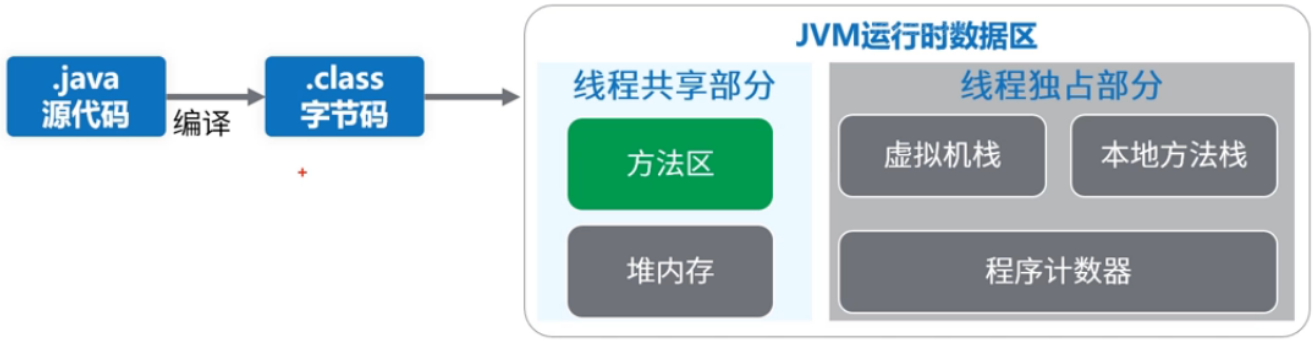
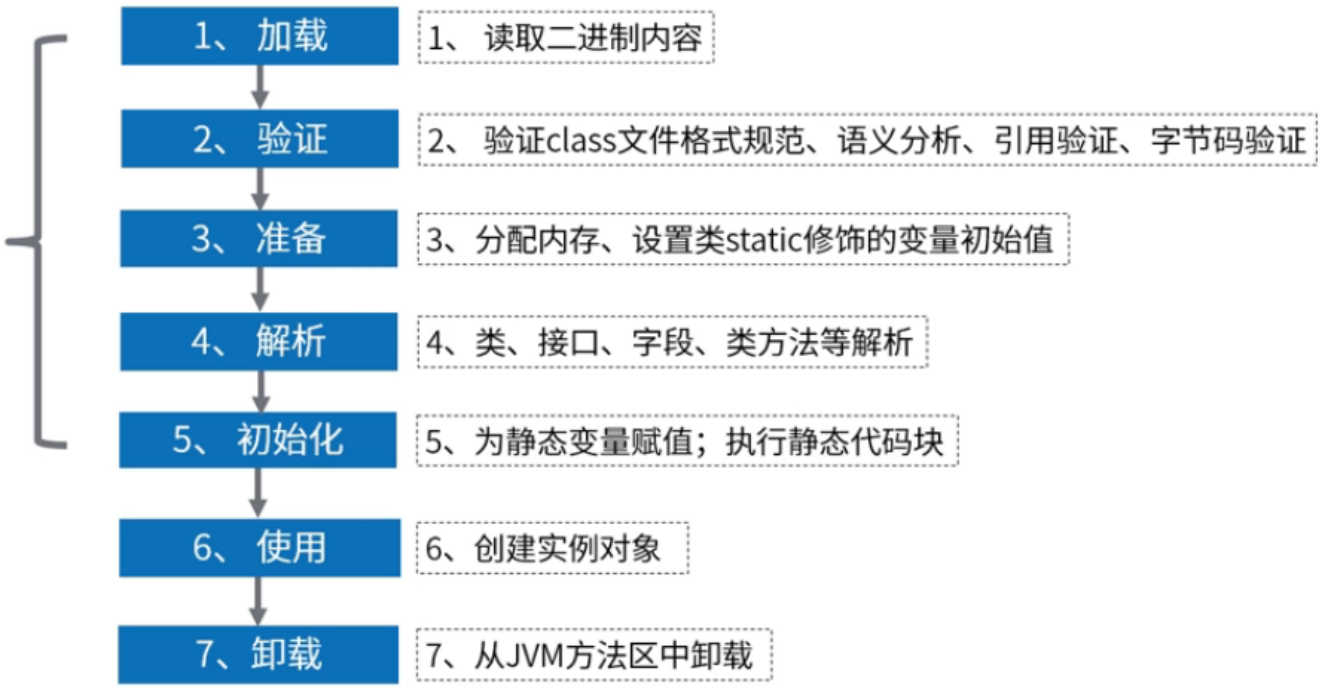
类加载机制

