百知教育 — Spring系列课程 — AOP编程

第一章、静态代理设计模式

1. 为什么需要代理设计模式

1.1 问题

- 在JavaEE分层开发开发中，那个层次对于我们来讲最重要

```
DAO ---> Service --> Controller

JavaEE分层开发中，最为重要的是Service层
```

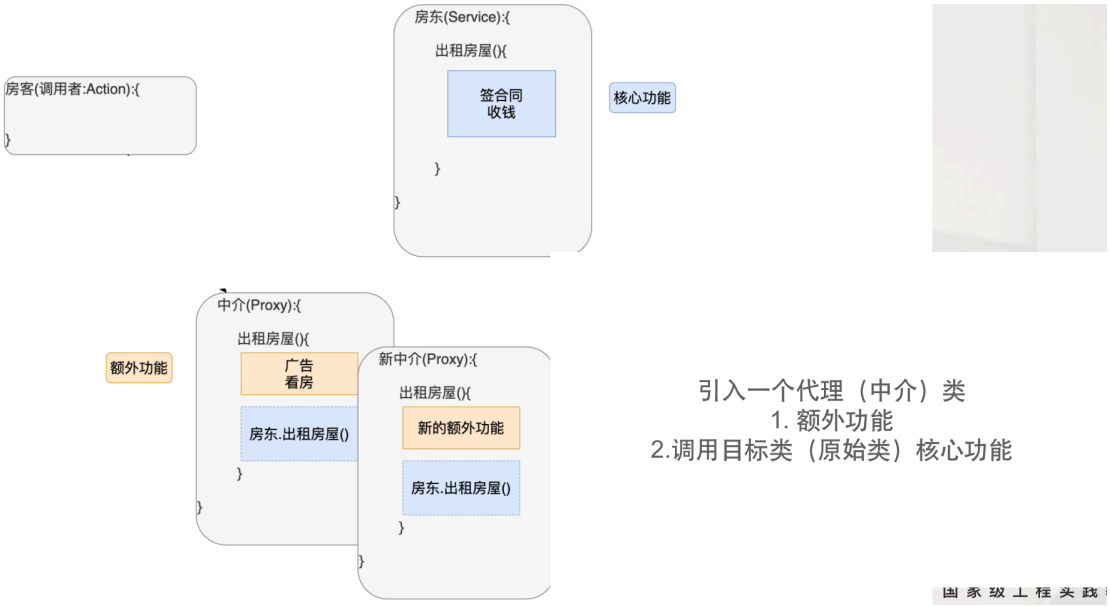
- Service层中包含了哪些代码?

```
 Service层中 = 核心功能(几十行 上百代码) + 额外功能(附加功能)
 1. 核心功能
    业务运算
    DAO调用
 2. 额外功能
    1. 不属于业务
    2. 可有可无
    3. 代码量很小

    事务、日志、性能...

```

- 额外功能书写在Service层中好不好?

```
Service层的调用者的角度（Controller）:需要在Service层书写额外功能。
                              软件设计者：Service层不需要额外功能

```

- 现实生活中的解决方式



2. 代理设计模式

1.1 概念

```
通过代理类，为原始类（目标）增加额外的功能
好处：利于原始类(目标)的维护
```

1.2名词解释

```
1. 目标类 原始类
   指的是 业务类 (核心功能 --> 业务运算 DAO调用)
2. 目标方法，原始方法
   目标类(原始类)中的方法 就是目标方法(原始方法)
3. 额外功能 (附加功能)
   日志，事务，性能
```

1.3 代理开发的核心要素

```
代理类 = 目标类(原始类) + 额外功能 + 原始类(目标类)实现相同的接口

房东 ---> public interface UserService{
                m1
                m2
          }
          UserServiceImpl implements UserService{
                m1 ---> 业务运算 DAO调用
                m2
          }
          UserServiceProxy implements UserService
                m1
                m2
```

1.4 编码

静态代理：为每一个原始类，手工编写一个代理类 (.java .class)

```java
public class UserServiceProxy implements UserService {
    private UserServiceImpl userService = new UserServiceImpl();

    @Override
    public void register(User user) {
        System.out.println("----log-----");
        userService.register(user);
    }

    @Override
    public boolean login(String name, String password) {
        System.out.println("----log-----");
        return userService.login(name,password);
    }
}
```

1.5 静态代理存在的问题

```
1. 静态类文件数量过多，不利于项目管理
   UserServiceImpl  UserServiceProxy
   OrderServiceImpl OrderServiceProxy
2. 额外功能维护性差
   代理类中 额外功能修改复杂(麻烦)
```

第二章、Spring的动态代理开发

1. Spring动态代理的概念

```
概念：通过代理类为原始类(目标类)增加额外功能
好处：利于原始类(目标类)的维护
```

2. 搭建开发环境

```xml
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aop</artifactId>
  <version>5.1.14.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjrt</artifactId>
  <version>1.8.8</version>
</dependency>

<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjweaver</artifactId>
  <version>1.8.3</version>
</dependency>

