Java Unsafe类学习（jdk1.8）

最近跟着javasec.org学习Java安全，最近在探索在jdk1.8下面Unsafe类对类实例化的操作与classloader和类反射有什么区别。下面是源码与测试环境情况。

1、package内

在package内，我先后采用三种方法：classloader，正常reflect和通过反射Unsafe类的allocateInstance方法来调用类。

下面这段代码是class转字节码的程序

```
//将class转换为字节
package com.sec.cwm;

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

public class classTransfer extends ClassLoader{
    private String clName;
    // 构造函数1：不指定父类加载器，仅指定自定义类加载器的名称
    public classTransfer(String clName) {
        super();
        this.clName = clName;
    }
    // 构造函数2：指定父类记载器和类加载器的名称。
    public classTransfer(ClassLoader parent, String clName) {
        super(parent);
        this.clName = clName;
    }
    /**
     * loadClassData,将class文件读取为byte[]数组。
     * @param target 目标class文件路径
     * @return
     */
    /**
     * 转换为字节码
     * @param target
     * @return
     */
    private byte[] loadClassData(String target) {
        InputStream is = null;
        byte[] bytes = null;
        ByteArrayOutputStream os = null;
        int len;
        try {
            is = new FileInputStream(new File(target));
            os = new ByteArrayOutputStream();
```

```
            while(-1 != (len = is.read())) {
                while(len>128){
                    len=len-256;
                }
//                System.out.print(len);
//                System.out.print(", ");
                os.write(len);
            }
            bytes = os.toByteArray();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return bytes;
    }
    /**
     * 通过调用 defineClass()方法，将byte[] 字节数组转化为类对象
     * @param target
     * @return
     */
    public Class<?> findClass(String target, String className) {
        byte[] bytes = loadClassData(target);
        // clName 应该是 类的binary name
        return defineClass(className, bytes, 0, bytes.length);
    }
    // 以下代码会调用自定义的类加载器加载TestCode15
    public static classTransfer loader1=new classTransfer("loader1");
//    public static String
target="C:\\Users\\CWM\\IdeaProjects\\test\\target\\classes\\com\\sec\\cwm\\test
javaclass.class";
    public static String
target="C:\\Users\\CWM\\IdeaProjects\\test\\target\\classes\\com\\test\\cwm\\tes
tPrivateclass1.class";
    public static byte[] bytes=loader1.loadClassData(target);
    public static void main(String[] args) throws Exception {
//        classTransfer loader1 = new classTransfer("loader1");
//        String target =
"C:\\Users\\CWM\\IdeaProjects\\test\\target\\classes\\com\\sec\\cwm\\testjavacla
ss.class";
        String className = "com.test.cwm.testPrivateclass1";
        Class clazz = loader1.findClass(target, className);
        byte[] bytes=loader1.loadClassData(target);
        System.out.println(clazz.getClassLoader()); //
cn.com.ccxi.jvm.test.TestCode16@773de2bd
    }
}
```

跨包与包内通用classloader

```
//classloader字节码转换成类
package com.sec.cwm;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import static com.sec.cwm.classTransfer.bytes;
public class testCrossClassloader {
```

```
    public static String testClassName ="com.sec.cwm.testJavaClass";
    public static byte[] testClassByte =bytes;
    public static class ClassloaderA extends ClassLoader{
        public ClassloaderA(ClassLoader parent){
            super(parent);
        }
        {
            defineClass(testClassName, testClassByte,0, testClassByte.length);
        }
    }
    public static class ClassloaderB extends ClassLoader{
        public ClassloaderB(ClassLoader parent){
            super(parent);
        }
        {
            defineClass(testClassName, testClassByte,0, testClassByte.length);
        }
    }
    public static void main(String args[]) throws ClassNotFoundException,
InstantiationException, IllegalAccessException, NoSuchMethodException,
InvocationTargetException {
        ClassLoader parentClassLoader=ClassLoader.getSystemClassLoader();
        ClassloaderA aClassloader=new ClassloaderA(parentClassLoader);
        ClassloaderB bClassloader=new ClassloaderB(parentClassLoader);
        Class<?> aClass  = Class.forName(testClassName, true, aClassloader);
        Class<?> aaClass = Class.forName(testClassName, true, aClassloader);
        Class<?> bClass  = Class.forName(testClassName, true, bClassloader);
        System.out.println("aClass == aaClass: " + (aClass == aaClass));
        System.out.println("aClass == bClass: " + (aClass == bClass));

        System.out.println("\n" + aClass.getName() + "方法清单: ");

        // 获取该类所有方法
        Method[] methods = aClass.getDeclaredMethods();

        for (Method method : methods) {
            System.out.println(method);
        }

        // 创建类实例
        Object instanceA = aClass.newInstance();

        // 获取hello方法
        Method helloMethod = aClass.getMethod("hello");

        // 调用hello方法
        String result = (String) helloMethod.invoke(instanceA);

        System.out.println("\n反射调用: " + testClassName + "类" +
helloMethod.getName() + "方法，返回结果: " + result);

    }
}
```

