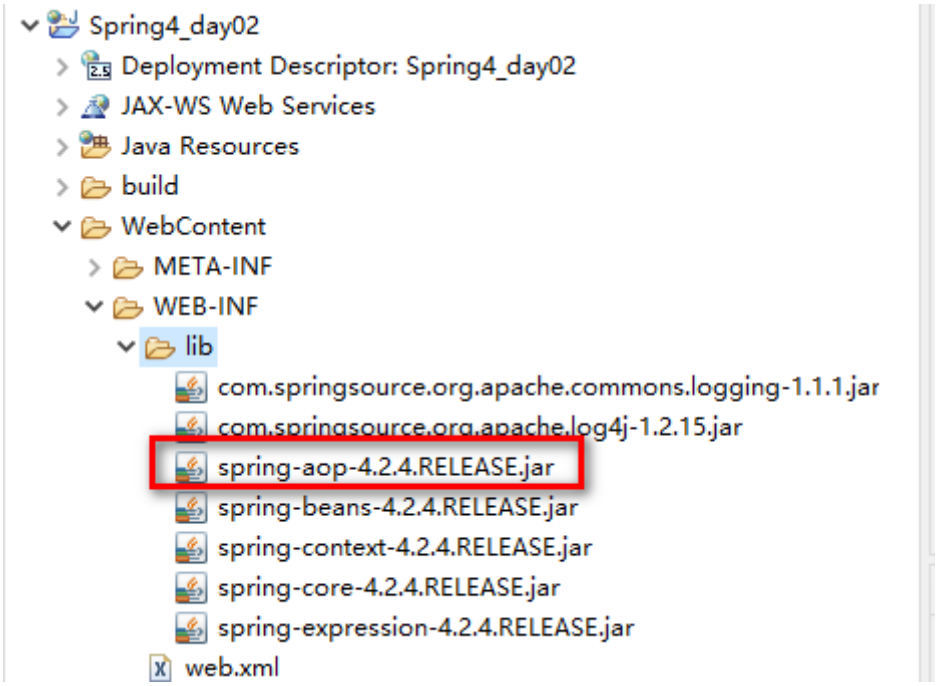
1.2 Spring 的 IOC 的注解开发

1.2.1 Spring 的 IOC 注解开发入门

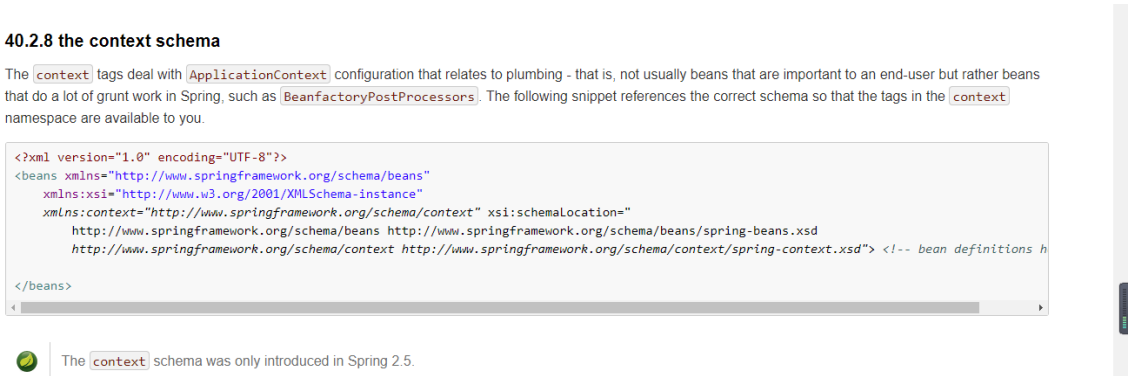
1.2.1.1 创建 web 项目,引入相关 jar 包

- 在 Spring4 的版本中,除了引入基本的开发包以外,还需要引入一个 AOP 的包



1.2.1.2 引入 Spring 的配置文件

- 在 src 下创建 applicationContext.xml
    - 引入约束:使用注解开发引入 context 约束
    - 约束:spring-framework-4.2.4.RELEASE\docs\spring-framework-reference\html\xsd-configuration.html



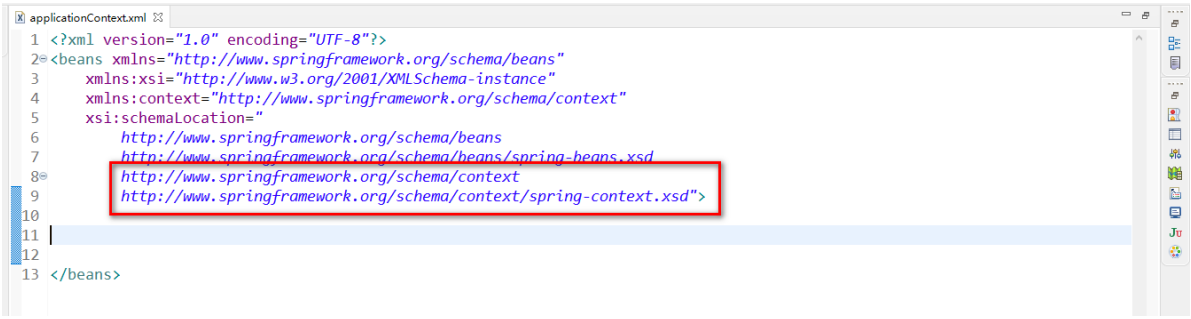
40.2.8 the context schema

The `context` tags deal with `ApplicationContext` configuration that relates to plumbing - that is, not usually beans that are important to an end-user but rather beans that do a lot of grunt work in Spring, such as `BeanfactoryPostProcessors`. The following snippet references the correct schema so that the tags in the `context` namespace are available to you.

```
<?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:context="http://www.springframework.org/schema/context" xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd"> <!-- bean definitions h

</beans>
```