```

3. Spring动态代理的开发步骤

1. 创建原始对象(目标对象)

```
 public class UserServiceImpl implements UserService {
     @Override
     public void register(User user) {
         System.out.println("UserServiceImpl.register 业务运算 +
DAO ");
     }

     @Override
     public boolean login(String name, String password) {
         System.out.println("UserServiceImpl.login");
         return true;
     }
 }

```

```
 <bean id="userService"
class="com.baizhiedu.proxy.UserServiceImpl"/>
```

2. 额外功能 MethodBeforeAdvice接口

```
 额外的功能书写在接口的实现中，运行在原始方法执行之前运行额外功能。
```

```
 public class Before implements MethodBeforeAdvice {
     /*
       作用：需要把运行在原始方法执行之前运行的额外功能，书写在before方法中
      */
     @Override
     public void before(Method method, Object[] args, Object
target) throws Throwable {
         System.out.println("-----method before advice log------
");
     }
 }
```

```
 <bean id="before" class="com.baizhiedu.dynamic.Before"/>
```

3. 定义切入点

```
 切入点：额外功能加入的位置

 目的：由程序员根据自己的需要，决定额外功能加入给那个原始方法
 register
 login

 简单的测试：所有方法都做为切入点，都加入额外的功能。
```

```
 <aop:config>
    <aop:pointcut id="pc" expression="execution(* *(..))"/>
 </aop:config>
```

4. 组装 (2 3整合)

```
表达的含义：所有的方法 都加入 before的额外功能
<aop:advisor advice-ref="before" pointcut-ref="pc"/>
```

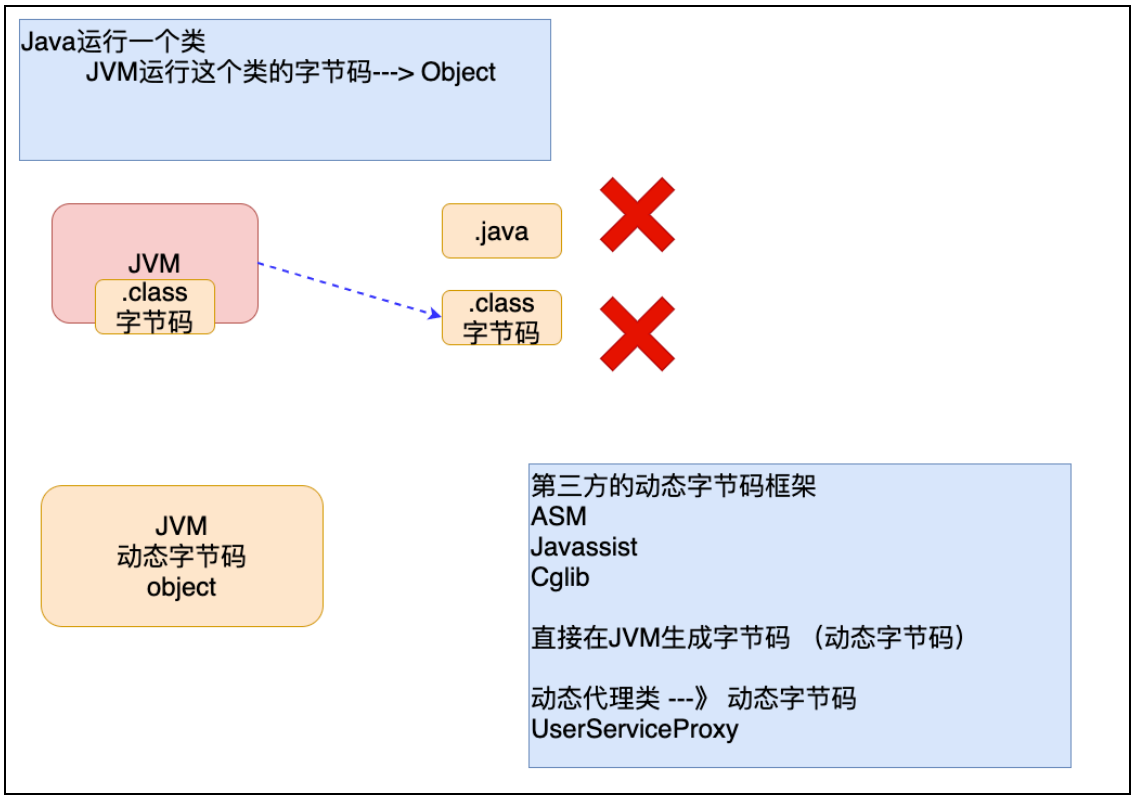
5. 调用

```
目的：获得Spring工厂创建的动态代理对象，并进行调用
ApplicationContext ctx = new ClassPathXmlApplicationContext("/applicationContext.xml");
注意：
    1. Spring的工厂通过原始对象的id值获得的是代理对象
    2. 获得代理对象后，可以通过声明接口类型，进行对象的存储

UserService userService=(UserService)ctx.getBean("userService");

userService.login("")
userService.register()
```

4. 动态代理细节分析

1. Spring创建的动态代理类在哪里?

```
Spring框架在运行时，通过动态字节码技术，在JVM创建的，运行在JVM内部，等程序结束后，会和JVM一起消失

什么叫动态字节码技术:通过第三个动态字节码框架，在JVM中创建对应类的字节码，进而创建对象，当虚拟机结束，动态字节码跟着消失。

结论：动态代理不需要定义类文件，都是JVM运行过程中动态创建的，所以不会造成静态代理，类文件数量过多，影响项目管理的问题。
```



