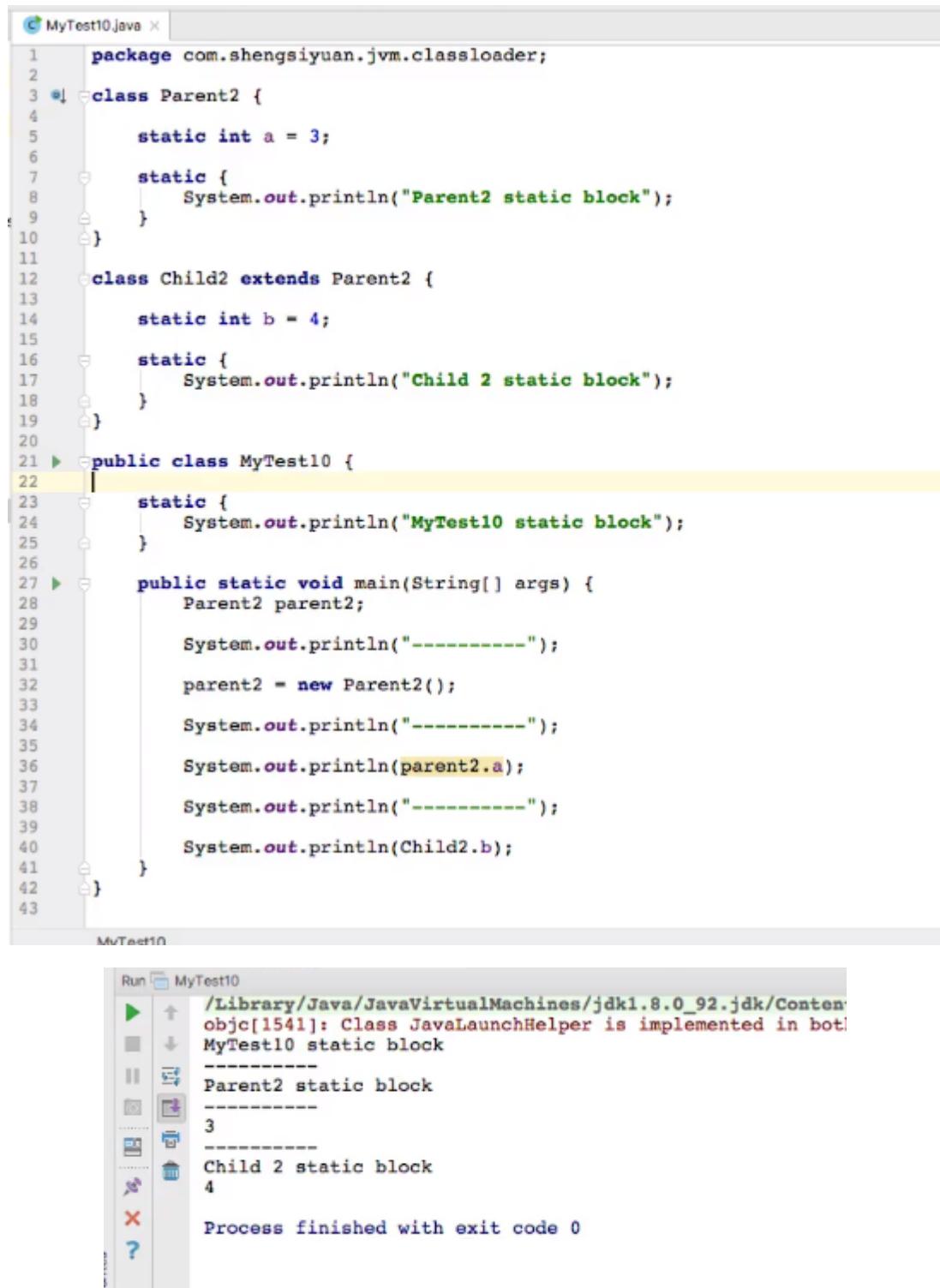
'a.io.IOException from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar'
'a.net.SocketException from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar'
'a.lang.Shutdown from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar'
'a.net.SocksSocketImpl$3 from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar'
'a.lang.Shutdown$Lock from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar'
'a.net.ProxySelector from /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar'
'a.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：



```

1 package com.shengsiyuan.jvm.classloader;
2
3 class Parent2 {
4     static int a = 3;
5     static {
6         System.out.println("Parent2 static block");
7     }
8 }
9
10 class Child2 extends Parent2 {
11     static int b = 4;
12     static {
13         System.out.println("Child 2 static block");
14     }
15 }
16
17 public class MyTest10 {
18     static {
19         System.out.println("MyTest10 static block");
20     }
21
22     public static void main(String[] args) {
23         Parent2 parent2;
24
25         System.out.println("-----");
26         parent2 = new Parent2();
27         System.out.println("-----");
28         System.out.println(parent2.a);
29         System.out.println("-----");
30         System.out.println(Child2.b);
31     }
32 }
33
34
35
36
37
38
39
40
41
42
43

```

Run MyTest10

```

objc[1541]: Class JavaLaunchHelper is implemented in both
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：

```
com>shengsiyuan>jvm>classloader>MyTest11.java
MyTest11.java
1 package com.shengsiyuan.jvm.classloader;
2
3 class Parent3 {
4     static int a = 3;
5
6     static {
7         System.out.println("Parent3 static block");
8     }
9
10    static void doSomething() {
11        System.out.println("do something");
12    }
13
14 }
15
16 class Child3 extends Parent3 {
17
18     static {
19         System.out.println("Child3 static block");
20     }
21
22 }
23
24 public class MyTest11 {
25
26     public static void main(String[] args) {
27         System.out.println(Child3.a);
28         Child3.doSomething();
29     }
30 }
```

```
External Libraries
MyTest11
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home
objc[1545]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/lib/server/libjava.dylib and /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/lib/server/libjava.dylib
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：

```

package com.shengsiyuan.jvm.classloader;

class CL {
    static {
        System.out.println("Class CL");
    }
}

// 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

public class MyTest12 {

    public static void main(String[] args) throws Exception {
        ClassLoader loader = ClassLoader.getSystemClassLoader();

        Class<?> clazz = loader.loadClass("com.shengsiyuan.jvm.classloader.CL");

        System.out.println(clazz);

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

        clazz = Class.forName("com.shengsiyuan.jvm.classloader.CL");

        System.out.println(clazz);
    }
}

```

获取class对象

获取class对象

```

objc[1547]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java and /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java
class com.shengsiyuan.jvm.classloader.CL
-----
Class CL
class com.shengsiyuan.jvm.classloader.CL
Process finished with exit code 0

```

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

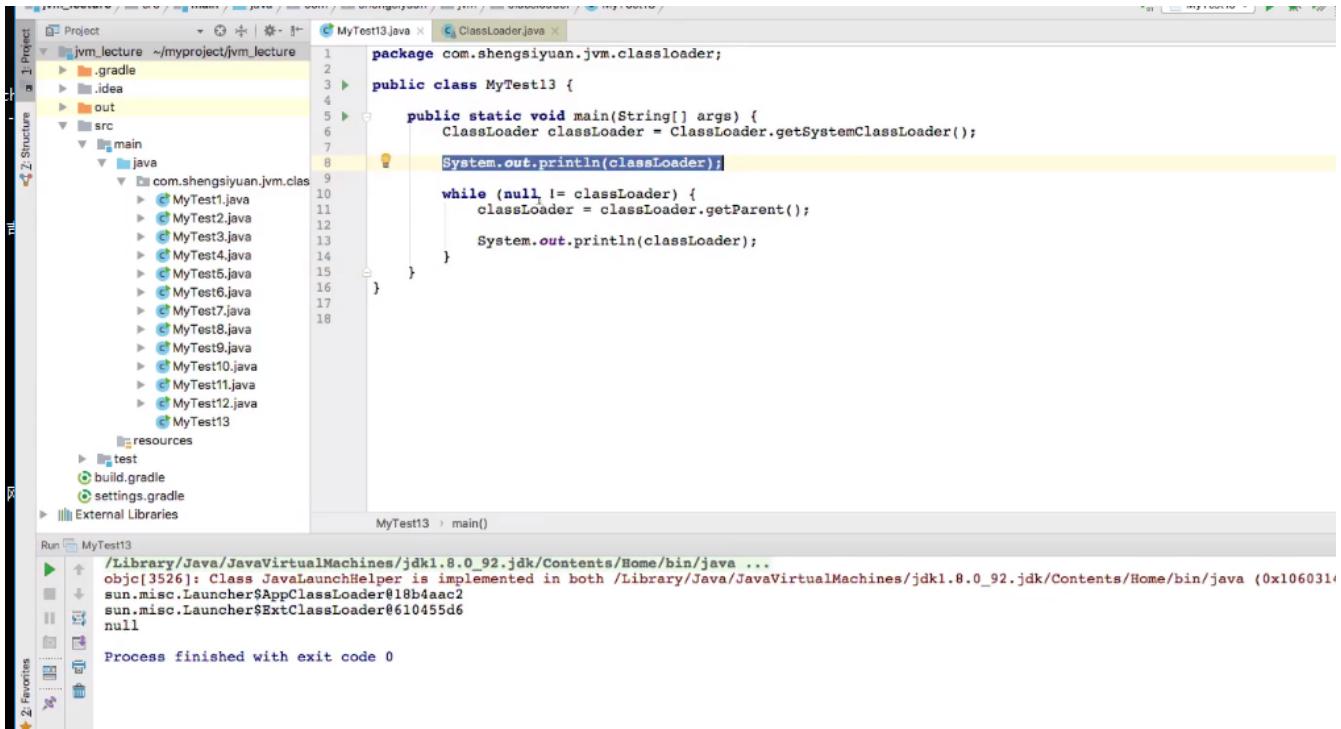
- ◦ ▪ public static ClassLoader getSystemClassLoader()

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。

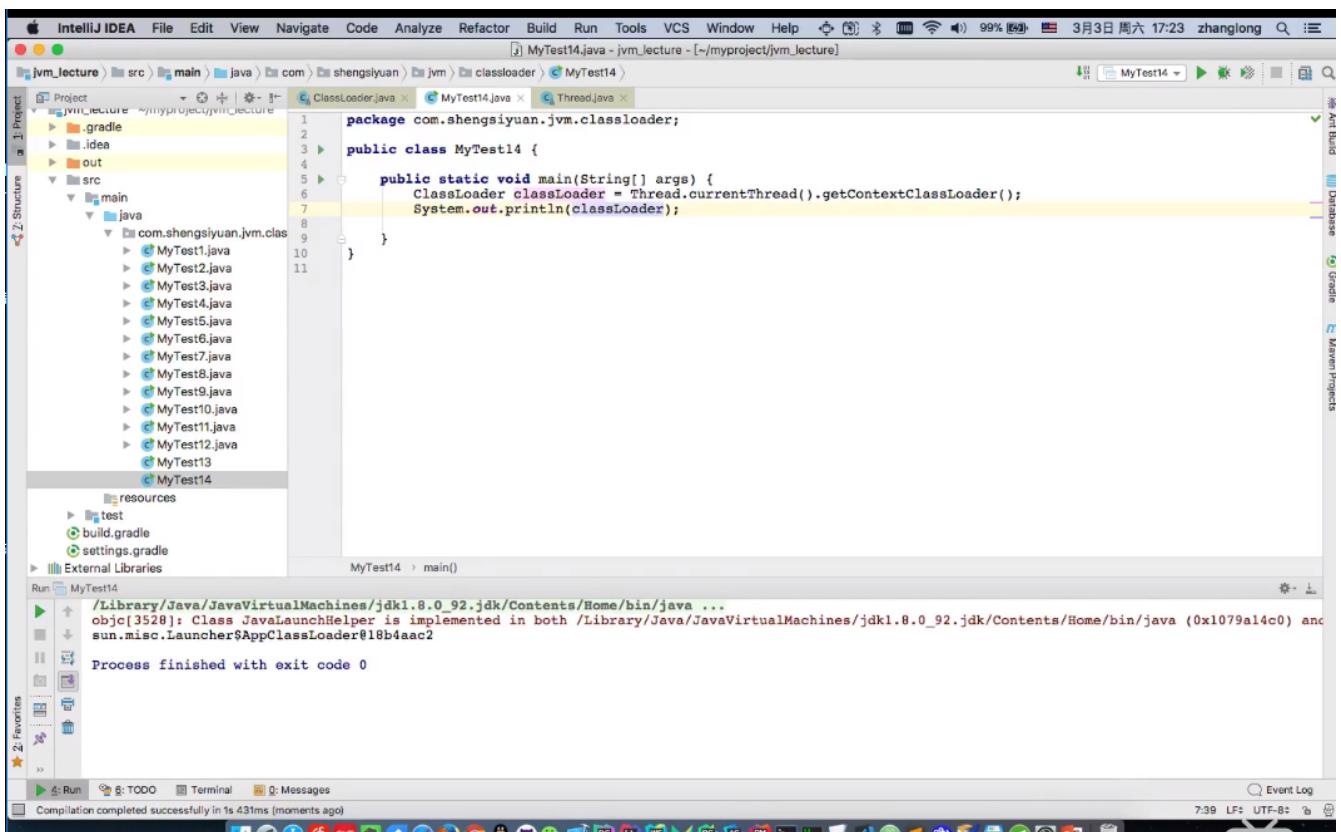
例1

获取类加载器：



```
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);
        }
    }
}
```

Run MyTest13
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
objc[352]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (0x106031400) and sun.misc.Launcher\$AppClassLoader@18b4aac2
sun.misc.Launcher\$ExtClassLoader@610455d6
null
Process finished with exit code 0