The `context` schema was only introduced in Spring 2.5.



1.2.1.3 创建接口和实现类

1.2.1.4 开启 Spring 的组件扫描

```
<?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:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- ================= Spring IOC 注解开发入门  ================= -->
    <!-- 使用 IOC 的注解开发,配置 Spring 的组件扫描(哪些包下的类使用注解) -->
    <context:component-scan base-package="com.admiral.spring.demo1" />

</beans>
```

1.2.1.5 在类上添加注解

```
/**
* @Title: UserDaoImpl.java
* @Package com.admiral.spring.demo1
* @Description: 用户 Dao 的实现类
* @author 白世鑫
* @date 2020-10-10
* @version V1.0
*/
package com.admiral.spring.demo1;

import org.springframework.stereotype.Component;

@Component("userDao")//相当于在 xml 中配置 <bean id="userDao"
class="com.admiral.spring.demo1.UserDaoImpl"/>
public class UserDaoImpl implements UserDao {

    @Override
    public void save() {
```

```
        System.out.println("UserDaoImpl save 执行了......");
    }

}
```

1.2.1.6 编写测试类

```
/**
* @Title: SpringDemo1.java
* @Package com.admiral.spring.demo1
* @Description: Spring IOC 注解开发测试类
* @author 白世鑫
* @date 2020-10-10
* @version V1.0
*/
package com.admiral.spring.demo1;

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

public class SpringDemo1 {

    @Test
    // 传统方式
    public void demo1() {
        UserDao userDao = new UserDaoImpl();
        userDao.save();
    }
    @Test
    // 注解方式
    public void demo2() {
        ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml");
        UserDao userDao = (UserDao) applicationContext.getBean("userDao");
        userDao.save();
    }
}
```

1.2.1.7 注解方式注入属性的值

- 如果使用注解方式,可以没有 set 方法
    - 属性如果有 set 方法,需要将属性注入的注解添加到 set 方法上
    - 属性如果没有 set 方法,需要将属性注入的注解添加到属性上.

```
/**
* @Title: UserDaoImpl.java
* @Package com.admiral.spring.demo1
```

```java
* @Description: 用户 Dao 的实现类
* @author 白世鑫
* @date 2020-10-10
* @version V1.0
*/
package com.admiral.spring.demo1;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component("userDao") // 相当于在 xml 中配置 <bean id="userDao"
                      // class="com.admiral.spring.demo1.UserDaoImpl"/>
public class UserDaoImpl implements UserDao {

    @Value("小白白")
    private String name;

    /*
    @Value("小白白")
    public void setName(String name) {
        this.name = name;
    }
    */
    @Override
    public void save() {
        System.out.println("UserDaoImpl save 执行了....." + name);
    }

}
```

1.2.2 Spring 的 IOC 的注解详解

1.2.2.1 @Component:组件

- 修饰一个类,将这个类交给 Spring 管理
- 这个注解有三个衍生注解(功能类似)
    - @Controller :web层
    - @Service :Service层
    - @Repository :Dao层

1.2.2.2 属性注入的注解

- 普通属性
    - @Value :设置普通属性的值
- 对象类型属性
    - @Autowired :设置对象类型属性的值,但是按照类型完成属性的注入
        - 我们习惯按照名称完成属性的注入:必须让@Autowired注解和@Qualifier 一起使用完成按照名称属性注入.
    - @Resource :完成对象类型的属性的注入,按照名称完成属性的注入

```java
/**
* @Title: UserDaoImpl.java
* @Package com.admiral.spring.demo1
* @Description: 用户 Dao 的实现类
* @author 白世鑫
* @date 2020-10-10
* @version V1.0
*/
package com.admiral.spring.demo1;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component("userDao222") // 相当于在 xml 中配置 <bean id="userDao"
class="com.admiral.spring.demo1.UserDaoImpl"/>
public class UserDaoImpl implements UserDao {

    @Value("小白白")
    private String name;

    /*
    @Value("小白白")
    public void setName(String name) {
        this.name = name;
    }
    */
    @Override
    public void save() {
        System.out.println("UserDaoImpl save 执行了....." + name);
    }

}
```

```java
/**
* @Title: SpringDemo1.java
* @Package com.admiral.spring.demo1
* @Description: Spring IOC 注解开发测试类
* @author 白世鑫
* @date 2020-10-10
* @version V1.0
*/
package com.admiral.spring.demo1;

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

public class SpringDemo1 {

    @Test
    // 传统方式
    public void demo1() {
```

```
        UserDao userDao = new UserDaoImpl();
        userDao.save();
    }
    @Test
    // 注解方式
    public void demo2() {
        ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml");
        UserDao userDao = (UserDao) applicationContext.getBean("userDao");
        userDao.save();
    }

    @Test
    // 注解方式
    public void demo3() {
        ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml");
        UserService userService = (UserService)
applicationContext.getBean("userService");
        userService.save();
    }
}
```

```
/**
* @Title: SpringDemo1.java
* @Package com.admiral.spring.demo1
* @Description: Spring IOC 注解开发测试类
* @author 白世鑫
* @date 2020-10-10
* @version V1.0
*/
package com.admiral.spring.demo1;

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

public class SpringDemo1 {



    @Test
    public void demo3() {
        ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml");
        UserService userService = (UserService)
applicationContext.getBean("userService");
        userService.save();
    }
}
```

