

Spring5

SSM:SpringMVC + Spring + Mybatis

下面是要导的maven的包

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>5.3.16</version>
</dependency>
```

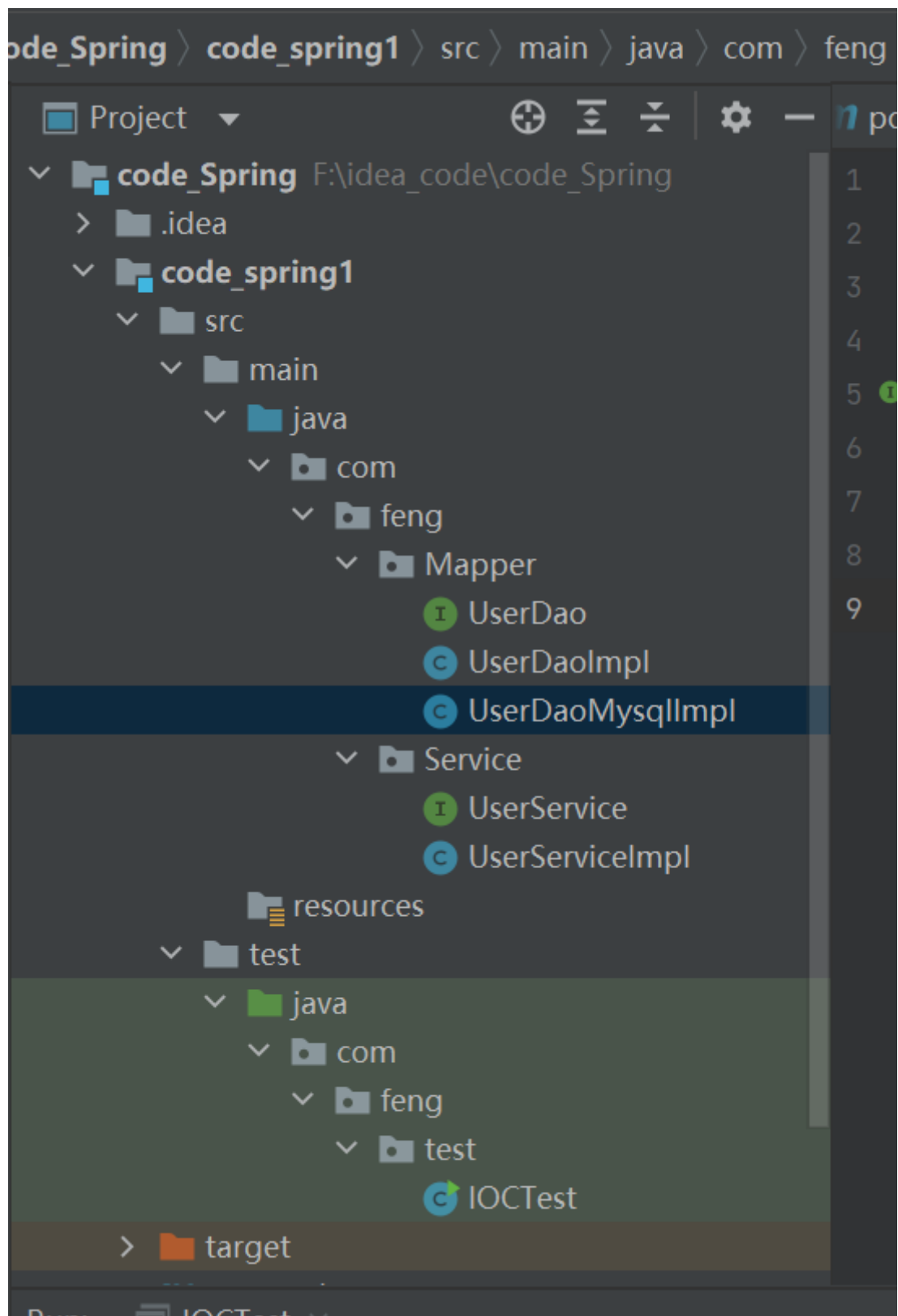
```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-jdbc -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>5.3.16</version>
</dependency>
```

控制反转IOC 面向切面AOP

1.IOC理论推导 important!

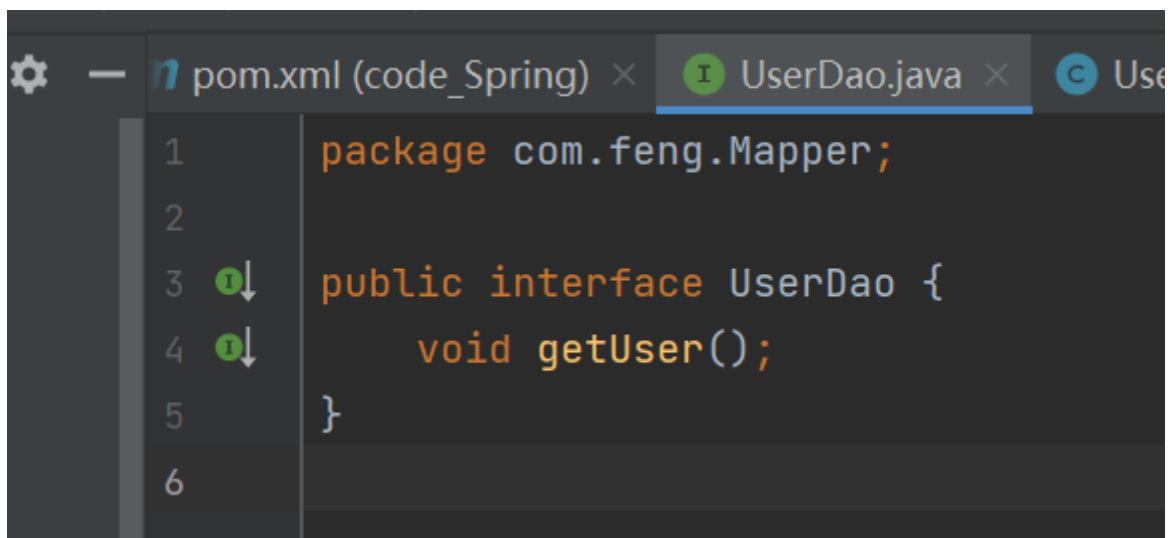
这里有接口层和服务层

1.文件布局



2.用户层

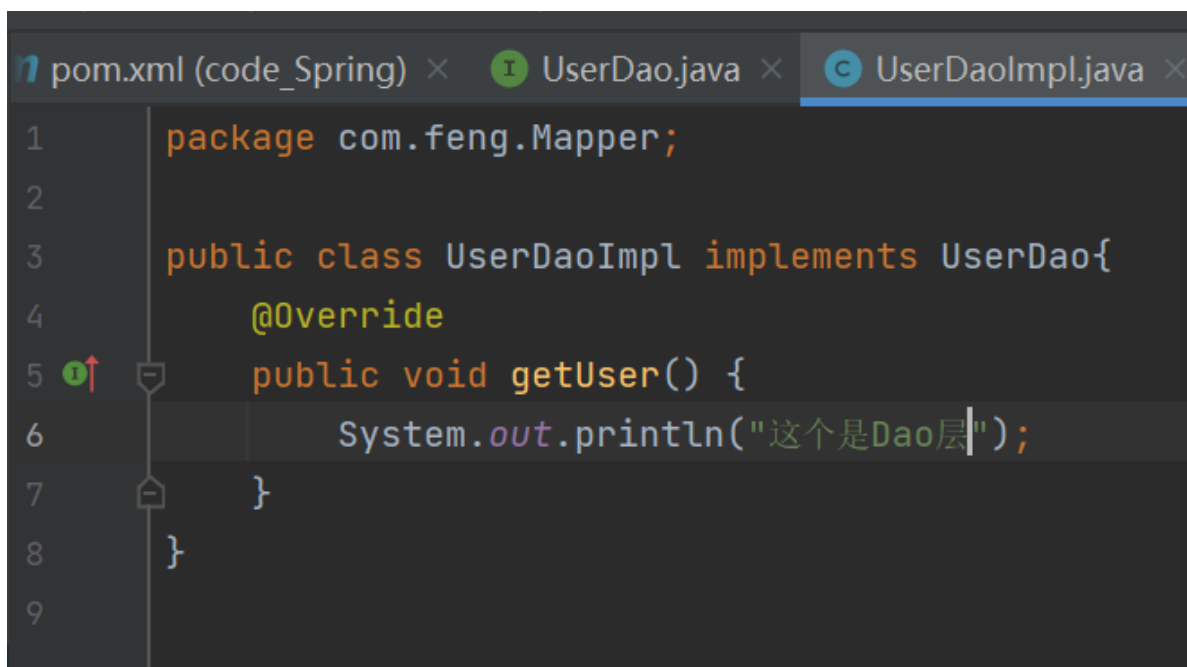
1.用户接口层