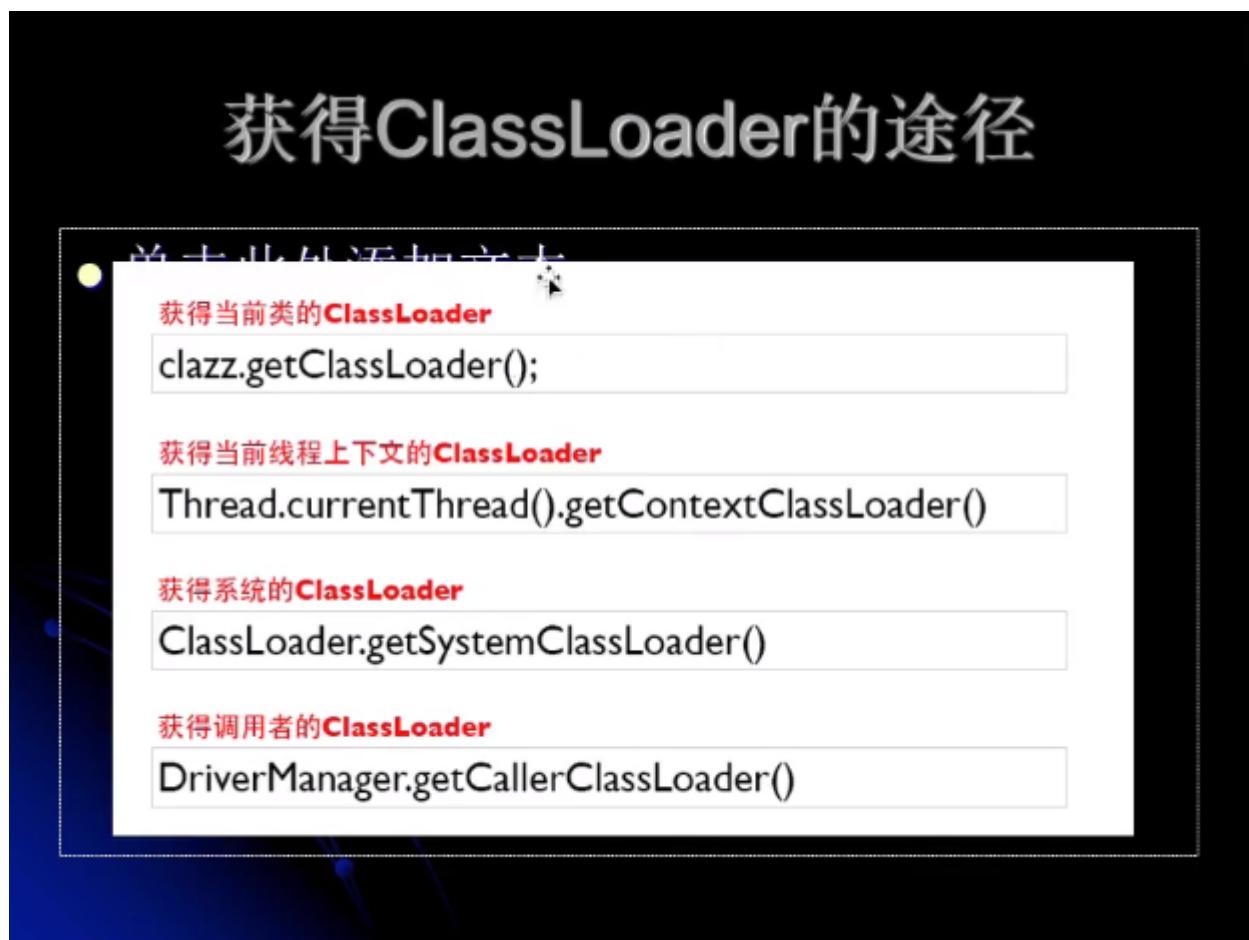
```
package com.shengsiyuan.jvm.classloader;
public class MyTest14 {
    public static void main(String[] args) {
        ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
        System.out.println(classLoader);
    }
}
```

Run MyTest14
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
objc[352]: 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
sun.misc.Launcher\$ExtClassLoader@610455d6
null
Process finished with exit code 0

获取资源：

```
MyTest14.java - jvm_lecture - [~/myproject/jvm_lecture]
Project: jvm_lecture ~/myproject/jvm_lecture
File | Edit | View | Navigate | Code | Analyze | Refactor | Build | Run | Tools | VCS | Window | Help | 99% 3月3日 周六 17:28 zhanglong | 3| Project: jvm_lecture ~/myproject/jvm_lecture
src | main | java | com | shengsiyuan | jvm | classloader | MyTest14 |
C ClassLoader.java C MyTest14.java Thread.java
1 package com.shengsiyuan.jvm.classloader;
2
3 import java.io.IOException;
4 import java.net.URL;
5 import java.util.Enumeration;
6
7 public class MyTest14 {
8
9     public static void main(String[] args) throws IOException {
10        ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
11
12        String resourceName = "com/shengsiyuan/jvm/classloader/MyTest13.class";
13
14        Enumeration<URL> urls = classLoader.getResources(resourceName);
15
16        while (urls.hasMoreElements()) {
17            URL url = urls.nextElement();
18            System.out.println(url);
19        }
20    }
21
22 }
```

Run MyTest14
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
objc[3530]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (0x1064074c0) and
file:/Users/zhanglong/myproject/jvm_lecture/out/production/classes/com/shengsiyuan/jvm/classloader/MyTest13.class
Process finished with exit code 0



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"
```

Since:
1.0

\$表示一个内部类，1表示匿名内部类

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.

```
package com.shengsiyuan.jvm.classloader;  
public class MyTest15 {  
    public static void main(String[] args) {  
        String[] strings = new String[2];  
        System.out.println(strings.getClass().getClassLoader());  
        System.out.println("-----");  
        MyTest15[] myTest15s = new MyTest15[2];  
        System.out.println(myTest15s.getClass().getClassLoader());  
        System.out.println("-----");  
        int[] ints = new int[2];  
        System.out.println(ints.getClass().getClassLoader());  
    }  
}
```

表示启动类加载器

系统类加载器

原生类型，没有加载器

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：

```
1 package com.shengsiyuan.jvm.classloader;
2
3 import com.sun.org.apache.xml.internal.resolver.readers.ExtendedXMLCatalogReader;
4
5 import java.io.ByteArrayOutputStream;
6 import java.io.File;
7 import java.io.FileInputStream;
8 import java.io.InputStream;
9
10 public class MyTest16 extends ClassLoader {
11
12     private String classLoaderName;
13
14     private final String fileExtension = ".class";
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     @Override
27     public String toString() {
28         return "[" + this.classLoaderName + "]";
29     }
30
31     @Override
32     protected Class<?> findClass(String className) throws ClassNotFoundException {
33         byte[] data = this.loadClassData(className);
34
35         return this.defineClass(className, data, 0, data.length);
36     }
37
38     private byte[] loadClassData(String name) {
39         InputStream is = null;
40         byte[] data = null;
41         ByteArrayOutputStream baos = null;
42
43         try {
44             is = this.getResourceAsStream(name);
45             if (is == null) {
46                 throw new ClassNotFoundException("未找到类文件");
47             }
48
49             baos = new ByteArrayOutputStream();
50             byte[] buffer = new byte[1024];
51             int len;
52             while ((len = is.read(buffer)) != -1) {
53                 baos.write(buffer, 0, len);
54             }
55
56             data = baos.toByteArray();
57         } catch (IOException e) {
58             e.printStackTrace();
59         } finally {
60             if (is != null) {
61                 try {
62                     is.close();
63                 } catch (IOException e) {
64                     e.printStackTrace();
65                 }
66             }
67         }
68
69         return data;
70     }
71 }
```

```

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;
}
}

```

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Structure:** The left sidebar shows the project structure under "Project". It includes "src", "main", "java", "com", "shengsiyuan", "jvm", "classloader", and "MyTest16". Inside "MyTest16", there are files like MyTest1.java through MyTest16.java.
- Code Editor:** The right pane displays the code for "ClassLoader.java". The code is identical to the one shown in the previous code block, handling file input and output to return class data as a byte array.
- Toolbars and Status Bar:** The top bar shows the current file as "MyTest16.java" and the status bar indicates "MyTest15".

```

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);
    }
}

```

```

> Task :MyClassLoader.main()
hello
successful
public class com.jlu.jvm.MyClassLoader
[null] ← 为什么是null
BUILD SUCCESSFUL in 0s

```

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：

The screenshot shows an IDE interface with a code editor and a terminal window.

Code Editor:

```

    ...
    return data;
}

public static void main(String[] args) throws Exception {
    MyTest16 loader1 = new MyTest16("loader1");
    //loader1.setPath("/Users/zhanglong/myproject/jvm_lecture/out/production/classes/");
    loader1.setPath("/Users/zhanglong/Desktop/");
    Class<?> clazz = loader1.loadClass("com.shengsiyuan.jvm.classloader.MyTest1");
    System.out.println("class: " + clazz.hashCode());
    Object object = clazz.newInstance();
    System.out.println(object);

    System.out.println();
    MyTest16 loader2 = new MyTest16("loader2");
    loader2.setPath("/Users/zhanglong/Desktop/");

    Class<?> clazz2 = loader2.loadClass("com.shengsiyuan.jvm.classloader.MyTest1");
    System.out.println("class: " + clazz2.hashCode());
    Object object2 = clazz2.newInstance();
    System.out.println(object2);

    System.out.println();
}

```

Terminal Window:

```

Run  MyTest16
▶  /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
obj[1365]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (0)
class: 1627674070
com.shengsiyuan.jvm.classloader.MyTest1@511d50c0

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

Process finished with exit code 0

```

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

The screenshot shows an IDE interface with two tabs: 'MyTest16.java' and 'Classloader.java'. The 'MyTest16.java' tab contains the following code:

```
MyTest16.java
1 package com.shengsiyuan.jvm.classloader;
2
3 public class MyTest16 {
4     public static void main(String[] args) throws Exception {
5         MyTest16 loader1 = new MyTest16("loader1");
6         loader1.setPath("/Users/zhanglong/myproject/jvm_lecture/out/production/classes/");
7         loader1.setPath("/Users/zhanglong/Desktop/");
8
9         Class<?> clazz = loader1.loadClass("com.shengsiyuan.jvm.classloader.MyTest1");
10        System.out.println("class: " + clazz.hashCode());
11        Object object = clazz.newInstance();
12        System.out.println(object);
13
14        System.out.println();
15
16        MyTest16 loader2 = new MyTest16("loader2");
17        loader2.setPath("/Users/zhanglong/Desktop/");
18
19        Class<?> clazz2 = loader2.loadClass("com.shengsiyuan.jvm.classloader.MyTest1");
20        System.out.println("class: " + clazz2.hashCode());
21        Object object2 = clazz2.newInstance();
22        System.out.println(object2);
23
24        System.out.println();
25    }
26 }
```

The 'Classloader.java' tab contains the following code:

```
Classloader.java
1 package com.shengsiyuan.jvm.classloader;
2
3 public class MyTest16 {
4     public static void main(String[] args) throws Exception {
5         MyTest16 loader1 = new MyTest16("loader1");
6         loader1.setPath("/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...");
7         objc[1368]; Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (0x1058f14c)
8         findClass invoked: com.shengsiyuan.jvm.classloader.MyTest1
9         class loader name: loader1
10        class: 1625635731
11        com.shengsiyuan.jvm.classloader.MyTest1@5e2de80c
12
13        findClass invoked: com.shengsiyuan.jvm.classloader.MyTest1
14        class loader name: loader2
15        class: 1872034366
16        com.shengsiyuan.jvm.classloader.MyTest1@5e481248
17
18    }
19 }
```

The terminal output shows the results of the class loading process:

```
Process finished with exit code 0
```

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：

命名空间

- 每个类加载器都有自己的命名空间，命名空间由该加载器及所有父加载器所加载的类组成。
- 在同一个命名空间中，不会出现类的完整名字（包括类的包名）相同的两个类
- 在不同的命名空间中，有可能会出现类的完整名字（包括类的包名）相同的两个类

18类的命名空间与卸载详解及jvisualvm的使用

The screenshot shows an IDE interface with a project named "jvm_lecture". The code editor displays "MyTest16.java" and "ClassLoader.java". The "ClassLoader.java" file contains code for creating multiple class loaders and loading the same class from different paths. The "MyTest16.java" file contains the main method which prints the class names and their hash codes. The run output shows the execution results.

```
public static void main(String[] args) throws Exception {
    MyTest16 loader1 = new MyTest16("loader1");
    //loader1.setPath("/Users/zhanglong/myproject/jvm_lecture/out/production/classes/");
    loader1.setPath("/Users/zhanglong/Desktop/");

    Class<?> clazz = loader1.loadClass("com.shengsiyuan.jvm.classloader.MyTest1");
    System.out.println("class: " + clazz.hashCode());
    Object object = clazz.newInstance();
    System.out.println(object);

    MyTest16 loader2 = new MyTest16(loader1, "loader2");
    loader2.setPath("/Users/zhanglong/Desktop/");

    Class<?> clazz2 = loader2.loadClass("com.shengsiyuan.jvm.classloader.MyTest1");
    System.out.println("class: " + clazz2.hashCode());
    Object object2 = clazz2.newInstance();
    System.out.println(object2);

    System.out.println();

    MyTest16 loader3 = new MyTest16("loader3");
    loader3.setPath("/Users/zhanglong/Desktop/");

    Class<?> clazz3 = loader3.loadClass("com.shengsiyuan.jvm.classloader.MyTest1");
    System.out.println("class: " + clazz3.hashCode());
```

Run MyTest16
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。

The screenshot shows an IDE interface with a project named "jvm_lecture". The code editor displays "MyTest16.java" and "ClassLoader.java". The "ClassLoader.java" file contains code for creating three class loaders (loader1, loader2, loader3) and loading the same class from different paths. The "MyTest16.java" file contains the main method which prints the class names and their hash codes. The run output shows the execution results.

```
public static void main(String[] args) throws Exception {
    MyTest16 loader1 = new MyTest16("loader1");
    //loader1.setPath("/Users/zhanglong/myproject/jvm_lecture/out/production/classes/");
    loader1.setPath("/Users/zhanglong/Desktop/");

    Class<?> clazz = loader1.loadClass("com.shengsiyuan.jvm.classloader.MyTest1");
    System.out.println("class: " + clazz.hashCode());
    Object object = clazz.newInstance();
    System.out.println(object);

    MyTest16 loader2 = new MyTest16(loader1, "loader2");
    loader2.setPath("/Users/zhanglong/Desktop/");

    Class<?> clazz2 = loader2.loadClass("com.shengsiyuan.jvm.classloader.MyTest1");
    System.out.println("class: " + clazz2.hashCode());
    Object object2 = clazz2.newInstance();
    System.out.println(object2);

    System.out.println();

    MyTest16 loader3 = new MyTest16("loader3");
    loader3.setPath("/Users/zhanglong/Desktop/");

    Class<?> clazz3 = loader3.loadClass("com.shengsiyuan.jvm.classloader.MyTest1");
    System.out.println("class: " + clazz3.hashCode());
```

Run MyTest16
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

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Structure:** Shows a tree view of class files: MyGrandpa.class, MyGrandpa\$1.class, MyParent1.class, MyParent2.class, MyParent3.class, MyParent4.class, MyParent5.class, MyParent5_1.class, MyTest2.class, MyTest3.class, MyTest4.class, MyTest5.class, MyTest6.class, MyTest7.class, MyTest8.class, MyTest9.class, MyTest10.class, MyTest11.class, MyTest12.class, MyTest13.class, and MyTest14.class.
- Code Editor:** Displays the `MyTest16.java` file with the following code:public static void main(String[] args) throws Exception {
 MyTest16 loader1 = new MyTest16("loader1");
 //loader1.setPath("/Users/zhanglong/myproject/jvm_lecture/out/production/classes/");
 loader1.setPath("/Users/zhanglong/Desktop/");

 Class<?> clazz = loader1.loadClass("com.shengsiyuan.jvm.classloader.MyTest1");
 System.out.println("class: " + clazz.hashCode());
 Object object = clazz.newInstance();
 System.out.println(object);

 System.out.println();

 MyTest16 loader2 = new MyTest16(loader1, "loader2");
 loader2.setPath("/Users/zhanglong/Desktop/");

 Class<?> clazz2 = loader2.loadClass("com.shengsiyuan.jvm.classloader.MyTest1");
 System.out.println("class: " + clazz2.hashCode());
 Object object2 = clazz2.newInstance();
 System.out.println(object2);

 System.out.println();
}
- Run Tab:** Shows the command run: `/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...`. The output window shows:objc[1431]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (0x101e5b4c0) and /System/Library/Java/JavaVirtualMachines/1.6.0_65/jdk/Contents/Home/bin/java (0x100000000). One version of class JavaLaunchHelper will be used.
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

findClass invoked: com.shengsiyuan.jvm.classloader.MyTest1
class loader name: loader3
class: 1581781576
com.shengsiyuan.jvm.classloader.MyTest1@866d3c617

Process finished with exit code 0

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：

```
public static void main(String[] args) throws Exception {
    MyTest16 loader1 = new MyTest16("loader1");
    //loader1.setPath("/Users/zhanglong/myproject/jvm_lecture/out/production/classes/");
    loader1.setPath("/Users/zhanglong/Desktop/");