2. 动态代理编程简化代理的开发

```
1    在额外功能不改变的前提下，创建其他目标类（原始类）的代理对象时，只需要指定原始(目标)对象即可。
```

3. 动态代理额外功能的维护性大大增强

第三章、Spring动态代理详解

1. 额外功能的详解

- MethodBeforeAdvice分析

```
1. MethodBeforeAdvice接口作用：额外功能运行在原始方法执行之前，进行额外功能操作。

public class Before1 implements MethodBeforeAdvice {
    /*
      作用：需要把运行在原始方法执行之前运行的额外功能，书写在before方法中

      Method：额外功能所增加给的那个原始方法
              login方法

              register方法

              showOrder方法

      Object[]：额外功能所增加给的那个原始方法的参数。String name,String password
                                                  User

       Object：额外功能所增加给的那个原始对象  UserServiceImpl
                                               OrderServiceImpl
     */
    @Override
    public void before(Method method, Object[] args, Object target) throws Throwable {
        System.out.println("-----new method before advice log-----");
    }
}

2. before方法的3个参数在实战中，该如何使用。
   before方法的参数，在实战中，会根据需要进行使用，不一定都会用到，也有可能都不用。

   Servlet{
       service(HttpRequest request,HttpResponse response){
            request.getParameter("name") -->

            response.getWriter() --->

       }

   }
```

```
38
```

- MethodInterceptor(方法拦截器)

```
methodinterceptor接口：额外功能可以根据需要运行在原始方法执行 前、后、前后。
```

```
public class Arround implements MethodInterceptor {
    /*
         invoke方法的作用:额外功能书写在invoke
                        额外功能   原始方法之前
                                   原始方法之后
                                   原始方法执行之前 之后
         确定：原始方法怎么运行

         参数：MethodInvocation  (Method):额外功能所增加给的那个原始方法
                    login
                    register
               invocation.proceed() ---> login运行
                                         register运行

          返回值：Object：原始方法的返回值

         Date convert(String name)
     */



    @Override
    public Object invoke(MethodInvocation invocation) throws
Throwable {
          System.out.println("-----额外功能 log----");
          Object ret = invocation.proceed();

          return ret;
    }
}


```

额外功能运行在原始方法执行之后

```
@Override
public Object invoke(MethodInvocation invocation) throws Throwable
{
  Object ret = invocation.proceed();
  System.out.println("-----额外功能运行在原始方法执行之后----");

  return ret;
}
```

额外功能运行在原始方法执行之前，之后

```
什么样的额外功能 运行在原始方法执行之前，之后都要添加?
事务

@Override
public Object invoke(MethodInvocation invocation) throws Throwable
{
  System.out.println("-----额外功能运行在原始方法执行之前----");
  Object ret = invocation.proceed();
  System.out.println("-----额外功能运行在原始方法执行之后----");

  return ret;
}
```

额外功能运行在原始方法抛出异常的时候

```
@Override
public Object invoke(MethodInvocation invocation) throws Throwable
{

  Object ret = null;
  try {
    ret = invocation.proceed();
  } catch (Throwable throwable) {

    System.out.println("-----原始方法抛出异常 执行的额外功能 ---- ");
    throwable.printStackTrace();
  }


  return ret;
}
```

MethodInterceptor影响原始方法的返回值

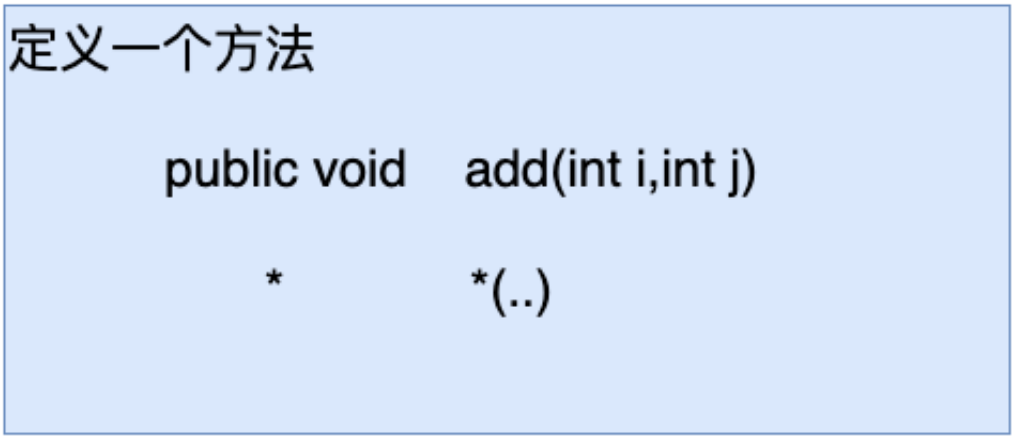
```
原始方法的返回值，直接作为invoke方法的返回值返回，MethodInterceptor不会影响原始方法的返回值

MethodInterceptor影响原始方法的返回值
Invoke方法的返回值，不要直接返回原始方法的运行结果即可。

@Override
public Object invoke(MethodInvocation invocation) throws Throwable
{
   System.out.println("------log-----");
   Object ret = invocation.proceed();
   return false;
}
```