```
1 package com.feng.Mapper;
2
3 public interface UserDao {
4     void getUser();
5 }
6
```


2.用户实例层 可以分层

1.这个是普通类



```
1 package com.feng.Mapper;
2
3 public class UserDaoImpl implements UserDao{
4     @Override
5     public void getUser() {
6         System.out.println("这个 Dao层");
7     }
8 }
9
```

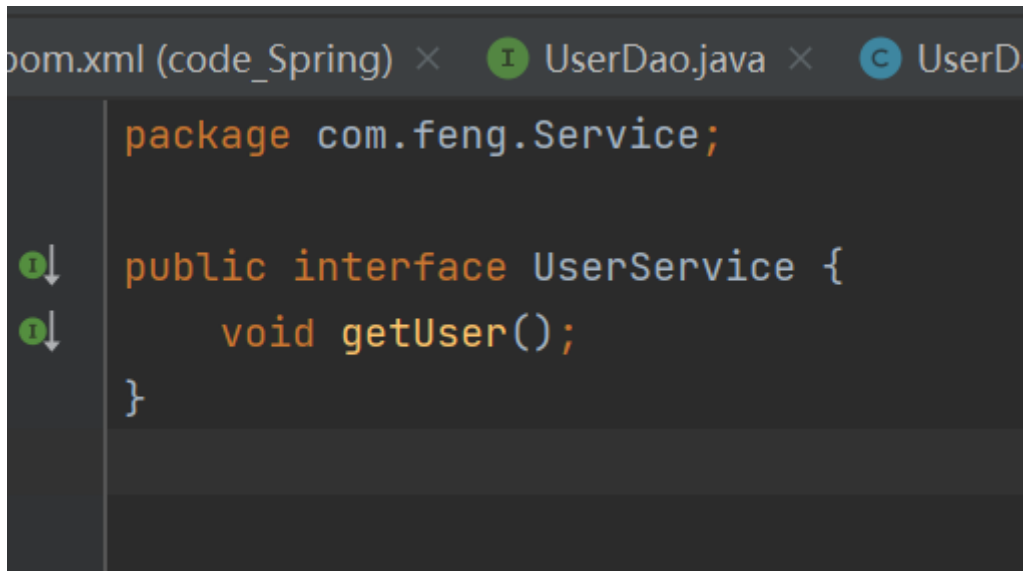
2.这个是Mysql层



```
1 package com.feng.Mapper;
2
3 public class UserDaoMysqlImpl implements UserDao{
4     @Override
5     public void getUser() {
6         System.out.println("这个 Mysql");
7     }
8 }
9
```

2.服务层

1.服务接口层



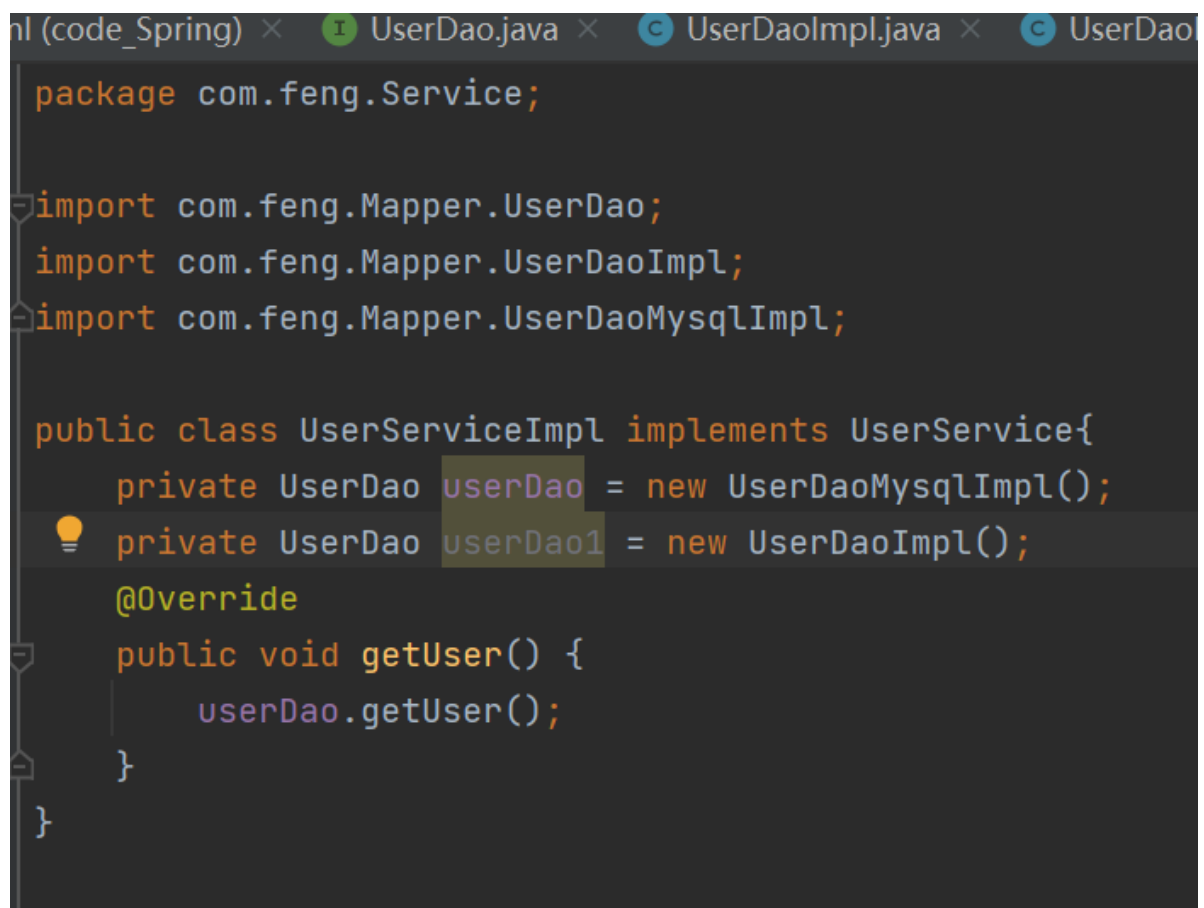
```
com.xml (code_Spring) × UserDao.java × UserDaoImpl.java × UserDaoMysqlImpl.java ×