运行时数据区

JVM用来存储加载的类信息、常亮、静态变量、编译后的代码等数据。虚拟机规范中这是一个逻辑区划，具体实现根据不同虚拟机来实现。oracle的HotSpot在java7中放在永久代，java8放在元数据空间，并且通过GC机制对这个区域进行管理。

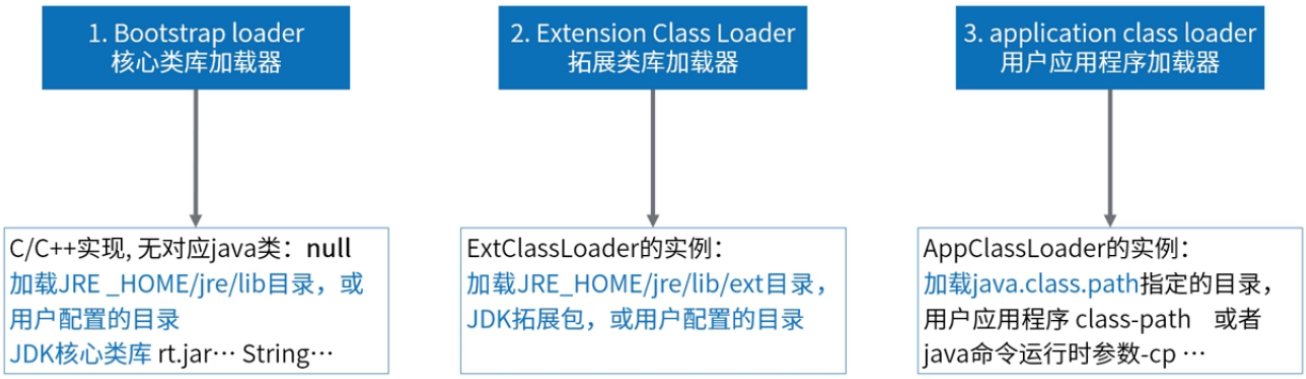


类的生命周期



类加载器

类加载器负责装入类，搜索网络、jar、zip、文件夹、二进制数据、内存等指定位置的类资源。一个java程序运行，最少有三个类加载器实例，负责不同类的加载。



查看类对应的加载器

1. Bootstrap Class Loader核心类加载器。
2. Extension Class Loader拓展类库加载器。
3. Application Class Loader用户应用程序加载器。

通过JDK-API进行查看：java.lang.Class.getClassLoader()返回装载类的类加载器。如果这个类是由BootstrapClassLoader加载的，那么这个方法在这种实现中将返回null。

```
public class ClassLoaderView {
    public static void main(String[] args) throws Exception {
        // 加载核心类库的 BootStrap ClassLoader
        System.out.println("核心类库加载器："
                +
ClassLoaderView.class.getClassLoader().loadClass("java.lang.String").getClassLoader());
        // 加载拓展库的 Extension ClassLoader
        System.out.println("拓展类库加载器：" + ClassLoaderView.class.getClassLoader()
                .loadClass("com.sun.nio.zipfs.ZipCoder").getClassLoader());
        // 加载应用程序的
        System.out.println("应用程序库加载器：" + ClassLoaderView.class.getClassLoader());

        // 双亲委派模型 Parents Delegation Model
        System.out.println("应用程序库加载器的父类：" +
ClassLoaderView.class.getClassLoader().getParent());
        System.out.println(
                "应用程序库加载器的父类的父类：" +
ClassLoaderView.class.getClassLoader().getParent().getParent());
    }
}

核心类库加载器：null
拓展类库加载器：sun.misc.Launcher$ExtClassLoader@2503dbd3
应用程序库加载器：sun.misc.Launcher$AppClassLoader@232204a1
应用程序库加载器的父类：sun.misc.Launcher$ExtClassLoader@2503dbd3
应用程序库加载器的父类的父类：null
```