反射获取类

```
package com.sec.cwm;
import org.apache.commons.io.IOUtils;

import java.io.IOException;
import java.io.InputStream;
import java.lang.reflect.*;

public class simpleReflect {
    public static void main(String args[]) throws ClassNotFoundException,
NoSuchMethodException, InvocationTargetException, InstantiationException,
IllegalAccessException, IOException, NoSuchFieldException {
        // 获取Runtime类对象

        Class runtimeClass1 = Class.forName("java.lang.Runtime");

        // 获取构造方法

        Constructor constructor = runtimeClass1.getDeclaredConstructor();

        constructor.setAccessible(true);

        // 创建Runtime类示例，等价于 Runtime rt = new Runtime();

        Object runtimeInstance = constructor.newInstance();

        Method runtimeMethod = runtimeClass1.getMethod("exec",String.class);

//        Method[] methods=runtimeClass1.getDeclaredMethods();

// 调用exec方法，等价于 rt.exec(cmd);

        Process process = (Process) runtimeMethod.invoke(runtimeInstance,
"calc");

// 获取命令执行结果

        InputStream in = process.getInputStream();

// 输出命令执行结果

        System.out.println(IOUtils.toString(in, "UTF-8"));

//获取hackjava类对象

        Class runtimeClass2= Class.forName("com.sec.cwm.hackJava");

// 获取构造方法

        Constructor constructor1 = runtimeClass2.getDeclaredConstructor();

        constructor1.setAccessible(true);

//创建hackjava实例

        Object runtimeInstance1 = constructor1.newInstance();
```

```
//获取hackjava的amd变量
        Field fields=runtimeClass2.getDeclaredField("amd");
//取消封装
        fields.setAccessible(true);
//更改变量
        fields.set(runtimeInstance1,"no");
        System.out.println(fields.get(runtimeInstance1));
// 获取构造方法
        Class runtimeClass3=Class.forName("com.sec.cwm.testJavaClass");
        Constructor constructor2 = runtimeClass3.getDeclaredConstructor();
        constructor2.setAccessible(true);
//创建实例
        Object runtimeInstance2 = constructor2.newInstance();
//获取hello方法
        Method runtimeMethod1=runtimeClass3.getMethod("hello");//无参不需要传参，不用写参数类型
//输出返回值
        System.out.println(runtimeMethod1.invoke(runtimeInstance2));



//直接运行runtime
//
System.out.println(IOUtils.toString(Runtime.getRuntime().exec("whoami").getInputStream(), "UTF-8"));
    }
}
```

Unsafe类转化实例

```
package com.sec.cwm;


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

public class testCrossUnsafe {
```