1.2.2.3 Bean 的其他注解

- 生命周期相关注解(了解)
  - @PostConstruct//相当于 init-method
  - @PreDestroy //相当于 init-destroy

```
<?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:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- ================= Spring IOC 注解开发入门  ================= -->
    <!-- 使用 IOC 的注解开发,配置 Spring 的组件扫描(哪些包下的类使用注解) -->
    <context:component-scan base-package="com.admiral.spring" />

</beans>
```

```
/**
* @Title: CustomerService.java
* @Package com.admiral.spring.demo2
* @Description:
* @author 白世鑫
* @date 2020-10-10
* @version V1.0
*/
package com.admiral.spring.demo2;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

import org.springframework.stereotype.Service;

@Service("customerService")
public class CustomerService {

    @PostConstruct//相当于 init-method
    public void init() {
        System.out.println("CustomerService 被初始化了.....");
    }


    public void save() {
        System.out.println("CustomerService 的 save 方法执行了......");
    }

    @PreDestroy //相当于 init-destroy
    public void destroy() {
```

```
        System.out.println("CustomerService 被销毁了.....");
    }
}
```

```
/**
* @Title: SpringDemo2.java
* @Package com.admiral.spring.demo2
* @Description: Bean 的其他相关注解测试类
* @author 白世鑫
* @date 2020-10-10
* @version V1.0
*/
package com.admiral.spring.demo2;

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

public class SpringDemo2 {

    @Test
    // 生命周期相关注解
    public void demo1() {
        ClassPathXmlApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml");
        CustomerService customerService = (CustomerService)
applicationContext.getBean("customerService");
        System.out.println(customerService);
        applicationContext.close();
    }
}
```

- Bean 作用范围的注解
    - @Scope
        - singleton :默认,单例
        - prototype :多例
        - request
        - session
        - globalsession

```
/**
* @Title: CustomerService.java
* @Package com.admiral.spring.demo2
* @Description:
* @author 白世鑫
* @date 2020-10-10
* @version V1.0
*/
```

```java
package com.admiral.spring.demo2;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Service;

@Service("customerService")
@Scope("prototype")
public class CustomerService {

    @PostConstruct//相当于 init-method
    public void init() {
        System.out.println("CustomerService 被初始化了.....");
    }


    public void save() {
        System.out.println("CustomerService 的 save 方法执行了......");
    }

    @PreDestroy //相当于 init-destroy
    public void destroy() {
        System.out.println("CustomerService 被销毁了......");
    }
}
```

```java
/**
* @Title: SpringDemo2.java
* @Package com.admiral.spring.demo2
* @Description: Bean 的其他相关注解测试类
* @author 白世鑫
* @date 2020-10-10
* @version V1.0
*/
package com.admiral.spring.demo2;

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

public class SpringDemo2 {

    @Test
    // Bean 作用范围
    public void demo2() {
        ClassPathXmlApplicationContext applicationContext = new
ClassPathXmlApplicationContext(
                "applicationContext.xml");
        CustomerService customerService1 = (CustomerService)
applicationContext.getBean("customerService");
```

```
        System.out.println(customerService1);

        CustomerService customerService2 = (CustomerService)
applicationContext.getBean("customerService");
        System.out.println(customerService2);
        applicationContext.close();
    }
}
```

1.2.3 IOC 的 XML 和 注解 开发比较

1.2.3.1 XML 和 注解 的比较

- 使用场景
  - XML: 可以使用任何场景
    - 结构清晰,维护方便
  - 注解: 有些地方用不了,这个类不是自己提供的
    - 开发方便

1.2.3.2 XML 和 注解 的整合开发

- XML 管理 bean , 注解完成属性的注入

ProductService.java

```
/**
* @Title: ProductService.java
* @Package com.admiral.spring.demo3
* @Description:
* @author 白世鑫
* @date 2020-10-10
* @version V1.0
*/
package com.admiral.spring.demo3;

import javax.annotation.Resource;

public class ProductService {

    @Resource(name = "productDao")
    private ProductDao productDao;
    @Resource(name = "orderDao")
    private OrderDao orderDao;

    /*
    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
    }
```

```
    public void setOrderDao(OrderDao orderDao) {
        this.orderDao = orderDao;
    }
    */

    public void save() {
        System.out.println("ProductService 的 Save 方法执行了......");
        productDao.save();
        orderDao.save();
    }
}
```

