注解和反射

1. 注解

> 注解
>
> 1.回顾我们之前所接触过的注解 @Override @Deprecated SuppressWarnings @FunctionalInterface
>
> 分析：以上注解 可以书写在不同的位置 有不同的作用 有的需要写值 有的不能写值
>
> 2.如何控制这些注解书写的位置 、如何使这些注解有不同的作用 等等
>
> 通过@Target注解控制注解可以书写的位置
>
> 我们也可以通过自定义注解来实现这些效果
>
> 3.注解是由来
>
> 注解是JDK1.5新增的内容
>
> 4.注解用来解决什么问题呢?
>
> 早期的Java项目中 复杂度非常之高 所以项目中会存在很多的配置文件 阅读性差 书写性差
>
> 所以 在JDK1.5引入了注解来解决这个问题 最初的梦想是 零配置(配置文件全部消失)
>
> 目前的情况：注解 + 配置文件
>
> 注解也确实简略 省略了 大量的配置文件 但是配置文件依然是不能完全杜绝的
>
> 注解采用了一种思想：约定大于配置
>
> 5.元注解
>
> 用于修饰注解的注解 称之为元注解
>
> @Target 用于规定注解书写的位置 不写表示此注解可以加在任何位置
>
> @Retention 用于规定注解的保留策略
>
> CLASS 表示在二进制文件中生效 默认为此效果
>
> RESOURCE 表示在源代码中生效
>
> RUNTIME 表示在运行期间生效
>
> @Documented 被此注解修饰的注解可以保存在帮助文档中
>
> @Inherited 被此注解修饰的注解 可以被子类继承

> 注解属性和赋值
>
> 注解属性支持的数据类型： 八种基本数据类型、String、枚举、Class类型 和 以上类型对应的数组类型

> 注解中的属性必须有值
>
> 注解属性的赋值:
>
> 1.如果注解中只有一个属性 并且属性名为value 则可以直接写值
>
> 2.如果为数组类型 一个元素直接赋值 多个元素 加上大括号
>
> 3.否则其他的情况都必须写为 属性名 = 属性值 这种写法
>
> 4.我们也可以使用default关键字给注解加上默认值

2. JUnit单元测试

> JUnit Java Unit 单元测试框架 是一个专业的测试框架(别人写好的一些类 接口 方法 等等)
>
> 回顾我们之前怎么测试我们写的代码 使用main方法 但是一个类只能有一个main方法 所以 不能满足我们实际的开发需求
>
> 我们可以使用Junit单元测试框架
>
> 我们目前所使用src目录 表示源文件目录 是用于存放Java源文件的 所以规范而言 测试的代码 不应该存放在 src 目录下
>
> 应该单独创建一个测试目录
>
> 1.在项目下创建文件夹 取名为test
>
> 2.右键将此目录标记为测试资源根目录

> @Test 注解 加在方法上 表示此方法可以单独执行
>
> @Before 表示此类中的@Test修饰的方法执行之前都执行一次
>
> @After 表示此类中的@Test修饰的方法执行之后都执行一次
>
> @BeforeClass 本类中的方法执行之前只执行一次
>
> @AfterClass 本类中的方法执行之后只执行一次

```java
package com.atguigu.test;

import org.junit.*;

/**
 * @author WHD
 * @description TODO
 * @date 2023/6/20 10:43
 *  @Test 注解 加在方法上 表示此方法可以单独执行
 *  @Before 表示此类中的@Test修饰的方法执行之前都执行一次
 *  @After 表示此类中的@Test修饰的方法执行之后都执行一次
 *
 *  @BeforeClass 本类中的方法执行之前只执行一次
 *  @AfterClass 本类中的方法执行之后只执行一次
 */
```

```
public class TestJUnit {
    @Before
    public void before(){
        System.out.println("执行前置操作");
    }

    @After
    public void after(){
        System.out.println("执行后置操作");
    }

    @BeforeClass
    public static void beforeClass(){
        System.out.println("本类中的方法执行之前只执行一次");
    }


    @AfterClass
    public static void afterClass(){
        System.out.println("本类中的方法执行之后只执行一次");
    }


    @Test
    public void m1(){
        System.out.println("m1方法执行");
    }

    @Test
    public void m2(){
        System.out.println(10 / 1);
    }


    @Test
    public void m3(){
        System.out.println("m3方法执行");
    }

}
```