2. 切入点详解

```
切入点决定额外功能加入位置(方法)

<aop:pointcut id="pc" expression="execution(* *(..))"/>
exection(* *(..)) ---> 匹配了所有方法     a  b  c

1. execution()  切入点函数
2. * *(..)      切入点表达式
```

2.1 切入点表达式

1. 方法切入点表达式




```
*  *(..)  --> 所有方法

* ---> 修饰符 返回值
* ---> 方法名
()---> 参数表
..---> 对于参数没有要求 (参数有没有，参数有几个都行，参数是什么类型的都行)
```

- 定义login方法作为切入点

```
* login(..)

# 定义register作为切入点
* register(..)
```

- 定义login方法且login方法有两个字符串类型的参数 作为切入点

```
* login(String,String)

#注意：非java.lang包中的类型，必须要写全限定名
* register(com.baizhiedu.proxy.User)

# ..可以和具体的参数类型连用
* login(String,..)  -->
login(String),login(String,String),login(String,com.baizhiedu
.proxy.User)
```

- 精准方法切入点限定

```
修饰符 返回值          包.类.方法(参数)

    *
com.baizhiedu.proxy.UserServiceImpl.login(..)
    *
com.baizhiedu.proxy.UserServiceImpl.login(String,String)
```

2. 类切入点

```
指定特定类作为切入点(额外功能加入的位置)，自然这个类中的所有方法，都会加上对应的额外功能
```

- 语法1

```
#类中的所有方法加入了额外功能
* com.baizhiedu.proxy.UserServiceImpl.*(..)
```

- 语法2

```
#忽略包
1. 类只存在一级包  com.UserServiceImpl
* *.UserServiceImpl.*(..)

2. 类存在多级包    com.baizhiedu.proxy.UserServiceImpl
* *..UserServiceImpl.*(..)
```

3. 包切入点表达式 实战

```
指定包作为额外功能加入的位置，自然包中的所有类及其方法都会加入额外的功能
```

- 语法1

```
#切入点包中的所有类，必须在proxy中，不能在proxy包的子包中
* com.baizhiedu.proxy.*.*(..)
```

- 语法2

```
#切入点当前包及其子包都生效
* com.baizhiedu.proxy..*.*(..)
```

2.2 切入点函数

```
切入点函数：用于执行切入点表达式
```

1. execution

```
最为重要的切入点函数，功能最全。
执行 方法切入点表达式 类切入点表达式 包切入点表达式

弊端：execution执行切入点表达式 ， 书写麻烦
     execution(* com.baizhiedu.proxy..*.*(..))

注意：其他的切入点函数 简化是execution书写复杂度，功能上完全一致
```

2. args

```
作用：主要用于函数(方法) 参数的匹配

切入点：方法参数必须得是2个字符串类型的参数

execution(* *(String,String))

args(String,String)
```

3. within

```
 作用：主要用于进行类、包切入点表达式的匹配

 切入点：UserServiceImpl这个类

 execution(* *..UserServiceImpl.*(..))

 within(*..UserServiceImpl)

 execution(* com.baizhiedu.proxy..*.*(..))

 within(com.baizhiedu.proxy..*)

```

4.@annotation

```
作用：为具有特殊注解的方法加入额外功能

<aop:pointcut id="" expression="@annotation(com.baizhiedu.Log)"/>
```

5. 切入点函数的逻辑运算

```
 指的是 整合多个切入点函数一起配合工作，进而完成更为复杂的需求
```

- and与操作

```
 案例：login 同时 参数 2个字符串

 1. execution(* login(String,String))

 2. execution(* login(..)) and args(String,String)

 注意：与操作不同用于同种类型的切入点函数

 案例：register方法 和 login方法作为切入点

 execution(* login(..)) or  execution(* register(..))

```

- or或操作

```
 案例：register方法 和 login方法作为切入点

 execution(* login(..)) or  execution(* register(..))
```

第四章、AOP编程

1. AOP概念

```
AOP (Aspect Oriented Programing)   面向切面编程 = Spring动态代理开发
以切面为基本单位的程序开发，通过切面间的彼此协同，相互调用，完成程序的构建
切面 = 切入点 + 额外功能

OOP (Object Oritened Programing)   面向对象编程 Java
以对象为基本单位的程序开发，通过对象间的彼此协同，相互调用，完成程序的构建

POP (Producer Oriented Programing) 面向过程(方法、函数)编程 C
以过程为基本单位的程序开发，通过过程间的彼此协同，相互调用，完成程序的构建
```