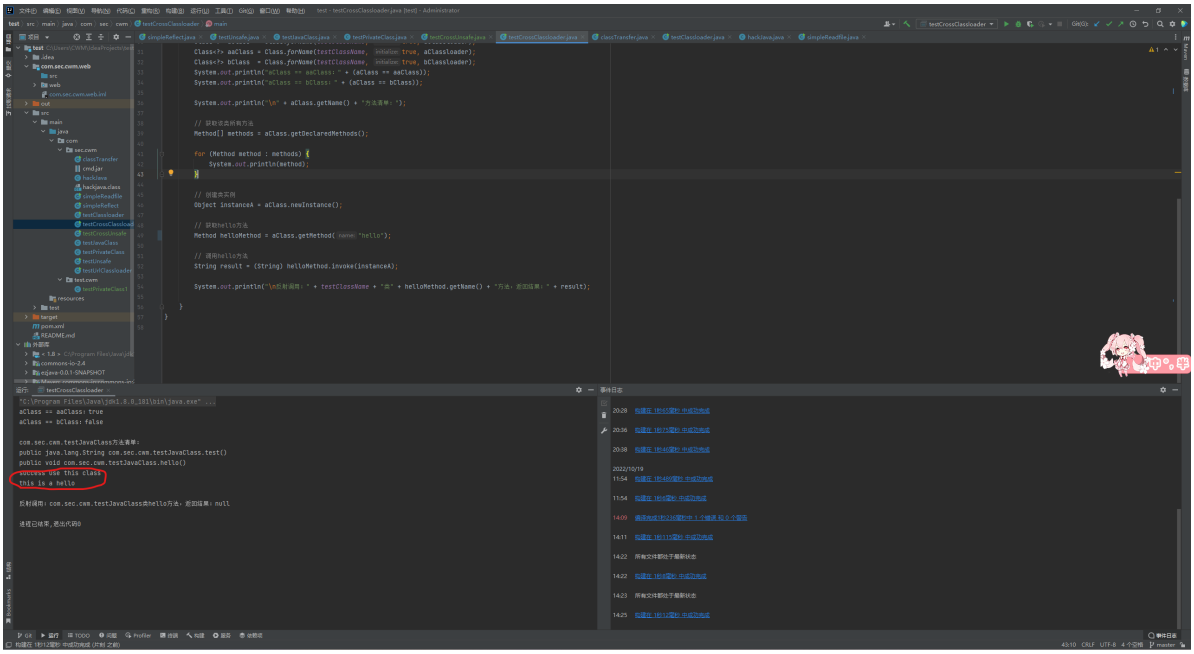
```
    public static void main(String args[]) throws ClassNotFoundException,
NoSuchMethodException, InvocationTargetException, InstantiationException,
IllegalAccessException {
        Class unsafeClass= Class.forName("sun.misc.Unsafe");
        Constructor unsafeConstructor=unsafeClass.getDeclaredConstructor();
        unsafeConstructor.setAccessible(true);
        Object unsafeObject=unsafeConstructor.newInstance();
        Method unsafeMethod=unsafeClass.getDeclaredMethod("allocateInstance",
Class.class);
        testJavaClass tjc=(testJavaClass)
unsafeMethod.invoke(unsafeObject,testJavaClass.class);
        tjc.hello();
    }
}
```

下面是testJavaClass源码

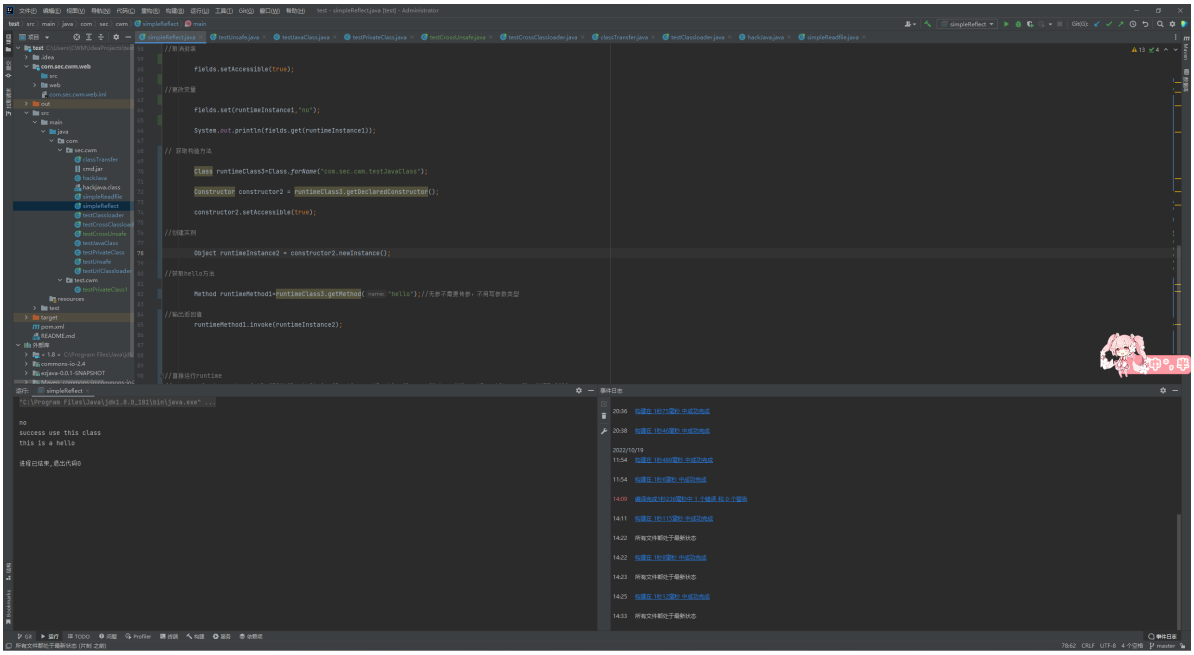
```
package com.sec.cwm;

public class testJavaClass {
    public void hello(){
        System.out.println("this is a hello");
    }
    public String test(){
        return "this is a test";
    }
    public testJavaClass(){
        System.out.println("success use this class");
    }
}
```