JVM如何知道我们的类在哪里

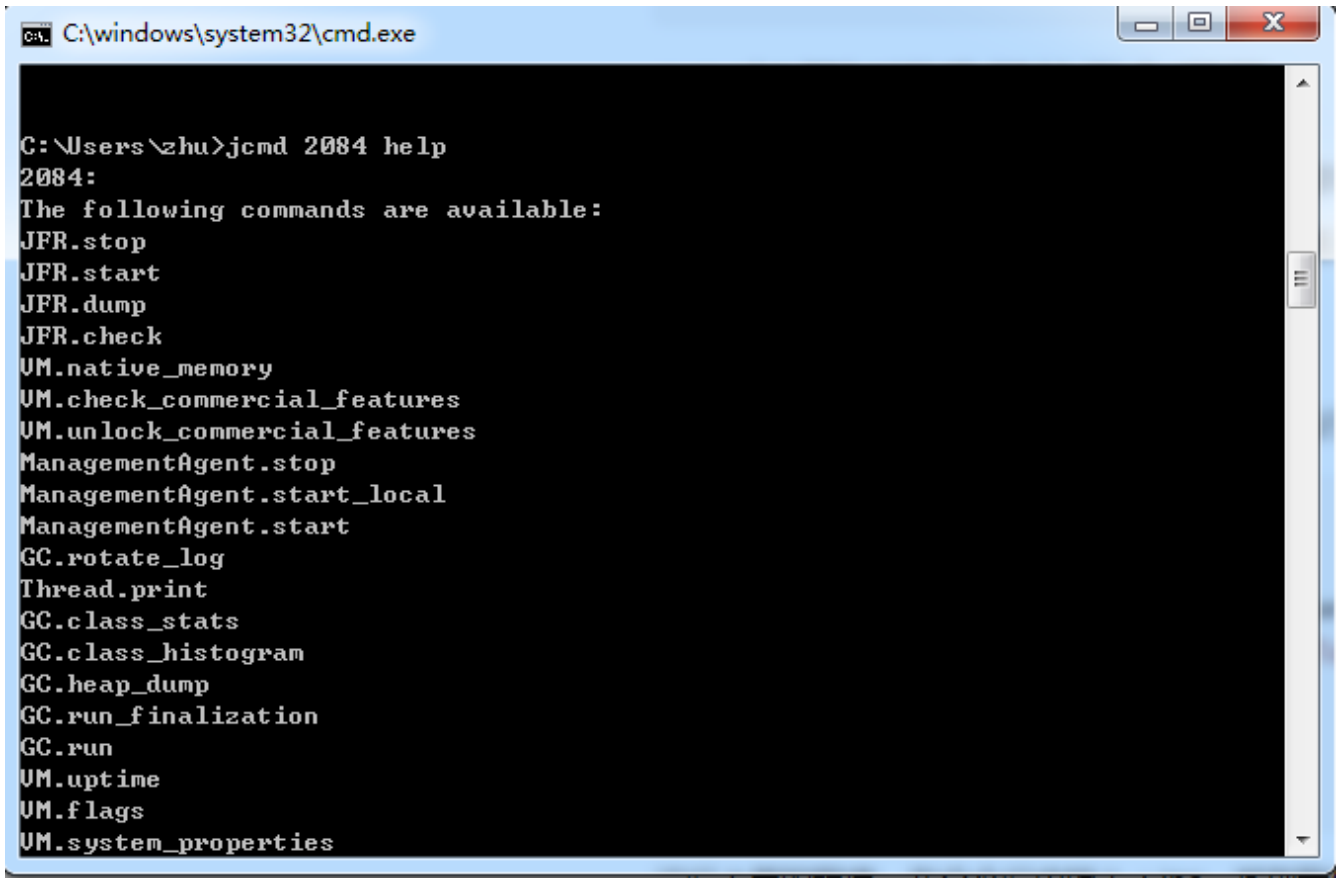
1. class信息存放在不同的位置，桌面jar、项目bin目录、target目录等。

2. 查看AppClassLoader的源码，我们可以知道通过读取java.class.path系统变量，指定去那些地址加载类资源。我们可以通过使用jps，jcmd命令。
3. jps查看本机JAVA进程。
4. jcmd 进程号 VM.system_properties查看运行时配置信息。