ProductDao.java

```
/**
* @Title: PeoductDao.java
* @Package com.admiral.spring.demo3
* @Description:
* @author 白世鑫
* @date 2020-10-10
* @version V1.0
*/
package com.admiral.spring.demo3;

public class ProductDao {

    public void save() {
        System.out.println("ProductDao 的 save 方法执行了......");
    }
}
```

OrderDao.java

```
/**
* @Title: OrderDao.java
* @Package com.admiral.spring.demo3
* @Description:
* @author 白世鑫
* @date 2020-10-10
* @version V1.0
*/
package com.admiral.spring.demo3;

public class OrderDao {

    public void save() {
        System.out.println("OrderDao 的 save 方法执行了......");
```

```
    }
}
```

applicationContext.xml

```
<?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:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- ================== Spring IOC 注解开发入门  ================== -->
    <!-- 使用 IOC 的注解开发,配置 Spring 的组件扫描(哪些包下的类使用注解) -->
<!--    <context:component-scan base-package="com.admiral.spring" /> -->



    <!-- ==================  IOC 的 XML 和 注解 整合开发  ================== -->
    <!-- 在没有扫描的情况下,使用属性注入的注解 @Resource @Value @Autowired @Qualifier--
>
    <context:annotation-config />
    <bean id="productService" class="com.admiral.spring.demo3.ProductService" >
</bean>


    <bean id="productDao" class="com.admiral.spring.demo3.ProductDao" />
    <bean id="orderDao" class="com.admiral.spring.demo3.OrderDao" />

</beans>
```

测试类

```
/**
* @Title: SpringDemo3.java
* @Package com.admiral.spring.demo3
* @Description: IOC XML 和注解 整合开发测试类
* @author 白世鑫
* @date 2020-10-10
* @version V1.0
*/
package com.admiral.spring.demo3;

import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
```

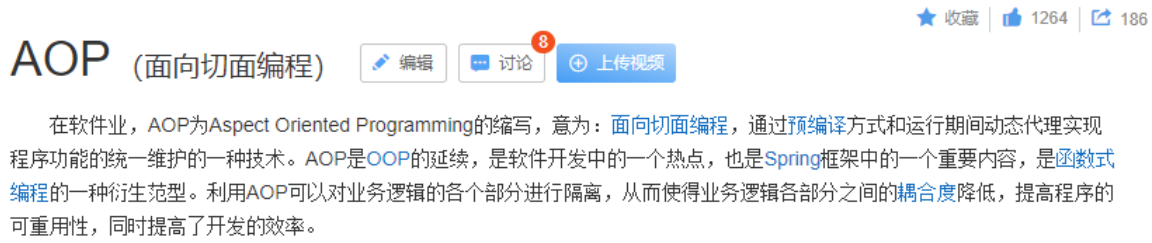
```
public class SpringDemo3 {

    @Test
    public void demo1() {
        ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml");
        ProductService productService = (ProductService)
applicationContext.getBean("productService");
        productService.save();
    }
}
```

1.3 Spring 的 AOP的 XML 开发

1.3.1 AOP 的基本概述

1.3.1.1 什么是 AOP



AOP:面向切面编程,AOP 是 OOP 的扩展和延伸,解决 OOP 开发遇到的问题

1.3.2 Spring 底层的AOP实现原理

- 动态代理
  - JDK动态代理: 只能对实现了接口的类产生代理
  - Cglib动态代理(类似于javassist第三方的代理技术): 对没有实现接口的类产生代理对象,生成子类对象

JDK 动态代理

UserDao.java

```
/**
* @Title: UserDao.java
* @Package com.admiral.spring.demo1
```

```
* @Description:
* @author 白世鑫
* @date 2020-10-10
* @version V1.0
*/
package com.admiral.spring.demo1;

public interface UserDao {

    public void save();
    public void update();
    public void delete();
    public void find();

}
```

UserDaoImpl.java

```
/**
* @Title: UserDaoImpl.java
* @Package com.admiral.spring.demo1
* @Description:
* @author 白世鑫
* @date 2020-10-10
* @version V1.0
*/
package com.admiral.spring.demo1;

public class UserDaoImpl implements UserDao {

    @Override
    public void save() {
        System.out.println("save....");
    }

    @Override
    public void update() {
        System.out.println("update....");
    }

    @Override
    public void delete() {
        System.out.println("delete....");
    }

    @Override
    public void find() {
        System.out.println("find....");
    }

}
```

JdkProxy.java

```
/**
* @Title: JdkProxy.java
* @Package com.admiral.spring.demo1
* @Description: 使用JDK动态代理产生UserDao代理
* @author 白世鑫
* @date 2020-10-10
* @version V1.0
*/
package com.admiral.spring.demo1;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class JdkProxy implements InvocationHandler {

    private UserDao userDao;

    public JdkProxy(UserDao userDao) {
        this.userDao = userDao;
    }

    /**
     * 返回代理对象
     */
    public UserDao createProxy() {

        UserDao userDaoProxy = (UserDao)
Proxy.newProxyInstance(userDao.getClass().getClassLoader(),
                userDao.getClass().getInterfaces(), this);

        return userDaoProxy;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        // 判断是否是你要增强的方法
        if ("delete".equals(method.getName())) {
            System.out.println("被增强了......");
        }
        return method.invoke(userDao, args);
    }

}
```