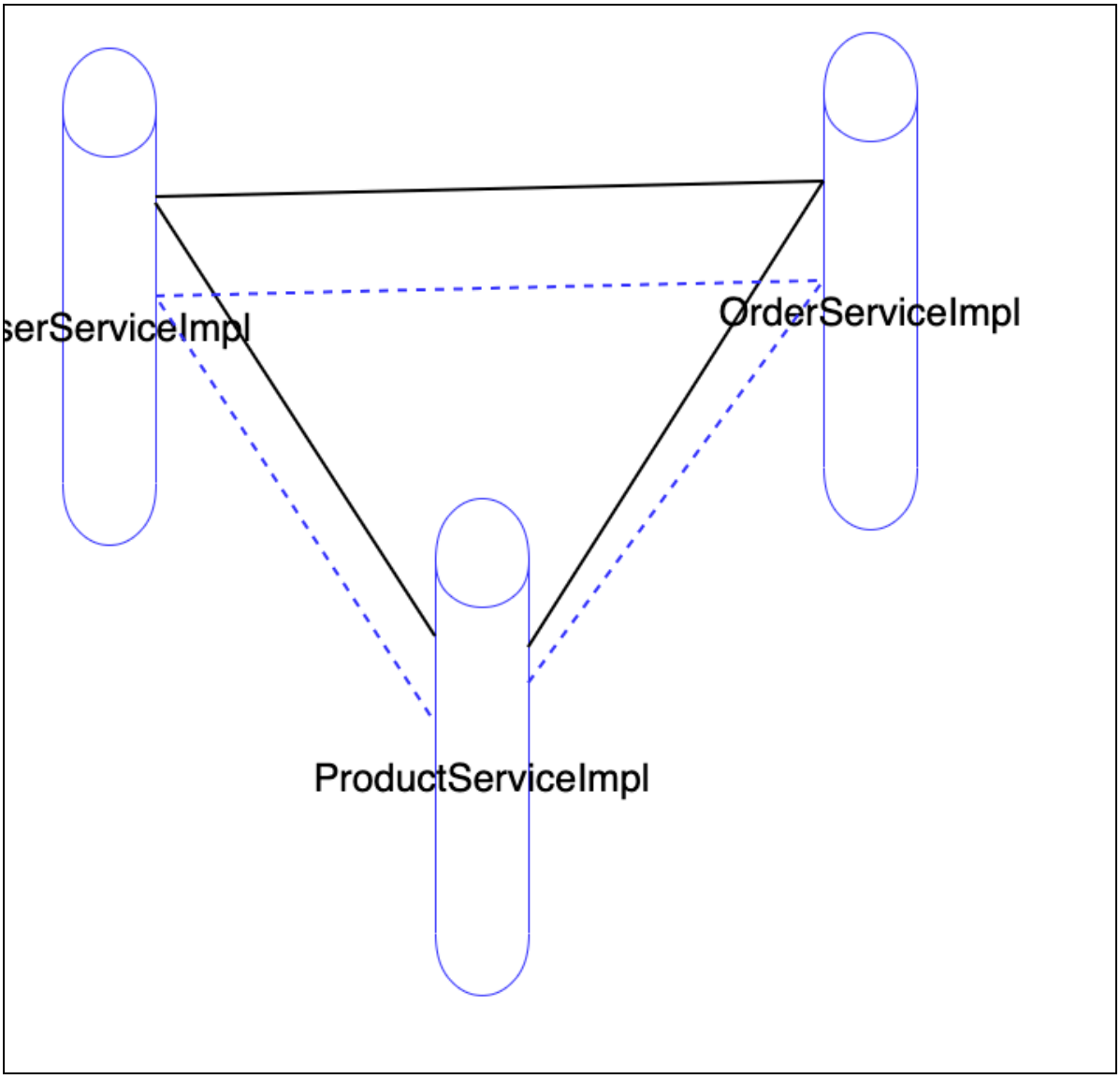
```
AOP的概念：
      本质就是Spring得动态代理开发，通过代理类为原始类增加额外功能。
      好处：利于原始类的维护

注意：AOP编程不可能取代OOP，OOP编程有意补充。
```

