

Spring 学习总结（二）——静态代理、JDK 与 CGLIB 动态代理、AOP+IoC

目录

- [一、为什么需要代理模式](#)
- [二、静态代理](#)
- [三、动态代理，使用 JDK 内置的 Proxy 实现](#)
- [四、动态代理，使用 cglib 实现](#)
- [五、使用 Spring 实现 AOP](#)
- [六、使用 IOC 配置的方式实现 AOP](#)
- [七、使用 XML 配置 Spring AOP 切面](#)
- [八、示例下载](#)

一、为什么需要代理模式

假设需实现一个计算的类 **Math**、完成加、减、乘、除功能，如下所示：



```
1 package com. zhangguo. Spring041. aop01;
2
3 public class Math {
4     //加
5     public int add(int n1, int n2) {
6         int result=n1+n2;
7         System.out.println(n1+" "+n2+"="+result);
8         return result;
9     }
10
11
12     //减
13     public int sub(int n1, int n2) {
14         int result=n1-n2;
15         System.out.println(n1+"-"+n2+"="+result);
16         return result;
17     }
```

```
18
19 //乘
20 public int mut(int n1, int n2) {
21     int result=n1*n2;
22     System.out.println(n1+"X"+n2+"="+result);
23     return result;
24 }
25
26 //除
27 public int div(int n1, int n2) {
28     int result=n1/n2;
29     System.out.println(n1+"/"+n2+"="+result);
30     return result;
31 }
32 }
```



现在需求发生了变化, 要求项目中所有的类在执行方法时输出执行耗时。最直接的办法是修改源代码, 如下所示:



```
1 package com. zhangguo. Spring041. aop01;
2
3 import java. util. Random;
4
5 public class Math {
6     //加
7     public int add(int n1, int n2) {
8         //开始时间
9         long start=System. currentTimeMillis();
10        lazy();
11        int result=n1+n2;
12        System.out. println(n1+" "+n2+"="+result);
13        Long span= System. currentTimeMillis()-start;
14        System.out. println("共用时: "+span);
15        return result;
16    }
17
18    //减
19    public int sub(int n1, int n2) {
20        //开始时间
21        long start=System. currentTimeMillis();
22        lazy();
23        int result=n1-n2;
24        System.out. println(n1+"-"+n2+"="+result);
```

```
25         Long span= System.currentTimeMillis()-start;
26         System.out.println("共用时: "+span);
27         return result;
28     }
29
30     //乘
31     public int mut(int n1, int n2) {
32         //开始时间
33         long start=System.currentTimeMillis();
34         lazy();
35         int result=n1*n2;
36         System.out.println(n1+"X"+n2+"="+result);
37         Long span= System.currentTimeMillis()-start;
38         System.out.println("共用时: "+span);
39         return result;
40     }
41
42     //除
43     public int div(int n1, int n2) {
44         //开始时间
45         long start=System.currentTimeMillis();
46         lazy();
47         int result=n1/n2;
48         System.out.println(n1+"/"+n2+"="+result);
49         Long span= System.currentTimeMillis()-start;
50         System.out.println("共用时: "+span);
51         return result;
52     }
53
54     //模拟延时
55     public void lazy()
56     {
57         try {
58             int n=(int)new Random().nextInt(500);
59             Thread.sleep(n);
60         } catch (InterruptedException e) {
61             e.printStackTrace();
62         }
63     }
64 }
```



测试运行:



```
package com. zhangguo. Spring041. aop01;
```

```
public class Test {
```

```
    @org. junit. Test
```

```
    public void test01()
```

```
    {
```

```
        Math math=new Math();
```

```
        int n1=100,n2=5;
```

```
        math.add(n1, n2);
```

```
        math.sub(n1, n2);
```

```
        math.mut(n1, n2);
```

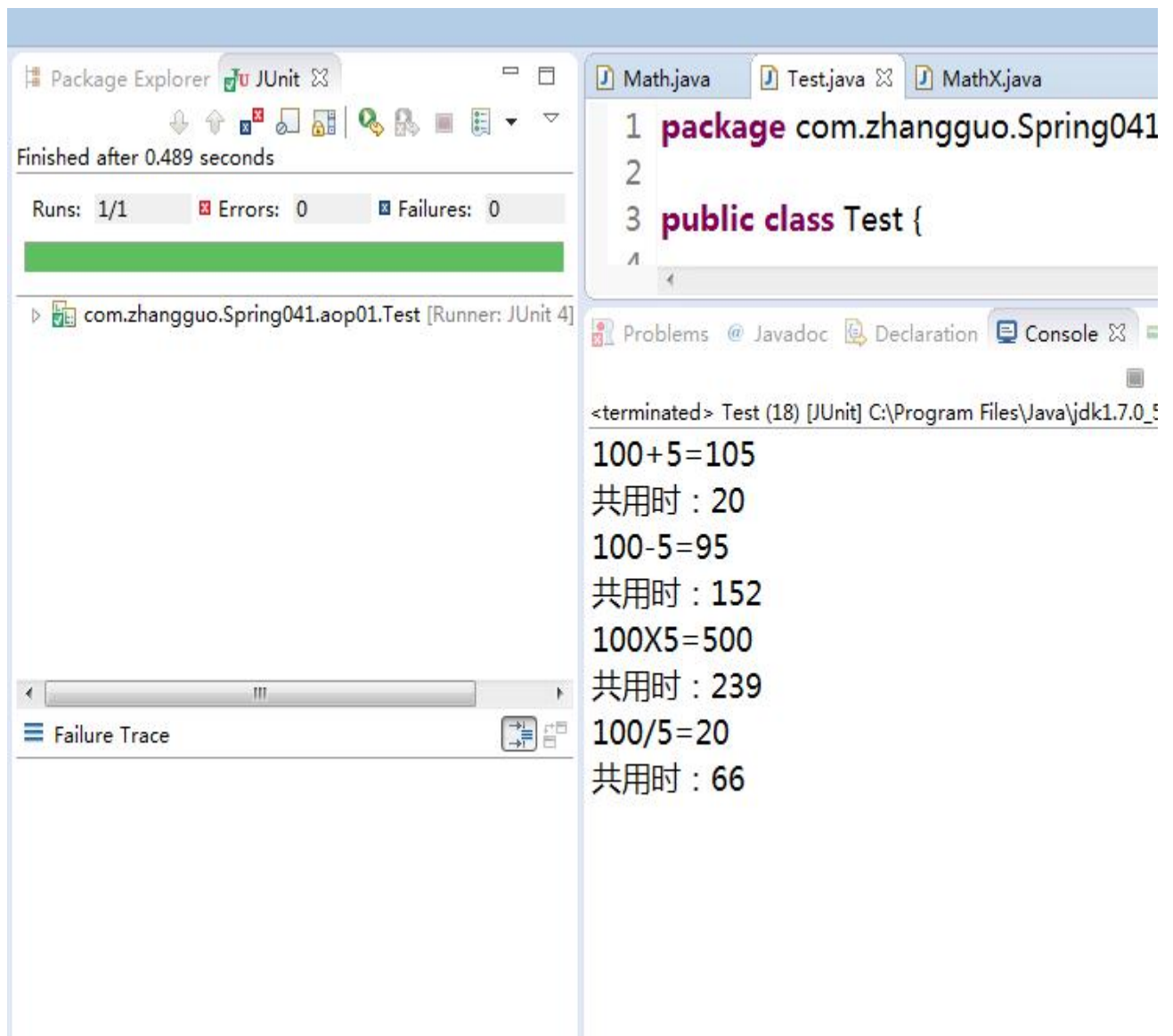
```
        math.div(n1, n2);
```

```
    }
```

```
}
```



运行结果:



缺点：

- 1、工作量特别大，如果项目中有多个类，多个方法，则要修改多次。
- 2、违背了设计原则：开闭原则（OCP），对扩展开放，对修改关闭，而为了增加功能把每个方法都修改了，也不便于维护。
- 3、违背了设计原则：单一职责（SRP），每个方法除了要完成自己本身的功能，还要计算耗时、延时；每一个方法引起它变化的原因就有多种。
- 4、违背了设计原则：依赖倒转（DIP），抽象不应该依赖细节，两者都应该依赖抽象。而在 `Test` 类中，`Test` 与 `Math` 都是细节。

