一、AOP

1、AOP 介绍

(1)、什么是 AOP

a、在软件业，AOP 为 Aspect Oriented Programming 的缩写，意为：面向切面编程，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。AOP 是 OOP（面向对象编程）的延续，是软件开发中的一个热点，也是 Spring 框架中的一个重要内容，是函数式编程的一种衍生范型。利用 AOP 可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率；

b、AOP 采取**横向抽取**机制，取代了传统**纵向继承**体系重复性代码；

c、经典应用：事务管理、性能监视、安全检查、缓存 、日志等；

d、Spring AOP 使用纯 Java 实现，不需要专门的编译过程和类加载器，在运行期通过代理方式向目标类织入增强代码。

2、AspectJ：是一个基于 Java 语言的 **AOP 框架**，Spring2.0 开始，Spring AOP 引入对 Aspect 的支持，AspectJ 扩展了 Java 语言，提供了一个专门的编译器，在编译时提供横向代码的织入。

3、AOP 实现原理

（1）、aop 底层将采用代理机制进行实现。

（2）、接口 + 实现类 ：spring 采用 jdk 的动态代理 Proxy。

（3）、实现类：spring 采用 cglib 字节码增强。

4、AOP 术语

1、target：目标类，需要被代理的类。例如：UserService

2、Joinpoint(连接点):所谓连接点是指那些可能被拦截到的方法。例如：所有的方法

3、PointCut 切入点：已经被增强的连接点。例如：addUser()

4、advice 通知/增强，增强代码。例如：after、before

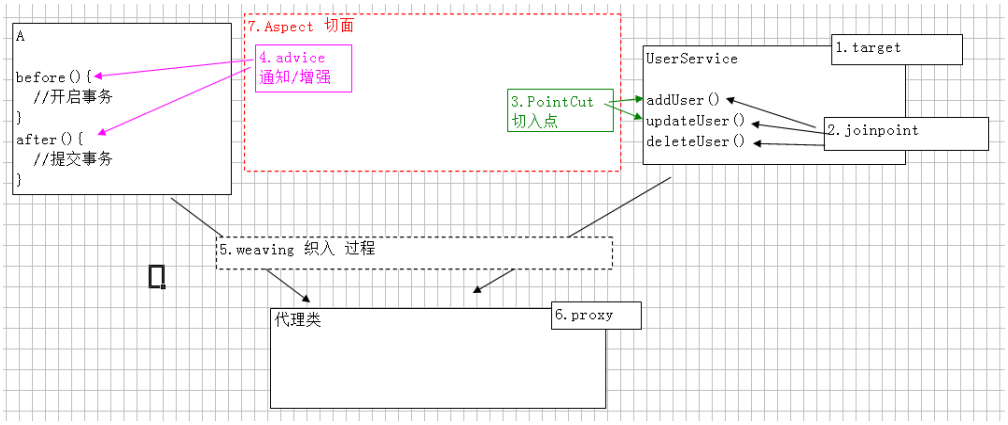
5、 Weaving(织入):是指把增强 advice 应用到目标对象 target 来创建新的代理对象 proxy 的过程.

6、proxy 代理类

7、 Aspect(切面): 是切入点 pointcut 和通知 advice 的结合

一个线是一个特殊的面。

一个切入点和一个通知，组成成一个特殊的面。



5、AOP 联盟通知类型（Spring 框架和 AOP 结合）

（1）、AOP 联盟为通知 Advice 定义了 org.aopalliance.aop.Advice

（2）、Spring 按照通知 Advice 在目标类方法的连接点位置，可以分为 5 类

a、前置通知 org.springframework.aop.MethodBeforeAdvice：在目标方法执行前实施增强；

b、后置通知 org.springframework.aop.AfterReturningAdvice：在目标方法执行后实施增强；

c、环绕通知 org.aopalliance.intercept.MethodInterceptor：在目标方法执行前后实施增强；

注意：环绕通知，必须手动执行目标方法

```
Try
{
    //前置通知
    //执行目标方法
    //后置通知
} catch()
{
    //抛出异常通知
}
```