2. AOP编程的开发步骤

```
1. 原始对象
2. 额外功能 (MethodInterceptor)
3. 切入点
4. 组装切面 (额外功能+切入点)
```

3. 切面的名词解释

```
切面 = 切入点 + 额外功能

几何学
   面 = 点 + 相同的性质
```



第五章、AOP的底层实现原理

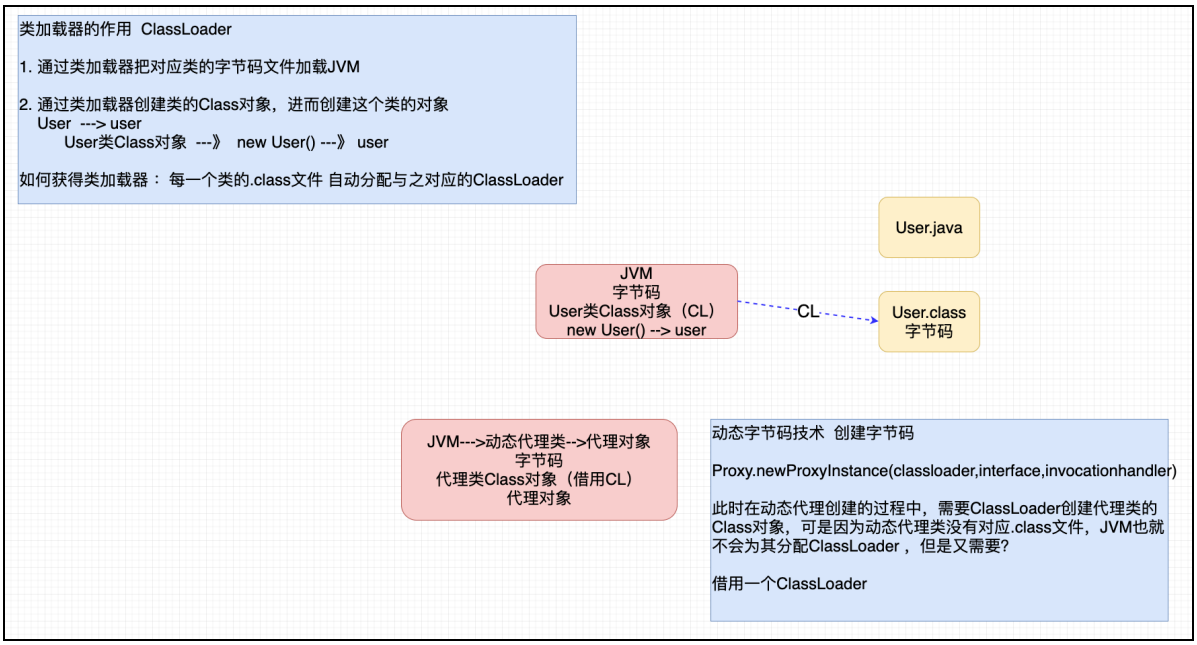
1. 核心问题

```
1. AOP如何创建动态代理类(动态字节码技术)
2. Spring工厂如何加工创建代理对象
   通过原始对象的id值，获得的是代理对象
```

2. 动态代理类的创建

2.1 JDK的动态代理

- Proxy.newProxyInstance方法参数详解



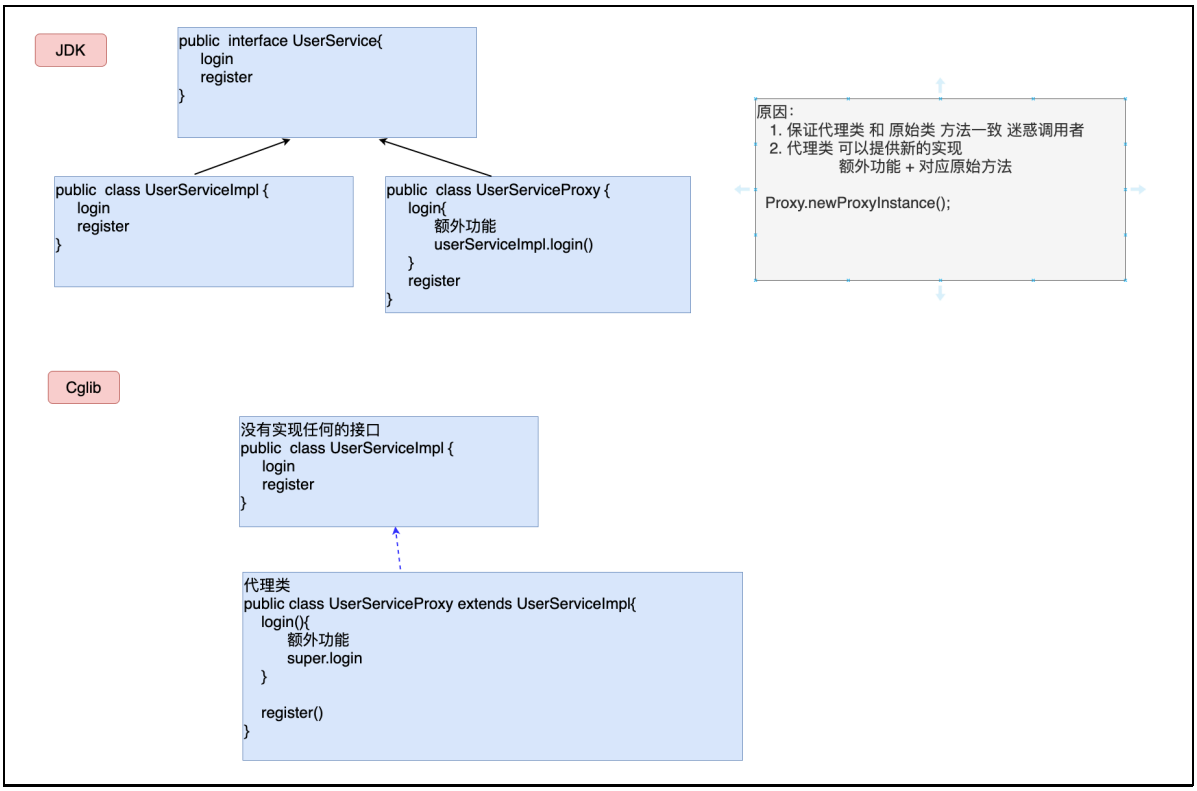
- 编码

```
1   public class TestJDKProxy {
2
3       /*
4           1. 借用类加载器  TestJDKProxy
5                           UserServiceImpl
6           2. JDK8.x前
7
8               final UserService userService = new UserServiceImpl();
9        */
10      public static void main(String[] args) {
11          //1 创建原始对象
12          UserService userService = new UserServiceImpl();
13
14          //2 JDK创建动态代理
15          /*
```

```
16
17          */
18
19          InvocationHandler handler = new InvocationHandler(){
20              @Override
21              public Object invoke(Object proxy, Method method,
    Object[] args) throws Throwable {
22                  System.out.println("------proxy  log --------");
23                  //原始方法运行
24                  Object ret = method.invoke(userService, args);
25                  return ret;
26              }
27          };
28
29          UserService userServiceProxy =
    (UserService)Proxy.newProxyInstance(UserServiceImpl.class.getClass
    Loader(),userService.getClass().getInterfaces(),handler);
30
31          userServiceProxy.login("suns", "123456");
32          userServiceProxy.register(new User());
33      }
34  }
35
```