可以看到我没有对构造方法进行private限制，所以不论是classloader、普通reflect还是unsafe都可以获取hello()方法。

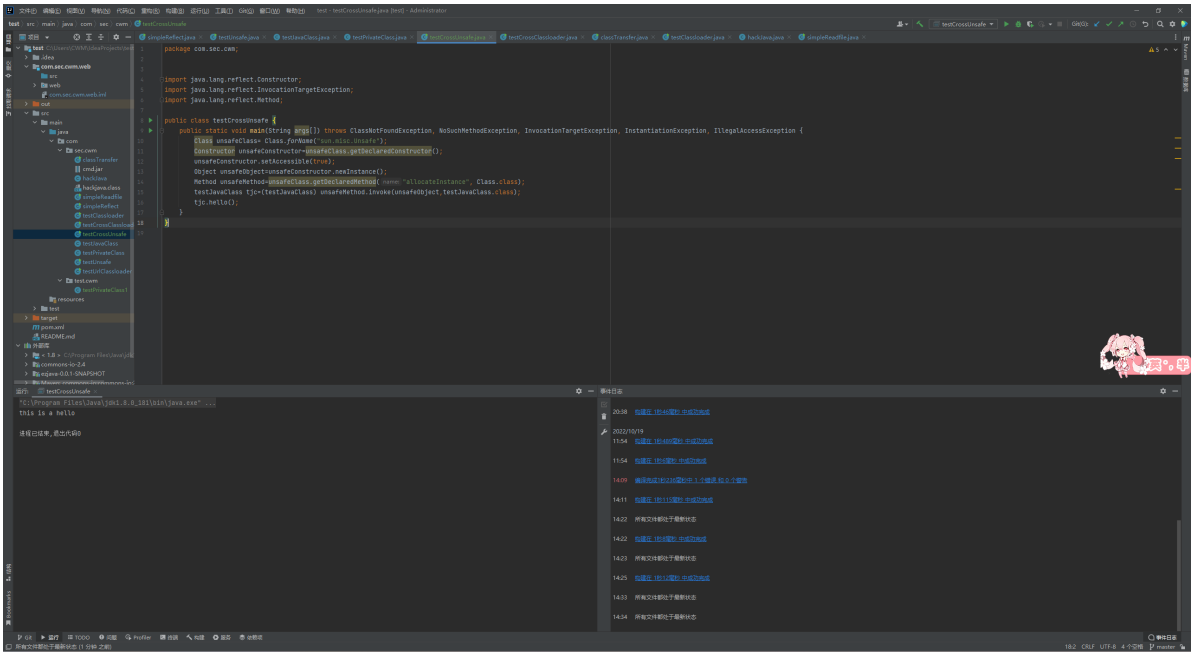
普通classloader



reflect 类调用



reflect Unsafe类调用allocateInstance方法来调用类方法



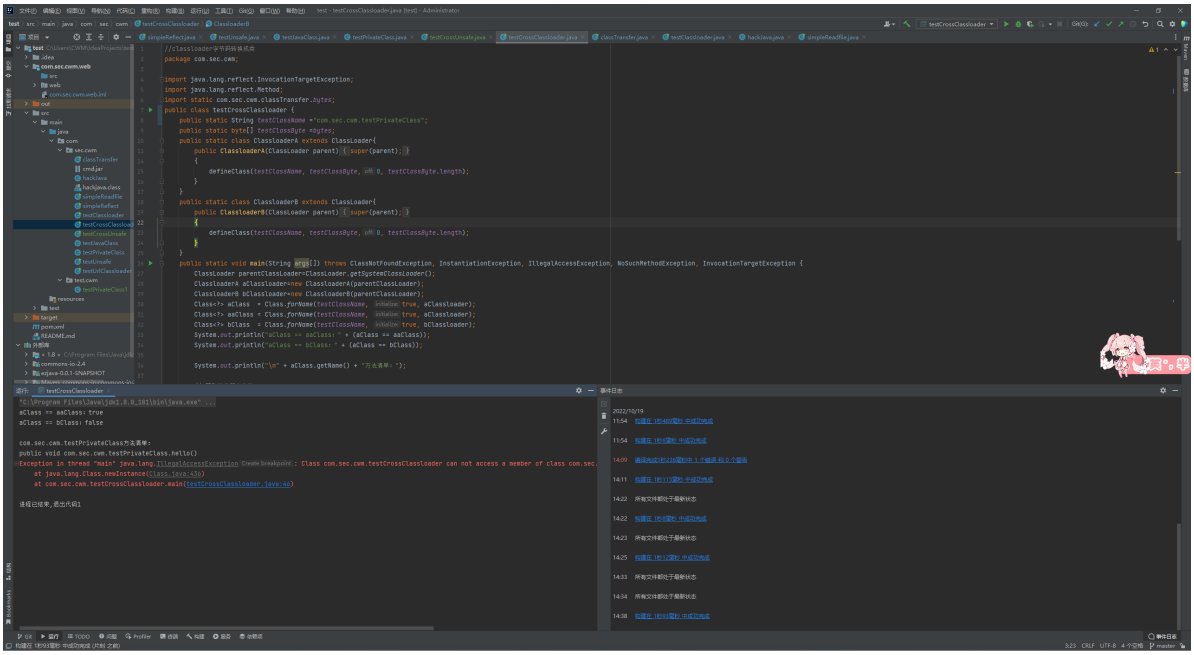
可以发现classloader和reflect都成功触发了构建方法，而Unsafe没有触发

我们换成包内private构建方法来看看