异常抛出通知 org.springframework.aop.ThrowsAdvice：在方法抛出异常后实施增强；

引介通知 org.springframework.aop.IntroductionInterceptor：在目标类中添加一些新的方法和属性；

二、 手动方式实现 AOP

1、JDK 动态代理

JDK 动态代理 对“装饰者”设计模式 简化。使用前提：必须有接口。

1、 目标类：接口 + 实现类

```
public interface UserService
{
   public void addUser();
   public void updateUser();
   public void deleteUser();
}
```

2、 切面类：用于存通知 MyAspect

```
public class MyAspect
{
    public void before()
    {
        System.out.println("鸡首");
    }
    public void after()
    {
        System.out.println("牛后");
    }
}
```

3、 工厂类：编写工厂生成代理

（1）、代理类：将目标类（切入点）和 切面类（通知） 结合 --> 切面，Proxy.newProxyInstance。

a、参数1：loader ，类加载器，动态代理类 运行时创建，任何类都需要类加载器将其加载到内存。

一般情况：当前类.class.getClassLoader()或者目标类实例.getClass().get ClassLoader()；

b、参数2：Class[] interfaces 代理类需要实现的所有接口

方式1：目标类实例.getClass().getInterfaces();注意：只能获得自己接口，不能获得父元素接口

方式2：new Class[]{UserService.class}

例如：jdbc 驱动 --> DriverManager获得接口 Connection

C、参数3：InvocationHandler 处理类，接口，必须进行实现类，一般采用匿名内部提供 invoke 方法，代理类的每一个方法执行时，都将调用一次invoke

a）、参数31：Object proxy ：代理对象

b）、参数32：Method method ：代理对象当前执行的方法的描述对象（反射）

执行方法名：method.getName()

执行方法：method.invoke(对象，实际参数)

c）、参数33：Object[] args :方法实际参数

```
public class MyBeanFactory
{
  public static UserService createService()
  {
    //1 目标类
    final UserService userService = new UserServiceImpl();
    //2切面类
    final MyAspect myAspect = new MyAspect();
    UserService proxService = (UserService)Proxy.newProxyInstance
                  (   MyBeanFactory.class.getClassLoader(),
                      userService.getClass().getInterfaces(),
                      new InvocationHandler()
                     {
                      @Override
                      public Object invoke(Object proxy, Method method, Object[] args)
                      {
                         //前执行
                         myAspect.before();
                         //执行目标类的方法
                         Object obj = method.invoke(userService, args);
                         //后执行
                         myAspect.after();
                         return obj;
                      }
                     }
                  );
          return proxService;
        }
  }
```

4、测试

```
@Test
public void demo01()
{
    UserService userService = MyBeanFactory.createService();
    userService.addUser();
    userService.updateUser();
    userService.deleteUser();
}
```