package com.feng.Service;

public interface UserService {
    void getUser();
}
```

2.服务实体类

在这个类中可以实现多类调用，起选择服务的作用



```
com.xml (code_Spring) × UserDao.java × UserDaoImpl.java × UserDaoMysqlImpl.java ×

package com.feng.Service;

import com.feng.Mapper.UserDao;
import com.feng.Mapper.UserDaoImpl;
import com.feng.Mapper.UserDaoMysqlImpl;

public class UserServiceImpl implements UserService{
    private UserDao userDao = new UserDaoMysqlImpl();
    private UserDao userDao1 = new UserDaoImpl();

    @Override
    public void getUser() {
        userDao.getUser();
    }
}
```

在这个方法中是十分不好的，因为在选择服务类型的时候，会动态修改源代码

下面的代码十分重要

在这个代码中，添加了setUserDao的方法导致不需要再改底层代码，实现到动态切换业务的功能

```

public class UserServiceImpl implements UserService{
    private UserDao userDao;

    public UserDao getUserDao() {
        return userDao;
    }

    public void setUserDao(UserDao userDao) {
        this.userDao = userDao;
    }

    @Override
    public void getUser() {
        userDao.getUser();
    }
}

```

测试类的可以传入实例的对象来实现动态切换业务

```

package com.feng.test;

import com.feng.Mapper.UserDaoMySQLImpl;
import com.feng.Service.UserServiceImpl;

public class IOCTest {
    public static void main(String[] args) {
        UserServiceImpl userService = new UserServiceImpl();
        userService.setUserDao(new UserDaoMySQLImpl());
        userService.getUser();
    }
}

```

2.HelloSpring

1.实体类

```
UserService.java × UserServiceImpl.java × IOCTest.java  
  
package com.feng.Mapper;  
  
public class Hello {  
    private String str;  
  
    public Hello() {  
    }  
  
    public Hello(String str) {  
        this.str = str;  
    }  
  
    public String getStr() {  
        return str;  
    }  
  
    public void setStr(String str) {  
        this.str = str;  
    }  
  
    @Override  
    public String toString() {  
        return "Hello{" +  
            "str='" + str + '\'' +  
            '}';  
    }  
}
```

2.beans.xml文件

beans.xml是注册类的配置文件，也就是自动创建类的对象，本质就是调用类创建的set的方法，如果跟下图一样的配置下，注意该实体类中必须有无参构造器和相应变量的set的方法,这里的property是相应实体类下的属性名，bean的标签本质就是实体类的set方法来创建属性的值

```
e.java x UserServiceImpl.java x IOCTest.java x pom.xml (code_spring2) x IOCTest.java x pom.xml (code_Spring)
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">
  <bean id="hello" class="com.feng.Mapper.Hello">
    <property name="str" value="Spring"/>
  </bean>
</beans>
```

这里附上xml文件中的基本代码

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

</beans>
```

1.当然Spring是一个十分灵活的框架，怎么可能只让你使用无参构造器呢，下面就加入一种的新的参数，这下面的index是有参构造器中的变量的索引值，由他可以定向地设置相应的属性值，所以可以使用多条语句来调用有参构造器

```
e_spring2) x IOCTest.java x UserService.java x pom.xml (code_Spring) x Hello.java x code_spring1\...\beans.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"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">
  <bean id="hello" class="com.feng.Mapper.Hello">
    <constructor-arg index="0" value="Hello Spring"/>
  </bean>
</beans>
```

2.属性索引赋值的操作，一样可以实现上述的操作

```
e_spring2) x IOCTest.java x UserService.java x pom.xml (code_Spring) x Hello.java x code_spring1\...\beans.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"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">
  <bean id="hello" class="com.feng.Mapper.Hello">
    <constructor-arg type="java.lang.String" value="Hello Spring"/>
  </bean>
</beans>
```

3.名字索引，但是和上面的不一样，用的标签不一样

```
<bean id="hello" class="com.feng.Mapper.Hello" name="hello2">
    <constructor-arg name="str" value="Hello Spring"/>
</bean>
```

ApplicationContext in module code_spring2. File is included in 4 contexts.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="hello" class="com.feng.Mapper.Hello" name="hello2 hello4;hello5,hello6">
        <constructor-arg name="str">
            <value>Hello Spring</value>
        </constructor-arg>
    </bean>
    <alias name="hello" alias="hello3"/>
</beans>
```

以上三种就是设置相应变量的方法，推荐name的方法，这样准确

3.测试文件

这里要注意ClassPathXmlApplicationContext类的调用，还有该例子实例化为ApplicationContext

```
package com.feng;

import com.feng.Mapper.Hello;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class IOCTest {
    @Test
    public void test1(){
        ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
        Hello hello = (Hello) context.getBean("hello");
        System.out.println(hello.toString());
    }
}
```

4.总结

写了上面的思路后，就可把之前的代码可优化了，这里的bean中可以设置多变量实体类，在相应的引用类中，加入相应的配置就行了，ref是指的是引用类型的

代码优化如下：

在相应的resources下加入相应的xml文件就行了


```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="userdaoimpl" class="com.feng.Mapper.UserDaoImpl"/>
    <bean id="userdaomysqlimpl" class="com.feng.Mapper.UserDaoMysqlImpl"/>
    <bean id="userserviceimpl" class="com.feng.Service.UserServiceImpl">
        <property name="userDao" ref="userdaomysqlimpl"/>
    </bean>
</beans>
```

2.Spring配置

2.1别名

这种方法可以取别名，也就是多取一个名字，但是这个方法没啥用的

```
</bean>
<alias name="hello" alias="hello3"/>
</beans>
```

2.2Bean的配置

1.id

id是唯一标识该实体类实例化的变量名

2.class

class是指出该bean是实例化哪些实体类

3.name

name是和上面的alias标签有一样的功能的，所以上面的alias是没啥用的，这里面取得别名可以是多个的，两个之间的分隔符可以是多种的