3.反射

> 反射 在程序运行期间 动态的获取某个类的信息(属性、方法、构造方法) 并且访问
>
> 也就是不通过new对象的方式 依然可以访问类中的属性 方法 和 构造方法
>
> 生活中的反射：倒车镜 拍X光片 IDE的自动提示功能 等等
>
> 综合生活中常见的反射的操作 我们发现 反射在有些情况下是必不可少的
>
> JUnit框架

> 如果在程序运行过程中 不知道需要什么类型的对象 以及 不知道需要获取多少个对象 该如何创建对象呢?
>
> Spring IOC Inversion Of Control 容器 就是使用反射技术帮我们创建对象的
>
> 万物皆对象：类也是对象 方法也是对象 属性也是对象 构造器也是对象
>
> java.lang.Class 用于表示类类型
>
> java.lang.reflect.Field 字段类 类型
>
> java.lang.reflect.Method 方法类 类型
>
> java.lang.reflect.Constructor 构造器类 类型

3.1 获取属性

> Field类 所有的字段都属于此类的对象
>
> 通过Class类提供的如下方法获取Field字段
>
> Field getField(String fieldName) 根据指定的名称获取public修饰的属性
>
> Field [] getFields() 获取本类中所有的public修饰的属性

```
package com.atguigu.test6;

import java.lang.reflect.Field;

/**
 * @author WHD
 * @description TODO
 * @date 2023/6/20 15:08
 *  Field类 所有的字段都属于此类的对象
 *  通过Class类提供的如下方法获取Field字段
 *
 *  Field getField(String fieldName) 根据指定的名称获取public修饰的属性
 *  Field [] getFields() 获取本类中所有的public修饰的属性
 */
public class TestFiled1 {
    public static void main(String[] args) {
        try {
            // 根据全限定名获取Class对象
            Class<?> stuClass = Class.forName("com.atguigu.test6.Student");

            // 根据具体的名称获取单个字段对象
            Field heightField = stuClass.getField("height");

            // 打印字段对象的名称 和 类型
            System.out.println(heightField.getName() + "===" + heightField.getType());

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

            // 获取所有的public修饰的字段对象
            Field[] fields = stuClass.getFields();
```

```
            // 遍历数组
            for(Field f : fields){
                // 打印字段对象的名称 和 类型
                System.out.println(f.getName() + "===" + f.getType());
            }

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




        } catch (ClassNotFoundException e) {
            throw new RuntimeException(e);
        } catch (NoSuchFieldException e) {
            throw new RuntimeException(e);
        }

    }
}
```

获取非公开的属性

Field getDeclaredField(String fieldName) : 根据属性名获取一个属性对象 可以是任何访问修饰符修饰

Field [] getDeclaredFields() : 获取所有属性对象 可以是任何访问修饰符修饰

Field类方法

第一个参数 表示给哪个对象的name属性赋值

第二个参数 具体的属性值

set(Object obj,Object value)