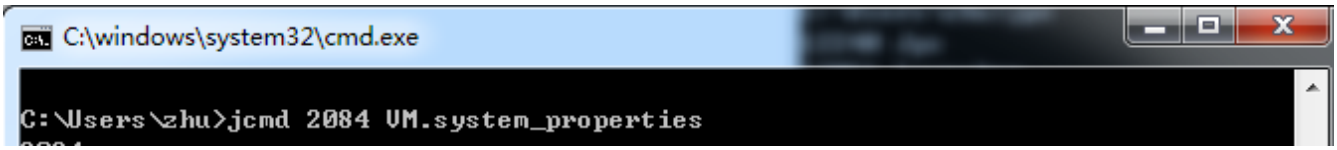
```
public class Main {

    public static void main(String[] args) throws IOException {
        System.out.println("Hello World!");
        System.in.read();
    }
}
```

运行上面的程序，在命令行中输入：jps，找到我们运行的main程序的进程号。

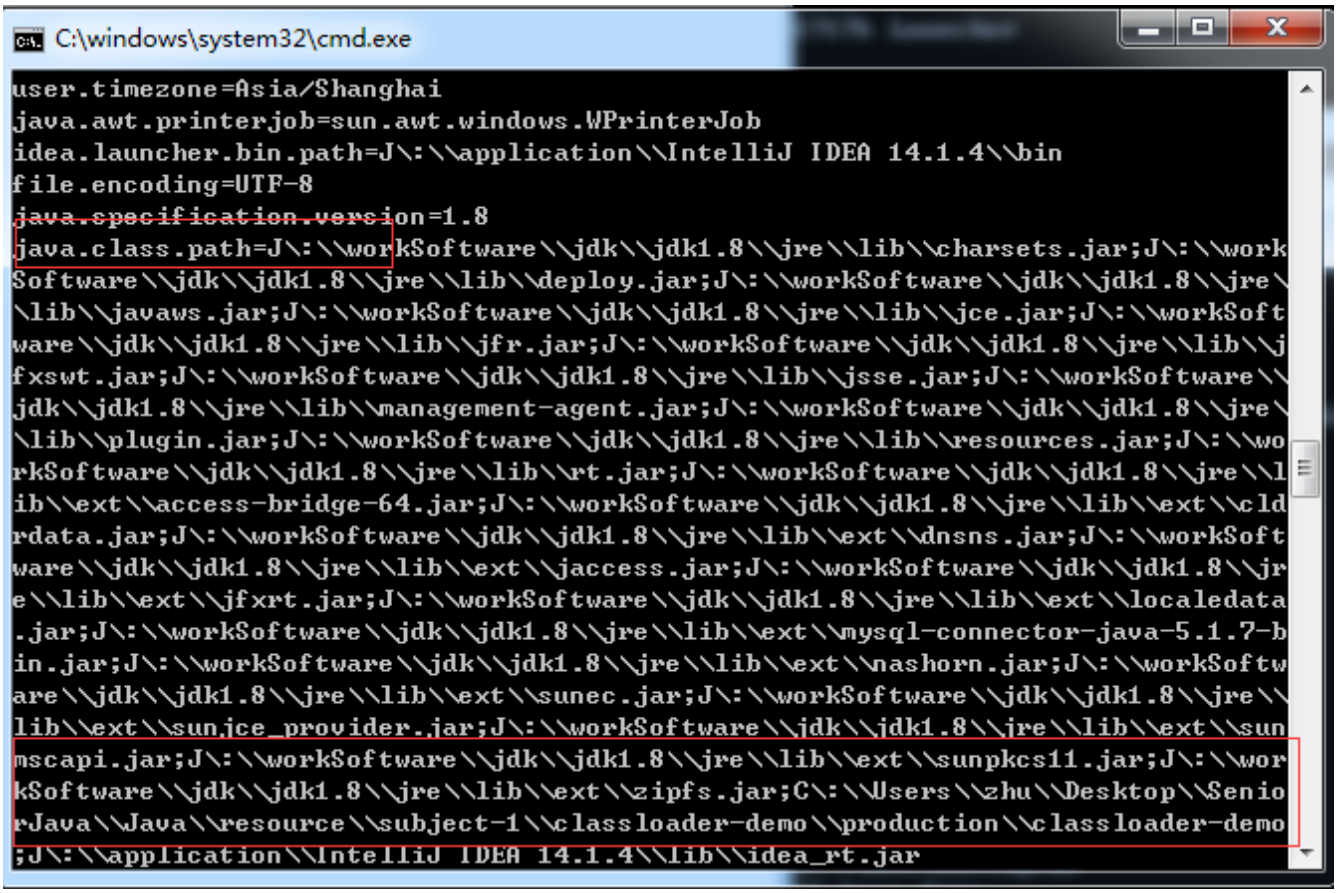
输入jcmd 进程号 help可以看我们能够查询的配置信息。



通过下面的命令查询系统变量。



查看java.class.path变量指定了我们自定义类的加载路径。



类不会重复加载

1. 类的唯一性：同一个类加载器，类名一样，代表的是同一个类。
2. 识别方式：ClassLoader Instance id + PackageName + ClassName。
3. 验证：使用类加载器，对同一个class类的不同版本，进行多次加载，检查是否会加载到最新代码。

```
public class LoaderTest {
    public static void main(String[] args) throws Exception {
        URL classUrl = new URL("file:D:\\");//jvm 类放在位置

        URLClassLoader parentLoader = new URLClassLoader(new URL[]{classUrl});

        URLClassLoader loader = new URLClassLoader(new URL[]{classUrl});
        while (true) {
            // 创建一个新的类加载器

            // 问题：静态块触发
            Class clazz = loader.loadClass("HelloService");
            System.out.println("HelloService所使用的类加载器：" + clazz.getClassLoader());
```

```
            Object newInstance = clazz.newInstance();
            Object value = clazz.getMethod("test").invoke(newInstance);
            System.out.println("调用getValue获得的返回值为：" + value);

            Thread.sleep(3000L); // 1秒执行一次
            System.out.println();
        }
    }
}

public class HelloService {
    private static int max = getValue();

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

    private static int getValue() {
        System.out.println("static field");
        return 1;
    }

    public void test() {
        System.out.println("test12323");
    }
}

HelloService所使用的类加载器：java.net.URLClassLoader@14ae5a5
static field
static code
test
调用getValue获得的返回值为：null

HelloService所使用的类加载器：java.net.URLClassLoader@14ae5a5
test
调用getValue获得的返回值为：null

HelloService所使用的类加载器：java.net.URLClassLoader@14ae5a5
test
调用getValue获得的返回值为：null
```

我们前编译好HelloService，并且在运行修改test方法的打印输出，发现LoaderTest中通过反射调用test方法的输出一直都不变，说明类不会被同一个类加载器重复加载。

类的卸载

1. 类什么时候被卸载：该Class所有的实例都已经被GC；加载该类的ClassLoader实例已经被GC;
2. 验证：jvm启动参数中增加-verbose:class参数，输出类加载和卸载的日志信息。

```
public class LoaderTest {