    Class<?> clazz = loader1.loadClass("com.shengsiyuan.jvm.classloader.MyTest1");
    System.out.println("class: " + clazz.hashCode());
    Object object = clazz.newInstance();
    System.out.println(object);

    System.out.println();

    loader1 = new MyTest16("loader1");
    loader1.setPath("/Users/zhanglong/Desktop/");

    clazz = loader1.loadClass("com.shengsiyuan.jvm.classloader.MyTest1");
    System.out.println("class: " + clazz.hashCode());
    object = clazz.newInstance();
    System.out.println(object);

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

Run MyTest16

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
objc[1441]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/
findClass invoked: com.shengsiyuan.jvm.classloader.MyTest1
class loader name: loader1
class: 1625635731
com.shengsiyuan.jvm.classloader.MyTest1@5e2de80c

findClass invoked: com.shengsiyuan.jvm.classloader.MyTest1
class loader name: loader1
class: 1872034366
com.shengsiyuan.jvm.classloader.MyTest1@5e481248

Process finished with exit code 0
```

使用参数jvm来查看类的卸载或者使用jvisualvm

```
public static void main(String[] args) throws Exception {
    MyTest16 loader1 = new MyTest16("loader1");
    //loader1.setPath("/Users/zhanglong/myproject/jvm_lecture/out/production/classes/");
    loader1.setPath("/Users/zhanglong/Desktop/");

    Class<?> clazz = loader1.loadClass("com.shengsiyuan.jvm.classloader.MyTest1");
    System.out.println("class: " + clazz.hashCode());
    Object object = clazz.newInstance();
    System.out.println(object);

    System.out.println();

    loader1 = null;
    clazz = null;
    object = null; ← 小技巧

    System.gc();

    loader1 = new MyTest16("loader1");
    loader1.setPath("/Users/zhanglong/Desktop/");

    clazz = loader1.loadClass("com.shengsiyuan.jvm.classloader.MyTest1");
    System.out.println("class: " + clazz.hashCode());
    object = clazz.newInstance();
    System.out.println(object);
}
```

Run MyTest16

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
objc[1445]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (0x10a
findClass invoked: com.shengsiyuan.jvm.classloader.MyTest1
class loader name: loader1
class: 1625635731
com.shengsiyuan.jvm.classloader.MyTest1@5e2de80c

[Unloading class com.shengsiyuan.jvm.classloader.MyTest1 0x00000007c0061028]
findClass invoked: com.shengsiyuan.jvm.classloader.MyTest1
class loader name: loader1
class: 1872034366
com.shengsiyuan.jvm.classloader.MyTest1@5e481248

Process finished with exit code 0
```

19 自定义类加载器在复杂类加载情况下的分析

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

public class MyCat {

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

The screenshot shows the project structure and code for `MySample.java`. The code prints the class loader of `MySample` and creates an instance of `MyCat`.

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

public class MySample {

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

The screenshot shows the project structure and code for `MyTest17.java`. It uses reflection to load `MySample` from `MyCat` and create its instance.

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

public class MyTest17 {

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

        Class<?> clazz = loader1.loadClass("com.shengsiyuan.jvm.classloader.MySample");
        System.out.println("class: " + clazz.hashCode());

        Object object = clazz.newInstance();
    }
}
```

系统类加载器去加载的。

```

package com.shengsiyuan.jvm.classloader;
public class MyTest17 {
    public static void main(String[] args) throws Exception{
        MyTest16 loader1 = new MyTest16("loader1");
        Class<?> clazz = loader1.loadClass("com.shengsiyuan.jvm.classloader.MySample");
        System.out.println("class: " + clazz.hashCode());
        // Object object = clazz.newInstance();
    }
}

```

Run MyTest17
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
objc[5886]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (0x1065
class: 1360875712
Process finished with exit code 0

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

// 如果注释掉该行，那么并不会实例化MySample对象，即MySample构造方法不会被调用
// 因此不会实例化MyCat对象，即没有对MyCat进行主动使用，这里就不会加载MyCat Class
Object object = clazz.newInstance();

20 类加载器命名空间实战剖析与透彻理解

例子：

```

package com.shengsiyuan.jvm.classloader;
public class MyTest17_1 {
    public static void main(String[] args) throws Exception{
        MyTest16 loader1 = new MyTest16("loader1");
        loader1.setPath("/Users/zhanglong/Desktop/");
        Class<?> clazz = loader1.loadClass("com.shengsiyuan.jvm.classloader.MySample");
        System.out.println("class: " + clazz.hashCode());
        // 如果注释掉该行，那么并不会实例化MySample对象，即MySample构造方法不会被调用
        // 因此不会实例化MyCat对象，即没有对MyCat进行主动使用，这里就不会加载MyCat Class
        Object object = clazz.newInstance();
    }
}

```

MyTest17_1
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
objc[5958]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java
findClass invoked: com.shengsiyuan.jvm.classloader.MySample
class loader name: loader1
class: 1580066828
MySample is loaded by: com.shengsiyuan.jvm.classloader.MyTest16@511d50c0
findClass invoked: com.shengsiyuan.jvm.classloader.MyCat
class loader name: loader1
MyCat is loaded by: com.shengsiyuan.jvm.classloader.MyTest16@511d50c0
Process finished with exit code 0

删除MyCat的class文件

```

package com.shengsiyuan.jvm.classloader;
public class MyTest17_1 {
    public static void main(String[] args) throws Exception{
        MyTest16 loader1 = new MyTest16("loader1");
        loader1.setPath("/Users/zhanglong/Desktop/");
        Class<?> clazz = loader1.loadClass("com.shengsiyuan.jvm.classloader.MySample");
        System.out.println("class: " + clazz.hashCode());
        // 如果注释掉该行，那么并不会实例化MySample对象，即MySample构造方法不会被调用
        // 因此不会实例化MyCat对象，即没有对MyCat进行主动使用，这里就不会加载MyCat Class
        Object object = clazz.newInstance();
    }
}

```

Run MyTest17_1

```

/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
objc[5962]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (0x1005aa4c0
class: 1360875712
MySample is loaded by: sun.misc.Launcher$AppClassLoader@18b4aac2
Exception in thread "main" java.lang.NoClassDefFoundError: com/shengsiyuan/jvm/classloader/MyCat
at com.shengsiyuan.jvm.classloader.MySample.<init>(MySample.java:8) <4 internal calls>
at java.lang.Class.newInstance(Class.java:42)
at com.shengsiyuan.jvm.classloader.MyTest17_1.main(MyTest17_1.java:14)
Caused by: java.lang.ClassNotFoundException: com.shengsiyuan.jvm.classloader.MyCat
at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:331)
at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
... 7 more
Process finished with exit code 1

```

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

```

package com.shengsiyuan.jvm.classloader;
public class MyTest17_1 {
    public static void main(String[] args) throws Exception{
        MyTest16 loader1 = new MyTest16("loader1");
        loader1.setPath("/Users/zhanglong/Desktop/");
        Class<?> clazz = loader1.loadClass("com.shengsiyuan.jvm.classloader.MySample");
        System.out.println("class: " + clazz.hashCode());
        // 如果注释掉该行，那么并不会实例化MySample对象，即MySample构造方法不会被调用
        // 因此不会实例化MyCat对象，即没有对MyCat进行主动使用，这里就不会加载MyCat Class
        Object object = clazz.newInstance();
    }
}

```

Run MyTest17_1

```

/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
objc[5965]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (0x1058a64c
findClass invoked: com.shengsiyuan.jvm.classloader.MySample
class loader name: loader1
class: 1580066828
MySample is loaded by: com.shengsiyuan.jvm.classloader.MyTest16@511d50c0
MyCat is loaded by: sun.misc.Launcher$AppClassLoader@18b4aac2
Process finished with exit code 0

```

这里的加载MySimple是自定义的类加载器，在加载MySimple时遇到new MyCat()，这个自定义的加载器尝试去加载MyCat，但是MyCat的class文件在，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。

The screenshot displays two Java IDE windows, likely IntelliJ IDEA, illustrating the class loading process for two different classes: MyCat and MyTest17_1.

Top Window (MyCat.java):

- Project structure shows a package com.shengsiyuan.jvm.classloader containing several class files: C.class, Child.class, Child2.class, Child3.class, CL.class, FinalTest.class, MyCat.class, MyChild1.class, MyChild5.class, MyGrandpa.class, MyGrandpa5_1.class, MyParent1.class, MyParent2.class, MyParent3.class, MyParent4.class, MyParent5.class, MyParent5_1.class, MyTest1.class, and MyTest2.class.
- The code for MyCat is shown:

```
package com.shengsiyuan.jvm.classloader;
public class MyCat {
    public MyCat() {
        System.out.println("MyCat is loaded by: " + this.getClass().getClassLoader());
        System.out.println("from MyCat: " + MySample.class);
    }
}
```

- Output window shows the following log:

```
MyCat is loaded by: com.shengsiyuan.jvm.classloader.loader1
MyCat is loaded by: sun.misc.Launcher$AppClassLoader@18b4aac2
Process finished with exit code 0
```

Bottom Window (MyTest17_1.java):

- Project structure shows a package com.shengsiyuan.jvm.classloader containing several class files: C.class, Child.class, Child2.class, Child3.class, CL.class, FinalTest.class, MyCat.class, MyChild1.class, MyChild5.class, MyGrandpa.class, MyGrandpa5_1.class, MyParent1.class, and MyParent2.class.
- The code for MyTest17_1 is shown:

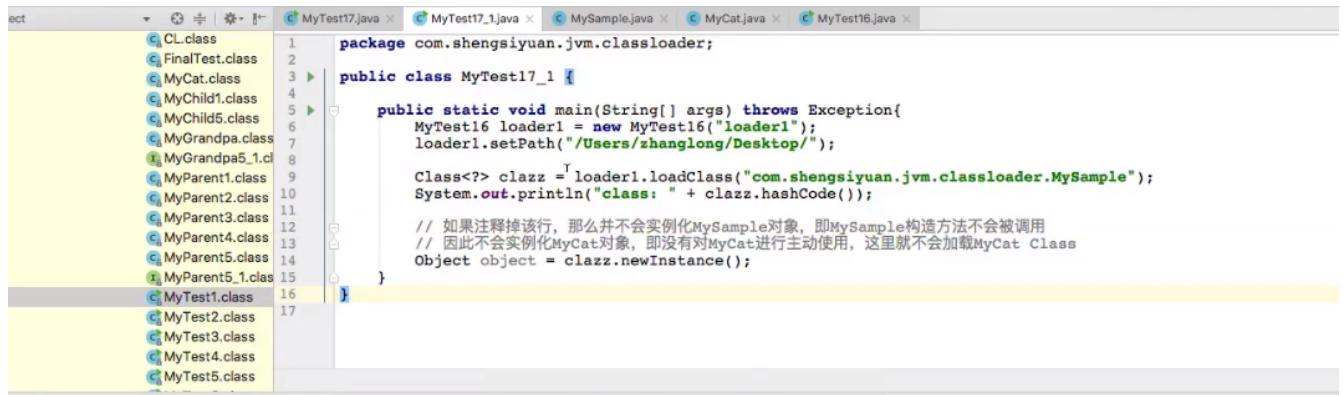
```
package com.shengsiyuan.jvm.classloader;
public class MyTest17_1 {
    public static void main(String[] args) throws Exception {
        MyTest16 loader1 = new MyTest16("loader1");
        loader1.setPath("/Users/zhanglong/Desktop/");
        Class<?> clazz = loader1.loadClass("com.shengsiyuan.jvm.classloader.MySample");
        System.out.println("class: " + clazz.hashCode());
        // If the annotation is removed, it won't instantiate MySample object, so MySample constructor won't be called
        // Therefore, MyCat won't be instantiated, as there is no active use of MyCat, so it won't load MyCat Class
        Object object = clazz.newInstance();
    }
}
```

- Output window shows the following log, indicating a ClassNotFoundException for MySample:

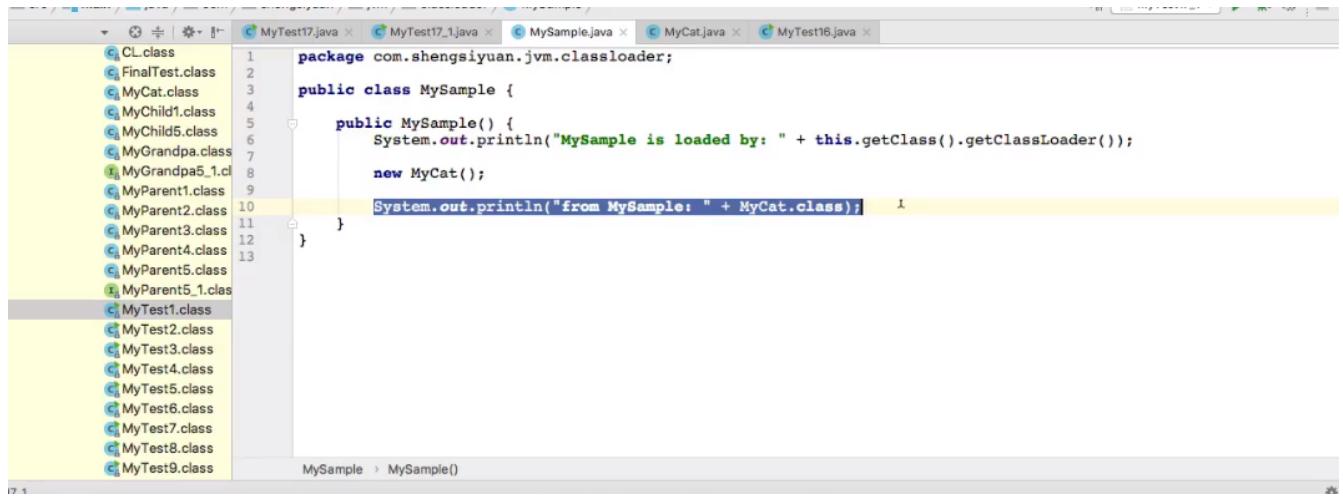
```
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
objc[5974]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (0x1058a64c0) and
findClass invoked: com.shengsiyuan.jvm.classloader.MySample
class loader name: loader1
class: 1580066828
MySample is loaded by: com.shengsiyuan.jvm.classloader.MyTest16@511d50c0
MyCat is loaded by: sun.misc.Launcher$AppClassLoader@18b4aac2
Exception in thread "main" java.lang.NoClassDefFoundError: com/shengsiyuan/jvm/classloader/MySample
at com.shengsiyuan.jvm.classloader.MyCat.<init>(MyCat.java:8)
at com.shengsiyuan.jvm.classloader.MySample.<init>(MySample.java:8) <4 internal calls>
at java.lang.Class.newInstance(Class.java:442)
at com.shengsiyuan.jvm.classloader.MyTest17_1.main(MyTest17_1.java:14)
Caused by: java.lang.ClassNotFoundException: com.shengsiyuan.jvm.classloader.MySample
at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
at java.lang.ClassLoader.loadClass(CLassLoader.java:424)
at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:331)
at java.lang.ClassLoader.loadClass(CLassLoader.java:357)
... 8 more
Process finished with exit code 1
```

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空间。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。

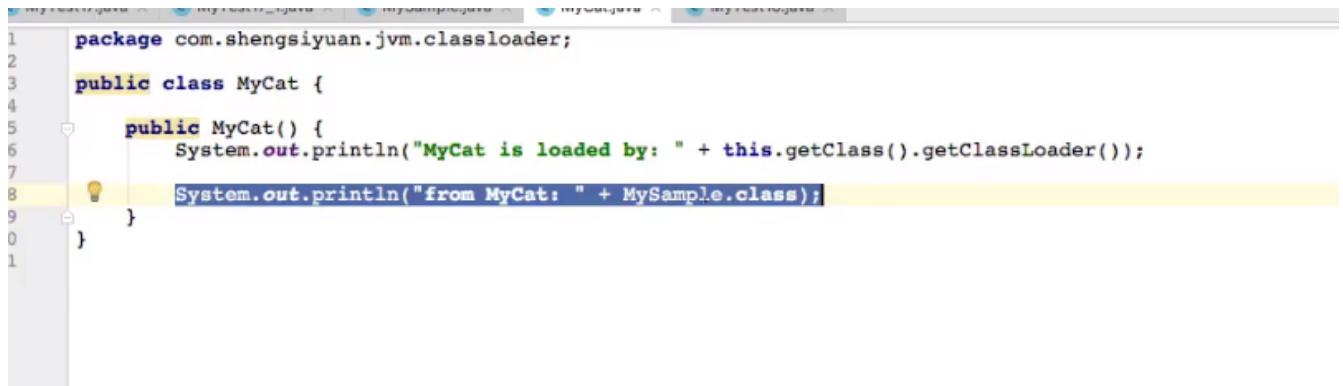
还是删掉mysimple，但是也删掉MyCat对MySimple的引用。