```
<bean id="hello" class="com.feng.Mapper.Hello" name="hello2 hello4;hello5,hello6">
    <constructor-arg name="str" value="Hello Spring"/>
</bean>
```

2.3import的配置

这个import适合团队开发，这个xml可以实现汇总所有的xml文件，文件中的内容相同的时候也会自动去重，将不一样的留下来，一样只出现一次

```
UserService.java x pom.xml (code_Spring) x Hello.java x code_spring1\...\beans.xml x code_spring2\...\b
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">
    <import resource="beans.xml"/>
</beans>
```

3.属性依赖注入

1.构造器注入

前面的bean本质就是构造器注入，所以在这里不再累述

2.Set方式注入 important!

这里实现各种对象的注入，例如String,Address(这是一个实体类),String[],List<>,Map<,>,Set<>,Properties

实体类中的属性

```
public class SetInsert {  
    private String name;  
    private Address address;  
    private String[] books;  
    private List<String> hobbies;  
    private Map<String,String> card;  
    private Set<String> games;  
    private String wife;  
    private Properties info;
```

下面就是这些set调用构建的实例对象

```
<bean id="address" class="com.feng.Mapper.Address">
    <constructor-arg index="0">
        <value>浙江省杭州市</value>
    </constructor-arg>
</bean>
<bean id="setInsert" class="com.feng.Mapper.SetInsert">
    <property name="name">
        <value>java</value>
    </property>
    <property name="address">
        <ref bean="address"/>
    </property>
    <property name="books">
        <array>
            <value>西游记</value>
            <value>红楼梦</value>
            <value>水浒传</value>
            <value>三国演义</value>
        </array>
    </property>
</bean>
```

```

<property name="card">
  <map>
    <entry key="身份证" value="ru9uw9"/>
    <entry key="银行卡" value="r9w9"/>
  </map>
</property>
<property name="games">
  <set>
    <value>LOL</value>
    <value>COC</value>
  </set>
</property>
<property name="hobbies">
  <list>
    <value>敲代码</value>
    <value>听歌</value>
  </list>
</property>

```

```

<property name="info">
  <props>
    <prop key="password">fjiosf</prop>
    <prop key="username">hsdhf</prop>
  </props>
</property>
<property name="wife">
  <value></value>
</property>

```

注意这里的null属性的设置有两种方式

```
<property name="wife">
    <null/>
</property>
```

还有注意这里的测试类的写法，这里通过后面指定的实体类的类型来明确指示类型，所以可以直接实现，不需要强转

```
SetInsert setInsert = context.getBean("setInsert", SetInsert.class);
```

3.C命名和P命名空间方式注入

这里C命名空间和P命名空间的引入需要xml语句的支持，这两句语句加在xml文件的最上面

```
xmlns:p="http://www.springframework.org/schema/p"
xmlns:c="http://www.springframework.org/schema/c"
```

```
<?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"
    xmlns:c="http://www.springframework.org/schema/c"
    xmlns:p="http://www.springframework.org/schema/p"
    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">
```

C命名空间就是Constructs的方法，就是调用构造器来实现建立实体类型，下面的这两种方法就是相互对应的

```
<bean id="user1" class="com.feng.Mapper.User">
    <constructor-arg index="0">
        <value>fishd</value>
    </constructor-arg>
    <constructor-arg index="1">
        <value>jidos</value>
    </constructor-arg>
</bean>
<bean id="user2" class="com.feng.Mapper.User" c:passWord="jioasjf" c:userName="afsf"/>
```

P命名空间就是property的方法，就是通过set的方法，不过这里需要无参构造器的存在，下面就是这两种方法的对比

```
<bean id="user3" class="com.feng.Mapper.User">
    <property name="passWord">
        <value>hsidfd</value>
    </property>
    <property name="userName">
        <value>jsdijfi</value>
    </property>
</bean>
<bean id="user4" class="com.feng.Mapper.User" p:passWord="jiosjfo" p:userName="fisdjif"/>
```

4.Bean的作用域

Bean的作用域有这几个singleton,prototype,request,session和global session

1.singleton

单例模式

singleton就是指指的是，一个实体类生成的实例是同一个实体，id中的名字相同，相应的对象就相同

```
<bean id="user2" class="com.feng.Mapper.User" c:passWord="jioasjf" c:userName="afsf" scope="singleton"/>
```

Spring的代理模式默认都是单例的

2.prototype

多例模式

```
<bean id="user2" class="com.feng.Mapper.User" c:passWord="jioasjf" c:userName="afsf" scope="prototype"/>
```

prototype就是多例模式，指的是相同的id在再次查询下和之前查询的不同，就是hashCode不同

其他的参数都是只能在javaweb中使用的

3.Bean的自动装配

在Spring中有三种装配的方式

1.在xml中的配置

在xml文件中可以设置autowire的属性值，其中有byName和byType的值

下面这个是byName的方法，这个方法是通过和set方法下的属性名相同来进行自动赋值，前面的bean的注册是必要的，这要和相应的set的属性名要相同，这个点是十分重要的

```
<?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"
       xmlns:c="http://www.springframework.org/schema/c"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springfram

<bean id="dog" class="com.feng.Mapper.Dog"/>
<bean id="cat" class="com.feng.Mapper.Cat"/>
<bean id="people" class="com.feng.Mapper.People" autowire="byName">
    <property name="name">
        <value>feng</value>
    </property>
</bean>
</beans>
```

下面是autowire的byType的属性的运用，看下面的这个cat的id的是不和set方法的属性名的，但是通过设置byType的属性的方式，可以通过类型来自动匹配，十分重要，在这个方法中还可以把构建的id去掉

```

<?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"
       xmlns:c="http://www.springframework.org/schema/c"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd http://www.s
<bean id="dog" class="com.feng.Mapper.Dog"/>
<bean id="cat1" class="com.feng.Mapper.Cat"/>
<bean id="people" class="com.feng.Mapper.People" autowire="byType">
    <property name="name">
        <value>feng</value>
    </property>
</bean>
</beans>

```

people中的setCat的方法，这个方法的名字是十分重要的

```

public void setCat(Cat cat) {
    this.cat = cat;
}

```

2.利用注解实现自动装配

这个里面有@Autowired的属性设置，也就是自动装配属性的值，不过要在xml文件中设置相应的文件设置，在这个注解下，实体类文件中都可以不设置set的方法

```

<?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">

    <context:annotation-config/>

</beans>

```

xml文件的配置

```
<?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"
       xmlns:c="http://www.springframework.org/schema/c"
       xmlns:p="http://www.springframework.org/schema/p"
       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">

    <context:annotation-config/>
    <bean id="dog" class="com.feng.Mapper.Dog"/>
    <bean id="cat" class="com.feng.Mapper.Cat"/>
    <bean id="people" class="com.feng.Mapper.People">
        <property name="name">
            <value>feng</value>
        </property>
    </bean>
</beans>
```

实体类的配置，@Autowired