使用静态代理可以解决部分问题。

二、静态代理

1、定义抽象主题接口。



```
package com.zhangguo.Spring041.aop02;
```

```
/**
 * 接口
 * 抽象主题
 */
public interface IMath {
    //加
    int add(int n1, int n2);

    //减
    int sub(int n1, int n2);

    //乘
    int mut(int n1, int n2);

    //除
    int div(int n1, int n2);
}
```



2、主题类，算术类，实现抽象接口。



```
package com.zhangguo.Spring041.aop02;
```


```
/**
 * 被代理的目标对象
 * 真实主题
 */
public class Math implements IMath {
    //加
    public int add(int n1, int n2) {
        int result=n1+n2;
        System.out.println(n1+" "+n2+"="+result);
        return result;
    }
}
```

```
}


//减
public int sub(int n1, int n2) {
    int result=n1-n2;
    System.out.println(n1+"-"+n2+"="+result);
    return result;
}

//乘
public int mut(int n1, int n2) {
    int result=n1*n2;
    System.out.println(n1+"X"+n2+"="+result);
    return result;
}

//除
public int div(int n1, int n2) {
    int result=n1/n2;
    System.out.println(n1+"/"+n2+"="+result);
    return result;
}
}
```



3、代理类



```
1 package com. zhangguo. Spring041. aop02;
2
3 import java. util. Random;
4
5 /**
6  * 静态代理类
7  */
8 public class MathProxy implements IMath {
9
10     //被代理的对象
11     IMath math=new Math();
12
13     //加
14     public int add(int n1, int n2) {
15         //开始时间
16         long start=System.currentTimeMillis();
```

```
17     lazy();
18     int result=math.add(n1, n2);
19     Long span= System.currentTimeMillis()-start;
20     System.out.println("共用时: "+span);
21     return result;
22 }
23
24 //减法
25 public int sub(int n1, int n2) {
26     //开始时间
27     long start=System.currentTimeMillis();
28     lazy();
29     int result=math.sub(n1, n2);
30     Long span= System.currentTimeMillis()-start;
31     System.out.println("共用时: "+span);
32     return result;
33 }
34
35 //乘
36 public int mut(int n1, int n2) {
37     //开始时间
38     long start=System.currentTimeMillis();
39     lazy();
40     int result=math.mut(n1, n2);
41     Long span= System.currentTimeMillis()-start;
42     System.out.println("共用时: "+span);
43     return result;
44 }
45
46 //除
47 public int div(int n1, int n2) {
48     //开始时间
49     long start=System.currentTimeMillis();
50     lazy();
51     int result=math.div(n1, n2);
52     Long span= System.currentTimeMillis()-start;
53     System.out.println("共用时: "+span);
54     return result;
55 }
56
57 //模拟延时
58 public void lazy()
59 {
60     try {
```



```
61         int n=(int)new Random().nextInt(500);
62         Thread.sleep(n);
63     } catch (InterruptedException e) {
64         e.printStackTrace();
65     }
66 }
67 }
```

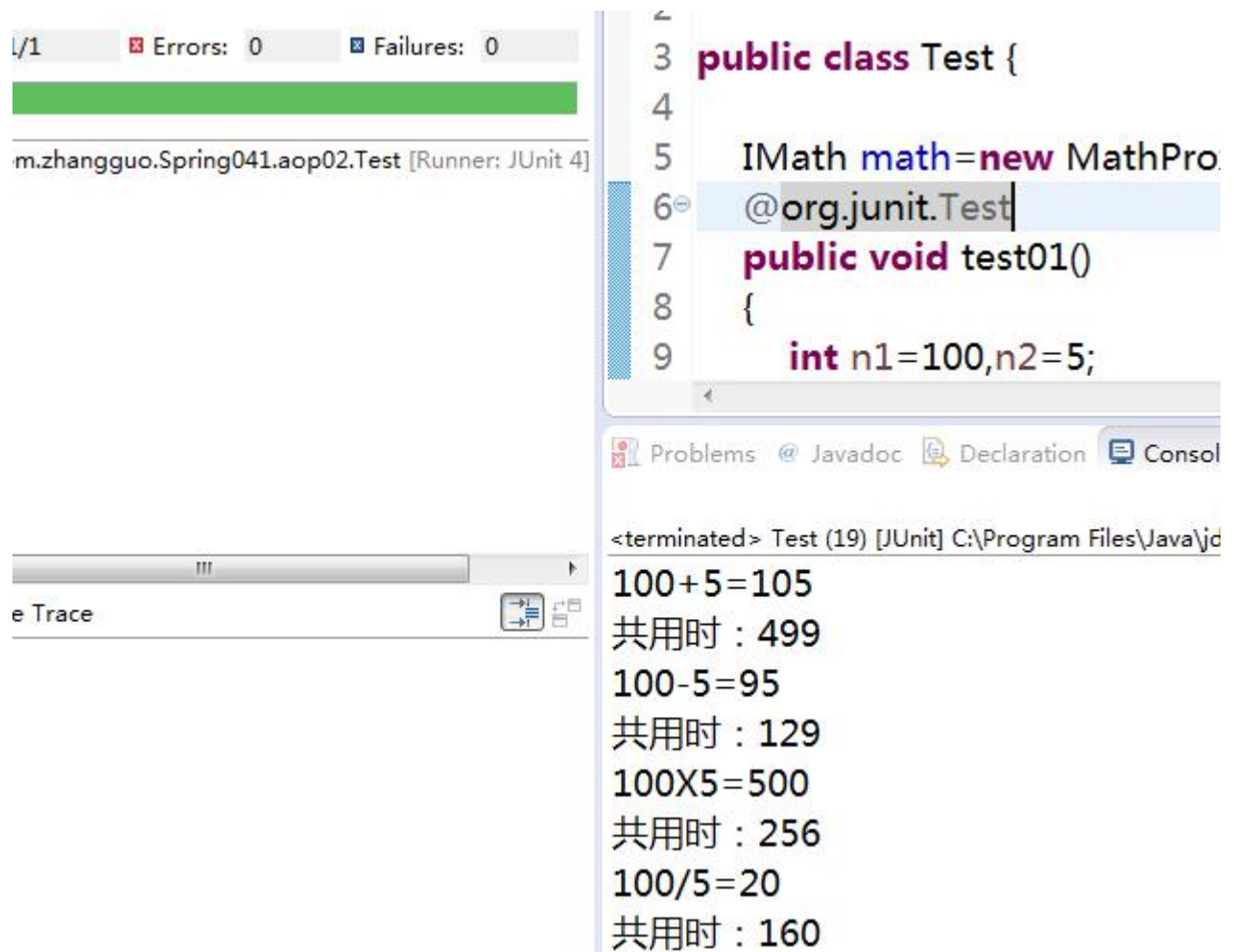


4、测试运行



```
1 package com.zhangguo.Spring041.aop02;
2
3 public class Test {
4
5     IMath math=new MathProxy();
6     @org.junit.Test
7     public void test01()
8     {
9         int n1=100,n2=5;
10        math.add(n1, n2);
11        math.sub(n1, n2);
12        math.mul(n1, n2);
13        math.div(n1, n2);
14    }
15 }
```





The screenshot shows an IDE with a Java test class `Test` and its execution output. The class `Test` is defined as follows:

```
public class Test {  
    IMath math=new MathPro.  
    @org.junit.Test  
    public void test01()  
    {  
        int n1=100,n2=5;  
    }  
}
```

The execution output, titled "<terminated> Test (19) [JUnit] C:\Program Files\Java\jdk", shows the following results:

```
100+5=105  
共用时：499  
100-5=95  
共用时：129  
100X5=500  
共用时：256  
100/5=20  
共用时：160
```