2、CGLIB 字节码增强

（1）、特点：

（a）、没有接口，只有实现类。

（b）、采用字节码增强框架 cglib，在运行时 创建目标类的子类，从而对目标类进行增强。

（c）、导入 jar 包，自己导包（了解）；

核心：hibernate-distribution-3.6.10.Final\lib\bytecode\cglib\cglib-2.2.jar

依赖：struts-2.3.15.3\apps\struts2-blank\WEB-INF\lib\asm-3.3.jar

spring-core..jar 已经整合以上两个内容

（2）、工厂类

```
public class MyBeanFactory
{
  public static UserServiceImpl createService()
   {
      //1 目标类
      final UserServiceImpl userService = new UserServiceImpl();
      //2切面类
      final MyAspect myAspect = new MyAspect();
      // 3.代理类 ， 采用cglib，底层创建目标类的子类
      //3.1 核心类
      Enhancer enhancer = new Enhancer();
      //3.2 确定父类
      enhancer.setSuperclass(userService.getClass());
      /* 3.3 设置回调函数 , MethodInterceptor接口 等效 jdk InvocationHandler接口
       *  intercept() 等效 jdk  invoke()
       *       参数1、参数2、参数3：以invoke一样
       *       参数4：methodProxy 方法的代理
       */
      enhancer.setCallback(new MethodInterceptor()
        {
          @Override
          public Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy)
            {
              myAspect.before();
              Object obj = method.invoke(userService, args);
              // * 执行代理类的父类 ，执行目标类 （目标类和代理类 父子关系）
              methodProxy.invokeSuper(proxy, args);
              myAspect.after();
              return obj;
          }
      });
      //3.4 创建代理
      UserServiceImpl proxService = (UserServiceImpl) enhancer.create();
      return proxService;
  }
}
```

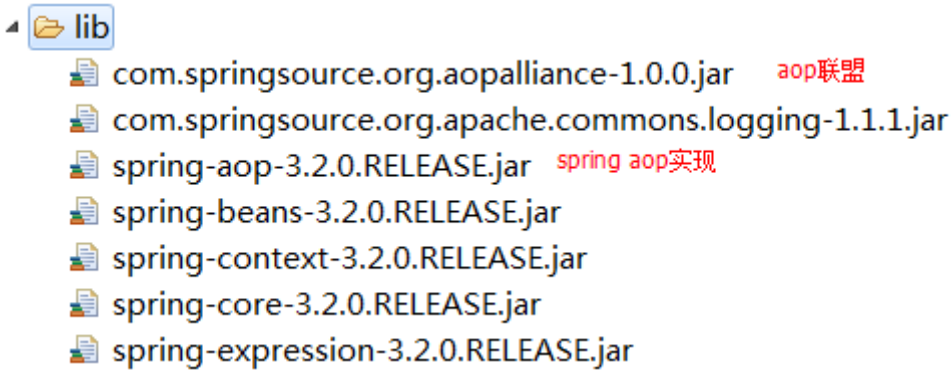
3、spring 编写代理:半自动

（1）、思路：让 spring 创建代理对象，从 spring 容器中手动的获取代理对象。

导入 jar 包：

核心：4+1

AOP：AOP 联盟（规范）、spring-aop （实现）



（2）、实现步骤：

a、目标类：

```
public interface UserService
{
    public void addUser();
    public void updateUser();
    public void deleteUser();
}
```

b、切面类：

```
/*
 * 切面类中确定通知，需要实现不同接口，接口就是规范，从而就确定方法名称。
 * 采用“环绕通知” MethodInterceptor
*/
public class MyAspect implements MethodInterceptor
{
  @Override
  public Object invoke(MethodInvocation mi) throws Throwable
   {
      System.out.println("前3");
      //手动执行目标方法
      Object obj = mi.proceed();
      System.out.println("后3");
      return obj;
  }
}
```

c、spring 配置

```
<!-- 1 创建目标类 -->
<bean id="userServiceId" class="com.itheima.b_factory_bean.UserServiceImpl"></bean>
<!-- 2 创建切面类 -->
<bean id="myAspectId" class="com.itheima.b_factory_bean.MyAspect"></bean>
<!-- 3 创建代理类
 a、使用工厂bean（FactoryBean），底层调用 getObject() 返回特殊bean;ProxyFactoryBean 用于创建
    代理工厂bean，生成特殊代理对象；
 b、参数：
   （a）、interfaces ： 确定接口们，通过<array>可以设置多个值只有一个值时，value="";
   （b）、target ： 确定目标类;
   （c）、interceptorNames ： 通知切面类的名称，类型String[]，如果设置一个值 value="";
   （d）、optimize :强制使用cglib:<property name="optimize" value="true"></property>.
 c、底层机制：
          如果目标类有接口，采用jdk动态代理
          如果没有接口，采用cglib字节码增强
          如果声明 optimize = true ，无论是否有接口，都采用cglib
-->
<bean id="proxyServiceId" class="org.springframework.aop.framework.ProxyFactoryBean">
      <property name="interfaces" value="com.itheima.b_factory_bean.UserService">
      </property>
      <property name="target" ref="userServiceId"></property>
      <property name="interceptorNames" value="myAspectId"></property>
</bean>
```