```
public class People {
    private String name;
    @Autowired
    private Dog dog;
    @Autowired
    private Cat cat;

    public People() {
    }
}
```

@Autowired中的required的属性值可以设置false和true，false代表是该属性可以为空，就是不设置

```
private String name;
@Autowired(required = false)
private Dog dog;
@Autowired
private Cat cat;
```

上面的方法可以通过设置@Nullable的注解的设置来实现上述的功能


```

public class People {
    private String name;
    @Autowired
    @Nullable
    private Dog dog;
    @Autowired
    private Cat cat;
}

```

在xml文件中的配置是这样的

```

<context:annotation-config/>
<bean id="cat" class="com.feng.Mapper.Cat"/>
<bean id="people" class="com.feng.Mapper.People">
    <property name="name">
        <value>feng</value>
    </property>
</bean>

```

在有些时候会使用到多个实体类，但是@Autowired中是不允许多个实体类的，所以要加入一个新的注解@Qualifier，来搭配@Autowired的相互使用

实体类的配置

```

public class People {
    private String name;
    @Autowired
    @Qualifier("dog111")
    private Dog dog;
    @Autowired
    private Cat cat;

    public People() {
    }
}

```

xml文件的设置，在这个文件中dog的id太多了，定位设置类的实例对象

```
http://www.springframework.org/sch
<context:annotation-config/>
<bean id="dog111" class="com.feng.Mapper.Dog"/>
<bean id="dog222" class="com.feng.Mapper.Dog"/>
<bean id="cat" class="com.feng.Mapper.Cat"/>
<bean id="people" class="com.feng.Mapper.People">
    <property name="name">
        <value>feng</value>
    </property>
</bean>
</beans>
```

和@Autowired功能相同的还有@Resource的注解，这个注解的功能和@Autowired的功能不同，@Autowired是通过属性的类型来查找，但是@Resource是先通过属性的名字来查找，再通过属性的类型来查找

实体类文件的配置

```
public class People {
    private String name;
    @Resource
    private Dog dog;
    @Resource
    private Cat cat;
}
```

xml文件的设置

```
<context:annotation-config/>
<bean id="dog111" class="com.feng.Mapper.Dog"/>S
<bean id="cat" class="com.feng.Mapper.Cat"/>
<bean id="people" class="com.feng.Mapper.People">
    <property name="name">
        <value>feng</value>
    </property>
</bean>
</beans>
```

在@Resource中还可以设置name的属性来实现上面的@Qualifier的功能

实体类文件的配置

```
public class People {
    private String name;
    @Resource(name = "dog222")
    private Dog dog;
    @Resource
    private Cat cat;
}
```

xml文件的配置

```

    http://www.springframework.org/schema/context
<context:annotation-config/>
<bean id="dog111" class="com.feng.Mapper.Dog"/>
<bean id="dog222" class="com.feng.Mapper.Dog"/>
<bean id="cat" class="com.feng.Mapper.Cat"/>
<bean id="people" class="com.feng.Mapper.People">
    <property name="name">
        <value>feng</value>
    </property>
</bean>
</beans>

```

beans > bean

4.使用注解开发

前提：注意aop的maven的导入包是否存在，Spring5已经自动加入了

1.注解的使用

1.注解@Component

实体类的配置

```

@Component
public class User {
    public String name = "feng";
}

```

xml文件的配置

```

resources / UserBeans.xml
MyTest.java × code_spring3\...\UserBeans.xml × PeopleBeans.xml × pom.xml (code_spring4) × User.java
<?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"
       xmlns:c="http://www.springframework.org/schema/c"
       xmlns:p="http://www.springframework.org/schema/p"
       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">
    <context:annotation-config/>
    <context:component-scan base-package="com.feng.Mapper"/>
</beans>

```

测试类的配置

这其中的getBean中的实体类名就是类名，而且只能是类名，别的名字报错，你改了类名就必须改bean中的名字，而且只能是小写的

```

@Test
public void test1(){
    ApplicationContext context = new ClassPathXmlApplicationContext( configLocation: "UserBeans.xml");
    User user = context.getBean( s: "user", User.class);
    System.out.println(user.name);
}

```

这上面的三个图就是实验环境，@Component就是xml文件中的标签的缩写在这个注解下的实体类会自动实体化，xml文件中的

```
<context:component-scan base-package="com.feng.Mapper"/>
```

这句话就是指定要扫描的包，这个package下的所有类都会被实例化，将这下面的包的注解生效

3. 衍生的注解

@Component 有几个衍生注解，我们在web开发

- dao 【@Repository】
- service 【@Service】
- controller 【@Controller】 I

4. 自动装配

2.@Value注解

这个注解是搭配上面的注解进行组合使用的，直接在该类中将属性进行赋值，这个注解可以放在属性名上还可以放在对应的set方法上

实体类中的配置

```
@Component
public class User {
    @Value("kuang")
    public String name;
}
```

其他和上面的保持一样

3.@Scope注解

这个注解就是xml文件中的scope的属性值的设置，和上面的功能是一样的

```
@Component
@Scope("prototype")
public class User {
    @Value("kuang")
    public String name;
}
```

2.纯java的注解使用 javaconfig

实体类的配置

```
@Component
public class User {
    public String name;

    public User() {
    }

    public User(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    @Value("feng")
    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "User{" +
```

config实体类的配置

@Configuration声明这个类是配置类，就是xml文件的配置

@ComponentScan和上面的bean.xml文件中的是一样的作用

@Bean是注册类的注解，所以这个方法名就是xml文件Bean中的id属性，这个bean的获取就是在getBean中设置方法名和该类的名字，这其中的new User()就是要实例化的对象，而且其中还能定义自己的方法在其中

@Import的使用就和xml文件中的一样的

```
code_spring3\...\UserBeans.xml × PeopleBeans.xml × pom.xml (code_spring4) ×  
  
package com.feng.config;  
  
import com.feng.Mapper.User1;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
@ComponentScan("com.feng.Mapper")  
public class UserConfig2 {  
    @Bean  
    public User1 setUser(){  
        return new User1();  
    }  
}
```

```
code_spring3\...\UserBeans.xml × PeopleBeans.xml × pom.xml (code_spring4) ×  
  
package com.feng.config;  
  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.Import;  
  
@Configuration  
@ComponentScan("com.feng.Mapper")  
@Import(UserConfig2.class)  
public class UserConfig {  
  
}
```

注意下面的测试类的书写

```
@Test  
public void test2(){  
    ApplicationContext context = new AnnotationConfigApplicationContext(UserConfig.class);  
    User user = context.getBean("user", User.class);  
    System.out.println(user);  
}
```

在这个测试类中使用了一个全新的类对象，AnnotationConfigApplicationContext，注解类内容

而且getBean中的bean对象是user，而且getUser一样可以使用来获取bean的对象，这个是Spring内部解决的事务

5.代理模式

1.静态代理

就是在用户和服务端之间建立一个代理商，来实现其他的功能

接口类的配置

```
package com.feng.Mapper;

public interface Rent {
    public void rent();
}
```

实体类的配置

```
package com.feng.Mapper;

public class Host implements Rent{

    @Override
    public void rent() {
        System.out.println("我是房东，我来租房了");
    }
}
```

代理实体类的配置，这里是组合，因为中介就是一个租房和收钱的组合体