    public static void main(String[] args) throws Exception {
        URL classUrl = new URL("file:D:\\");//jvm 类放在位置
```

```
        URLClassLoader parentLoader = new URLClassLoader(new URL[]{classUrl});

        URLClassLoader loader = new URLClassLoader(new URL[]{classUrl});
        while (true) {
            // 创建一个新的类加载器
            if (loader == null) {
                break;
            }

            // 问题：静态块触发
            Class clazz = loader.loadClass("HelloService");
            System.out.println("HelloService所使用的类加载器：" + clazz.getClassLoader());

            Object newInstance = clazz.newInstance();
            Object value = clazz.getMethod("test").invoke(newInstance);
            System.out.println("调用getValue获得的返回值为：" + value);

            Thread.sleep(3000L); // 1秒执行一次
            System.out.println();

            // gc
            newInstance = null;
            loader = null;
        }

        System.gc();
        Thread.sleep(20000);
    }
}
```

```
[Loaded java.net.InetAddressImplFactory from J:\workSoftware\jdk\jdk1.8\jre\lib\rt.jar]
[Loaded java.net.InetAddressImpl from J:\workSoftware\jdk\jdk1.8\jre\lib\rt.jar]
[Loaded java.net.Inet6AddressImpl from J:\workSoftware\jdk\jdk1.8\jre\lib\rt.jar]
[Loaded sun.net.spi.nameservice.NameService from J:\workSoftware\jdk\jdk1.8\jre\lib\rt.jar]
[Loaded java.net.InetAddress$2 from J:\workSoftware\jdk\jdk1.8\jre\lib\rt.jar]
[Loaded java.net.Inet4AddressImpl from J:\workSoftware\jdk\jdk1.8\jre\lib\rt.jar]
[Loaded java.net.Inet4Address from J:\workSoftware\jdk\jdk1.8\jre\lib\rt.jar]
[Loaded sun.net.NetHooks from J:\workSoftware\jdk\jdk1.8\jre\lib\rt.jar]
[Loaded java.net.Inet6Address from J:\workSoftware\jdk\jdk1.8\jre\lib\rt.jar]
[Loaded java.net.Inet6Address$Inet6AddressHolder from J:\workSoftware\jdk\jdk1.8\jre\lib\rt.jar]
[Loaded java.net.Socket from J:\workSoftware\jdk\jdk1.8\jre\lib\rt.jar]

[Unloading class HelloService 0x000000013f440828]
[Loaded java.lang.Shutdown from J:\workSoftware\jdk\jdk1.8\jre\lib\rt.jar]
[Loaded java.lang.Shutdown$Lock from J:\workSoftware\jdk\jdk1.8\jre\lib\rt.jar]

Process finished with exit code 0
```