测试类:

```
/**
* @Title: SpringDemo1.java
* @Package com.admiral.spring.demo1
* @Description:
```

```
* @author 白世鑫
* @date 2020-10-10
* @version V1.0
*/
package com.admiral.spring.demo1;

import org.junit.Test;

public class SpringDemo1 {

    @Test
    public void demo1() {
        UserDao userDao = new UserDaoImpl();

        //创建代理
        UserDao proxy = new JdkProxy(userDao).createProxy();
        proxy.save();
        proxy.update();
        proxy.delete();
        proxy.find();
    }

}
```

Cglib 动态代理

CustomerDao.java

```
package com.admiral.spring.demo2;

public class CustomerDao {

    public void save() {
        System.out.println("save....");
    }

    public void update() {
        System.out.println("update....");
    }

    public void delete() {
        System.out.println("delete....");
    }

    public void find() {
        System.out.println("find....");
    }

}
```

CglibProxy.java

```
/**
* @Title: CglibProxy.java
* @Package com.admiral.spring.demo2
* @Description: Cglib  动态代理
* @author 白世鑫
* @date 2020-10-10
* @version V1.0
*/
package com.admiral.spring.demo2;

import java.lang.reflect.Method;

import org.springframework.cglib.proxy.Enhancer;
import org.springframework.cglib.proxy.MethodInterceptor;
import org.springframework.cglib.proxy.MethodProxy;

public class CglibProxy implements MethodInterceptor{

    private CustomerDao customerDao;

    public CglibProxy(CustomerDao customerDao) {
        this.customerDao = customerDao;
    }

    /**
     * 返回代理对象
     */
    public CustomerDao createProxy() {
        //1.创建 Cglib 的核心对象
        Enhancer enhancer = new Enhancer();
        //2.设置父类
        enhancer.setSuperclass(customerDao.getClass());
        //3.设置回调
        enhancer.setCallback(this);
        //4.创建代理对象
        CustomerDao proxy = (CustomerDao) enhancer.create();
        return proxy;
    }

    @Override
    public Object intercept(Object proxy, Method method, Object[] args,
MethodProxy methodProxy) throws Throwable {
        //判断是否是要增强的方法
        if("delete".equals(method.getName())) {
            System.out.println("权限校验.....");
        }
        return methodProxy.invokeSuper(proxy, args);
    }
}
```

测试类:

```
/**
* @Title: SpringDemo2.java
* @Package com.admiral.spring.demo2
* @Description:
* @author 白世鑫
* @date 2020-10-10
* @version V1.0
*/
package com.admiral.spring.demo2;

import org.junit.Test;

public class SpringDemo2 {

    @Test
    public void demo1() {
        CustomerDao customerDao = new CustomerDao();
        //创建代理对象
        CustomerDao proxy = new CglibProxy(customerDao).createProxy();
        proxy.save();
        proxy.update();
        proxy.delete();
        proxy.find();
    }
}
```

1.3.3 Spring 的 AOP 的开发

1.3.3.1 Spring 的 AOP 简介

- AOP 思想最早是由 AOP 联盟组织提出的. Spring 是使用这种思想最好的框架.
    - Spring 的 AOP 有自己的实现方式(非常繁琐),AspectJ 是一个 AOP 框架,Spring 引入 AspectJ 作为自身的 AOP 开发
    - Spring 有两套 AOP 开发方式
        - Spring 传统方式(弃用)
        - Spring 基于 AspectJ 的 AOP 开发(应用广泛)

1.3.3.2 AOP 开发相关术语

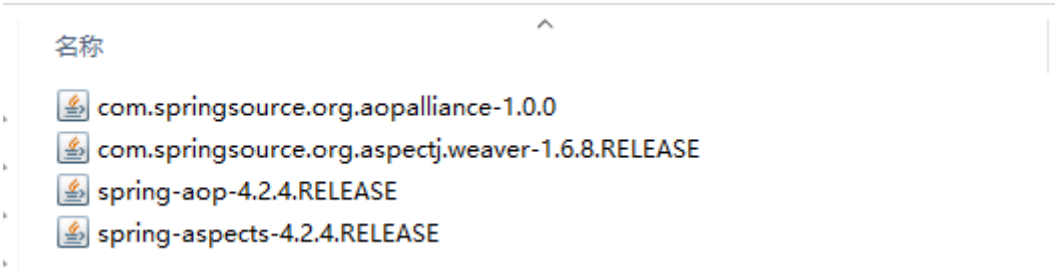
- Joinpoint(连接点) :
    - 所谓连接点是指那些被拦截到的点。在spring中,这些点指的是方法,因为spring只支持方法类型的连接点.
- Pointcut(切入点) :
    - 所谓切入点是指我们要对哪些Joinpoint进行拦截的定义.
- Advice(通知/增强) :
    - 所谓通知是指拦截到Joinpoint之后所要做的事情就是通知.通知分为前置通知,后置通知,异常通知,最终通知,环绕通知(切面要完成的功能)