```
package com.feng.Mapper;

public class Proxy implements Rent{
    private Host host;

    public Proxy() {
    }

    public Proxy(Host host) {
        this.host = host;
    }

    @Override
    public void rent() {
        seeHouse();
        host.rent();
        Fare();
    }

    public void seeHouse(){
        System.out.println("我是中介，我来帶你看房了");
    }

    public void Fare(){
        System.out.println("我是中介，我来收中介费了");
    }
}
```

测试类的配置

```

import com.feng.Mapper.Host;
import com.feng.Mapper.Proxy;
import org.junit.Test;

public class MyTest {
    @Test
    public void test1(){
        Host host = new Host();
        Proxy proxy = new Proxy(host);
        proxy.rent();
    }
}

```

在这个代理类中，实现中介代理事务并且完成他自己的任务，实现其他的功能，这就是代理的作用

2.动态代理

这个动态代理涉及一个类的实现，Proxy类和InvocationHandler接口，这个动态代理主要是实现了投影的功能，将要代理的事务投影到该本代理上，实现功能的再次实现

动态代理类的配置

这个可以作为一个工具类，在加层实体类，就可以实现动态代理加具体功能的实现了

```

public class Proxy1 implements InvocationHandler {
    private Object target;

    public Proxy1(Object target) {
        this.target = target;
    }

    public Object getProxy(){
        return Proxy.newProxyInstance(this.getClass().getClassLoader(), target.getClass().getInterfaces(), this);
    }

    public void seeHouse(){
        System.out.println("看房子了");
    }

    public void Fare(){
        System.out.println("收钱了");
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        seeHouse();
        Object invoke = method.invoke(target, args);
        Fare();
        return invoke;
    }
}

```

测试类的配置

```

import com.feng.Mapper.Host;
import com.feng.Mapper.Rent;
import com.feng.Utills.Proxy1;
import org.junit.Test;

public class MyTest {
    @Test
    public void test1(){
        Host host = new Host();
        Proxy1 proxy1 = new Proxy1(host);
        Rent proxy = (Rent) proxy1.getProxy();
        proxy.rent();
    }
}

```

其中的InvocationHandler是一个接口，要重写invoke方法，其中invoke的参数具体含义是

proxy:代理对象，只是反射机制调用方法的需要

method: proxy被反射机制用于调用的方法对象，可以调用getName方法来知道方法的名字

args: 调用方法的参数列表

Proxy.newProxyInstance是生成一个相应的代理类的方法，其中的参数的具体含义是

loader:用哪个类加载器去加载代理对象，也就是类对象

interfaces:动态代理类需要去实现的接口，也就是类接口

h:动态代理方法在执行时，会调用h里面的invoke方法去执行

6.AOP

0.介绍AOP

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。

导入maven的依赖

```

<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.8</version>
</dependency>

```

xml文件需要导入的值

```
<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"
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/context
http://www.springframework.org/schema/context/spring-context.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd">
</beans>
```

1.AOP实现方式一

1.建立一个业务接口层

```
package com.feng.pojo;

public interface UserService {
    public void add();
    public void delete();
}
```

2.建立一个业务实现层

```
package com.feng.pojo;

public class UserServiceImpl implements UserService{
    @Override
    public void add() {
        System.out.println("增加一个新的用户");
    }

    @Override
    public void delete() {
        System.out.println("减少了一个老用户");
    }
}
```

3.建立其他业务层

这里通过实现日志记录的功能来解释这个功能

第一张图中是方法实现之前调用的方法，**MethodBeforeAdvice**中有个需要重写的方法，其中的参数的含义为

- method:是要执行的目标对象的方法
- args:是需要的参数
- target:是目标对象，就是传入的对象

```
pom.xml (code_Spring) x UserService.java x UserServiceImpl.java x BeforeLog.java x AfterLog.java x
package com.feng.pojo;

import org.springframework.aop.MethodBeforeAdvice;
import java.lang.reflect.Method;

public class BeforeLog implements MethodBeforeAdvice {
    @Override
    public void before(Method method, Object[] args, Object target) throws Throwable {
        System.out.println(target.getClass().getName()+"的"+method.getName()+"被执行了");
    }
}
```

下面的这张图中是通过实现接口**AfterReturningAdvice**的接口来实现方法执行后被调用的方法，其中的参数和前面的**MethodBeforeAdvice**是一样的，这个returnValue是方法执行后需要返回的值，**这个参数的值就是中间调用的method的返回参数的值**，还有要注意的是，实现后置通知必须实现**AfterReturningAdvice**接口，同前置通知一样，后置通知有机会得到调用的方法、传入的参数以及目标对象，亦可以获得被通知方法的返回值。

```
pom.xml (code_Spring) x UserService.java x UserServiceImpl.java x BeforeLog.java x AfterLog.java x UserServiceBeans.xml x MyTest
package com.feng.pojo;

import org.springframework.aop.AfterReturningAdvice;
import java.lang.reflect.Method;

public class AfterLog implements AfterReturningAdvice {
    @Override
    public void afterReturning(Object returnValue, Method method, Object[] args, Object target) throws Throwable {
        System.out.println(target.getClass().getName()+"的"+method.getName()+"被执行了"+returnValue);
    }
}
```

4.xml文件的配置

其中的aop标签是通过beans中引入的

- aop:config是aop切片的配置标签头
- aop:pointcut是规定切片点，其中的expression的参数execution是十分重要的

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?
name-pattern(param-pattern) throws-pattern?)
```

- modifiers-pattern?是表示修饰符就是访问修饰符，如public,protected
- ret-type-pattern是返回类型，这个是必填的，如果使用通配符*代表任意的返回类型
- declaring-type-pattern?表示声明方法的类
- name-pattern表示方法名
- param-pattern表示方法的参数，如果使用通配符..则代表任意的参数
- throw-pattern?表示方法抛出的异常

例子：

```
execution(public * com.example..say*(..))
```

aop:advisor就是将前面的advice类来插入到pointcut中，实现AOP的分片的功能

```
pom.xml (code_Spring) × UserService.java × UserServiceImpl.java × BeforeLog.java × AfterLog.java × UserServiceBeans.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"
       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/context
http://www.springframework.org/schema/context/spring-context.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd">
  <bean id="userService" class="com.feng.pojo.UserService" />
  <bean id="beforeLog" class="com.feng.pojo.BeforeLog" />
  <bean id="afterLog" class="com.feng.pojo.AfterLog" />
  <aop:config>
    <aop:pointcut id="ServicePointCut" expression="execution(* com.feng.pojo.UserServiceImpl.*(..))" />
    <aop:advisor advice-ref="beforeLog" pointcut-ref="ServicePointCut" />
    <aop:advisor advice-ref="afterLog" pointcut-ref="ServicePointCut" />
  </aop:config>
</beans>
```