5、小结

通过静态代理，是否完全解决了上述的 4 个问题：

已解决：

5.1、解决了“开闭原则(OCP)”的问题，因为并没有修改 `Math` 类，而扩展出了 `MathProxy` 类。

5.2、解决了“依赖倒转(DIP)”的问题，通过引入接口。

5.3、解决了“单一职责(SRP)”的问题，`Math` 类不再需要去计算耗时与延时操作，但从某些方面讲 `MathProxy` 还是存在该问题。

未解决：

5.4、如果项目中有多个类，则需要编写多个代理类，工作量大，不好修改，不好维护，不能应对变化。

如果要解决上面的问题，可以使用动态代理。

三、动态代理，使用 JDK 内置的 Proxy 实现

只需要一个代理类，而不是针对每个类编写代理类。

在上一个示例中修改代理类 MathProxy 如下：



```
1 package com.zhangguo.Spring041.aop03;
2
3 import java.lang.reflect.InvocationHandler;
4 import java.lang.reflect.Method;
5 import java.lang.reflect.Proxy;
6 import java.util.Random;
7
8 /**
9  * 动态代理类
10 */
11 public class DynamicProxy implements InvocationHandler {
12
13     //被代理的对象
14     Object targetObject;
15
16     /**
17      * 获得被代理后的对象
18      * @param object 被代理的对象
19      * @return 代理后的对象
20      */
21     public Object getProxyObject(Object object) {
22         this.targetObject=object;
23         return Proxy.newProxyInstance(
24             targetObject.getClass().getClassLoader(), //类加载器
25             targetObject.getClass().getInterfaces(), //获得被代理对象的所有接口
26             this); //InvocationHandler 对象
27     //loader:一个 ClassLoader 对象，定义了由哪个 ClassLoader 对象来生成代理对象进行加载
```

```
28         //interfaces:一个 Interface 对象的数组, 表示的是我将来给我需要代理的对象提供一组什么接口, 如果我提供了一组接口给它, 那么这个代理对象就宣称实现了该接口(多态), 这样我就能调用这组接口中的方法了
29         //h:一个 InvocationHandler 对象, 表示的是当我这个动态代理对象在调用方法的时候, 会关联到哪一个 InvocationHandler 对象上, 间接通过 invoke 来执行
30     }
31
32
33     /**
34      * 当用户调用对象中的每个方法时都通过下面的方法执行, 方法必须在接口
35      * proxy 被代理后的对象
36      * method 将要被执行的方法信息(反射)
37      * args 执行方法时需要的参数
38      */
39     public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
40         //被织入的内容, 开始时间
41         long start=System.currentTimeMillis();
42         lazy();
43
44         //使用反射在目标对象上调用方法并传入参数
45         Object result=method.invoke(targetObject, args);
46
47         //被织入的内容, 结束时间
48         Long span= System.currentTimeMillis()-start;
49         System.out.println("共用时: "+span);
50
51         return result;
52     }
53
54     //模拟延时
55     public void lazy()
56     {
57         try {
58             int n=(int)new Random().nextInt(500);
59             Thread.sleep(n);
60         } catch (InterruptedException e) {
61             e.printStackTrace();
62         }
63     }
64
65 }
```

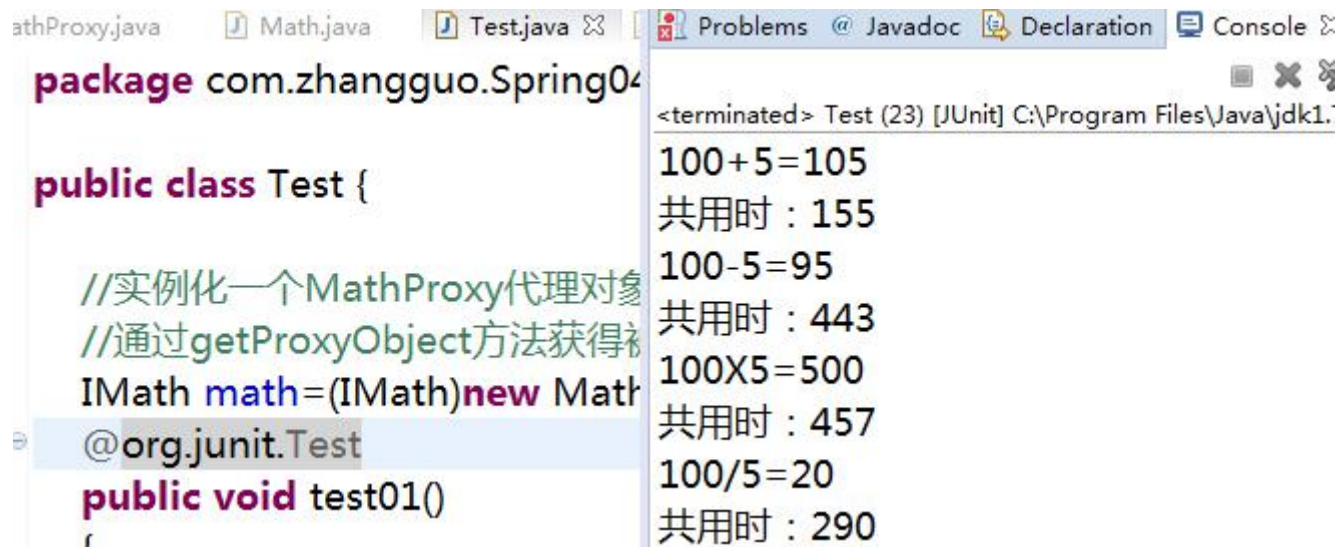


测试运行:



```
1 package com.zhangguo.Spring041.aop03;
2
3 public class Test {
4
5     //实例化一个 MathProxy 代理对象
6     //通过 getProxyObject 方法获得被代理后的对象
7     IMath math=(IMath)new DynamicProxy().getProxyObject(new
Math());
8     @org.junit.Test
9     public void test01()
10    {
11        int n1=100,n2=5;
12        math.add(n1, n2);
13        math.sub(n1, n2);
14        math.mul(n1, n2);
15        math.div(n1, n2);
16    }
17
18    IMessage message=(IMessage) new
DynamicProxy().getProxyObject(new Message());
19    @org.junit.Test
20    public void test02()
21    {
22        message.message();
23    }
24 }
```





The screenshot shows an IDE with two panes. The left pane displays the source code of a Java class named `Test` in the package `com.zhangguo.Spring04`. The code includes a comment in Chinese about proxying a `MathProxy` object, an annotation `@org.junit.Test`, and a method `test01()`. The right pane shows the console output of the test, which lists several arithmetic operations and their execution times in milliseconds.

```
package com.zhangguo.Spring04

public class Test {

    //实例化一个MathProxy代理对象
    //通过getProxyObject方法获得代理对象
    IMath math=(IMath)new MathProxy()

    @org.junit.Test
    public void test01()
    {
        //...
    }
}
```

```
<terminated> Test (23) [JUnit] C:\Program Files\Java\jdk1.8.0_101\bin\java.exe
100+5=105
共用时：155
100-5=95
共用时：443
100X5=500
共用时：457
100/5=20
共用时：290
```

小结：

JDK 内置的 Proxy 动态代理可以在运行时动态生成字节码，而没必要针对每个类编写代理类。中间主要使用到了一个接口 `InvocationHandler` 与 `Proxy.newProxyInstance` 静态方法，参数说明如下：

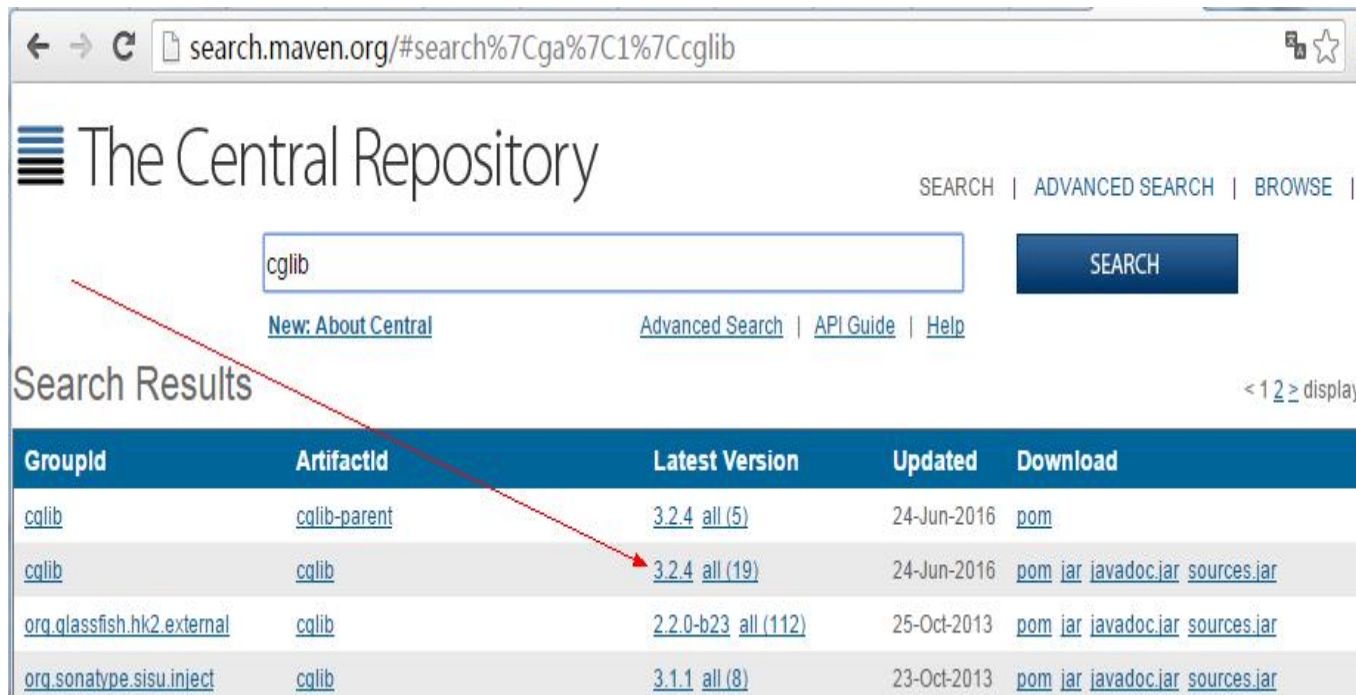
使用内置的 Proxy 实现动态代理有一个问题：被代理的类必须实现接口，未实现接口则没办法完成动态代理。