```
1 package com.shengsiyuan.jvm.classloader;
2
3 public class MyTest17_1 {
4
5     public static void main(String[] args) throws Exception{
6         MyTest16 loader1 = new MyTest16("loader1");
7         loader1.setPath("/Users/zhanglong/Desktop/");
8
9         Class<?> clazz = loader1.loadClass("com.shengsiyuan.jvm.classloader.MySample");
10        System.out.println("class: " + clazz.hashCode());
11
12        // 如果注释掉该行，那么并不会实例化MySample对象，即MySample构造方法不会被调用
13        // 因此不会实例化MyCat对象，即没有对MyCat进行主动使用，这里就不会加载MyCat Class
14        Object object = clazz.newInstance();
15    }
16
17 }
```

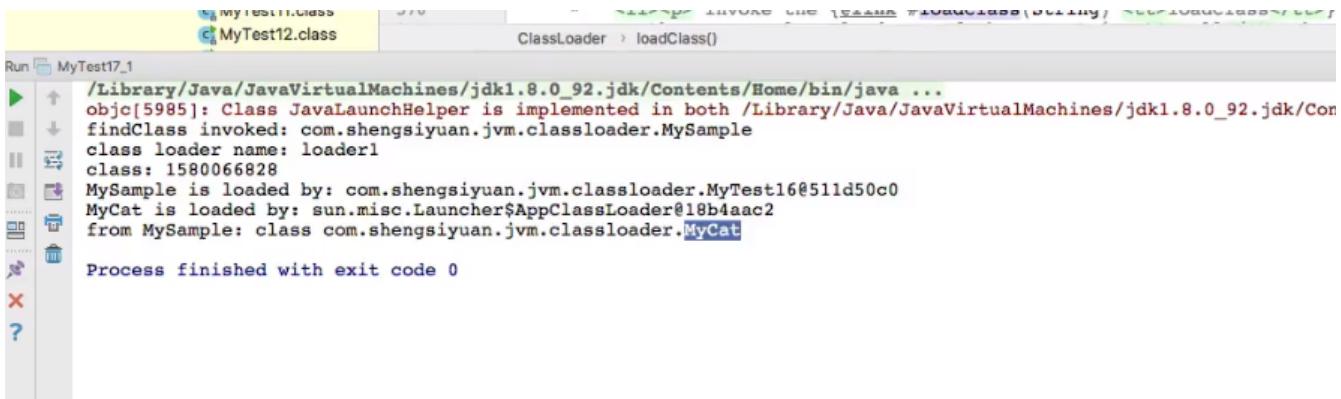


```
1 package com.shengsiyuan.jvm.classloader;
2
3 public class MySample {
4
5     public MySample() {
6         System.out.println("MySample is loaded by: " + this.getClass().getClassLoader());
7
8         new MyCat();
9
10        System.out.println("from MySample: " + MyCat.class);
11    }
12
13 }
```



```
1 package com.shengsiyuan.jvm.classloader;
2
3 public class MyCat {
4
5     public MyCat() {
6         System.out.println("MyCat is loaded by: " + this.getClass().getClassLoader());
7
8         System.out.println("from MyCat: " + MySample.class);
9     }
10
11 }
```

注释掉标记的行。



结论：

```
4      关于命名空间的重要说明
5
6
7      1. 子加载器所加载的类能够访问父加载器所加载的类
8      2. 父加载器所加载的类无法访问到子加载器所加载的类
9
10
11 > public class MyTest17_1 {
```

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"));
    }
}
```

IDEA Screenshot showing the project structure and code execution output.

Project Structure:

- HelloGradle
- .gradle
- .idea
- build
 - classes
 - java
 - main
 - com
 - jlu
 - jvm
 - MyClassLoader.class
 - MyTest18.class
 - tmp
 - gradle
 - src
 - main
 - java
 - com.jlu.jvm
- com.jlu.jvm

Code Editor (MyTest18.java):

```

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"));
    }
}

```

Run Output:

```

10:23:57: Executing task 'MyTest18.main()'...

> Task :compileJava UP-TO-DATE
> Task :processResources NO-SOURCE
> Task :classes UP-TO-DATE

> Task :MyTest18.main()
null
null
E:\HelloGradle\build\classes\java\main;E:\HelloGradle\build\resources\main

BUILD SUCCESSFUL in 1s

```

为什么是null

IntelliJ IDEA Screenshot showing the project structure and code execution output.

Project Structure:

- resources.jar
- Project
- Structure
- MyTest18
- java
 - com.shengsiyuan.jvm.classloader
 - MyTest18.java
- resources
- bin
- lib
- src

Code Editor (MyTest18.java):

```

package com.shengsiyuan.jvm.classloader;

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"));
    }
}

```

Run Output:

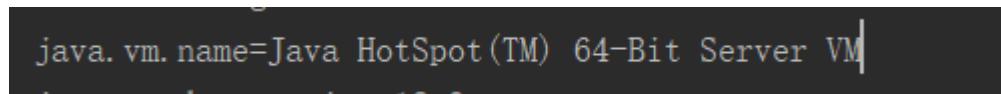
```

MyTest18
'/bin/java ...
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (0x100abe4c0) and /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Hom
'/jre/lib/resources.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/ext:/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Content
alMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/ext:/Library/Java/Extensions:/Network/Library/Java/Extensions:/System/Library/Java/Extensions:/usr
'/jre/lib/charsets.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/deploy.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Content

```

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



放到这个目录下

```

1 package com.shengsiyuan.jvm.classloader;
2
3 public class MyTest18_1 {
4
5     public static void main(String[] args) throws Exception {
6         MyTest16 loader1 = new MyTest16("loader1");
7         loader1.setPath("/Users/zhanglong/Desktop/");
8
9         Class<?> clazz = loader1.loadClass("com.shengsiyuan.jvm.classloader.MyTest1");
10
11        System.out.println("class: " + clazz.hashCode());
12        System.out.println("class loader: " + clazz.getClassLoader());
13    }
14
15 }

```

结果：

根类加载器

```

1 package com.shengsiyuan.jvm.classloader;
2
3 public class MyTest18_1 {
4
5     public static void main(String[] args) throws Exception {
6         MyTest16 loader1 = new MyTest16("loader1");
7         loader1.setPath("/Users/zhanglong/Desktop/");
8
9         Class<?> clazz = loader1.loadClass("com.shengsiyuan.jvm.classloader.MyTest1");
10
11        System.out.println("class: " + clazz.hashCode());
12        System.out.println("class loader: " + clazz.getClassLoader());
13    }
14
15 }

```

/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
objc[6161]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (0x106b5a4c0)
class: 1627674070
class loader: null

Process finished with exit code 0

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：

The screenshot shows the IntelliJ IDEA interface. In the code editor, the file `MyTest19.java` is open, containing the following code:

```

1 package com.shengsiyuan.jvm.classloader;
2
3 import com.sun.crypto.provider.AESKeyGenerator;
4
5 public class MyTest19 {
6
7     public static void main(String[] args) {
8         AESKeyGenerator aesKeyGenerator = new AESKeyGenerator();
9
10        System.out.println(aesKeyGenerator.getClass().getClassLoader());
11        System.out.println(MyTest19.class.getClassLoader());
12    }
13}
14

```

In the run tab, the output shows:

```

/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

```

The screenshot shows a terminal window with the following command and output:

```

+ jvm_lecture cd out/production/classes
+ classes ls
com
+ classes java -Djava.ext.dirs=/ com.shengsiyuan.jvm.classloader.MyTest19
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

```

Red annotations are present: "当前目录" (Current Directory) above the command line, and "修改属性" (Modify Properties) next to the error message.

例子2：

The screenshot shows the IntelliJ IDEA interface with a different project structure. The code editor has `MyTest20.java` open, containing:

```

1 package com.shengsiyuan.jvm.classloader;
2
3 public class MyPerson {
4
5     private MyPerson myPerson;
6
7     public void setMyPerson(Object object) {
8         this.myPerson = (MyPerson) object;
9     }
10}
11

```

```

1 package com.shengsiyuan.jvm.classloader;
2 import java.lang.reflect.Method;
3
4 public class MyTest20 {
5
6     public static void main(String[] args) throws Exception{
7         MyTest16 loader1 = new MyTest16("loader1");
8         MyTest16 loader2 = new MyTest16("loader2");
9
10        Class<?> clazz1 = loader1.loadClass("com.shengsiyuan.jvm.classloader.MyPerson");
11        Class<?> clazz2 = loader2.loadClass("com.shengsiyuan.jvm.classloader.MyPerson");
12
13        System.out.println(clazz1 == clazz2);
14
15        Object object1 = clazz1.newInstance();
16        Object object2 = clazz2.newInstance();
17
18        Method method = clazz1.getMethod("setMyPerson", Object.class);
19        method.invoke(object1, object2);
20    }
21
22 }
23

```

对象 参数 方法名 参数

```

* @since JDK1.1
*/
@CallerSensitive
public Method getMethod(String name, Class<?>... parameterTypes)
    throws NoSuchMethodException, SecurityException {
    checkMemberAccess(Member.PUBLIC, Reflection.getCallerClass(), true);
    Method method = getMethod0(name, parameterTypes, true);
    if (method == null) {
        throw new NoSuchMethodException(getName() + "." + name + argumentTypesToString(parameterTy
    }

    * PROVOKED BY THIS METHOD CALLS.
*/
@CallerSensitive
public Object invoke(Object obj, Object... args)
    throws IllegalAccessException, IllegalArgumentException,
    InvocationTargetException
{
    if (!override) {
        if (!Reflection.quickCheckMemberAccess(clazz, modifiers)) {

```

class 类型的可变参数，0
个
或者多个

例子3：

删除 person class文件，将其放到桌面

```

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", Object.class);
        method.invoke(object1, object2);
    }
}

```

```
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) and
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。

不同类加载器的命名空间关系

同一个命名空间内的类是相互可见的。

子加载器的命名空间包含所有父加载器的命名空间。因此由子加载器加载的类能看见父加载器加载的类。例如系统类加载器加载的类能看见根类加载器加载的类。

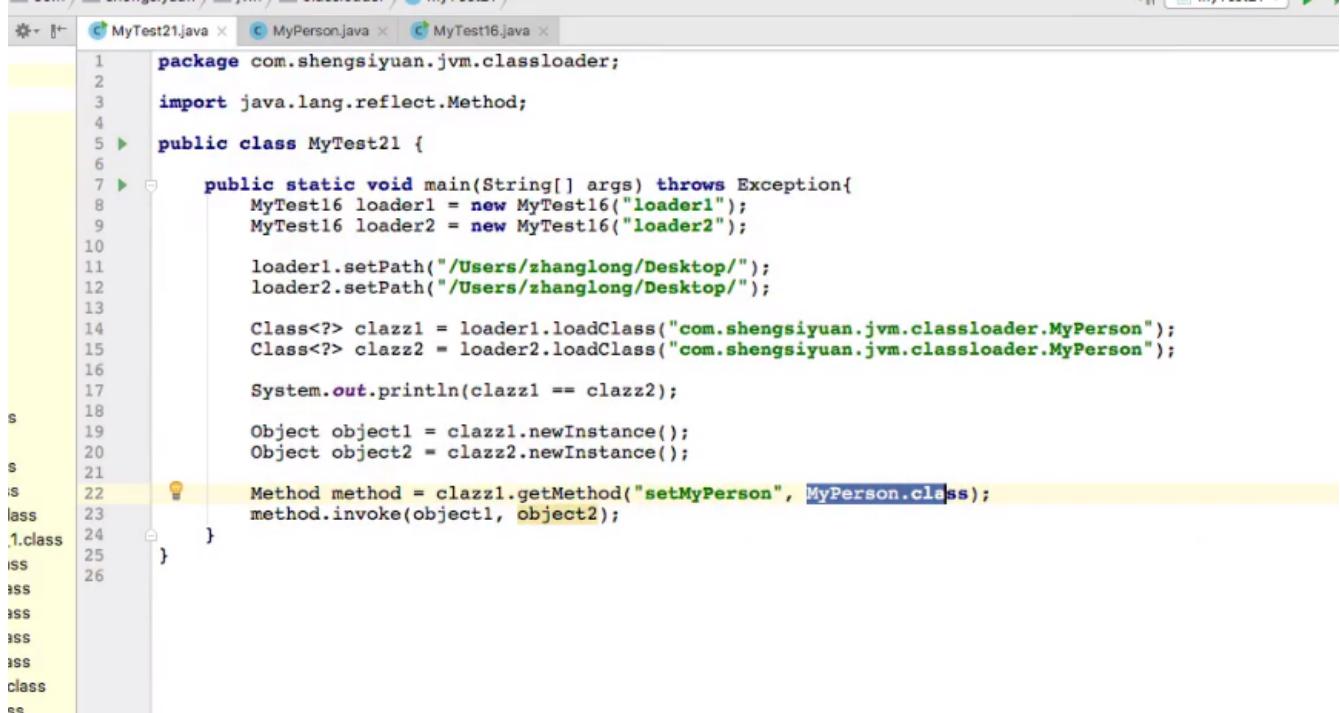
由父加载器加载的类不能看见子加载器加载的类。

如果两个加载器之间没有直接或间接的父子关系，那么它们各自加载的类相互不可见。

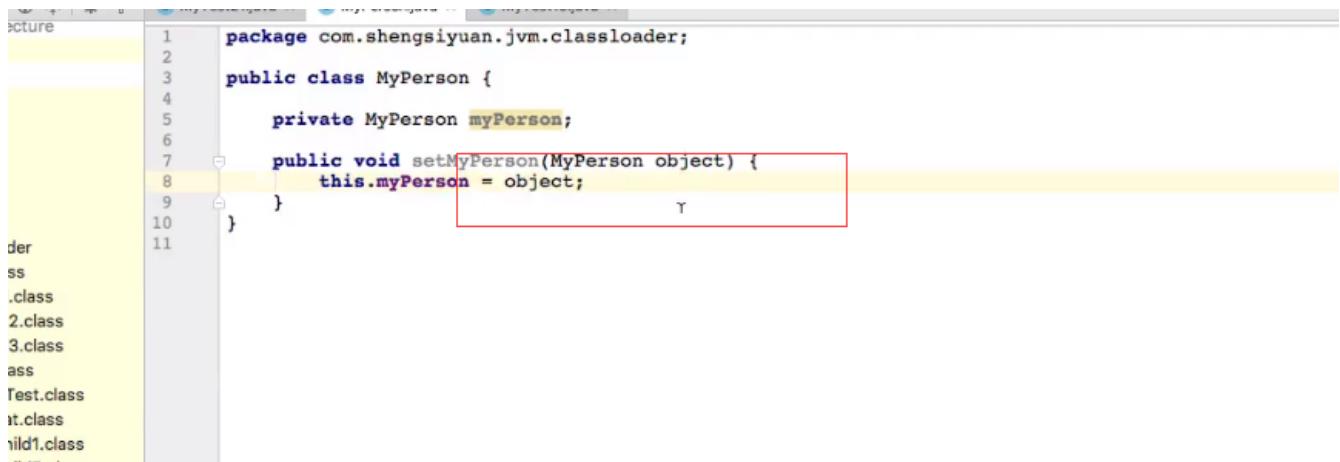
修改：

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

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