取值传参 表示声明 访问哪个对象的此属性值

get(Object obj)

```
package com.atguigu.test6;

import java.lang.reflect.Field;

/**
 * @author WHD
 * @description TODO
 * @date 2023/6/20 15:19
 *  获取非公开的属性
 *  Field getDeclaredField(String fieldName) : 根据属性名获取一个属性对象 可以是任何访问修饰符修饰
 *  Field [] getDeclaredFields() : 获取所有属性对象 可以是任何访问修饰符修饰
 *
 *  Field类方法
 *  第一个参数 表示给哪个对象的name属性赋值
 *  第二个参数 具体的属性值
 *  set(Object obj,Object value)
```

```
 *
 *  取值传参 表示声明 访问哪个对象的此属性值
 *  get(Object obj)
 */
public class TestField2 {
    public static void main(String[] args) {
        try {
            Class<?> stuClass = Class.forName("com.atguigu.test6.Student");

            Field[] declaredFields = stuClass.getDeclaredFields();

            Object obj = stuClass.newInstance();

            for (Field declaredField : declaredFields) {
                System.out.println(declaredField.getName() + "====" + declaredField.getType());
            }

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

            Field nameField = stuClass.getDeclaredField("name");

            // 调用此方法表示忽略JVM安全检查 即不再抛出异常 即可以访问了
            nameField.setAccessible(true);

            // 第一个参数 表示给哪个对象的name属性赋值
            // 第二个参数 具体的属性值
            nameField.set(obj,"赵四");

            // 取值传参 表示声明 访问哪个学生对象的 name属性值
            System.out.println(nameField.get(obj));

        } catch (ClassNotFoundException e) {
            throw new RuntimeException(e);
        } catch (NoSuchFieldException e) {
            throw new RuntimeException(e);
        } catch (InstantiationException e) {
            throw new RuntimeException(e);
        } catch (IllegalAccessException e) {
            throw new RuntimeException(e);
        }


    }
}
```

3.2 获取方法

> Class类提供的两个方法
>
> Method getMethod(String name,Class<?>... parameterType)

> 根据方法名和形参列表获取一个public或者继承自父类的方法
>
> Method [] getMethods() 获取本类中所有public修饰的 和 继承自父类的方法

```
package com.atguigu.test7;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

/**
 * @author WHD
 * @description TODO
 * @date 2023/6/20 15:34
 *  Class类提供的两个方法
 *  Method getMethod(String name,Class<?>... parameterType)
 *  根据方法名和形参列表获取一个public或者继承自父类的方法
 *
 *  Method [] getMethods() 获取本类中所有public修饰的 和 继承自父类的方法
 */
public class TestMethod1 {
    public static void main(String[] args) {
        try {
            Class<?> stuClass = Class.forName("com.atguigu.test6.Student");

            Method[] methods = stuClass.getMethods();

            for (Method method : methods) {
                System.out.println(method.getName() + "-----" +   method.getParameterCount());
            }

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

            Method noParameterM4 = stuClass.getMethod("m4");

            Object obj = stuClass.newInstance();

            noParameterM4.invoke(obj);
            System.out.println("--------------------------------------");


            Method intParameterM4 = stuClass.getMethod("m4", int.class);

            intParameterM4.invoke(obj,100);

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

            Method twoParameterM4 = stuClass.getMethod("m4", String.class, int.class);

            twoParameterM4.invoke(obj, "abc",100);


        } catch (ClassNotFoundException e) {
            throw new RuntimeException(e);
```

```
        } catch (NoSuchMethodException e) {
            throw new RuntimeException(e);
        } catch (InstantiationException e) {
            throw new RuntimeException(e);
        } catch (IllegalAccessException e) {
            throw new RuntimeException(e);
        } catch (InvocationTargetException e) {
            throw new RuntimeException(e);
        }
    }
}
```

> Method getDeclaredMethod(String name,Class<?>... parameterType)
>
> 根据方法名和形参列表获取一个任意修饰符修饰的 本类中定义的方法 (不包括继承自父类的方法)
>
> Method [] getDeclaredMethods() 获取本类中所有的任意修饰符修饰的已定义的方法(不包括继承自父类的方法)

```
package com.atguigu.test7;

import java.lang.reflect.Method;

/**
 * @author WHD
 * @description TODO
 * @date 2023/6/20 15:46
 *  Method getDeclaredMethod(String name,Class<?>... parameterType)
 *   根据方法名和形参列表获取一个任意修饰符修饰的 本类中定义的方法 (不包括继承自父类的方法)
 *
 *    Method [] getDeclaredMethods() 获取本类中所有的任意修饰符修饰的已定义的方法(不包括继承自父类的方法)
 */
public class TestMethod2 {
    public static void main(String[] args) throws Exception {
        Class<?> stuClass = Class.forName("com.atguigu.test6.Student");

        Method[] declaredMethods = stuClass.getDeclaredMethods();

        for (Method declaredMethod : declaredMethods) {

            System.out.println(declaredMethod.getName() + "---" +
declaredMethod.getParameterCount());
        }

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

        Method m1 = stuClass.getDeclaredMethod("m1");

        m1.setAccessible(true); // 忽略JVM安全检查 即可以访问

        Object obj = stuClass.newInstance();

        m1.invoke(obj);
```

```
    }
}
```