如果项目中有些类没有实现接口，则不应该为了实现动态代理而刻意去抽出一些没有实例意义的接口，通过 cglib 可以解决这个问题。

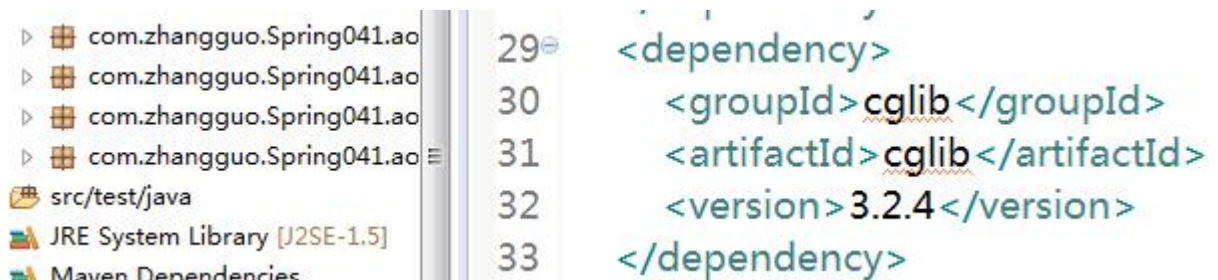
四、动态代理，使用 cglib 实现

CGLIB(Code Generation Library)是一个开源项目,是一个强大的，高性能，高质量的 Code 生成类库，它可以在运行期扩展 Java 类与实现 Java 接口，通俗说 cglib 可以在运行时动态生成字节码。

4.1、引用 cglib，通过 maven



修改 pom.xml 文件，添加依赖



保存 pom.xml 配置文件，将自动从共享资源库下载 cglib 所依赖的 jar 包，主要有如下几个：



4.2、使用 cglib 完成动态代理，大概的原理是：cglib 继承被代理的类，重写方法，织入通知，动态生成字节码并运行，因为是继承所以 final 类是没有办法动态代理的。具体实现如下：



```
1 package com. zhangguo. Spring041. aop04;
```



```
2
3 import java.lang.reflect.Method;
4 import java.util.Random;
5
6 import net.sf.cglib.proxy.Enhancer;
7 import net.sf.cglib.proxy.MethodInterceptor;
8 import net.sf.cglib.proxy.MethodProxy;
9
10 /*
11  * 动态代理类
12  * 实现了一个方法拦截器接口
13  */
14 public class DynamicProxy implements MethodInterceptor {
15
16     // 被代理对象
17     Object targetObject;
18
19     //Generate a new class if necessary and uses the specified
20     //callbacks (if any) to create a new object instance.
21     //Uses the no-arg constructor of the superclass.
22     //动态生成一个新的类，使用父类的无参构造方法创建一个指定了特定
23     //回调的代理实例
24     public Object getProxyObject(Object object) {
25         this.targetObject = object;
26         //增强器，动态代码生成器
27         Enhancer enhancer=new Enhancer();
28         //回调方法
29         enhancer.setCallback(this);
30         //设置生成类的父类类型
31         enhancer.setSuperclass(targetObject.getClass());
32         //动态生成字节码并返回代理对象
33         return enhancer.create();
34     }
35
36     // 拦截方法
37     public Object intercept(Object object, Method method, Object[]
38     args, MethodProxy methodProxy) throws Throwable {
39         // 被织入的横切内容，开始时间 before
40         long start = System.currentTimeMillis();
41         lazy();
42
43         // 调用方法
44         Object result = methodProxy.invoke(targetObject, args);
```



```
43         // 被织入的横切内容, 结束时间
44         Long span = System.currentTimeMillis() - start;
45         System.out.println("共用时: " + span);
46
47         return result;
48     }
49
50     // 模拟延时
51     public void lazy() {
52         try {
53             int n = (int) new Random().nextInt(500);
54             Thread.sleep(n);
55         } catch (InterruptedException e) {
56             e.printStackTrace();
57         }
58     }
59
60 }
```



测试运行:



```
package com.zhangguo.Spring041.aop04;

public class Test {
    //实例化一个 DynamicProxy 代理对象
    //通过 getProxyObject 方法获得被代理后的对象
    Math math=(Math)new DynamicProxy().getProxyObject(new Math());
    @org.junit.Test
    public void test01()
    {
        int n1=100,n2=5;
        math.add(n1, n2);
        math.sub(n1, n2);
        math.mul(n1, n2);
        math.div(n1, n2);
    }
    //另一个被代理的对象, 不再需要重新编辑代理代码
    Message message=(Message) new DynamicProxy().getProxyObject(new
    Message());
    @org.junit.Test
    public void test02()
    {
        message.message();
    }
}
```

```

    }
}

```

运行结果:

The screenshot shows an IDE with two tabs: `DynamicProxy.java` and `Test.java`. The `Test.java` tab is active, displaying the following code:

```

1 package com.zhangguo.Spring04
2
3 public class Test {
4     //实例化一个DynamicProxy代理
5     //通过getProxyObject方法获得被代理对象
6     Math math=(Math)new DynamicProxy().getProxyObject(new Math());
7     @org.junit.Test
8     public void test01()
9     {
10         int n1=100,n2=5;
11         math.add(n1, n2);
12         math.sub(n1, n2);
13         math.mul(n1, n2);
14         math.div(n1, n2);
15     }
16     //另一个被代理的对象,不再需要重新编辑代理代码
17     Message message=(Message) new DynamicProxy().getProxyObject(new Message());

```

The `Console` tab on the right shows the execution output:

```

<terminated> Test (24) [JUnit] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (2016
Hello AOP
共用时 : 78
100+5=105
共用时 : 184
100-5=95
共用时 : 345
100X5=500
共用时 : 334
100/5=20
共用时 : 325

```

4.3、小结

使用 `cglib` 可以实现动态代理,即使被代理的类没有实现接口,但被代理的类必须不是 `final` 类。

五、使用 Spring 实现 AOP

SpringAOP 中,通过 `Advice` 定义横切逻辑, Spring 中支持 5 种类型的 `Advice`:

通知类型	连接点	实现接口
前置通知	方法方法前	org.springframework.aop.MethodBeforeAdvice
后置通知	方法后	org.springframework.aop.AfterReturningAdvice
环绕通知	方法前后	org.aopalliance.intercept.MethodInterceptor
异常抛出通知	方法抛出异常	org.springframework.aop.ThrowsAdvice
引介通知	类中增加新的方法属性	org.springframework.aop.IntroductionInterceptor