```
1 package com.shengsiyuan.jvm.classloader;
2
3 import java.lang.reflect.Method;
4
5 public class MyTest21 {
6
7     public static void main(String[] args) throws Exception{
8         MyTest16 loader1 = new MyTest16("loader1");
9         MyTest16 loader2 = new MyTest16("loader2");
10
11         loader1.setPath("/Users/zhanglong/Desktop/");
12         loader2.setPath("/Users/zhanglong/Desktop/");
13
14         Class<?> clazz1 = loader1.loadClass("com.shengsiyuan.jvm.classloader.MyPerson");
15         Class<?> clazz2 = loader2.loadClass("com.shengsiyuan.jvm.classloader.MyPerson");
16
17         System.out.println(clazz1 == clazz2);
18
19         Object object1 = clazz1.newInstance();
20         Object object2 = clazz2.newInstance();
21
22         Method method = clazz1.getMethod("setMyPerson", MyPerson.class);
23         method.invoke(object1, object2);
24     }
25
26 }
```



```
1 package com.shengsiyuan.jvm.classloader;
2
3 public class MyPerson {
4
5     private MyPerson myPerson;
6
7     public void setMyPerson(MyPerson object) {
8         this.myPerson = object;
9     }
10
11 }
```

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 initializer
va.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：

```

sun.misc.Launcher$AppClassLoader@73d16e93
+ classes jar cvf test.jar com/shengsiyuan/jvm/classloader/MyTest1.class

```

The screenshot shows an IDE interface with a project tree on the left and a code editor on the right.

Project Tree:

- test.jar
- 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
 - MyTest13.java

Code Editor (MyTest22.java):

```

2
3 > public class MyTest22 {
4
5     static {
6         System.out.println("MyTest22 initializer");
7     }
8
9 >     public static void main(String[] args) {
10        System.out.println(MyTest22.class.getClassLoader());
11
12        System.out.println(MyTest1.class.getClassLoader());
13    }
14
15 }

```

Terminal:

```

+ classes java -Djava.ext.dirs= ./ com.shengsiyuan.jvm.classloader.MyTest22
MyTest22 initializer
sun.misc.Launcher$AppClassLoader@2a139a55
sun.misc.Launcher$ExtClassLoader@3d4eac69
+ classes

```

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

The screenshot shows an IDE interface with a project tree on the left and a code editor on the right.

Project Tree:

- 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
 - MyTest13.java

Code Editor (MyTest23.java):

```

1 package com.shengsiyuan.jvm.classloader;
2
3 public class MyTest23 {
4
5     public static void main(String[] args) {
6         System.out.println(System.getProperty("sun.boot.class.path"));
7         System.out.println(System.getProperty("java.ext.dirs"));
8         System.out.println(System.getProperty("java.class.path"));
9     }
10
11 }

```

Terminal:

```

+ classes java com.shengsiyuan.jvm.classloader.MyTest23
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/resources.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/rt.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/sunrsasign.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/jce.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/jsse.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/charsets.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/jfr.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/classes
/Users/zhanglong/Library/Java/Extensions:/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/ext:/Library/Java/Extensions:/Network/Library/Java/Extensions:/System/Library/Java/Extensions:/usr/lib/java
+ classes

```

点 标识当前路径

```

package com.shengsiyuan.jvm.classloader;

/*
在运行期，一个Java类是由该类的完全限定名 (binary name, 二进制名) 和用于加载该类的定义类加载器 (defining loader) 所共同决定的。
如果同样名字 (即相同的完全限定名) 的类是由两个不同的加载器所加载，那么这些类就是不同的，即便.class文件的字节码完全一样，并且从相同的位置加载亦如此。
*/

```

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。？？

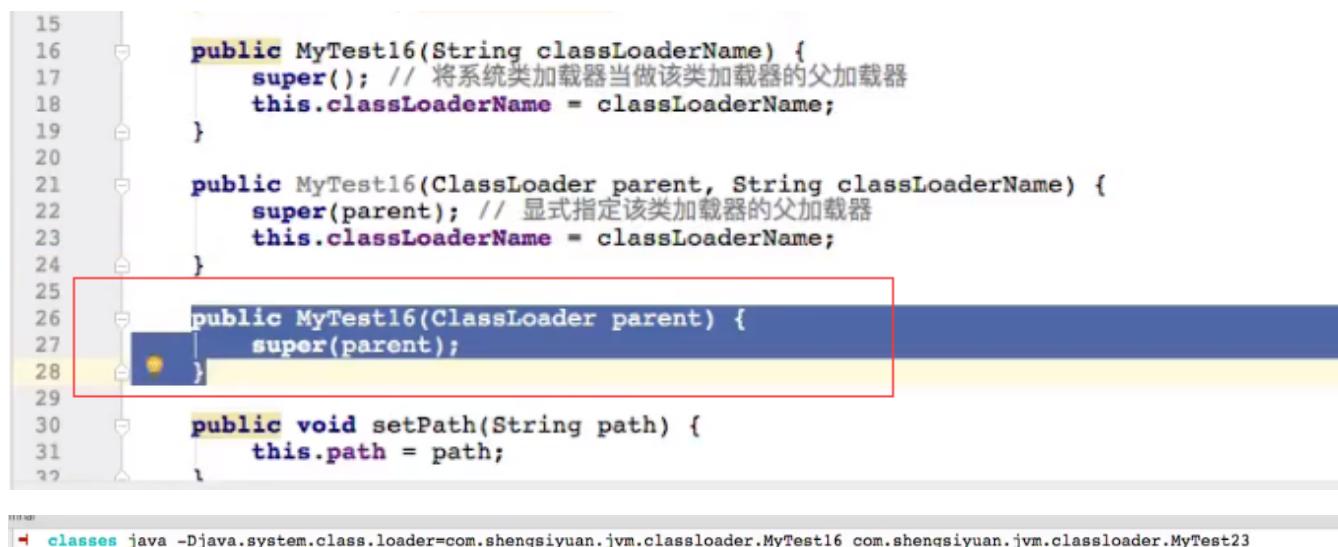
这些类是由启动类加载器加载。

```
/*
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；

```
/Users/zhanglong/Library/Java/Extensions:/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/lib/ext:/Library  
/Library/Java/Extensions:/System/Library/Java/Extensions:/usr/lib/java  
.  
null  
null  
-----  
com.shengsiyuan.jvm.classloader.MyTest16  
sun.misc.Launcher$AppClassLoader@18b4aac2  
sun.misc.Launcher$AppClassLoader@18b4aac2  
com.shengsiyuan.jvm.classloader.MyTest16@4e25154f  
└ classes
```

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。

```
est24.java x Thread.java x MyTest23 ▾ ▶ 🔍
* ClassLoader context of the parent Thread. The context ClassLoader of the
* primordial thread is typically set to the class loader used to load the
* application.
*
* <p>If a security manager is present, and the invoker's class loader is not
* {@code null} and is not the same as or an ancestor of the context class
* loader, then this method invokes the security manager's {@link
* SecurityManager#checkPermission(java.security.Permission) checkPermission}
* method with a {@link RuntimePermission RuntimePermission}{@code
* ("getClassLoader")} permission to verify that retrieval of the context
* class loader is permitted.
*
* @return the context ClassLoader for this Thread, or {@code null}
* indicating the system class loader (or, failing that, the
* bootstrap class loader)
*
* @throws SecurityException
*         if the current thread cannot get the context ClassLoader
*
* @since 1.2
*/
@CallerSensitive
public ClassLoader getContextClassLoader() {
    if (contextClassLoader == null)
        return null;
    SecurityManager sm = System.getSecurityManager();
    if (sm != null) {
        ClassLoader.checkClassLoaderPermission(contextClassLoader,
            Reflection.getCallerClass());
    }
    return contextClassLoader;
}
```

```

package com.shengsiyuan.jvm.classloader;
public class MyTest24 {
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getContextClassLoader());
        System.out.println(Thread.currentThread().getClass().getClassLoader());
    }
}

```

Run MyTest24

```

/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
sun.misc.Launcher$AppClassLoader@18b4aac2
null
objc[1119]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (
Process finished with exit code 0

```

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 c)分别用来获取和设置上下文类加载器。

如果没有通过setContextClassLoader(ClassLoader c1)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

```
1 package com.shengsiyuan.jvm.classloader;
2
3 public class MyTest25 implements Runnable{
4
5     private Thread thread;
6
7     public MyTest25() {
8         thread = new Thread(this);
9         thread.start();
10    }
11
12    @Override
13    public void run() {
14        ClassLoader classLoader = this.thread.getContextClassLoader();
15
16        this.thread.setContextClassLoader(classLoader);
17
18        System.out.println("Class: " + classLoader.getClass());
19        System.out.println("Parent: " + classLoader.getParent().getClass());
20    }
21
22    public static void main(String[] args) {
23        new MyTest25();
24    }
25}
26
```



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。|
```

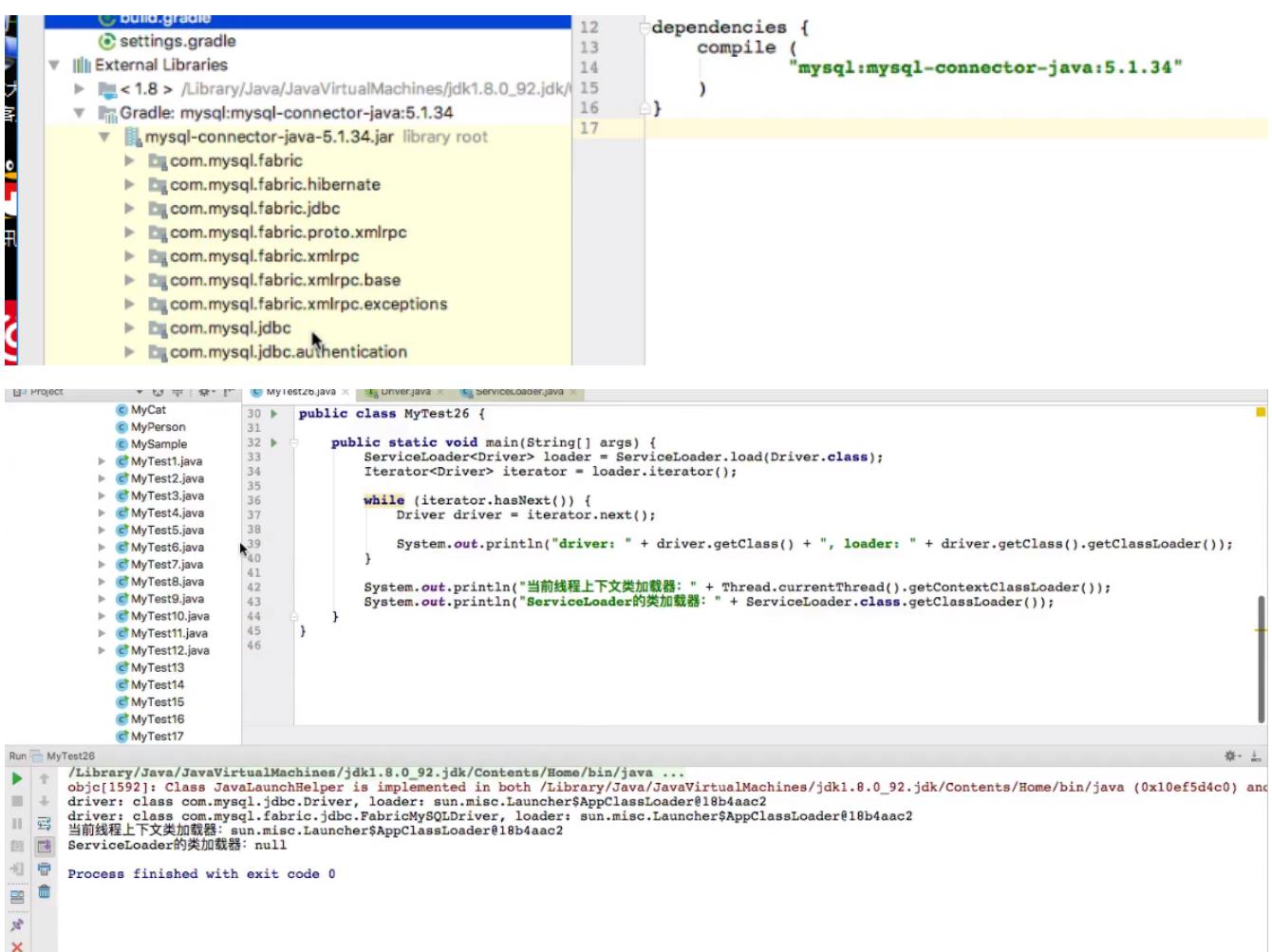
29 ServiceLoader在SPI中的重要作用分析(有时间再看一遍)

```

5
6     sourceCompatibility = 1.8
7
8     repositories {
9         mavenCentral()
10    }
11
12    dependencies {
13        compile(
14            "mysql:mysql-connector-java:5.1.34"
15        )
16    }
17

```

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。

The screenshot shows a Java documentation page for the `ServiceLoader` class. It includes sections on provider classes, configuration files, and security context. The code snippet at the top shows the `ServiceLoader` interface definition:

```
java.util
public final class ServiceLoader<S>
extends Object
implements Iterable<S>
```

Below the code, there is a note about a simple service-provider loading facility. It explains that 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 typically 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 or 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 particular 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 are ignored. The file must be encoded in UTF-8.

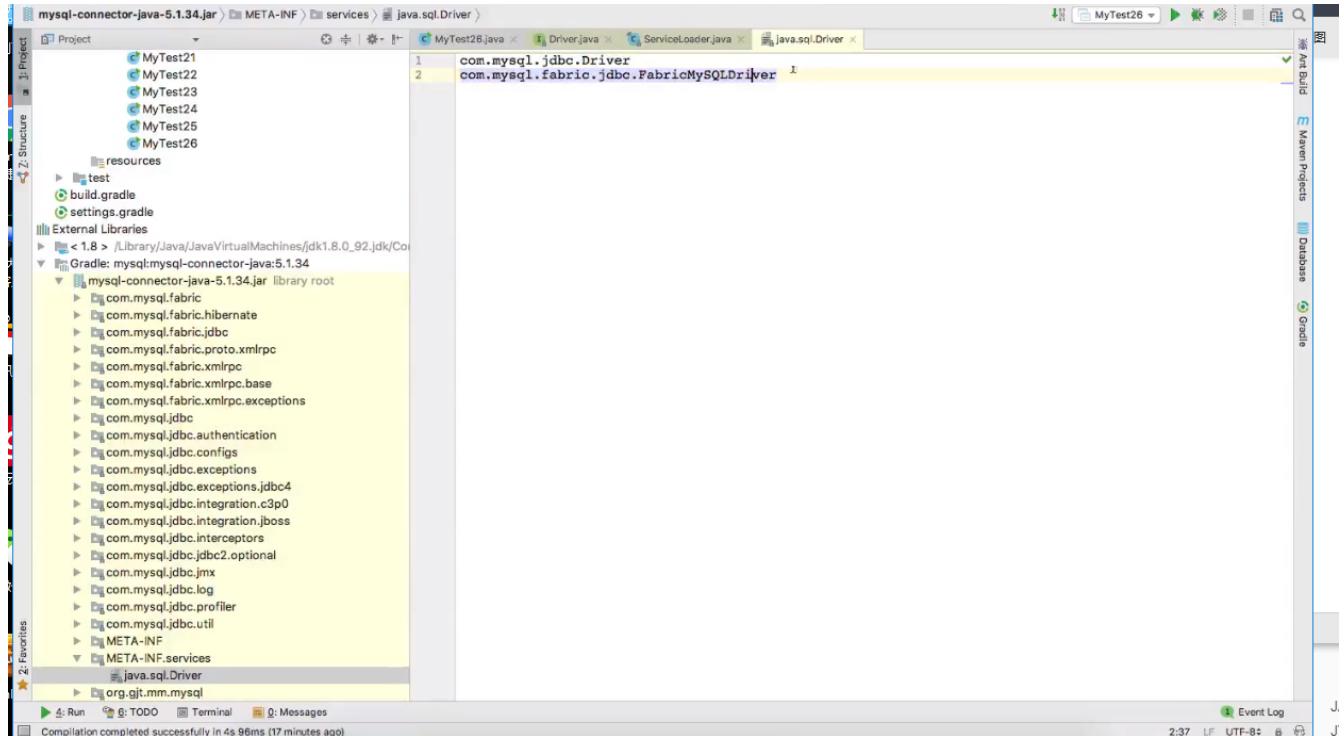
If a particular concrete provider class is named in more than one configuration file, or is named in the same configuration file more than once, then the duplicates are ignored. The configuration file naming a particular provider need not be in the same jar file or other distribution unit as the provider 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<InstantiatedProvider> instantiatedProviders = new ArrayList<>();
```

-

-

<

<https://javadoc.allmant.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。

~~classname 第二个参数boolean 表示是否去初始化。~~

The screenshot shows an IDE interface with a code editor containing Java code. The code is a test for class loaders, specifically focusing on the ServiceLoader mechanism. It imports java.sql.Driver, java.util.Iterator, and java.util.ServiceLoader. The main method creates a ServiceLoader<Driver> and iterates over its elements to print their class names and class loaders. It also prints the current thread's context class loader and the class loader of the ServiceLoader class itself.

```
25 import java.sql.Driver;
26 import java.util.Iterator;
27 import java.util.ServiceLoader;
28
29 public class MyTest26 {
30     public static void main(String[] args) {
31         Thread.currentThread().setContextClassLoader(MyTest26.class.getClassLoader().getParent());
32         ServiceLoader<Driver> loader = ServiceLoader.load(Driver.class);
33         Iterator<Driver> iterator = loader.iterator();
34
35         while (iterator.hasNext()) {
36             Driver driver = iterator.next();
37
38             System.out.println("driver: " + driver.getClass() + ", loader: " + driver.getClass().getClassLoader());
39
40         }
41
42         System.out.println("当前线程上下文类加载器: " + Thread.currentThread().getContextClassLoader());
43         System.out.println("ServiceLoader的类加载器: " + ServiceLoader.class.getClassLoader());
44
45     }
46 }
```

Below the code editor, the IDE's toolbars and panels are visible, including a run configuration for "MyTest26". The output window shows the execution results:

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
objc[1664]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (0x10cfad4c0) and /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (0x10cfad4c0)
当前线程上下文类加载器: sun.misc.Launcher$ExtClassLoader@60e53b93
ServiceLoader的类加载器: null
Process finished with exit code 0
```

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:

类的验证

- 类的验证的内容
 - -类文件的结构检查
 - -语义检查
 - -字节码验证
 - -二进制兼容性的验证

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：

```
public void main(String[] args) throws IOException {
    ClassLoader classLoader = Thread.currentThread()
        .getContextClassLoader();
    String resourceName = "java/lang/String.class";

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

类加载器的父亲委托机制

需要指出的是，加载器之间的父子关系实际上指的是加载器对象之间的包装关系，而不是类之间的继承关系。一对父子加载器可能是同一个加载器类的两个实例，也可能不是。在子加载器对象中包装了一个父加载器对象。例如以下 loader1 和 loader2 都是 MyClassLoader 类的实例，并且 loader2 包装了 loader1，loader1 是 loader2 的父加载器。

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
-

```
E:\HelloGradle\build\classes\java\main>javap -verbose com.jlu.bytecode.MyTest1
Classfile /E:/HelloGradle/build/classes/java/main/com/jlu bytecode/MyTest1.class
Last modified 2019年5月30日; size 479 bytes
MD5 checksum 2b0c21870fe1c25c00eaee02b451b3dc
Compiled from "MyTest1.java"
public class com.jlu.bytecode.MyTest1
    minor version: 0
    major version: 52
    flags: (0x0021) ACC_PUBLIC, ACC_SUPER
    this_class: #3                      // com/jlu bytecode/MyTest1
    super_class: #4                      // java/lang/Object
    interfaces: 0, fields: 1, methods: 3, attributes: 1
Constant pool:
#1 = Methodref      #4. #20          // java/lang/Object."<init>":()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           <init>
#8 = Utf8           ()V
#9 = Utf8           Code
#10 = Utf8          LineNumberTable
```

Terminal Build Run Debug TODO
IDE and Plugin Updates: IntelliJ IDEA is ready to update. (today 11:30)

常量池也可以有变量的。

字面量

Java字节码

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

u88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

```
memo.txt
1. 使用javap -verbose命令分析一个字节码文件时，将会分析该字节码文件的魔数、版本号、常量池、类信息、类的构造方法、类中的方法信息、类变量与成员变量等信息。
2. 魔数：所有的.class字节码文件的前4个字节都是魔数，魔数值为固定值：0xCAFEBAE。
3. 魔数之后的4个字节为版本信息，前两个字节表示minor version（次版本号），后两个字节表示major version（主版本号）。这里的版本号为00 00 00 34，换算成十进制，表示次版本号为0，主版本号为52。所以，该文件的版本号为：1.8.0。可以通过java -version命令来验证这一点。
4. 常量池（constant pool）：紧接着主版本号之后的就是常量池入口。一个Java类中定义的很多信息都是由常量池来维护和描述的，可以将常量池看作是Class文件的资源仓库，比如说Java类中定义的方法与变量信息，都是存储在常量池中。常量池中主要存储两类常量：字面量与符号引用。字面量如文本字符串，Java中声明为final的常量值等，而符号引用如类和接口的全局限定名，字段的名称和描述符，方法的名称和描述符等。
5. 常量池的总体结构：Java类所对应的常量池主要由常量池数量与常量池数组（常量表）这两部分共同构成。常量池数量紧跟在主版本号后面，占据2个字节；常量池数组则紧跟在常量池数量之后。常量池数组与一般的数组不同的是，常量池数组中不同的元素的类型、结构都是不同的，长度当然也就不同；但是，每一种元素的第一个数据都是一个u1类型，该字节是个标志位，占据1个字节。JVM在解析常量池时，会根据这个u1类型来获取元素的具体类型。值得注意的是，常量池数组中元素的个数 = 常量池数 - 1（其中0暂时不使用），目的是满足某些常量池索引值的数据在特定情况下需要表达『不引用任何一个常量池』的含义；根本原因在于，索引为0也是一个常量（保留常量），只不过它不位于常量表中，这个常量就对应null值；所以，常量池的索引从1而非0开始。
6. 在JVM规范中，每个变量/字段都有描述信息，描述信息主要的作用是描述字段的数据类型、方法的参数列表（包括数量、类型与顺序）与返回值。根据描述符规则，基本数据类型和代表无返回值的void类型都用一个大写字符来表示，对象类型则使用字符L加对象的全限定名称来表示。为了压缩字节码文件的体积，对于基本数据类型，JVM都只使用一个大写字母来表示，如下所示：B - byte, C - char, D - double, F - float, I - int, J - long, S - short, Z - boolean, V - void, L - 对象类型，如Ljava/lang/String;
```

7. 对于数组类型来说，每一个维度使用一个前缀的[来表示，如int[]被记录为[I, String[][]被记录为[[Ljava/lang/String;
8. 用描述符描述方法时，按照先参数列表，后返回值的顺序来描述。参数列表按照参数的严格顺序放在一组()之内，如方法：String getRealnameByIdAndNickname(int

#1 = Methodref #4.#20 // java/lang/Object."<":()V #2 = Fieldref #3.#21 // com/jlu/bytocode/MyTest1.a:I #3 = Class #22 // com/jlu/bytocode/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/bytocode/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/bytocode/MyTest1 #23 = Utf8 java/lang/Object

#表示引用或者索引项

java字节码结构

4个字节	Magic Number	魔数，值为0xCAFEBAE，Java创始人James Gosling制定
2 + 2个字节	Version	包括minor_version和major_version, minor_version: 1.1(45), 1.2(46), 1.3(47), 1.4(48), 1.5(49), 1.6 (50), 1.7(51)。指令集多年不变，但是版本号每次发布都变化。
2 + n个字节	Constant Pool	包括字符串常量、数值常量等
2个字节	Access Flags	
2个字节	This Class Name	
2个字节	Super Class Name	
2+n个字节	Interfaces	
2+n个字节	Fields	
2+n个字节	Methods	
2+n个字节	Attributes	

~~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

类索引、父类索引与接口索引

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

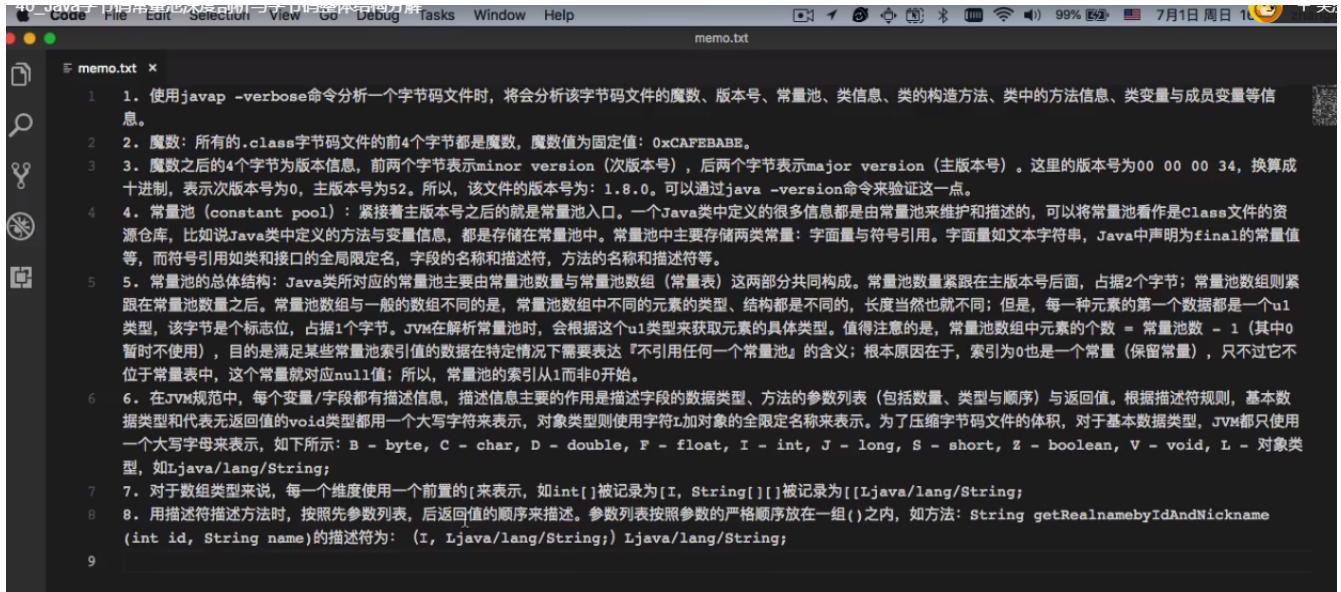
变量表是由自己的结构的。

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



```
memo.txt
1. 使用javap -verbose命令分析一个字节码文件时，将会分析该字节码文件的魔数、版本号、常量池、类信息、类的构造方法、类中的方法信息、类变量与成员变量等信息。
2. 魔数：所有的.class字节码文件的前4个字节都是魔数，魔数值为固定值：0xCAFEBABE。
3. 魔数之后的4个字节为版本信息，前两个字节表示minor version（次版本号），后两个字节表示major version（主版本号）。这里的版本号为00 00 00 34，换算成十进制，表示次版本号为0，主版本号为52。所以，该文件的版本号为：1.8.0。可以通过java -version命令来验证这一点。
4. 常量池（constant pool）：紧接着主版本号之后的就是常量池入口。一个Java类中定义的很多信息都是由常量池来维护和描述的，可以将常量池看作是Class文件的资源仓库，比如说Java类中定义的方法与变量信息，都是存储在常量池中。常量池中主要存储两类常量：字面量与符号引用。字面量如文本字符串，Java中声明为final的常量值等，而符号引用如类和接口的全局限定名，字段的名称和描述符，方法的名称和描述符等。
5. 常量池的总体结构：Java类所对应的常量池主要由常量池数量与常量池数组（常量表）这两部分共同构成。常量池数量紧跟在主版本号后面，占据2个字节；常量池数组则紧跟在常量池数量之后。常量池数组与一般的数组不同的是，常量池数组中不同的元素的类型、结构都是不同的，长度当然也就不同；但是，每一种元素的第一个数据都是一个u1类型，该字节是个标志位，占据1个字节。JVM在解析常量池时，会根据这个u1类型来获取元素的具体类型。值得注意的是，常量池数组中元素的个数 = 常量池数 - 1（其中0暂时不使用），目的是满足某些常量池索引值的数据在特定情况下需要表达『不引用任何一个常量池』的含义；根本原因在于，索引为0也是一个常量（保留常量），只不过它不位于常量表中，这个常量就对应null值；所以，常量池的索引从1而非0开始。
6. 在JVM规范中，每个变量/字段都有描述信息，描述信息主要的作用是描述字段的数据类型、方法的参数列表（包括数量、类型与顺序）与返回值。根据描述符规则，基本数据类型和代表无返回值的void类型都用一个大写字母来表示，对象类型则使用字符L加对象的全限定名称来表示。为了压缩字节码文件的体积，对于基本数据类型，JVM都只使用一个大写字母来表示，如下所示：B - byte, C - char, D - double, F - float, I - int, J - long, S - short, Z - boolean, V - void, L - 对象类型，如Ljava/lang/String;
7. 对于数组类型来说，每一个维度使用一个前置的[来表示，如int[]被记录为[I, String[][]]被记录为[[Ljava/lang/String;
8. 用描述符描述方法时，按照先参数列表，后返回值的顺序来描述。参数列表按照参数的严格顺序放在一组()之内，如方法：String getRealnameByIdAndNickname(int id, String name)的描述符为：(I, Ljava/lang/String;) Ljava/lang/String;
```