- Introduction(引介) :
  - 引介是一种特殊的通知在不修改类代码的前提下, Introduction可以在运行期为类动态地添加一些方法或Field.
- Target(目标对象) :
  - 代理的目标对象
- Weaving(织入) :
  - 是指把增强应用到目标对象来创建新的代理对象的过程.spring采用动态代理织入，而AspectJ采用编译期织入和类装在期织入
- Proxy（代理） :
  - 一个类被AOP织入增强后，就产生一个结果代理类
- Aspect(切面) :
  - 是切入点和通知（引介）的结合

1.3.4 Spring 的 AOP 的入门

1.3.4.1 创建 web 项目,引入 jar 包

- 引入基本开发包
- 引入 aop 开发相关 jar 包



1.3.4.2 引入 Spring 的配置文件

- 引入 aop 约束

```
<?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">

</beans>
```

1.3.4.3 编写目标类并完成配置

ProductDao.java

```
/**
* @Title: ProductDao.java
* @Package com.admiral.spring.demo3
* @Description:
* @author 白世鑫
* @date 2020-10-12
* @version V1.0
*/
package com.admiral.spring.demo3;

public interface ProductDao {

    public void save();
    public void update();
    public void delete();
    public void find();
}
```

ProductDaoImpl.java

```
/**
* @Title: ProductDaoImpl.java
* @Package com.admiral.spring.demo3
* @Description:
* @author 白世鑫
* @date 2020-10-12
* @version V1.0
*/
package com.admiral.spring.demo3;

public class ProductDaoImpl implements ProductDao {

    @Override
    public void save() {
        System.out.println("保存商品....");
    }

    @Override
    public void update() {
        System.out.println("修改商品....");
    }

    @Override
    public void delete() {
        System.out.println("删除商品....");
    }

    @Override
    public void find() {
        System.out.println("查询商品....");
```

```
    }

}
```

applicationContext.xml

```
<?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">

        <bean id="productDao" class="com.admiral.spring.demo3.ProductDaoImpl">
</bean>

</beans>
```

1.3.4.4 编写测试类

- Spring 整合单元测试

```
/**
* @Title: SpringDemo1.java
* @Package com.admiral.spring.demo3
* @Description:
* @author 白世鑫
* @date 2020-10-12
* @version V1.0
*/
package com.admiral.spring.demo3;

import javax.annotation.Resource;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class SpringDemo3 {

    @Resource(name = "productDao")
    private ProductDao productDao;

    @Test
    public void demo1() {
        productDao.save();
```

```
        productDao.update();
        productDao.delete();
        productDao.find();
    }

}
```

1.3.4.5 编写一个切面类

- 编写切面

```
/**
* @Title: MyAspectXML.java
* @Package com.admiral.spring.demo3
* @Description: 切面类
* @author 白世鑫
* @date 2020-10-12
* @version V1.0
*/
package com.admiral.spring.demo3;

public class MyAspectXML {


    /*
     * 权限校验
     */
    public void checkPri() {
        System.out.println("权限校验......");
    }
}
```

- 将切面类交给 Spring 管理

```
<?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">

        <!-- 配置目标对象:被增强的对象 -->
        <bean id="productDao" class="com.admiral.spring.demo3.ProductDaoImpl" />

        <!-- 将切面类交给 Spring 管理 -->
        <bean id="myAspect" class="com.admiral.spring.demo3.MyAspectXML" />
```

```
</beans>
```

1.3.4.6 通过 AOP 配置实现

```
<?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">

        <!-- 配置目标对象:被增强的对象 -->
        <bean id="productDao" class="com.admiral.spring.demo3.ProductDaoImpl" />

        <!-- 将切面类交给 Spring 管理 -->
        <bean id="myAspect" class="com.admiral.spring.demo3.MyAspectXML" />

        <!-- 通过 AOP 的配置完成对目标类产生代理 -->
        <aop:config>
            <!-- 表达式配置哪些类的那些方法需要进行增强 -->
            <aop:pointcut expression="execution(*
com.admiral.spring.demo3.ProductDaoImpl.delete(..))" id="pointcut1"/>
            <!-- 配置切面 -->
            <aop:aspect ref="myAspect">
                <aop:before method="checkPri" pointcut-ref="pointcut1"/>
            </aop:aspect>
        </aop:config>
</beans>
```

1.3.5 Spring 中的通知类型

1.3.5.1 前置通知:在目标方法执行之前进行操作

- 前置通知:可以获取切入点信息

```
/**
* @Title: MyAspectXML.java
* @Package com.admiral.spring.demo3
* @Description: 切面类
* @author 白世鑫
* @date 2020-10-12
* @version V1.0
*/
package com.admiral.spring.demo3;

import org.aspectj.lang.JoinPoint;

public class MyAspectXML {
```

```
    /*
     * 权限校验
     */
    public void checkPri(JoinPoint joinPoint) {
        System.out.println("权限校验......" + joinPoint);
    }
}
```