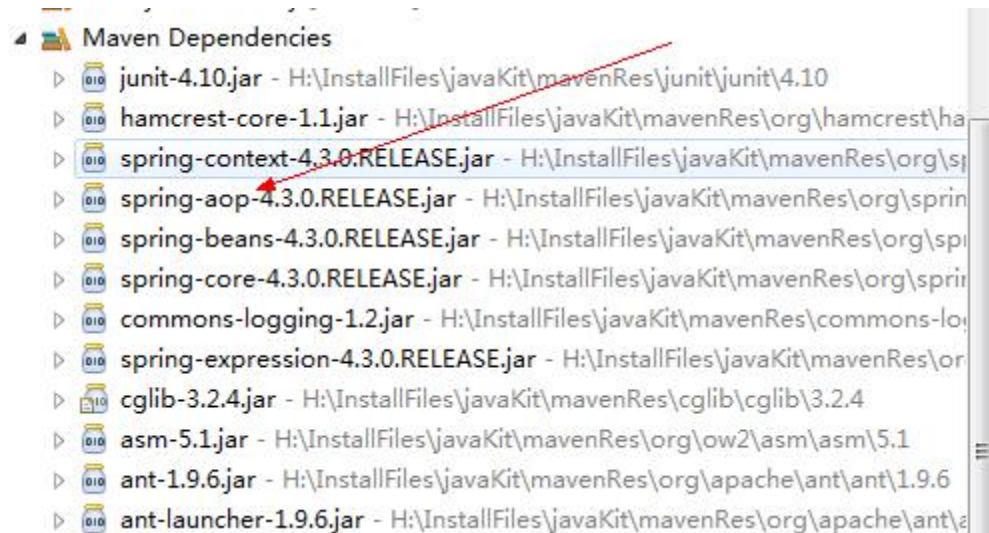
5.1、新建 一个 Maven 项目，在项目中引入 Spring 核心库与 AOP，修改 pom.xml 文件，
在 dependencies 中增加如下节点：



```
<dependency>
  <groupId>org. springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>4. 3. 0. RELEASE</version>
</dependency>
```



当保存 pom.xml 文件时会从远程共享库自动将需要引入的 jar 包下载到本地并引入项目中：



5.2、定义通知（Advice）

前置通知



```
1 package com. zhangguo. Spring041. aop05;
2
3 import java. lang. reflect. Method;
4
5 import org. springframework. aop. MethodBeforeAdvice;
6
7 /**
8  * 前置通知
9  */
10 public class BeforeAdvice implements MethodBeforeAdvice {
11
12     /**
13      * method 方法信息
14      * args 参数
15      * target 被代理的目标对象
16      */
17     public void before(Method method, Object[] args, Object target)
18     throws Throwable {
19         System. out. println("-----前置通知
20         -----");
21     }
22 }
```



后置通知



```
1 package com. zhangguo. Spring041. aop05;
2
3 import java. lang. reflect. Method;
4
5 import org. springframework. aop. AfterReturningAdvice;
6
7 /**
8  * 后置通知
9  *
10 */
11 public class AfterAdvice implements AfterReturningAdvice {
12
13     /*
14      * returnValue 返回值
15      * method 被调用的方法
16      * args 方法参数
17      * target 被代理对象
18      */
19     public void afterReturning(Object returnValue, Method method,
20 Object[] args, Object target) throws Throwable {
21         System.out.println("-----后置通知
22 -----");
23     }
24 }
```



环绕通知



```
1 package com. zhangguo. Spring041. aop05;
2
3 import org. aopalliance. intercept. MethodInterceptor;
4 import org. aopalliance. intercept. MethodInvocation;
5
6 /**
7  * 环绕通知
8  * 方法拦截器
9  *
10 */
11 public class SurroundAdvice implements MethodInterceptor {
12
13     public Object invoke(MethodInvocation i) throws Throwable {
14         //前置横切逻辑
15     }
16 }
```

```

15      System.out.println("方法" + i.getMethod() + " 被调用在对象"
+ i.getThis() + "上, 参数 " + i.getArguments());
16      //方法调用
17      Object ret = i.proceed();
18      //后置横切逻辑
19      System.out.println("返回值: " + ret);
20      return ret;
21  }
22 }

```



5.3、创建代理工厂、设置被代理对象、添加通知。



```

1 package com. zhangguo. Spring041. aop05;
2
3 import org. springframework. aop. framework. ProxyFactory;
4
5 public class Test {
6
7     @org. junit. Test
8     public void test01()
9     {
10         //实例化 Spring 代理工厂
11         ProxyFactory factory=new ProxyFactory();
12         //设置被代理的对象
13         factory.setTarget(new Math());
14         //添加通知, 横切逻辑
15         factory.addAdvice(new BeforeAdvice());
16         factory.addAdvice(new AfterAdvice());
17         factory.addAdvice(new SurroundAdvice());
18         //从代理工厂中获得代理对象
19         IMath math=(IMath) factory.getProxy();
20         int n1=100, n2=5;
21         math.add(n1, n2);
22         math.sub(n1, n2);
23         math.mul(n1, n2);
24         math.div(n1, n2);
25     }
26     @org. junit. Test
27     public void test02()
28     {
29         //message.message();
30     }

```

```
31 }
```



运行结果:

```

Problems @ Javadoc Declaration Console Progress
<terminated> Test (25) [JUnit] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (2016年6月28日 下午3:02:10)
-----前置通知-----
方法public int com.zhangguo.Spring041.aop05.Math.add(int,int) 被调用在对象com.zhangguo.Sp
100+5=105
返回值 : 105
-----后置通知-----
-----前置通知-----
方法public int com.zhangguo.Spring041.aop05.Math.sub(int,int) 被调用在对象com.zhangguo.Sp
100-5=95
返回值 : 95
-----后置通知-----
-----前置通知-----
方法public int com.zhangguo.Spring041.aop05.Math.mul(int,int) 被调用在对象com.zhangguo.Sp
100X5=500
返回值 : 500
-----后置通知-----
-----前置通知-----
方法public int com.zhangguo.Spring041.aop05.Math.div(int,int) 被调用在对象com.zhangguo.Sp
100/5=20

```

5.4、封装代理创建逻辑

在上面的示例中如果要代理不同的对象需要反复创建 **ProxyFactory** 对象, 代码会冗余。同样以实现方法耗时为例代码代码如下:

5.4.1、创建一个环绕通知:



```

1 package com. zhangguo. Spring041. aop05;
2
3 import java. util. Random;
4
5 import org. aopalliance. intercept. MethodInterceptor;

```



```
6 import org.aopalliance.intercept.MethodInvocation;
7
8 /**
9  * 用于完成计算方法执行时长的环绕通知
10 */
11 public class TimeSpanAdvice implements MethodInterceptor {
12
13     public Object invoke(MethodInvocation invocation) throws
Throwable {
14         // 被织入的横切内容, 开始时间 before
15         long start = System.currentTimeMillis();
16         lazy();
17
18         //方法调用
19         Object result = invocation.proceed();
20
21         // 被织入的横切内容, 结束时间
22         Long span = System.currentTimeMillis() - start;
23         System.out.println("共用时: " + span);
24
25         return result;
26     }
27
28     // 模拟延时
29     public void lazy() {
30         try {
31             int n = (int) new Random().nextInt(500);
32             Thread.sleep(n);
33         } catch (InterruptedException e) {
34             e.printStackTrace();
35         }
36     }
37 }
```



5.4.2、封装动态代理类



```
1 package com.zhangguo.Spring041.aop05;
2
3 import org.springframework.aop.framework.ProxyFactory;
4
5 /**
6  * 动态代理类
```



```
7  *
8  */
9  public abstract class DynamicProxy {
10     /**
11      * 获得代理对象
12      * @param object 被代理的对象
13      * @return 代理对象
14      */
15     public static Object getProxy(Object object) {
16         //实例化 Spring 代理工厂
17         ProxyFactory factory=new ProxyFactory();
18         //设置被代理的对象
19         factory.setTarget(object);
20         //添加通知, 横切逻辑
21         factory.addAdvice(new TimeSpanAdvice());
22         return factory.getProxy();
23     }
24 }
```