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      constant_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 <i>invokespecial</i> 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 <i>invokespecial</i> 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];
}
```

Code结构

- attribute_length表示attribute所包含的字节数，不包含attribute_name_index和attribute_length字段。
- max_stack表示这个方法运行的任何时刻所能达到的操作数栈的最大深度
- max_locals表示方法执行期间创建的局部变量的数目，包含用来表示传入的参数的局部变量

字节码查看工具 classlib

<https://github.com/ingekegel/jclasslib/>

46 synchronized 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

```

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/bytocode/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/bytocode/MyTest2;
0       6     1  x      I

```

moniterenter
moniterexit

虽然在这个方法中没有看到moniterenter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 1
   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  sun
```

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。

Code结构

- **attribute_length**表示**attribute**所包含的字节数，不包含**attribute_name_index**和**attribute_length**字段。
- **max_stack**表示这个方法运行的任何时刻所能达到的操作数栈的最大深度
- **max_locals**表示方法执行期间创建的局部变量的数目，包含用来表示传入的参数的局部变量

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Structure:** Shows a project named "jvm_lecture" with files like .gradle, .idea, build.gradle, settings.gradle, and src/main/java/com/shengsiyuan/jvm/bytcode/MyTest3.java.
- Code Editor:** Displays the Java code for MyTest3.java, specifically the test() method which handles file input and server socket operations.
- Terminal:** Shows the generated byte code for the test() method. The code includes instructions for creating FileInputStream and ServerSocket objects, performing I/O operations, and handling exceptions. A note in red highlights that there are three catch blocks but only one exception is thrown.

```

public void test() {
    try {
        InputStream is = new FileInputStream("test.txt");
        ServerSocket serverSocket = new ServerSocket(9999);
        serverSocket.accept();
    } catch (FileNotFoundException ex) {
    } catch (IOException ex) {
    } catch (Exception ex) {
    } finally {
        System.out.println("finally");
    }
}

```

```

0      5      0  this  Lcom/shengsiyuan/jvm/bytcode/MyTest3;
          locals=4
public void test();           this is serverSocket 还有一个异常对象,
descriptor: ()V             虽然有三个catch, 但是只会产生一个异常
flags: ACC_PUBLIC
Code:
  stack=3, locals=4, args_size=1
  0: new           #2           // class java/io/FileInputStream
  3: dup
  4: ldc           #3           // String test.txt
  6: invokevirtual #4           // Method java/io/FileInputStream."<init>":(Ljava/lang/String;)V
  9: astore_1
 10: new            #5           // class java/net/ServerSocket
 13: dup
 14: sipush         9999        args_size =1
                                this 参数

```

stack=3, 对应locals的个数。减去this

Code结构

- **code_length** 表示该方法所包含的字节码的字节数以及具体的指令码
- 具体字节码即是该方法被调用时，虚拟机所执行的字节码
- **exception_table**, 这里存放的是处理异常的信息
- 每个**exception_table**表项由**start_pc**, **end_pc**, **handler_pc**, **catch_type**组成

50 通过字节码分析Java异常的处理机制

The screenshot shows the JD-GUI interface with two panes. The left pane displays the Java code:MyTest3.java
public class MyTest3 {
 public void test() {
 try {
 InputStream is = new FileInputStream("test.txt");
 ServerSocket serverSocket = new ServerSocket(9999);
 serverSocket.accept();
 } catch (FileNotFoundException ex) {
 } catch (IOException ex) {
 } catch (Exception ex) {
 } finally {
 System.out.println("finally");
 }
 }
} The right pane shows the corresponding bytecode for the `test()` method. A red box highlights the exception handling part of the bytecode table.

Nr.	Start PC	End PC	Handler PC	Cat
0	0	26	37	cp_info #11 java/io/FileNotFoundExcep
1	0	26	49	cp_info #12 java/io/IOException
2	0	26	61	cp_info #13 java/lang/Exception
3	0	26	73	cp_info #0 any

This screenshot shows the full bytecode for the `MyTest3` class. A red box highlights the entire exception handling section of the bytecode table.

Nr.	Start PC	End PC	Handler PC	Cat
0	0	26	37	cp_info #11 java/io/FileNotFoundExcep
1	0	26	49	cp_info #12 java/io/IOException
2	0	26	61	cp_info #13 java/lang/Exception
3	0	26	73	cp_info #0 any

This screenshot shows the exception table for the `MyTest3` class. A red arrow points to the last row of the table, which corresponds to the `any` catch-all handler.

Nr.	Start PC	End PC	Handler PC	Cat
0	0	26	37	cp_info #11 java/io/FileNotFoundExcep
1	0	26	49	cp_info #12 java/io/IOException
2	0	26	61	cp_info #13 java/lang/Exception
3	0	26	73	cp_info #0 any

处理所有异常

```

/*
 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。



The screenshot shows a table titled "LineNumberTable" with three columns: "Nr.", "Start PC", and "Line Number". A red arrow points from the header "Line Number" to the column. Red annotations on the right side of the table identify the columns: "java" for "Start PC", "源码" (source code) for "Line Number", and "行号" (line number) for "Nr.". The table contains 16 rows of data, with the last row (Nr. 15) highlighted in blue. A red arrow points from the bottom-left of the table to the value "84" in the "Start PC" column of the last row, with the label "字节码行号" (bytecode line number) written below it.

Nr.	Start PC	Line Number
0	0	13
1	10	15
2	21	16
3	26	24
4	34	25
5	37	17
6	38	24
7	46	25
8	49	19
9	50	24
10	58	25
11	61	21
12	62	24
13	70	25
14	73	24
15	84	26

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

MyTest3.java x C:\ServerSocket.java x C:\FileInputStream.java x

```

/*
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，方法参数的数量总比参数的数量多一个（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。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引用。
*/
/*
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。
*/
public class MyTest3 {
    public void test() throws IOException, FileNotFoundException {
        try {
            InputStream is = new FileInputStream("test.txt");
            ServerSocket serverSocket = new ServerSocket(9999);
            serverSocket.accept();
        } catch (FileNotFoundException ex) {
        } catch (IOException ex) {
        } catch (Exception ex) {
        } finally {
            System.out.println("finally");
        }
    }
}

```

jclasslib MyTest3.class

General Information

- Constant Pool
- Interfaces
- Fields
- Methods
 - [0] <init>
 - [1] test
 - [0] Code
 - [1] Exceptions
- Attributes

Specific info

Bytecode	Exception
0 new #2 <ja	
3 dup	
4 ldc #3 <te	
6 invokespecial	
9 astore_1	
10 new #5 <ja	
13 dup	
14 pushpop 999	
17 invokespecial	
20 astore_2	
21 aload_2	
22 invokevirtual	
25 pop	
26 getstatic	
29 ldc #9 <fi	
31 invokevirtual	
34 goto #4 +	
37 astore_1	
38 getstatic	
41 ldc #9 <fi	
43 invokevirtual	
46 goto #4 +	
49 astore_1	
50 getstatic	
53 ldc #9 <fi	

MyTest3.java x C:\ServerSocket.java x C:\FileInputStream.java x

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引用。

异常的处理方式：

统一采用异常表的方式来对异常进行处理。

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。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。

```

public class MyTest3 {
    public void test() throws IOException, FileNotFoundException {
        try {
            InputStream is = new FileInputStream("test.txt");
            ServerSocket serverSocket = new ServerSocket(9999);
            serverSocket.accept();
        } catch (FileNotFoundException ex) {
        } catch (IOException ex) {
        } catch (Exception ex) {
        } finally {
            System.out.println("finally");
        }
    }
}

```

jclasslib MyTest3.class

General Information

- Constant Pool
- Interfaces
- Fields
- Methods
 - [0] <init>
 - [1] test
 - [0] Code
 - [1] Exceptions
- Attributes

Specific info

Exception	Verbose
0 cp_info #12	java/io/IOException
1 cp_info #11	java/io/FileNotFoundException

MyTest3.java x C:\ServerSocket.java x C:\FileInputStream.java x

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；接下来说一下异常的处理方式：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引用。

异常的处理方式：

统一采用异常表的方式来对异常进行处理。

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。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。

```

public class MyTest3 {
    public void test() throws IOException, FileNotFoundException, NullPointerException {
        try {
            InputStream is = new FileInputStream("test.txt");
            ServerSocket serverSocket = new ServerSocket(9999);
            serverSocket.accept();
        } catch (FileNotFoundException ex) {
        } catch (IOException ex) {
        } catch (NullPointerException ex) {
        } finally {
            System.out.println("finally");
        }
    }
}

```

jclasslib MyTest3.class

General Information

- Constant Pool
- Interfaces
- Fields
- Methods
 - [0] <init>
 - [1] test
 - [0] Code
 - [1] Exceptions
- Attributes

Specific info

Nr.	Exception	Verbose
0	cp_info #12	java/io/IOException
1	cp_info #11	java/io/FileNotFoundException
2	cp_info #34	java/lang/NullPointerException

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。

```
2
3  /*
4   * 栈帧 (stack frame)
5   * 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。
6   * 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。
7
8
9
10 */
11
```

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）~~

```
/*
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。

*/

5.2 方法重载与invokevirtual字节码指令的关系

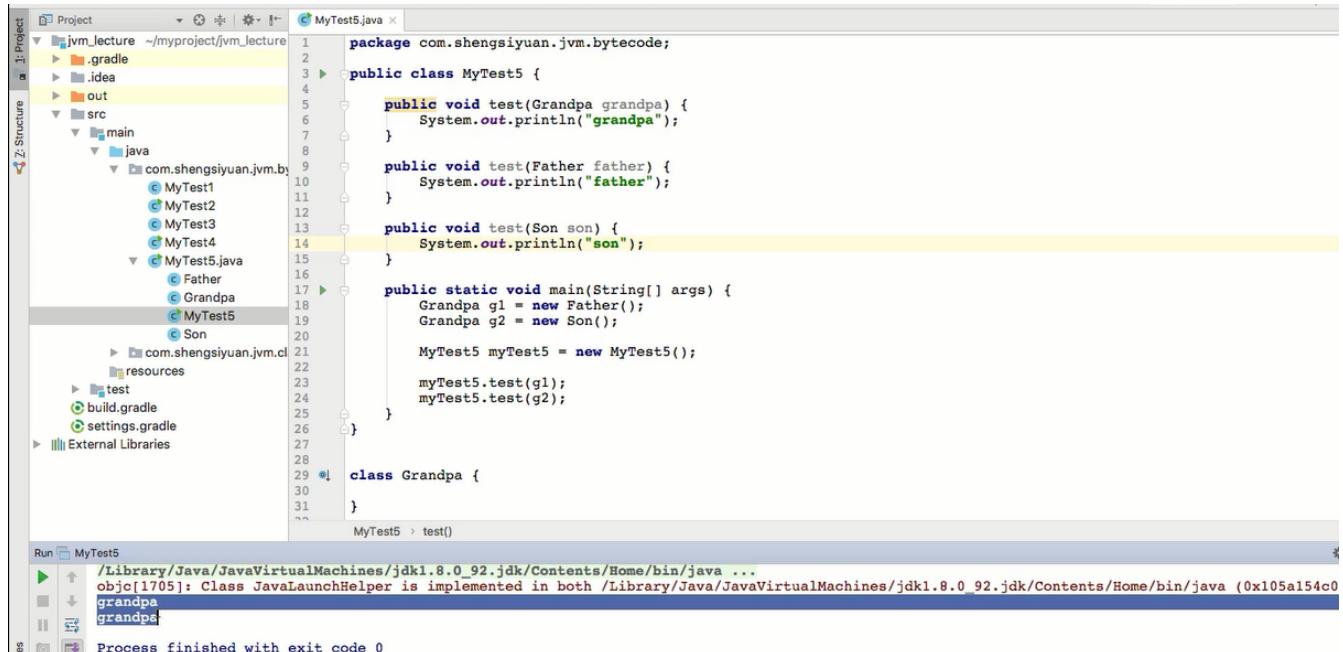
```
/*
1. invokeinterface: 调用接口中的方法，实际上是在运行期决定的，决定到底调用实现该接口的哪个对象的特定方法。
2. invokespecial: 调用静态方法。
3. invokespecial: 调用自己的私有方法、构造方法(<init>)以及父类的方法。
4. invokevirtual: 调用虚方法，运行期动态查找的过程。
5. invokedynamic: 动态调用方法。
*/
```

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。

```
*/
```

```
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

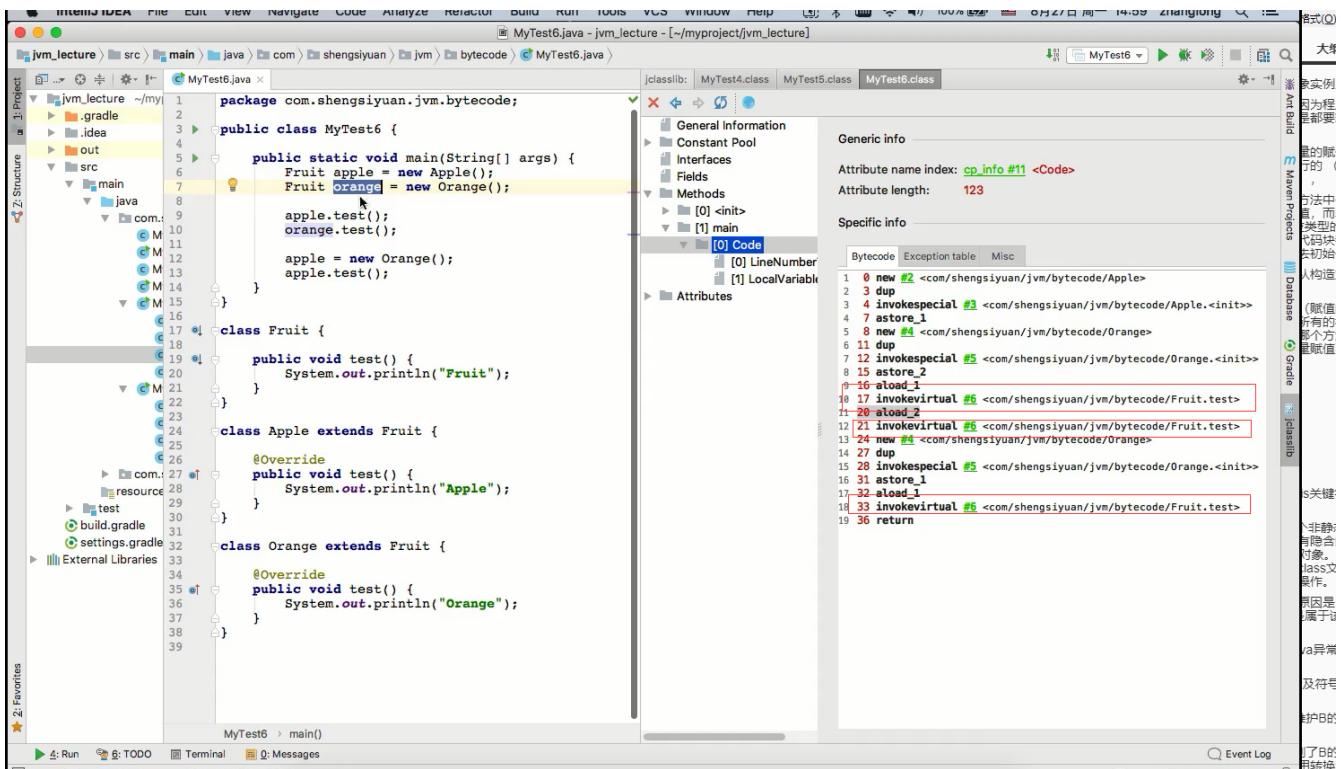
重载是静态的概念，而重写是动态的概念。

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 x
1 package com.shengsiyuan.jvm.bytecode;
2
3 public class MyTest6 {
4
5     public static void main(String[] args) {
6         Fruit apple = new Apple();
7         Fruit orange = new Orange();
8
9         apple.test();
10        orange.test();
11
12        apple = new Orange();
13        apple.test();
14    }
15
16    class Fruit {
17
18        public void test() {
19            System.out.println("Fruit");
20        }
21    }
22
23    class Apple extends Fruit {
24
25        @Override
26        public void test() {
27            System.out.println("Apple");
28        }
29    }
30
31    class Orange extends Fruit {
32
33        @Override
34        public void test() {
35            System.out.println("Orange");
36        }
37    }
38
39}
```