1.3.5.2 后置通知:在目标方法执行之后进行操作

- 后置通知:可以获取方法返回值

```
<?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">

        <!-- 配置目标对象:被增强的对象 -->
        <bean id="productDao" class="com.admiral.spring.demo3.ProductDaoImpl" />

        <!-- 将切面类交给 Spring 管理 -->
        <bean id="myAspect" class="com.admiral.spring.demo3.MyAspectXML" />

        <!-- 通过 AOP 的配置完成对目标类产生代理 -->
        <aop:config>
            <!-- 表达式配置哪些类的那些方法需要进行增强 -->
            <aop:pointcut expression="execution(*
com.admiral.spring.demo3.ProductDaoImpl.delete(..))" id="pointcut1"/>
            <aop:pointcut expression="execution(*
com.admiral.spring.demo3.ProductDaoImpl.save(..))" id="pointcut2"/>
            <!-- 配置切面 -->
            <aop:aspect ref="myAspect">
                <!-- 前置通知 -->
                <aop:before method="checkPri" pointcut-ref="pointcut1"/>
                <!-- 后置通知 -->
                <aop:after-returning method="writeLog" pointcut-ref="pointcut2"
returning="result"/>
            </aop:aspect>
        </aop:config>
</beans>
```

```
/**
* @Title: MyAspectXML.java
* @Package com.admiral.spring.demo3
* @Description: 切面类
* @author 白世鑫
* @date 2020-10-12
```

```
* @version V1.0
*/
package com.admiral.spring.demo3;

import org.aspectj.lang.JoinPoint;

public class MyAspectXML {


    /*
     * 前置通知:权限校验
     */
    public void checkPri(JoinPoint joinPoint) {
        System.out.println("权限校验 ======." + joinPoint);
    }

    /*
     * 后置通知:日志记录
     */
    public void writeLog(Object result) {
        System.out.println("日志记录 ======." + result);
    }


}
```

```
/**
* @Title: ProductDaoImpl.java
* @Package com.admiral.spring.demo3
* @Description:
* @author 白世鑫
* @date 2020-10-12
* @version V1.0
*/
package com.admiral.spring.demo3;

public class ProductDaoImpl implements ProductDao {

    @Override
    public String save() {
        System.out.println("保存商品....");
        return "小黑黑";
    }

    @Override
    public void update() {
        System.out.println("修改商品....");
    }

    @Override
    public void delete() {
        System.out.println("删除商品....");
    }

    @Override
    public void find() {
```

```
        System.out.println("查询商品....");
    }

}
```

1.3.5.3 环绕通知:在目标方法执行之前,之后进行操作

- 环绕通知

```
<?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">

        <!-- 配置目标对象:被增强的对象 -->
        <bean id="productDao" class="com.admiral.spring.demo3.ProductDaoImpl" />

        <!-- 将切面类交给 Spring 管理 -->
        <bean id="myAspect" class="com.admiral.spring.demo3.MyAspectXML" />

        <!-- 通过 AOP 的配置完成对目标类产生代理 -->
        <aop:config>
            <!-- 表达式配置哪些类的那些方法需要进行增强 -->
            <aop:pointcut expression="execution(*
com.admiral.spring.demo3.ProductDaoImpl.delete(..))" id="pointcut1"/>
            <aop:pointcut expression="execution(*
com.admiral.spring.demo3.ProductDaoImpl.save(..))" id="pointcut2"/>
            <aop:pointcut expression="execution(*
com.admiral.spring.demo3.ProductDaoImpl.find(..))" id="pointcut3"/>
            <!-- 配置切面 -->
            <aop:aspect ref="myAspect">
                <!-- 前置通知 -->
                <aop:before method="checkPri" pointcut-ref="pointcut1"/>
                <!-- 后置通知 -->
                <aop:after-returning method="writeLog" pointcut-ref="pointcut2"
returning="result"/>
                <!-- 环绕通知 -->
                <aop:around method="around" pointcut-ref="pointcut3"/>
            </aop:aspect>
        </aop:config>
</beans>
```

```
/**
* @Title: MyAspectXML.java
* @Package com.admiral.spring.demo3
* @Description: 切面类
* @author 白世鑫
* @date 2020-10-12
```

```
* @version V1.0
*/
package com.admiral.spring.demo3;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;

public class MyAspectXML {


    /*
     * 前置通知:权限校验
     */
    public void checkPri(JoinPoint joinPoint) {
        System.out.println("权限校验 ======." + joinPoint);
    }

    /*
     * 后置通知:日志记录
     */
    public void writeLog(Object result) {
        System.out.println("日志记录 ======." + result);
    }

    /*
     * 环绕通知:性能监控
     */
    public Object around(ProceedingJoinPoint joinPoint) throws Throwable {
        System.out.println("环绕前通知 ======.");
        Object obj = joinPoint.proceed();//相当于执行 目标程序
        System.out.println("环绕后通知 ======.");
        return obj;
    }


}
```