5.4.3、测试运行



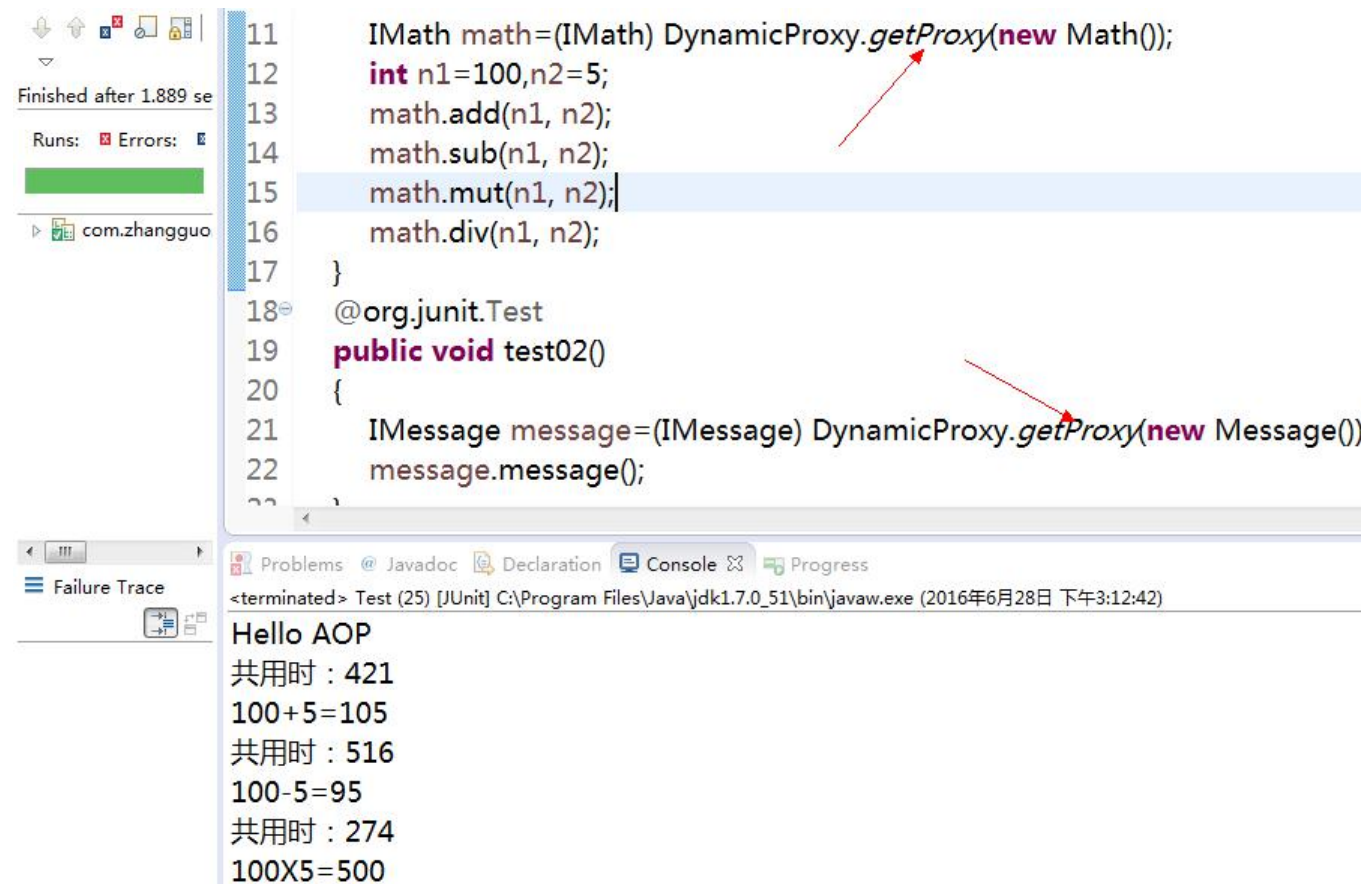
```
1  package com. zhangguo. Spring041. aop05;
2
3  import org. springframework. aop. framework. ProxyFactory;
4
5  public class Test {
6
7      @org. junit. Test
8      public void test01()
9      {
10         //从代理工厂中获得代理对象
11         IMath math=(IMath) DynamicProxy. getProxy(new Math());
12         int n1=100,n2=5;
13         math.add(n1, n2);
14         math.sub(n1, n2);
15         math.mut(n1, n2);
16         math.div(n1, n2);
17     }
18     @org. junit. Test
19     public void test02()
20     {
```

```

21     IMessage message=(IMessage) DynamicProxy.getProxy(new
Message());
22     message.message();
23 }
24 }

```

运行结果:



The screenshot shows an IDE with a Java file named `com.zhangguo`. The code defines an `IMath` interface and a `DynamicProxy` class that implements it. The `test02` method in the `DynamicProxy` class calls `DynamicProxy.getProxy(new Math())` to create a proxy for the `Math` class. The output window shows the results of the `test02` method, including the execution time for each operation and the results of the calculations.

```

11     IMath math=(IMath) DynamicProxy.getProxy(new Math());
12     int n1=100,n2=5;
13     math.add(n1, n2);
14     math.sub(n1, n2);
15     math.mul(n1, n2);
16     math.div(n1, n2);
17 }
18 @org.junit.Test
19 public void test02()
20 {
21     IMessage message=(IMessage) DynamicProxy.getProxy(new Message());
22     message.message();
23 }

```

Finished after 1.889 se
Runs: 1 Errors: 0
com.zhangguo

Failure Trace

Problems Javadoc Declaration Console Progress

<terminated> Test (25) [JUnit] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (2016年6月28日 下午3:12:42)

Hello AOP
共用时 : 421
100+5=105
共用时 : 516
100-5=95
共用时 : 274
100X5=500

六、使用 IOC 配置的方式实现 AOP

6.1、引入 Spring IOC 的核心 jar 包，方法与前面相同。

6.2、创建 IOC 的配置文件 `beans.xml`，内容如下：

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:p="http://www.springframework.org/schema/p"

```

```

4
xsi:schemaLocation="http://www.springframework.org/schema/beans
5
http://www.springframework.org/schema/beans/spring-beans.xsd">
6    <!-- 被代理的目标对象 -->
7    <bean id="target"
class="com.zhangguo.Spring041.aop06.Math"></bean>
8    <!--通知、横切逻辑-->
9    <bean id="advice"
class="com.zhangguo.Spring041.aop06.AfterAdvice"></bean>
10   <!--代理对象 -->
11   <!--interceptorNames 通知数组 -->
12   <!--p:target-ref 被代理的对象-->
13   <!--p:proxyTargetClass 被代理对象是否为一个类，如果是则使用
cglib, 否则使用 jdk 动态代理 -->
14   <bean id="proxy"
class="org.springframework.aop.framework.ProxyFactoryBean"
15       p:interceptorNames="advice"
16       p:target-ref="target"
17       p:proxyTargetClass="true"></bean>
18 </beans>

```



6.3、获得代理类的实例并测试运行



```

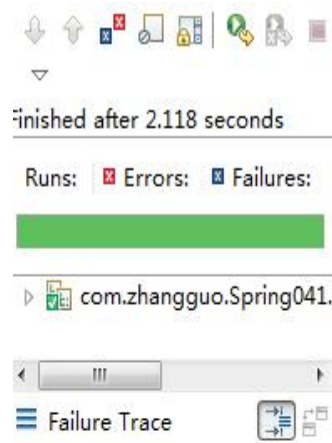
1 package com.zhangguo.Spring041.aop06;
2
3 import org.springframework.context.ApplicationContext;
4 import
org.springframework.context.support.ClassPathXmlApplicationContext;
5
6 public class Test {
7
8     @org.junit.Test
9     public void test01()
10    {
11        //容器
12        ApplicationContext ctx=new
ClassPathXmlApplicationContext("beans.xml");
13        //从代理工厂中获得代理对象
14        IMath math=(IMath)ctx.getBean("proxy");
15        int n1=100, n2=5;
16        math.add(n1, n2);

```

```

17     math.sub(n1, n2);
18     math.mul(n1, n2);
19     math.div(n1, n2);
20 }
21 }

```



```

1 package com.zhangguo.Spring041.aop06;
2
3 import org.springframework.context.ApplicationContext;
4 import org.springframework.context.support.ClassPathXmlApplicationContext;
5
6 public class Test {
7
8     @org.junit.Test
9     public void test01()

```

Problems Javadoc Declaration Console

<terminated> Test (26) [JUnit] C:\Program Files\Java\jdk1.7.0_51\bin\javaw.exe (2016年6月28日 下午 3:45)

信息: Loading XML bean definitions from class path resource [applicationContext.xml]

100+5=105
共用时 : 457

100-5=95
共用时 : 3

100X5=500
共用时 : 495

100/5=20
共用时 : 451

6.4、小结

这里有个值得注意的问题：从容器中获得 proxy 对象时应该是

org.springframework.aop.framework.ProxyFactoryBean 类型的对象(如下代码所示)，

但这里直接就转换成 IMath 类型了，这是因为：ProxyFactoryBean 本质上是一个用来生

产 Proxy 的 FactoryBean。如果容器中的某个对象持有某个 FactoryBean 的引用 它取得

的不是 `FactoryBean` 本身而是 `FactoryBean` 的 `getObject()` 方法所返回的对象。所以如果容器中某个对象依赖于 `ProxyFactoryBean` 那么它将会使用到 `ProxyFactoryBean` 的 `getObject()` 方法所返回的代理对象这就是 `ProxyFactoryBean` 得以在容器中使用的原因。