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



invokeVirtual 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

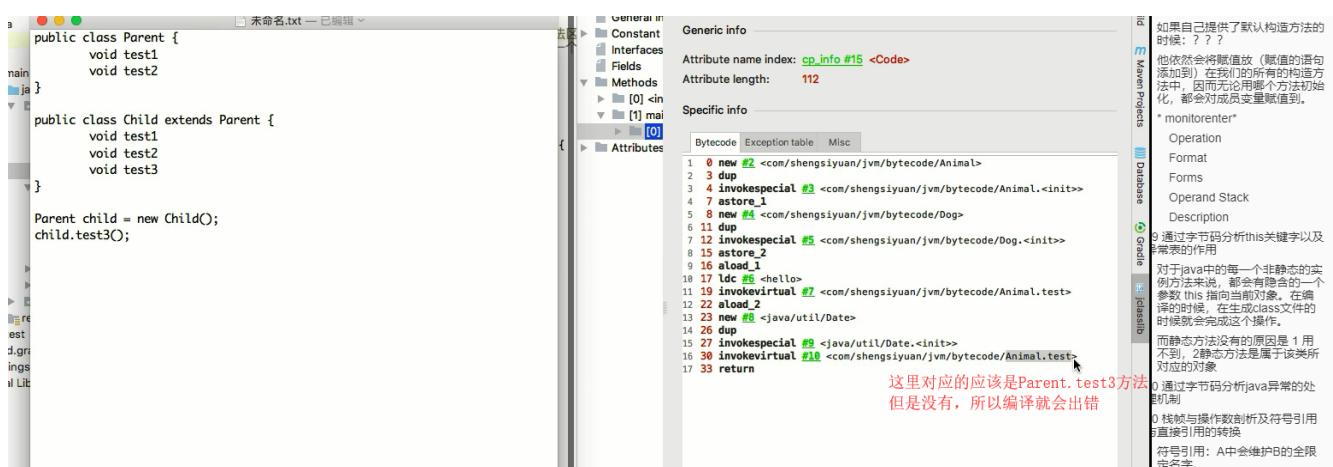
在apple类中找到与test方法完全一致的方法，那么就去执行（确定一个方法的方式 方法名+方法的描述符即方法的参数和返回值）

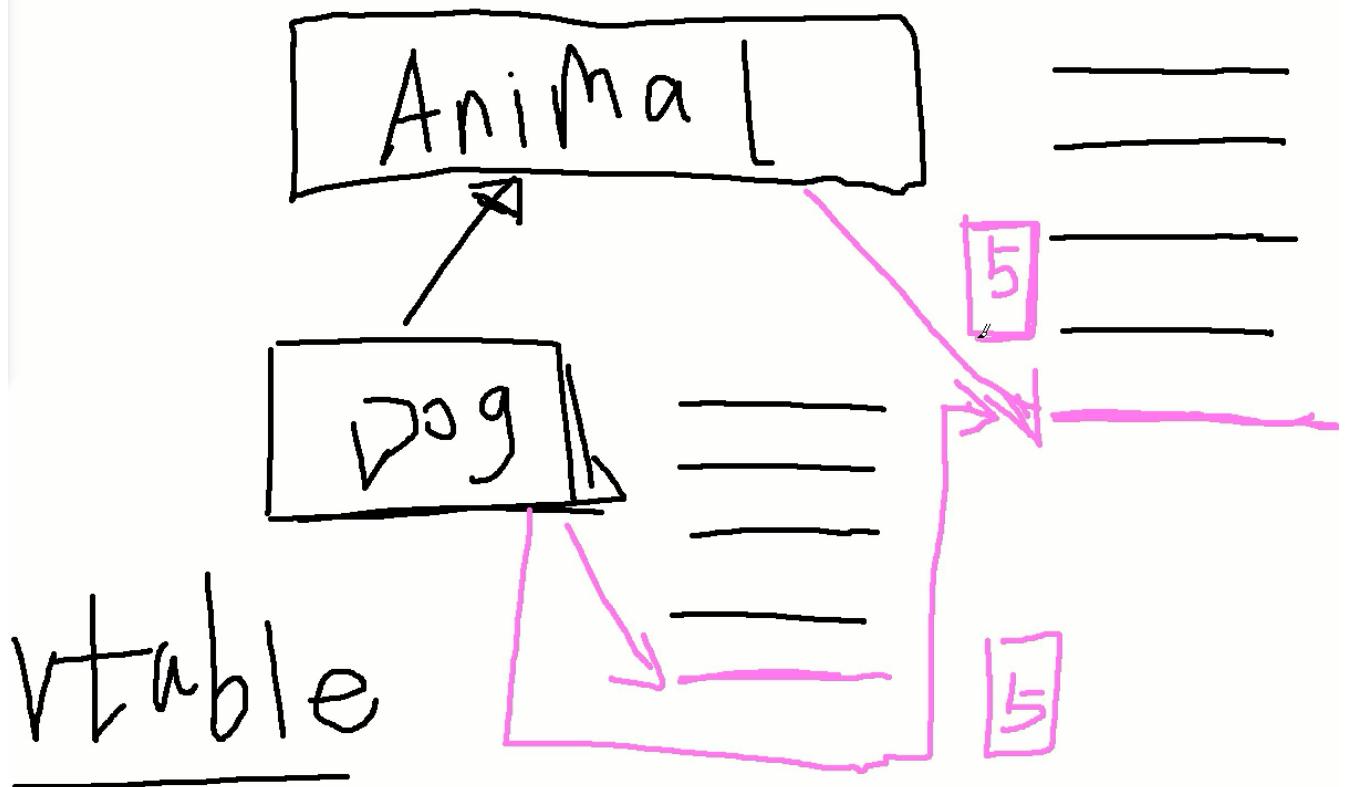
否则，就去找父类的方法，

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

6.5 基于栈的指令集与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。

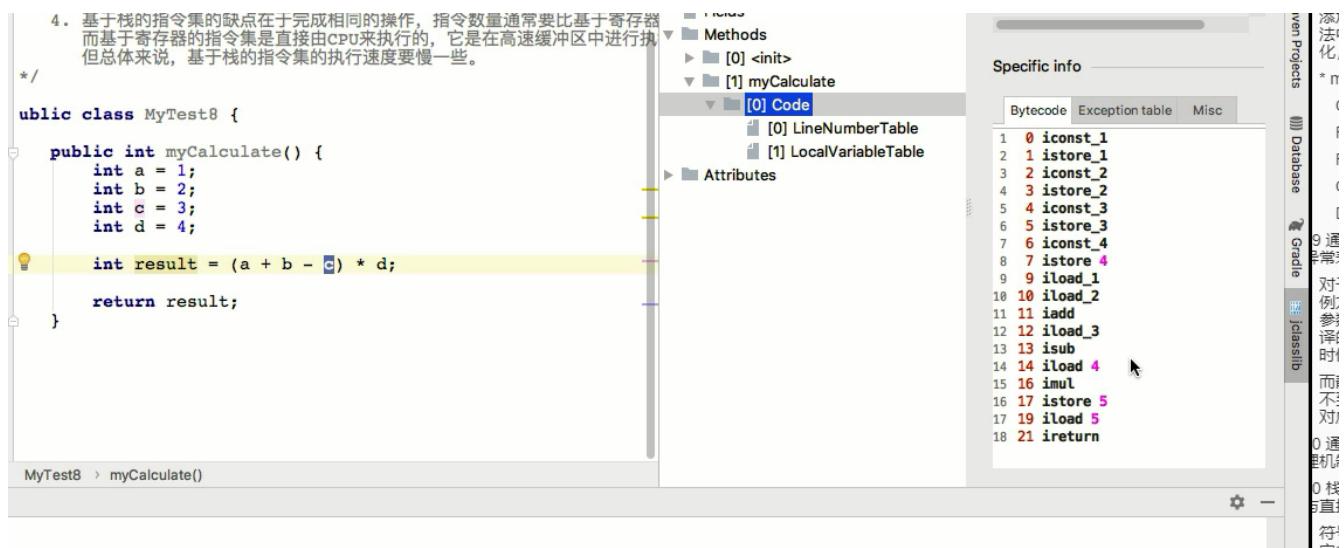
基于栈的指令集与基于寄存器的指令集之间的关系：

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

```
jvm > bytecode > Subject  
Subject.java  
1 package com.shengsiyuan.jvm.bytecode;  
2  
3 public interface Subject {  
4  
5     void request();  
6 }  
7
```

```
Subject.java > RealSubject.java  
RealSubject.java  
1 package com.shengsiyuan.jvm.bytecode;  
2  
3 public class RealSubject implements Subject {  
4  
5     @Override  
6     public void request() {  
7         System.out.println("From real subject");  
8     }  
9 }  
10
```

```
1 package com.shengsiyuan.jvm.bytecode;                                动态代理
2
3 import java.lang.reflect.InvocationHandler;
4 import java.lang.reflect.Method;
5
6 public class DynamicSubject implements InvocationHandler {
7
8     private Object sub;          代理的对象
9
10    public DynamicSubject(Object obj) {
11        this.sub = obj;
12    }
13
14    @Override
15    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
16        System.out.println("before calling: " + method);
17
18        method.invoke(this.sub, args);
19
20        System.out.println("after calling: " + method);
21
22        return null;
23    }
24}
25
```

```
1 package com.shengsiyuan.jvm.bytecode;
2
3 import java.lang.reflect.InvocationHandler;
4 import java.lang.reflect.Proxy;
5
6 public class Client {
7
8     public static void main(String[] args) {
9         RealSubject rs = new RealSubject();
10        InvocationHandler ds = new DynamicSubject(rs);
11        Class<?> cls = rs.getClass();
12
13        Subject subject = (Subject) Proxy.
14            newProxyInstance(cls.getClassLoader(), cls.getInterfaces(), ds);
15
16        subject.request();
17    }
18}
19
```

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Structure:** The project is named "jvm_lecture". The "src/main/java/com/shengsiyuan/jvm/bytocode" package contains several files: Client.java, Subject.java, RealSubject.java, DynamicSubject.java, and various MyTest1 through MyTest8 files.
- Code Editor:** The Client.java file is open, showing the following code snippet:


```

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();
        System.out.println(subject.getClass());
    }
}

```
- Run Tab:** The "Client" configuration is selected. The output window shows the following log:


```

/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java ...
objc[1741]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (0x10cbe34c0)
before calling: public abstract void com.shengsiyuan.jvm.bytecode.Subject.request()
From real subject
after calling: public abstract void com.shengsiyuan.jvm.bytecode.Subject.request()
Class com.sun.proxy.$Proxy0

```
- Status Bar:** The status bar at the bottom right indicates the date and time: "9月8日 周六 17:13".

这是动态生成的

This screenshot is identical to the one above, showing the IntelliJ IDEA interface with the "Client" run configuration completed. The output log shows the generated proxy class and its superclass.

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，要求其父类全部初始化完毕。**~~

使用反编译对class文件

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方
4 // 本质上，调用类并没有直接引用到定义常量
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MyTest2
7 // 甚至，我们可以将MyParent2的class文
8
9 public class MyTest2 {
10
11     public static void main(String args) {
12         System.out.println(MyPare
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

```

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

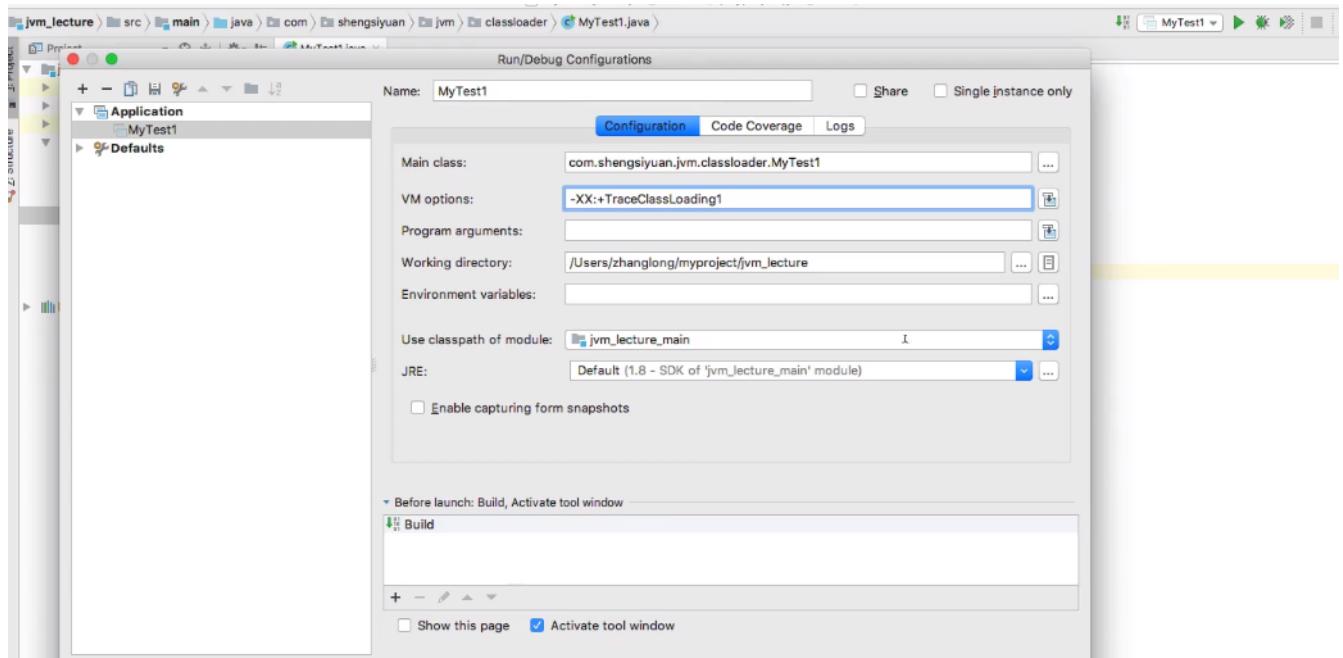
```

助记符

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

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

2.浮点型

1	类型	存储需求	bit数	备注
2	float	4字节	4*8	float类型的数值有一个后缀F(例如: 3.14F)
3	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

1	类型	存储需求	bit数
2	char	2字节	2*8

4.boolean类型

1	类型	存储需求	bit数	取值范围
2	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”。