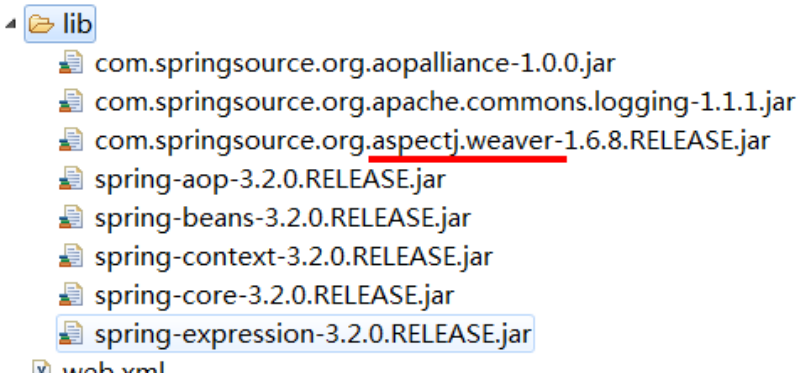
d、测试

```
@Test
public void demo01()
{
    String xmlPath = "com/itheima/b_factory_bean/beans.xml";
    ApplicationContext applicationContext = new ClassPathXmlApplicationContext(xmlPath);
    //获得代理类
    UserService userService = (UserService) applicationContext.getBean("proxyServiceId");
    userService.addUser();
    userService.updateUser();
    userService.deleteUser();
}
```

4、spring aop 编程：全自动

（1）、思路：

从 spring 容器获得目标类，如果配置 aop，spring 将自动生成代理。
要确定目标类，aspectj 切入点表达式，导入 jar 包。
spring-framework-3.0.2.RELEASE-dependencies\org.aspectj\
\com.springsource.org.aspectj.weaver\1.6.8.RELEASE



(2)、spring 配置

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       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.xsd">
```

```
<?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: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.xsd">
        <!-- 1 创建目标类 -->
        <bean id="userServiceId" class="com.itheima.c_spring_aop.UserServiceImpl">
        </bean>
        <!-- 2 创建切面类（通知） -->
        <bean id="myAspectId" class="com.itheima.c_spring_aop.MyAspect"></bean>
        <!-- 3 aop编程
             3.1 导入命名空间
             3.2 使用 <aop:config>进行配置
                      proxy-target-class="true" 声明时使用cglib代理
                     <aop:pointcut> 切入点 ， 从目标对象获得具体方法
                     <aop:advisor> 特殊的切面，只有一个通知 和 一个切入点
                      advice-ref 通知引用
                      pointcut-ref 切入点引用
             3.3 切入点表达式
                 execution(* com.itheima.c_spring_aop.*.*(..))
         -->
        <aop:config proxy-target-class="true">
            <aop:pointcut expression="execution(* com.itheima.c_spring_aop.*.*(..))"
                          id="myPointCut"/>
            <aop:advisor advice-ref="myAspectId" pointcut-ref="myPointCut"/>
        </aop:config>
    </beans>
```

（3）、测试

```
@Test
public void demo01()
{
    String xmlPath = "com/itheima/c_spring_aop/beans.xml";
    ApplicationContext applicationContext = new ClassPathXmlApplicationContext(xmlPath);
    //获得目标类
    UserService userService = (UserService) applicationContext.getBean("userServiceId");
    userService.addUser();
    userService.updateUser();
    userService.deleteUser();
}
```