testPrivateClass

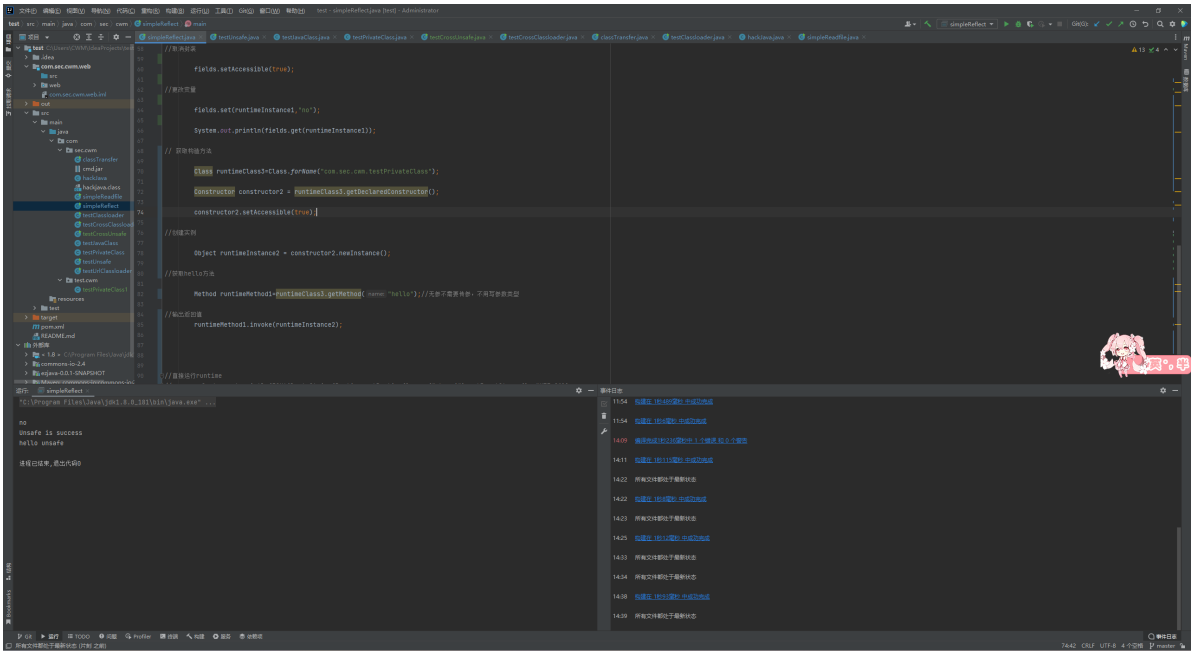
```
package com.sec.cwm;

public class testPrivateClass {
    private testPrivateClass(){
        System.out.println("Unsafe is success");
    }
    public void hello(){
         System.out.println("hello unsafe");
    }
}
```