3.3 获取构造器

> Class类提供如下方法获取构造器对象
>
> Constructor getConstructor(Class<?> ...parameterType) 根据参数列表获取一个public修饰的构造器
>
> Constructor [] getConstructors() 获取本类中所有的public修饰的构造器
>
> Class类中的newInstance() 和 Constructor类中的newInstance(Object...args)
>
> 注意区分

```
package com.atguigu.test8;

import com.atguigu.test6.Student;

import java.lang.reflect.Constructor;

/**
 * @author WHD
 * @description TODO
 * @date 2023/6/20 15:53
 *  Class类提供如下方法获取构造器对象
 *
 *  Constructor getConstructor(Class<?> ...parameterType) 根据参数列表获取一个public修饰的构造器
 *  Constructor [] getConstructors() 获取本类中所有的public修饰的构造器
 *
 *
 *  Class类中的newInstance()  和 Constructor类中的newInstance(Object...args)
 *  注意区分
 *
 *
 */
public class TestConstructors1 {
    public static void main(String[] args) throws Exception {
        Class<?> stuClass = Class.forName("com.atguigu.test6.Student");

        Constructor<?> constructor = stuClass.getConstructor(String.class,String.class);

        Object obj = constructor.newInstance("hello", "world");

        if(obj instanceof Student){

            Student stu = (Student) obj;
```

```
            System.out.println("stu = " + stu);
        }

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

        Constructor<?>[] constructors = stuClass.getConstructors();

        for (Constructor<?> con : constructors) {
            System.out.println(con.getName() + "=======" + con.getParameterCount());
        }


    }
}
```

Class类提供如下方法获取构造器对象

Constructor getDeclaredConstructor(Class<?> ...parameterType) 根据参数列表获取任意修饰符修饰的构造器