5.测试类

这其中的UserService是十分重要的，实现类和接口类的相互转换

```
java > MyTest > st1
pom.xml (code_Spring) × UserService.java × UserServiceImpl.java × BeforeLog.java × AfterLog.java × UserServiceBeans.xml ×
import com.feng.pojo.UserService;
import com.feng.pojo.UserServiceImpl;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MyTest {
    @Test
    public void test1(){
        ApplicationContext context = new ClassPathXmlApplicationContext("UserServiceBeans.xml");
        UserService userService = context.getBean("userService", UserService.class);
        userService.add();
    }
}
```

2.AOP实现方式二

自定义类来实现，在上面的基础加入一个新的类，新的标签

1.新建的类

```

package com.feng.pojo;

public class DiyPointCut {
    public void before(){
        System.out.println("这是之前的方法");
    }
    public void after(){
        System.out.println("这是之后的方法");
    }
}

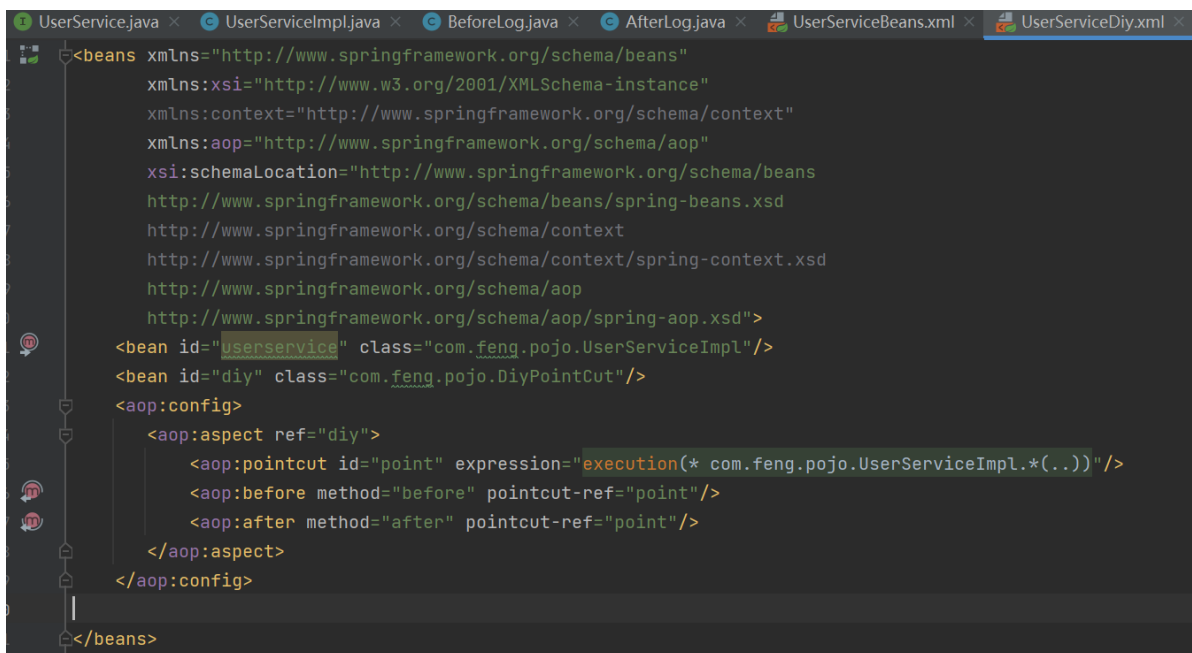
```

2.xml文件的配置

这个文件的配置是最重要的

其中有多之前没学过的标签

- aop:aspect这个代表是一个切面，通常是一个类，在里面可以定义切入点和通知
- aop:before这个是定义切点前的配置方法
- aop:after这个是定义切点后的配置方法



```

<?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"
       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/context
http://www.springframework.org/schema/context/spring-context.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd">
    <bean id="userService" class="com.feng.pojo.UserServiceImpl"/>
    <bean id="diy" class="com.feng.pojo.DiyPointCut"/>
    <aop:config>
        <aop:aspect ref="diy">
            <aop:pointcut id="point" expression="execution(* com.feng.pojo.UserServiceImpl.*(..))"/>
            <aop:before method="before" pointcut-ref="point"/>
            <aop:after method="after" pointcut-ref="point"/>
        </aop:aspect>
    </aop:config>
</beans>

```

3.注解实现AOP

这里要注意导入包是否正确

1.新建一个类

和上面的一样是用类来实现切口的

这里有@Aspect，@Before和@After的支持，当然还有好几个这样的方法，这其中的配置方法是和前面的配置是一样的，这个是在@Before中就实现了execution的接口，就相当于找到了接口处

```
eBeans.xml x UserServiceDiy.xml x pom.xml (code_Spring) x UserServiceAnnotation

package com.feng.pojo;

import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class UserServiceAnnotation {
    @Before("execution(* com.feng.pojo.UserServiceImpl.*(..))")
    public void before(){
        System.out.println("这个是之前的方法");
    }
    @After("execution(* com.feng.pojo.UserServiceImpl.*(..))")
    public void after(){
        System.out.println("这个是之后的方法");
    }
}
```

2.xml文件的配置

```
<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"
        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/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd">
    <aop:aspectj-autoproxy/>
</beans>
```

这个< aop:aspectj-autoproxy/>是代表开启注解实现AOP的意思

7.Mybatis-Spring

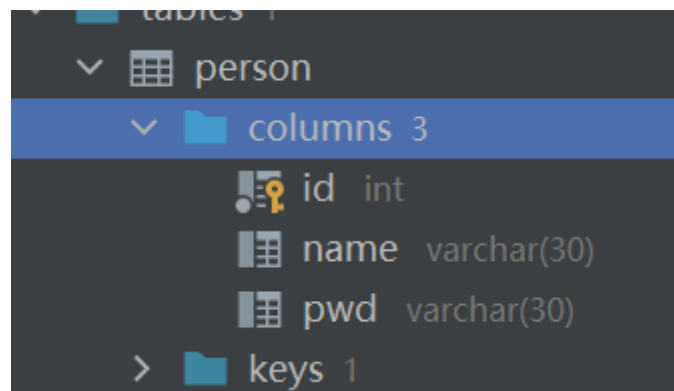
这下面的知识点十分的复杂，要注意逻辑

1.实体类的创建


```
package com.feng.pojo;

public class Person {
    private int id;
    private String name;
    private String pwd;
}
```

数据库的配置



2.接口类的配置

```
package com.feng.mapper;

import com.feng.pojo.Person;
import java.util.List;

public interface PersonMapper {
    List<Person> selectPerson();
}
```

3.xml文件的配置

```
mybatis-config.xml x Person.java x PersonMapper.java x PersonMapper.xml x
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
    PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.feng.mapper.PersonMapper">
    <select id="selectPerson" resultType="com.feng.pojo.Person">
        select * from person
    </select>
</mapper>
```