classloader

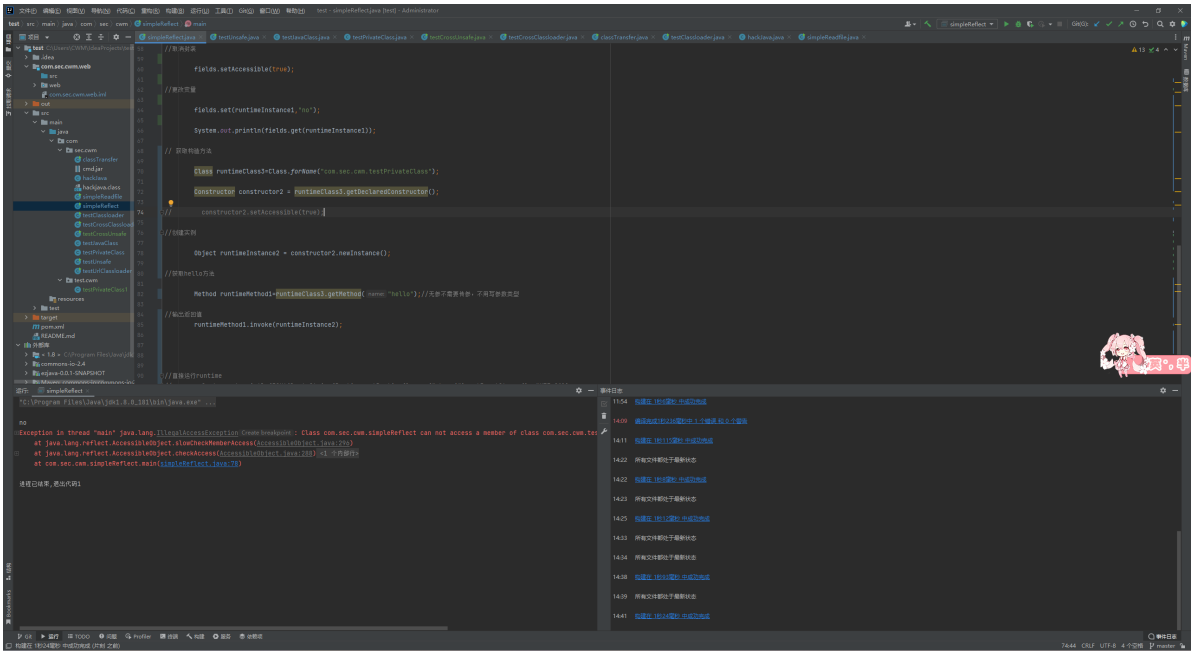


可以发现报错了，证明没有权限可以触发构造方法。

reflect

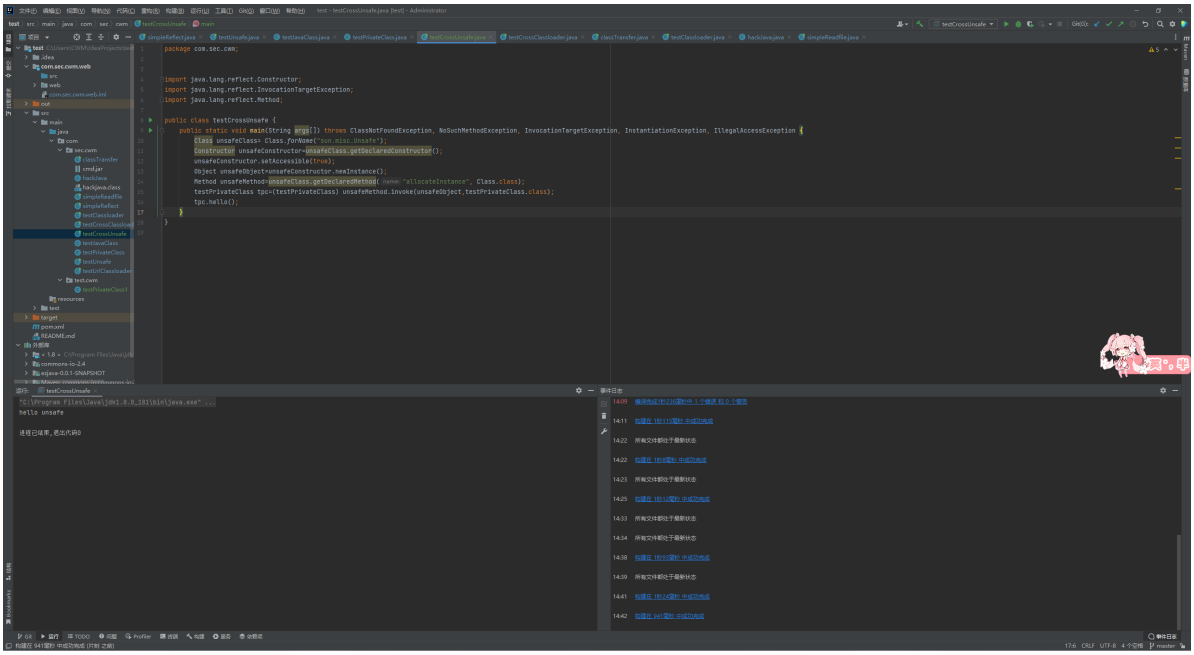


可以发现成功触发，究其原因在于constructor2.setAccessible(true)上，把类的构造方法更改为可访问。如果我们不添加这一句话：



弹出了一样的报错。