Constructor [] getDeclaredConstructors() 获取本类中所有任意修饰符修饰的构造器

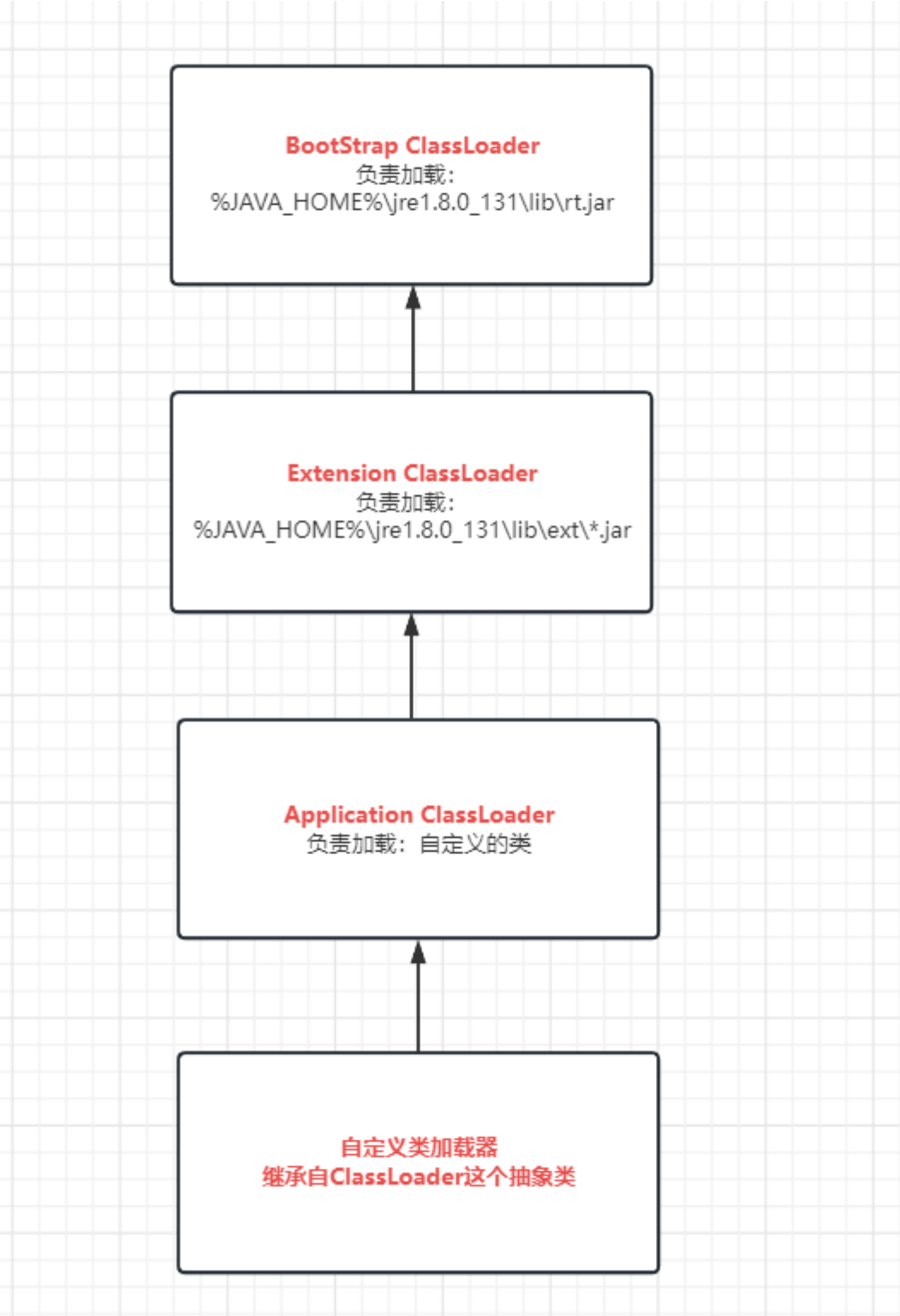
Class类中的newInstance() 和 Constructor类中的newInstance(Object...args)

注意区分

```
package com.atguigu.test8;

import com.atguigu.test6.Student;

import java.lang.reflect.Constructor;

/**
 * @author WHD
 * @description TODO
 * @date 2023/6/20 15:53
 *  Class类提供如下方法获取构造器对象
 *
 *  Constructor getDeclaredConstructor(Class<?> ...parameterType) 根据参数列表获取任意修饰符修饰的构造
器
 *  Constructor [] getDeclaredConstructors() 获取本类中所有任意修饰符修饰的构造器
 *
 *
 *  Class类中的newInstance()  和 Constructor类中的newInstance(Object...args)
 *  注意区分
 *
 *
 */
public class TestConstructors2 {
    public static void main(String[] args) throws Exception {
        Class<?> stuClass = Class.forName("com.atguigu.test6.Student");
```

```
        Constructor<?> constructor = stuClass.getDeclaredConstructor();

        constructor.setAccessible(true);

        Object obj = constructor.newInstance();

        if(obj instanceof Student){

            Student stu = (Student) obj;

            System.out.println("stu = " + stu);
        }

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


        Constructor<?>[] declaredConstructors = stuClass.getDeclaredConstructors();


        for (Constructor<?> con : declaredConstructors) {
            System.out.println(con.getName() + "----" + con.getParameterCount());
        }


    }
}
```

4. 类加载器



类加载器的作用:

加载类的

类加载的过程:

当程序主动使用某个类时，如果该类还未被加载到内存中，则系统会通过如下三个步骤来对该类进行初始化。

①加载类(load)： 将类的class文件读入内存，并为之创建一个java.lang.Class对象。此过程由类加载器完成

将class文件字节码内容加载到内存中，并将这些数据转换成方法区的运行时数据结构，然后生成一个代表这个类的java.lang.Class对象。这个加载的过程需要类加载器参与。 *