3. 静态代码块和静态变量赋值什么时候执行

```
public class LoaderTest {
    public static void main(String[] args) throws Exception {
        URL classUrl = new URL("file:D:\\");//jvm 类放在位置
        URLClassLoader loader = new URLClassLoader(new URL[]{classUrl});
        while (true) {
            // 创建一个新的类加载器
```

```
            if (loader == null) {
                break;
            }

            // 问题：静态块触发
            Class clazz = loader.loadClass("HelloService");
            System.out.println("HelloService所使用的类加载器：" + clazz.getClassLoader());
            loader = null;
        }

        System.gc();
        Thread.sleep(20000);
    }
}
```

```
[Loaded sun.reflect.DelegatingMethodAccessorImpl from J:\workSoftware\
[Loaded HelloService from file:D:/]
HelloService所使用的类加载器：java.net.URLClassLoader@14ae5a5
[Unloading class HelloService 0x000000013f440828]
```

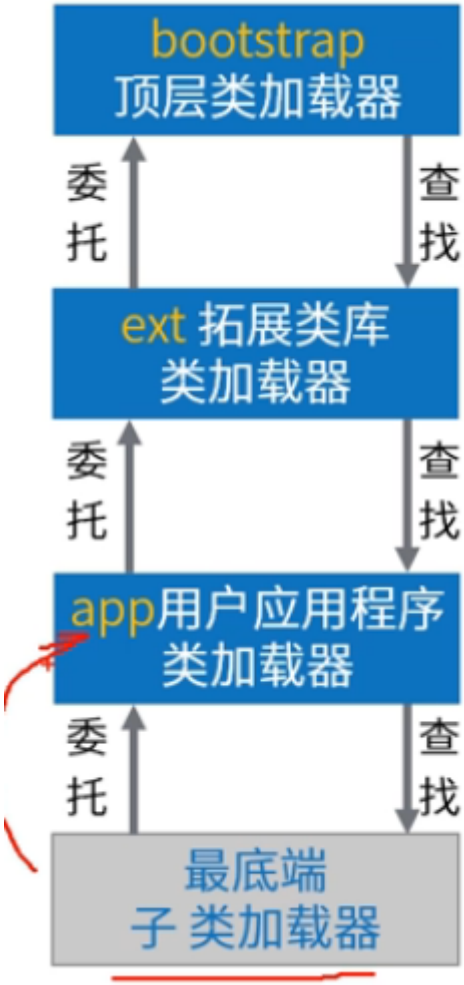
```
public class LoaderTest {
    public static void main(String[] args) throws Exception {
        URL classUrl = new URL("file:D:\\");//jvm 类放在位置
        URLClassLoader loader = new URLClassLoader(new URL[]{classUrl});
        while (true) {
            // 创建一个新的类加载器
            if (loader == null) {
                break;
            }

            // 问题：静态块触发
            Class clazz = loader.loadClass("HelloService");
             Object newInstance = clazz.newInstance();
            System.out.println("HelloService所使用的类加载器：" + clazz.getClassLoader());
            loader = null;
        }

        System.gc();
        Thread.sleep(20000);
    }
}
```

```
[Loaded HelloService from file:D:/]
HelloService所使用的类加载器：java.net.URLClassLoader@14ae5a5
static field
static code
[Unloading class HelloService 0x000000013f440828]
[Loaded java.net.Socket from J:\workSoftware\jdk\jdk1.8\jre\lib\rt.jar]
```