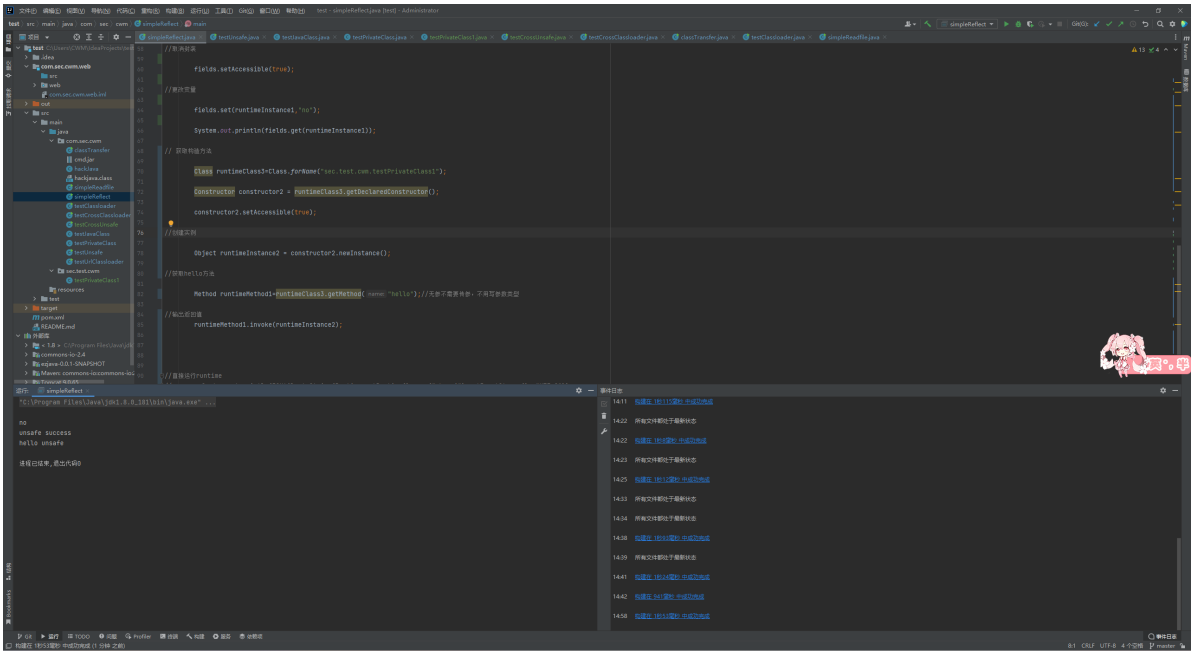
但如果我们用Unsafe的allocateInstance方法来调用的话：



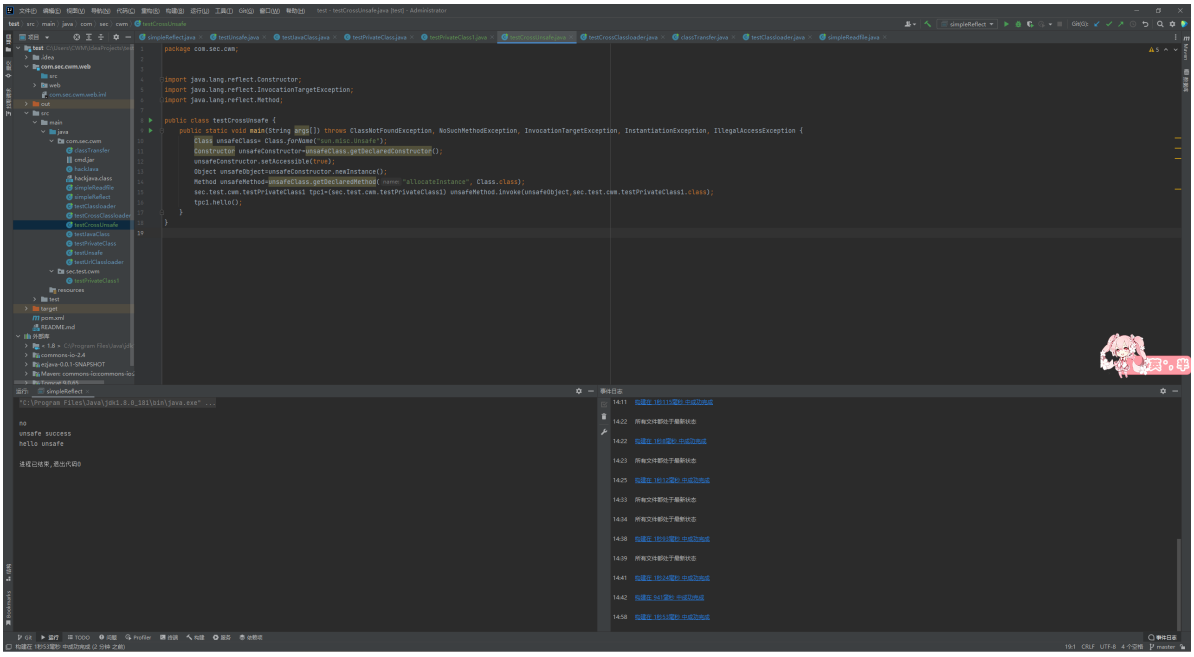
可以发现还是触发了的，这段代码中的setAccessible是把Unsafe的权限开放了，并不是testPrivateClass

这个时候如果更改为挎包进行的话，那么再来看看reflect方法与unsafe谁更有效。

发现构造器也是能调用的



Unsafe也一样可以调用



不过可以发现都没有调用构造方法，说明利用Unsafe创建实例可以绕过构造方法，让一些没有构造方法的类在反序列化利用的时候不受影响。

learn from :

1、Java魔法类：Unsafe应用解析 - 美团技术团队 (meituan.com)

2、聊聊Unsafe的一些使用技巧-51CTO.COM

3、[sun.misc.Unsafe · 攻击Java Web应用-Java Web安全] (javasec.org)