②链接(link)： 将类的二进制数据合并到JRE中

验证：确保加载的类信息符合JVM规范，例如：以cafe开头，没有安全方面的问题

准备：正式为类变量（static）分配内存并设置类变量默认初始值的阶段，这些内存都将在方法区中进行分配(静态区)。

解析：将类、接口、字段和方法的符号引用转为直接引用。

③初始化(Initialize) : JVM负责对类进行初始化

执行类构造器()方法的过程。类构造器()方法是由编译期自动收集类中所有类变量的赋值动作和静态代码块中的语句合并产生的。（类构造器是构造类信息的，不是构造该类对象的构造器）。

当初始化一个类的时候，如果发现其父类还没有进行初始化，则需要先触发其父类的初始化。

虚拟机会保证一个类的()方法在多线程环境中被正确加锁和同步。

BootStrap ClassLoader 顶层(核心)类加载器 C:\Program Files\Java\jre1.8.0_131\lib\ rt.jar

Extension ClassLoader 扩展类加载器 C:\Program Files\Java\jre1.8.0_131\lib\ext.*.jar

Application ClassLoader 应用程序类加载器 加载自定义的类

自定义类加载器 继承ClassLoad 重写 findClass()方法

双亲委派模型(机制)：

当JVM需要加载一个类时 先委托给上级的类加载器来加载 以此类推

这样的设计的目的是为了保证Java中的源代码不被污染 入侵

```
package com.atguigu.test3;

import sun.net.spi.nameservice.dns.DNSNameService;

/**
 * @author WHD
 * @description TODO
 * @date 2023/6/21 11:24
 *  类加载器 ClassLoader
 *
 *
 *  类加载作用? 加载类的
 *
     *  类加载的过程：
     *  当程序主动使用某个类时，如果该类还未被加载到内存中，则系统会通过如下三个步骤来对该类进行初始化。
     *  ①加载类(load) ： 将类的class文件读入内存，并为之创建一个java.lang.Class对象。此过程由类加载器完成
     *  将class文件字节码内容加载到内存中，并将这些数据转换成方法区的运行时数据结构，然后生成一个代表这个类的java.lang.Class对象。这个加载的过程需要类加载器参与。
     *
     *  ②链接(link) ： 将类的二进制数据合并到JRE中
```

```
     *       验证：确保加载的类信息符合JVM规范，例如：以cafe开头，没有安全方面的问题
     *       准备：正式为类变量（static）分配内存并设置类变量默认初始值的阶段，这些内存都将在方法区中进行分配(静态区)。
     *       解析：将类、接口、字段和方法的符号引用转为直接引用。
     *  ③初始化(Initialize) ： JVM负责对类进行初始化
     *  执行类构造器<clinit>()方法的过程。类构造器<clinit>()方法是由编译期自动收集类中所有类变量的赋值动作和静态代码块中的语句合并产生的。（类构造器是构造类信息的，不是构造该类对象的构造器）。
     *  当初始化一个类的时候，如果发现其父类还没有进行初始化，则需要先触发其父类的初始化。
     *  虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确加锁和同步。
 *
 *
 *  类加载器有哪些类型
 *  BootStrap ClassLoader  顶层(核心)类加载器 C:\Program Files\Java\jre1.8.0_131\lib\ rt.jar
 *  Extension ClassLoader   扩展类加载器 C:\Program Files\Java\jre1.8.0_131\lib\ext.*.jar
 *  Application ClassLoader 应用程序类加载器 加载自定义的类
 *  自定义类加载器 继承ClassLoad 重写 findClass()方法
 *
 *  双亲委派模型(机制) ：
 *  当JVM需要加载一个类的时候 先委托给上级的类加载器来加载 以此类推
 *
 *
 */
public class Note {

    public static void main(String[] args) throws Exception {
        Note note = new Note();
        ClassLoader classLoader = note.getClass().getClassLoader();
        System.out.println(classLoader);

        System.out.println(classLoader.getParent());
        System.out.println(classLoader.getParent().getParent());



        DNSNameService dnsNameService = new DNSNameService();
        System.out.println(dnsNameService.getClass().getClassLoader());
        System.out.println(dnsNameService.getClass().getClassLoader().getParent());


    }
}
```