```
1 ProxyFactoryBean message=new ProxyFactoryBean();
2 message.setTarget(new Message());
3 message.addAdvice(new SurroundAdvice());
4 ((IMessage)message.getObject()).message();
```

七、使用 XML 配置 Spring AOP 切面

7.1、添加引用，需要引用一个新的 jar 包：aspectjweaver，该包是 AspectJ 的组成部分。

可以去 <http://search.maven.org> 搜索后下载或直接在 maven 项目中添加依赖。

Search Results

GroupId	ArtifactId	Latest Version	Updated	Download
org.aspectj	aspectjweaver	1.8.9 all (29)	15-Mar-2016	pom jar javadoc.jar sources.jar
org.apache.geronimo.bundles	aspectjweaver	1.6.8.2 all (3)	13-Mar-2013	pom jar source-release.tar.gz source-release
aspectj	aspectjweaver	1.5.4 all (7)	19-Jan-2008	pom

示例中使用 `pom.xml` 文件如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.zhangguo</groupId>
  <artifactId>Spring041</artifactId>
```

```
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>
<name>Spring041</name>
<url>http://maven.apache.org</url>
<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <spring.version>4.3.0.RELEASE</spring.version>
</properties>
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <scope>test</scope>
    <version>4.10</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${spring.version}</version>
  </dependency>
  <dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.8.9</version>
  </dependency>
  <dependency>
    <groupId>cglib</groupId>
    <artifactId>cglib</artifactId>
    <version>3.2.4</version>
  </dependency>
</dependencies>
</project>
```



7.2、定义通知

该通知不再需要实现任何接口或继承抽象类，一个普通的 **bean** 即可，方法可以带一个

JoinPoint 连接点参数，用于获得连接点信息，如方法名，参数，代理对象等。



```
1 package com.zhangguo.Spring041.aop08;
2
```

```

3 import org.aspectj.lang.JoinPoint;
4
5 /**
6  * 通知
7  */
8 public class Advices {
9     //前置通知
10    public void before(JoinPoint jp)
11    {
12
13        System.out.println("-----before-----");
14        System.out.println("方法名: "+jp.getSignature()+"， 参数: "+jp.getArgs().length+"， 代理对象: "+jp.getTarget());
15    }
16    //后置通知
17    public void after(JoinPoint jp) {
18
19        System.out.println("-----after-----");
20    }
21 }

```



通知的类型有多种，有些参数会不一样，特别是环绕通知，通知类型如下：



```

1 //前置通知
2 public void beforeMethod(JoinPoint joinPoint)
3
4 //后置通知
5 public void afterMethod(JoinPoint joinPoint)
6
7 //返回值通知
8 public void afterReturning(JoinPoint joinPoint, Object result)
9
10 //抛出异常通知
11 //在方法出现异常时会执行的代码可以访问到异常对象，可以指定在出现特
    定异常时在执行通知代码
12 public void afterThrowing(JoinPoint joinPoint, Exception ex)
13
14 //环绕通知
15 //环绕通知需要携带 ProceedingJoinPoint 类型的参数
16 //环绕通知类似于动态代理的全过程：ProceedingJoinPoint 类型的参数可
    以决定是否执行目标方法。
17 //而且环绕通知必须有返回值，返回值即为目标方法的返回值
18 public Object aroundMethod(ProceedingJoinPoint pjd)

```



7.3、配置 IOC 容器依赖的 XML 文件 beansOfAOP.xml



```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xmlns:p="http://www.springframework.org/schema/p"
5     xmlns:aop="http://www.springframework.org/schema/aop"
6
xsi:schemaLocation="http://www.springframework.org/schema/beans
7
http://www.springframework.org/schema/beans/spring-beans.xsd
8     http://www.springframework.org/schema/aop
9
http://www.springframework.org/schema/aop/spring-aop-4.3.xsd">
10
11     <!--被代理的目标对象 -->
12     <bean id="math"
class="com.zhangguo.Spring041.aop08.Math"></bean>
13     <!-- 通知 -->
14     <bean id="advice"
class="com.zhangguo.Spring041.aop08.Advices"></bean>
15     <!-- AOP 配置 -->
16     <!-- proxy-target-class 属性表示被代理的类是否为一个没有实现接
口的类, Spring 会依据实现了接口则使用 JDK 内置的动态代理, 如果未实现接
口则使用 cglib -->
17     <aop:config proxy-target-class="true">
18         <!-- 切面配置 -->
19         <!--ref 表示通知对象的引用 -->
20         <aop:aspect ref="advice">
21             <!-- 配置切入点(横切逻辑将注入的精确位置) -->
22             <aop:pointcut expression="execution(*
com.zhangguo.Spring041.aop08.Math.*(..))" id="pointcut1"/>
23             <!--声明通知, method 指定通知类型, pointcut 指定切点, 就
是该通知应该注入那些方法中 -->
24             <aop:before method="before" pointcut-ref="pointcut1"/>
25             <aop:after method="after" pointcut-ref="pointcut1"/>
26         </aop:aspect>
27     </aop:config>
28 </beans>

```



加粗部分的内容是在原 IOC 内容中新增的，主要是为 AOP 服务，如果引入失败则没有智能提示。xmlns:是 xml namespace 的简写。xmlns:xsi: 其 xsd 文件是 xml 需要遵守的规范，通过 URL 可以看到，是 w3 的统一规范，后面通过 xsi:schemaLocation 来定位所有的解析文件，这里只能成偶数对出现。

`<bean id="advice" class="com.zhangguo.Spring041.aop08.Advices"></bean>` 表示通知 bean，也就是横切逻辑 bean。`<aop:config proxy-target-class="true">` 用于 AOP 配置，`proxy-target-class` 属性表示被代理的类是否为一个没有实现接口的类，Spring 会依据实现了接口则使用 JDK 内置的动态代理，如果未实现接口则使用 `cglib`；在 Bean 配置文件中，所有的 Spring AOP 配置都必须定义在 `<aop:config>` 元素内部。对于每个切面而言，都要创建一个 `<aop:aspect>` 元素来为具体的切面实现引用后端 Bean 实例。因此，切面 Bean 必须有一个标识符，供 `<aop:aspect>` 元素引用。

`aop:aspect` 表示切面配置，`ref` 表示通知对象的引用；`aop:pointcut` 是配置切入点，就是横切逻辑将注入的精确位置，那些包，类，方法需要拦截注入横切逻辑。


`aop:before` 用于声明通知，`method` 指定通知类型，`pointcut` 指定切点，就是该通知应该注入那些方法中。在 aop Schema 中，每种通知类型都对应一个特定地 XML 元素。通知元素需要 `pointcut-ref` 属性来引用切入点，或者用 `pointcut` 属性直接嵌入切入点表达式。

`method` 属性指定切面类中通知方法的名称。有如下几种：



```
<!-- 前置通知 -->
<aop:before method="before" pointcut-ref="pointcut1"/>
<!-- 后置通知 -->
<aop:after method="after" pointcut-ref="pointcut1"/>
<!-- 环绕通知 -->
<aop:around method="around" pointcut="execution(*
com.zhangguo.Spring041.aop08.Math.s*(..))"/>
<!-- 异常通知 -->
```

```
<aop:after-throwing method="afterThrowing" pointcut="execution(*
com.zhangguo.Spring041.aop08.Math.d*(..))" throwing="exp"/>
<!-- 返回值通知 -->
<aop:after-returning method="afterReturning" pointcut="execution(*
com.zhangguo.Spring041.aop08.Math.m*(..))" returning="result"/>
```



关于 **execution** 请查看另一篇文章:

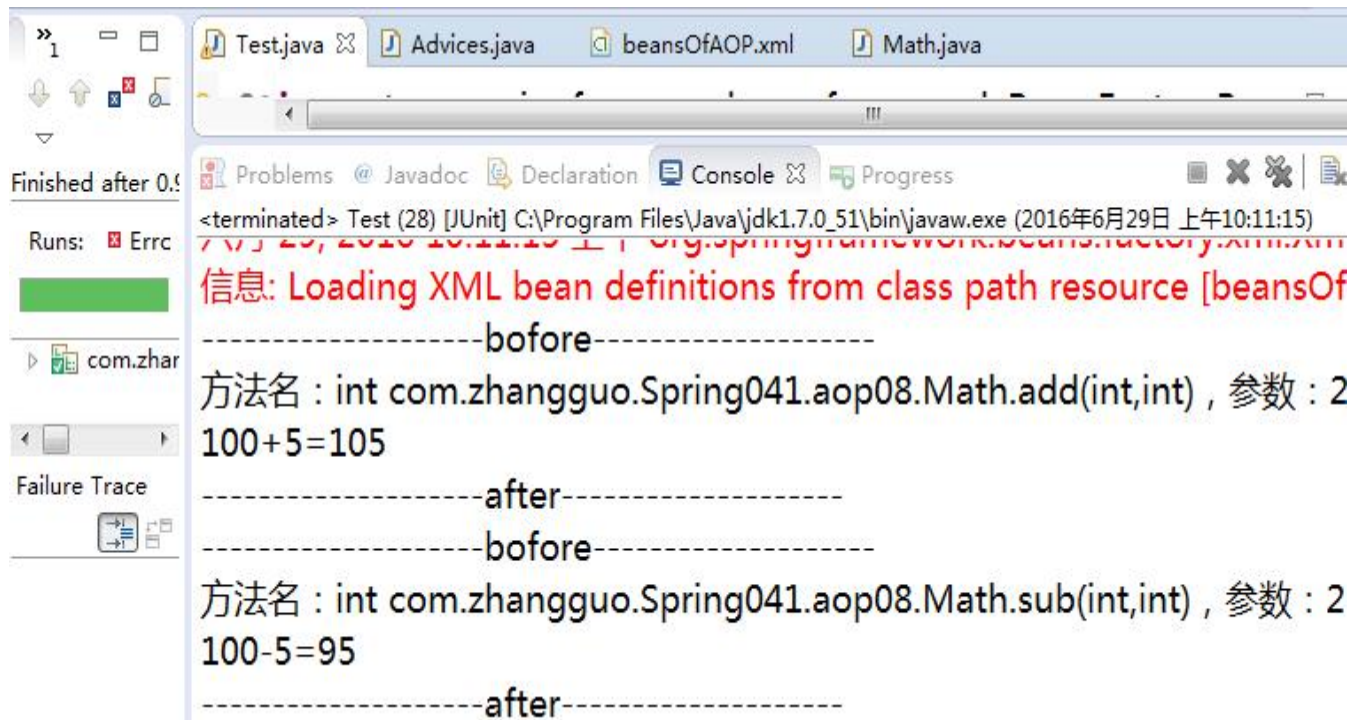
7.4、获得代理对象



```
1 package com.zhangguo.Spring041.aop08;
2
3 import org.springframework.aop.framework.ProxyFactoryBean;
4 import org.springframework.context.ApplicationContext;
5 import
org.springframework.context.support.ClassPathXmlApplicationContext;
6
7 public class Test {
8
9     @org.junit.Test
10     public void test01()
11     {
12         //容器
13         ApplicationContext ctx=new
ClassPathXmlApplicationContext("beansOfAOP.xml");
14         //从代理工厂中获得代理对象
15         IMath math=(IMath)ctx.getBean("math");
16         int n1=100,n2=5;
17         math.add(n1, n2);
18         math.sub(n1, n2);
19         math.mul(n1, n2);
20         math.div(n1, n2);
21     }
22 }
```



7.5、测试运行



7.6、环绕通知、异常后通知、返回结果后通知

在配置中我们发现共有 5 种类型的通知，前面我们试过了前置通知与后置通知，另外几种类型的通知如下代码所示：



```
package com.zhangguo.Spring041.aop08;
```

```
import org.aspectj.lang.JoinPoint;
```

```
import org.aspectj.lang.ProceedingJoinPoint;
```

```
/**
```

```
 * 通知
```

```
 */
```

```
public class Advices {
```

```
    //前置通知
```

```
    public void before(JoinPoint jp)
```

```
    {
```

```
        System.out.println("-----前置通知  
-----");
```

```
        System.out.println("方法名: "+jp.getSignature().getName()+"",  
        参数: "+jp.getArgs().length+", 被代理对象:
```

```
        "+jp.getTarget().getClass().getName());
```

```
    }
```

```
    //后置通知
```

```

    public void after(JoinPoint jp) {
        System.out.println("-----后置通知
-----");
    }
    //环绕通知
    public Object around(ProceedingJoinPoint pjd) throws Throwable{
        System.out.println("-----环绕开始
-----");
        Object object=pjd.proceed();
        System.out.println("-----环绕结束
-----");
        return object;
    }
    //异常后通知
    public void afterThrowing(JoinPoint jp,Exception exp)
    {
        System.out.println("-----异常后通知,发生了异常:
"+exp.getMessage()+"-----");
    }
    //返回结果后通知
    public void afterReturning(JoinPoint joinPoint, Object result)
    {
        System.out.println("-----返回结果后通知
-----");
        System.out.println("结果是: "+result);
    }
}

```



容器配置文件 **beansOfAOP.xml** 如下:



```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop

```

<http://www.springframework.org/schema/aop/spring-aop-4.3.xsd>>

<!--被代理的目标对象 -->

```

<bean id="math" class="com.zhangguo.Spring041.aop08.Math"></bean>
<!-- 通知 -->
<bean id="advice"
class="com.zhangguo.Spring041.aop08.Advices"></bean>
<!-- AOP 配置 -->
<!-- proxy-target-class 属性表示被代理的类是否为一个没有实现接口的
类, Spring 会依据实现了接口则使用 JDK 内置的动态代理, 如果未实现接口则
使用 cglib -->
<aop:config proxy-target-class="true">
    <!-- 切面配置 -->
    <!--ref 表示通知对象的引用 -->
    <aop:aspect ref="advice">
        <!-- 配置切入点(横切逻辑将注入的精确位置) -->
        <aop:pointcut expression="execution(*
com.zhangguo.Spring041.aop08.Math.a*(..))" id="pointcut1"/>
        <!--声明通知, method 指定通知类型, pointcut 指定切点, 就是
该通知应该注入那些方法中 -->
        <aop:before method="before" pointcut-ref="pointcut1"/>
        <aop:after method="after" pointcut-ref="pointcut1"/>
        <aop:around method="around" pointcut="execution(*
com.zhangguo.Spring041.aop08.Math.s*(..))"/>
        <aop:after-throwing method="afterThrowing"
pointcut="execution(* com.zhangguo.Spring041.aop08.Math.d*(..))"
throwing="exp"/>
        <aop:after-returning method="afterReturning"
pointcut="execution(* com.zhangguo.Spring041.aop08.Math.m*(..))"
returning="result"/>
    </aop:aspect>
</aop:config>
</beans>

```



测试代码:



```

package com.zhangguo.Spring041.aop08;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {

    @org.junit.Test
    public void test01()

```

```
{
    //容器
    ApplicationContext ctx=new
ClassPathXmlApplicationContext("beansOfAOP.xml");
    //从代理工厂中获得代理对象
    IMath math=(IMath)ctx.getBean("math");
    int n1=100, n2=0;
    math.add(n1, n2);
    math.sub(n1, n2);
    math.mul(n1, n2);
    math.div(n1, n2);
}
```



运行结果:

```
-----前置通知-----
方法名 : add , 参数 : 2 , 被代理对象 : com.zhangguo.Spring041.aop08.Math
100+0=100
-----后置通知-----
-----环绕开始-----|
100-0=100
-----环绕结束-----
100X0=0
-----返回结果后通知-----
结果是 : 0
-----异常后通知, 发生了异常 : / by zero-----
```

小结: 不同类型的通知参数可能不相同; `aop:after-throwing` 需要指定通知中参数的名称

`throwing="exp"`, 则方法中定义应该是这样: `afterThrowing(JoinPoint jp,Exception exp);`

`aop:after-returning` 同样需要设置 `returning` 指定方法参数的名称。通过配置切面的方法

使 AOP 变得更加灵活。