2.2 CGlib的动态代理

1 CGlib创建动态代理的原理：父子继承关系创建代理对象，原始类作为父类，代理类作为子类，这样既可以保证2者方法一致，同时在代理类中提供新的实现(额外功能+原始方法)



- CGlib编码

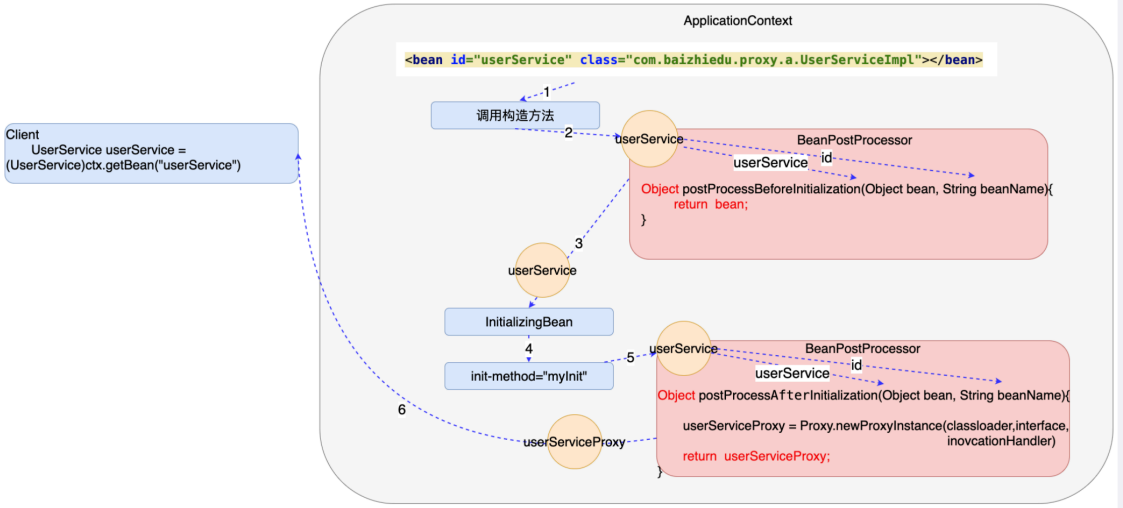
```java
package com.baizhiedu.cglib;

import com.baizhiedu.proxy.User;
import org.springframework.cglib.proxy.Enhancer;
import org.springframework.cglib.proxy.MethodInterceptor;
import org.springframework.cglib.proxy.MethodProxy;

import java.lang.reflect.Method;

public class TestCglib {
    public static void main(String[] args) {
        //1 创建原始对象
        UserService userService = new UserService();

        /*
          2 通过cglib方式创建动态代理对象

 Proxy.newProxyInstance(classloader,interface,invocationhandler)

            Enhancer.setClassLoader()
            Enhancer.setSuperClass()
            Enhancer.setCallback();  ---> MethodInterceptor(cglib)
            Enhancer.create() ---> 代理
         */

        Enhancer enhancer = new Enhancer();

        enhancer.setClassLoader(TestCglib.class.getClassLoader());
        enhancer.setSuperclass(userService.getClass());


        MethodInterceptor interceptor = new MethodInterceptor() {
            //等同于 InvocationHandler --- invoke
            @Override
            public Object intercept(Object o, Method method,
Object[] args, MethodProxy methodProxy) throws Throwable {
                System.out.println("---cglib log----");
                Object ret = method.invoke(userService, args);

                return ret;
            }
        };

        enhancer.setCallback(interceptor);

        UserService userServiceProxy = (UserService)
enhancer.create();

        userServiceProxy.login("suns", "123345");
        userServiceProxy.register(new User());
    }
}

```

- 总结

```
1. JDK动态代理    Proxy.newProxyInstance()  通过接口创建代理的实现类
2. Cglib动态代理 Enhancer                  通过继承父类创建的代理类
```