从上面实验可以看出，静态代码是在第一次创建类实例时进行的。

双亲委派模式

1. 为了避免重复加载和安全性，由上而下逐级委托，由上而下逐级查找。
2. 首先不会自己去尝试加载类，而是把这个请求委派给父加载器去完成。每一个层次的加载器都是如此，因此所有的类加载器请求都会传给上层的启动类加载器。
3. 只有当父加载器反馈自己无法完成该加载请求(该加载器的搜索范围中没有找到对应的类)时，子加载器才会尝试自己去加载。
4. 类加载器之间不存在父类子类关系，双亲只是翻译，可以理解为逻辑上定义的上下级关系。




```
public class LoaderTest1 {
    public static void main(String[] args) throws Exception {
        URL classUrl = new URL("file:D:\\");

        while (true) {
            // 创建一个新的类加载器，它的父加载器为上面的parentLoader
            URLClassLoader loader = new URLClassLoader(new URL[]{classUrl});

            Class clazz = loader.loadClass("HelloService");
            System.out.println("HelloService所使用的类加载器：" + clazz.getClassLoader());
            Object newInstance = clazz.newInstance();
            Object value = clazz.getMethod("test").invoke(newInstance);
            System.out.println("调用getValue获得的返回值为：" + value);

            Thread.sleep(3000L); // 1秒执行一次
```

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

执行上面的代码，在运行期间修改HelloSerivice的test方法，可以看到效果如下。

```
HelloService所使用的类加载器：java.net.URLClassLoader@12a3a380
static field
static code
test12323
调用getValue获得的返回值为：null

HelloService所使用的类加载器：java.net.URLClassLoader@6e0be858
static field
static code
test12323
调用getValue获得的返回值为：null

HelloService所使用的类加载器：java.net.URLClassLoader@511d50c0
static field
static code
test12323sdfsdfse
调用getValue获得的返回值为：null
```

我们发现对test的调用输出改变了，因为在每次加载类时，使用的都是新创建的类加载器，所以对HelloService的每次改动，类加载器都会重新加载，这就是热加载原理。

```
public class LoaderTest1 {
    public static void main(String[] args) throws Exception {
        URL classUrl = new URL("file:D:\\");
        // 测试双亲委派机制
        // 如果使用此加载器作为父加载器,则下面的热更新会失效,因为双亲委派机制,HelloService实际上是被这个类加载器加载的;
        URLClassLoader parentLoader = new URLClassLoader(new URL[]{classUrl});

        while (true) {
            // 创建一个新的类加载器，它的父加载器为上面的parentLoader
            URLClassLoader loader = new URLClassLoader(new URL[]{classUrl}, parentLoader);

            Class clazz = loader.loadClass("HelloService");
            System.out.println("HelloService所使用的类加载器：" + clazz.getClassLoader());
            Object newInstance = clazz.newInstance();
            Object value = clazz.getMethod("test").invoke(newInstance);
            System.out.println("调用getValue获得的返回值为：" + value);

            Thread.sleep(3000L); // 1秒执行一次
            System.out.println();
        }
    }
}
```

上面代码我们为每个新创建的类加载器指定了父类加载器，此时每次加载都会先委托父类加载，所以最终的类都是有同一个父类加载器加载的，所以改变HelloService的test方法，不会被重复加载。

```
HelloService所使用的类加载器：java.net.URLClassLoader@14ae5a5
test12323sdfsdfse
调用getValue获得的返回值为：null

HelloService所使用的类加载器：java.net.URLClassLoader@14ae5a5
test12323sdfsdfse
调用getValue获得的返回值为：null

HelloService所使用的类加载器：java.net.URLClassLoader@14ae5a5
test12323sdfsdfse
调用getValue获得的返回值为：null
```