1.3.5.4 异常抛出通知:在程序出现异常的时候进行操作

```
<?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">

        <!-- 配置目标对象:被增强的对象 -->
        <bean id="productDao" class="com.admiral.spring.demo3.ProductDaoImpl" />
```

```
        <!-- 将切面类交给 Spring 管理 -->
        <bean id="myAspect" class="com.admiral.spring.demo3.MyAspectXML" />

        <!-- 通过 AOP 的配置完成对目标类产生代理 -->
        <aop:config>
            <!-- 表达式配置哪些类的那些方法需要进行增强 -->
            <aop:pointcut expression="execution(*
com.admiral.spring.demo3.ProductDaoImpl.delete(..))" id="pointcut1"/>
            <aop:pointcut expression="execution(*
com.admiral.spring.demo3.ProductDaoImpl.save(..))" id="pointcut2"/>
            <aop:pointcut expression="execution(*
com.admiral.spring.demo3.ProductDaoImpl.find(..))" id="pointcut3"/>
            <aop:pointcut expression="execution(*
com.admiral.spring.demo3.ProductDaoImpl.update(..))" id="pointcut4"/>
            <!-- 配置切面 -->
            <aop:aspect ref="myAspect">
                <!-- 前置通知 -->
                <aop:before method="checkPri" pointcut-ref="pointcut1"/>
                <!-- 后置通知 -->
                <aop:after-returning method="writeLog" pointcut-ref="pointcut2"
returning="result"/>
                <!-- 环绕通知 -->
                <aop:around method="around" pointcut-ref="pointcut3"/>
                <!-- 异常抛出通知 -->
                <aop:after-throwing method="afterThrowing" pointcut-
ref="pointcut4" throwing="ex"/>
            </aop:aspect>
        </aop:config>
</beans>
```

```
/**
* @Title: MyAspectXML.java
* @Package com.admiral.spring.demo3
* @Description: 切面类
* @author 白世鑫
* @date 2020-10-12
* @version V1.0
*/
package com.admiral.spring.demo3;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;

public class MyAspectXML {


    /*
     * 前置通知:权限校验
     */
    public void checkPri(JoinPoint joinPoint) {
        System.out.println("权限校验 ======." + joinPoint);
    }

    /*
     * 后置通知:日志记录
```

```
     */
    public void writeLog(Object result) {
        System.out.println("日志记录 ======." + result);
    }

    /*
     * 环绕通知:性能监控
     */
    public Object around(ProceedingJoinPoint joinPoint) throws Throwable {
        System.out.println("环绕前通知 ======.");
        Object obj = joinPoint.proceed();//相当于执行 目标程序
        System.out.println("环绕后通知 ======.");
        return obj;
    }

    /*
     * 异常抛出通知
     */
    public void afterThrowing(Throwable ex) {
        System.out.println("异常抛出通知 ====== " + ex.getMessage());
    }


}
```

1.3.5.5 最终通知:无论代码是否有异常,总是会执行

```
<?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">

        <!-- 配置目标对象:被增强的对象 -->
        <bean id="productDao" class="com.admiral.spring.demo3.ProductDaoImpl" />

        <!-- 将切面类交给 Spring 管理 -->
        <bean id="myAspect" class="com.admiral.spring.demo3.MyAspectXML" />

        <!-- 通过 AOP 的配置完成对目标类产生代理 -->
        <aop:config>
            <!-- 表达式配置哪些类的那些方法需要进行增强 -->
            <aop:pointcut expression="execution(*
com.admiral.spring.demo3.ProductDaoImpl.delete(..))" id="pointcut1"/>
            <aop:pointcut expression="execution(*
com.admiral.spring.demo3.ProductDaoImpl.save(..))" id="pointcut2"/>
            <aop:pointcut expression="execution(*
com.admiral.spring.demo3.ProductDaoImpl.find(..))" id="pointcut3"/>
```

```
            <aop:pointcut expression="execution(*
com.admiral.spring.demo3.ProductDaoImpl.update(..))" id="pointcut4"/>
            <!-- 配置切面 -->
            <aop:aspect ref="myAspect">
                <!-- 前置通知 -->
                <aop:before method="checkPri" pointcut-ref="pointcut1"/>
                <!-- 后置通知 -->
                <aop:after-returning method="writeLog" pointcut-ref="pointcut2"
returning="result"/>
                <!-- 环绕通知 -->
                <aop:around method="around" pointcut-ref="pointcut3"/>
                <!-- 异常抛出通知 -->
                <aop:after-throwing method="afterThrowing" pointcut-
ref="pointcut4" throwing="ex"/>
                <!-- 最终通知 -->
                <aop:after method="after" pointcut-ref="pointcut4"/>
            </aop:aspect>
        </aop:config>
</beans>
```

1.3.5.6 引介通知(不用会)

1.3.6 Spring 的切入点表达式写法

- 基于 execution 函数完成的
- 语法
    - [访问修饰符] 方法返回值 包名.类名.方法名(参数)

```
public void com.admiral.spring.CustomerDao.save(..)
* *.*.*.*Dao.save(..)
* com.admiral.spring.CustomerDao+.save(..)
* com.admiral.spring..*.*(..)
```