3. Spring工厂如何加工原始对象

- 思路分析



- 编码

```
1   public class ProxyBeanPostProcessor implements BeanPostProcessor {
2       @Override
3       public Object postProcessBeforeInitialization(Object bean,
    String beanName) throws BeansException {
4           return bean;
5       }
6
7       @Override
8       /*
9            Proxy.newProxyInstance();
10       */
11      public Object postProcessAfterInitialization(Object bean,
    String beanName) throws BeansException {
12
13          InvocationHandler handler = new InvocationHandler() {
14              @Override
15              public Object invoke(Object proxy, Method method,
    Object[] args) throws Throwable {
16                  System.out.println("----- new Log-----");
17                  Object ret = method.invoke(bean, args);
18
19                  return ret;
20              }
21          };
22          return
    Proxy.newProxyInstance(ProxyBeanPostProcessor.class.getClassLoader
    (),bean.getClass().getInterfaces(),handler);
23      }
24  }
```

```
<bean id="userService"
class="com.baizhiedu.factory.UserServiceImpl"/>


<!--1. 实现BeanPostProcessor 进行加工
        2. 配置文件中对BeanPostProcessor进行配置
    -->

<bean id="proxyBeanPostProcessor"
class="com.baizhiedu.factory.ProxyBeanPostProcessor"/>

```

第六章、基于注解的AOP编程

1. 基于注解的AOP编程的开发步骤

1. 原始对象
2. 额外功能
3. 切入点
4. 组装切面

```
# 通过切面类 定义了 额外功能 @Around
            定义了 切入点   @Around("execution(* login(..))")
            @Aspect 切面类

package com.baizhiedu.aspect;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;


/*
        1. 额外功能
                public class MyArround implements
MethodInterceptor{

                    public Object invoke(MethodInvocation
invocation){

                                Object ret =
invocation.proceed();

                                return ret;

                    }

                }

        2. 切入点
              <aop:config
                  <aop:pointcut id=""  expression="execution(*
login(..))"/>
```

```
 */
@Aspect
public class MyAspect {

    @Around("execution(* login(..))")
    public Object arround(ProceedingJoinPoint joinPoint) throws Throwable {

        System.out.println("----aspect log ------");

        Object ret = joinPoint.proceed();


        return ret;
    }
}

```

```
  <bean id="userService" class="com.baizhiedu.aspect.UserServiceImpl"/>

    <!--
       切面
         1. 额外功能
         2. 切入点
         3. 组装切面


    -->
<bean id="arround" class="com.baizhiedu.aspect.MyAspect"/>

<!--告知Spring基于注解进行AOP编程-->
<aop:aspectj-autoproxy />
```

2. 细节

1. 切入点复用

```
 切入点复用：在切面类中定义一个函数 上面@Pointcut注解 通过这种方式，定义切入点表达式，后续更加有利于切入点复用。

@Aspect
public class MyAspect {
    @Pointcut("execution(* login(..))")
    public void myPointcut(){}

    @Around(value="myPointcut()")
    public Object arround(ProceedingJoinPoint joinPoint) throws Throwable {

        System.out.println("----aspect log ------");

        Object ret = joinPoint.proceed();


```

```
            return ret;
        }


        @Around(value="myPointcut()")
        public Object arround1(ProceedingJoinPoint joinPoint)
    throws Throwable {

            System.out.println("----aspect tx ------");

            Object ret = joinPoint.proceed();


            return ret;
        }

    }
```

2. 动态代理的创建方式

```
    AOP底层实现   2种代理创建方式
    1.  JDK   通过实现接口 做新的实现类方式 创建代理对象
    2.  Cglib通过继承父类 做新的子类       创建代理对象

    默认情况 AOP编程 底层应用JDK动态代理创建方式
    如果切换Cglib
         1. 基于注解AOP开发
            <aop:aspectj-autoproxy proxy-target-class="true" />
         2. 传统的AOP开发
            <aop:config proxy-target-class="true">
            </aop>
```

第七章、AOP开发中的一个坑

```
坑：在同一个业务类中，进行业务方法间的相互调用，只有最外层的方法,才是加入了额外功能(内部的方法，通过普通的方式调用，都调用的是原始方法)。如果想让内层的方法也调用代理对象的方法，就要AppicationContextAware获得工厂，进而获得代理对象。
public class UserServiceImpl implements UserService,
ApplicationContextAware {
    private ApplicationContext ctx;


    @Override
    public void setApplicationContext(ApplicationContext
applicationContext) throws BeansException {
            this.ctx = applicationContext;
    }

    @Log
    @Override
    public void register(User user) {
        System.out.println("UserServiceImpl.register 业务运算 + DAO ");
        //throw new RuntimeException("测试异常");
```

```
16
17          //调用的是原始对象的login方法 ---> 核心功能
18          /*
19              设计目的：代理对象的login方法 --->  额外功能+核心功能
20              ApplicationContext ctx = new
    ClassPathXmlApplicationContext("/applicationContext2.xml");
21              UserService userService = (UserService)
    ctx.getBean("userService");
22              userService.login();
23
24              Spring工厂重量级资源 一个应用中 应该只创建一个工厂
25           */
26
27          UserService userService = (UserService)
    ctx.getBean("userService");
28          userService.login("suns", "123456");
29      }
30
31      @Override
32      public boolean login(String name, String password) {
33          System.out.println("UserServiceImpl.login");
34          return true;
35      }
36  }
37
```

第八章、AOP阶段知识总结