注意下面的配置，和之前的都不一样

4. Mybatis-Spring.xml文件的配置

这个文件的设定就是将之前的mybatis-config.xml文件给代替了，这其中有一部分的代码是恒定的

在这个文件中使用spring配置mybatis的datasource的属性，在下面的

org.mybatis.spring.SqlSessionFactoryBean的属性中配置相应的mapper.xml文件的路径，在这其中还可以配置config.xml的路径，这是将setting等的配置还是遗留在了config.xml之中，所以还是可以配置config.xml的路径的，在mapper.xml文件中还是想以前一样的写法，在下面还配置**sqlSession**的存在，就是通过这样的设置来达到不需要想以前一样的配置utils类了

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="datasource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost:3306/user?useSSL=true&useUnicode=true&characterEncoding=UTF-8&serverTimezone=GMT%2B8"/>
        <property name="username" value="root"/>
        <property name="password" value="root"/>
    </bean>
    <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="configLocation" value="classpath:mybatis-config.xml"/>
        <property name="mapperLocations" value="classpath:mapper/*.xml"/>
        <property name="dataSource" ref="datasource"/>
    </bean>
    <bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">
        <!-- 这里没有set的方法 -->
        <constructor-arg index="0" ref="sqlSessionFactory"/>
    </bean>
    <bean id="people" class="com.feng.mapper.PersonMapperImpl">
        <property name="sqlSession" ref="sqlSession"/>
    </bean>
</beans>
```

5. 新的实体实现类

在这个类中实现了之前的人personmapper接口的实体方法，在这其中私有了**sqlSession**，这就导致，只需要在**Spring.xml**中配置**bean**就可以注册并调用这个**sqlSession**了，并可以直接调用类中的实例方法来实现相应的功能了

```

package com.feng.mapper;

import com.feng.pojo.Person;
import org.mybatis.spring.SqlSessionTemplate;

import java.util.List;

public class PersonMapperImpl implements PersonMapper{
    private SqlSessionTemplate sqlSession;

    public void setSqlSession(SqlSessionTemplate sqlSession) {
        this.sqlSession = sqlSession;
    }

    @Override
    public List<Person> selectPerson() {
        PersonMapper mapper = sqlSession.getMapper(PersonMapper.class);
        List<Person> people = mapper.selectPerson();
        return people;
    }
}

```

6.测试类的配置

这个类就比较简单了，直接像以前一样拿到Bean的实体类，并使用其中的方法就行了

```

public class MyTest {
    @Test
    public void test1(){
        ApplicationContext context = new ClassPathXmlApplicationContext( configLocation: "Springxml/Spring.xml");
        PersonMapper people = context.getBean( s: "people", PersonMapper.class);
        List<Person> people1 = people.selectPerson();
        for (Person person : people1) {
            System.out.println(person);
        }
    }
}

```

在这个配置下，增删改查和之前一样配置就行，这是要在接口类的实体类中实现该方法

这就是例子

```
void updatePerson(Map map);
```

```

<update id="updatePerson" parameterType="map">
    update person set name=#{name} where id = #{id};
</update>

```

```

@Override
public void updatePerson(Map map) {
    PersonMapper mapper = sqlSession.getMapper(PersonMapper.class);
    mapper.updatePerson(map);
}

```

8.声明式事务

0.tx的引入

```
xmlns:tx="http://www.springframework.org/schema/tx"

http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd
```

其实可以发现这个aop和tx的引入是有规律的，自己发现

这个事务可以保证数据库的完整性和一致性，就是不会因为什么误操作导致数据库的变化

1.Spring.xml文件的配置 [Spring的事务实现](#)

tx使用了jdbc的manager来构建自己，而这个manager中有内置了rollback等的方法，所以可以用它来管理数据库的一致性的保护，通过tx的方法来实现的事务解决冲突性，aop来建立切片来使得包括在属性中的类都受到tx的管辖

DataSourceTransactionManager 提供了事务控制方法

- dataSource : DataSource
- enforceReadOnly : boolean
- DataSourceTransactionManager()
- DataSourceTransactionManager(DataSource)
- setDataSource(DataSource) : void
- getDataSource() : DataSource
- setEnforceReadOnly(boolean) : void
- isEnforceReadOnly() : boolean
- afterPropertiesSet() : void
- getResourceFactory() : Object
- doGetTransaction() : Object
- isExistingTransaction(Object) : boolean
- doBegin(Object, TransactionDefinition) : void** 开启事务
- doSuspend(Object) : Object
- doResume(Object, Object) : void
- doCommit(DefaultTransactionStatus) : void** 提交事务
- doRollback(DefaultTransactionStatus) : void** 回滚事务
- doSetRollbackOnly(DefaultTransactionStatus) : void
- doCleanupAfterCompletion(Object) : void

```
<bean class="org.springframework.jdbc.datasource.DataSourceTransactionManager" id="manager">
    <property name="dataSource" ref="datasource"/>
</bean>
<tx:advice transaction-manager="manager" id="interceptor">
    <tx:attributes>
        <tx:method name="*" propagation="REQUIRED"/>
    </tx:attributes>
</tx:advice>
<aop:config>
    <aop:pointcut id="pointCut" expression="execution(* com.feng.mapper.*(..))"/>
    <aop:advisor advice-ref="interceptor" pointcut-ref="pointCut"/>
</aop:config>
```

这个tx:method中的属性propagation(传播)中的值可以上网搜到，还有这个name的值指的是在下面配置的excution中的类的方法，经过这样的配置会使得这些收到manager的保护，进而保护数据库的完整性和一致性

最后的pom.xml文件

```
.  
<dependency>  
    <groupId>org.springframework</groupId>  
    <artifactId>spring-webmvc</artifactId>  
    <version>5.3.16</version>  
</dependency>  
<dependency>  
    <groupId>junit</groupId>  
    <artifactId>junit</artifactId>  
    <version>4.11</version>  
    <scope>test</scope>  
</dependency>  
<dependency>  
    <groupId>org.aspectj</groupId>  
    <artifactId>aspectjweaver</artifactId>  
    <version>1.9.8</version>  
</dependency>  
<dependency>  
    <groupId>org.aspectj</groupId>  
    <artifactId>aspectjrt</artifactId>  
    <version>1.9.8</version>  
</dependency>  
<dependency>  
    <groupId>mysql</groupId>  
    <artifactId>mysql-connector-java</artifactId>  
    <version>8.0.28</version>  
</dependency>  
<dependency>  
    <groupId>org.mybatis</groupId>  
    <artifactId>mybatis</artifactId>  
    <version>3.5.9</version>  
</dependency>  
<dependency>  
    <groupId>org.springframework</groupId>  
    <artifactId>spring-jdbc</artifactId>  
    <version>5.3.16</version>  
</dependency>  
<dependency>  
    <groupId>org.mybatis</groupId>  
    <artifactId>mybatis-spring</artifactId>  
    <version>2.0.7</version>  
</dependency>  
<dependency>  
    <groupId>aopalliance</groupId>  
    <artifactId>aopalliance</artifactId>  
    <version>1.0</version>  
</dependency>
```

```
<artifactId>aopalliance</artifactId>  
<version>1.0</version>  
...
```