

Spring简介：2002年，首次推出了Spring框架的雏形->interface 2.1框架，Spring框架以interface 2.1框架为基础，经过重新设计，并不断丰富其内涵，于2004年3月24日发布了1.0正式版。

Rod Johnson, Spring Framework创始人

Spring理念：使现有的技术更加容易使用，本身是一个大杂烩，整合了现有的技术框架。

SSH: Struts2+Spring+Hibernate

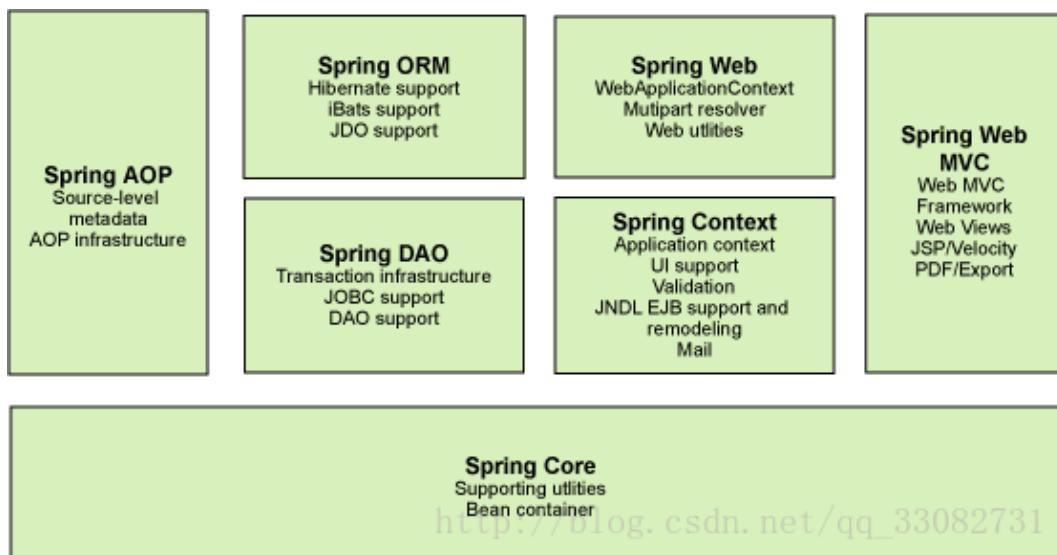
SSM: SpringMvc+Spring+Mybatis

Spring优点：1. 开源、免费的框架（容器）

2. 是一个轻量级的、非侵入式的框架
3. 控制反转（IOC），面向切面编程（AOP）
4. 支持事务的处理，对框架整合的支持

总结：Spring是一个轻量级的控制反转（IOC）和面向切面编程（AOP）的框架

Spring七大模块：



- SpringBoot
  - 1. 一个快速开发的脚手架
  - 2. 基于SpringBoot可以快速的开发单个微服务
  - 3. 约定大于配置
- SpringCloud
  - 1. SpringCloud是基于SpringBoot实现的

学习SpringBoot的前提，需要完全掌握Spring和SpringMVC，承上启下的作用。

## 2. 理论推导

- (1) UserDao接口
- (2) UserDaoImpl实现类
- (3) UserService业务接口
- (4) UserServiceImpl业务实现类

```

import com.kuang.dao.UserDao;

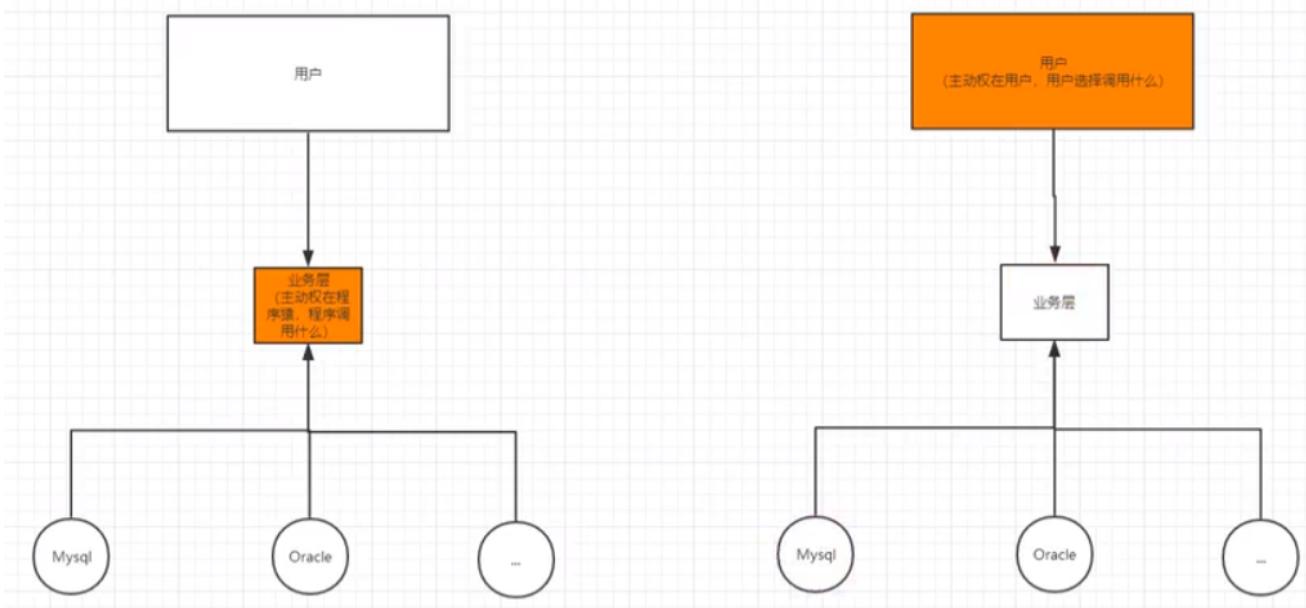
public class UserServiceImpl implements UserService {

    private UserDao userDao; // 利用set进行动态实现值的注入！

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

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

```



## IOC本质

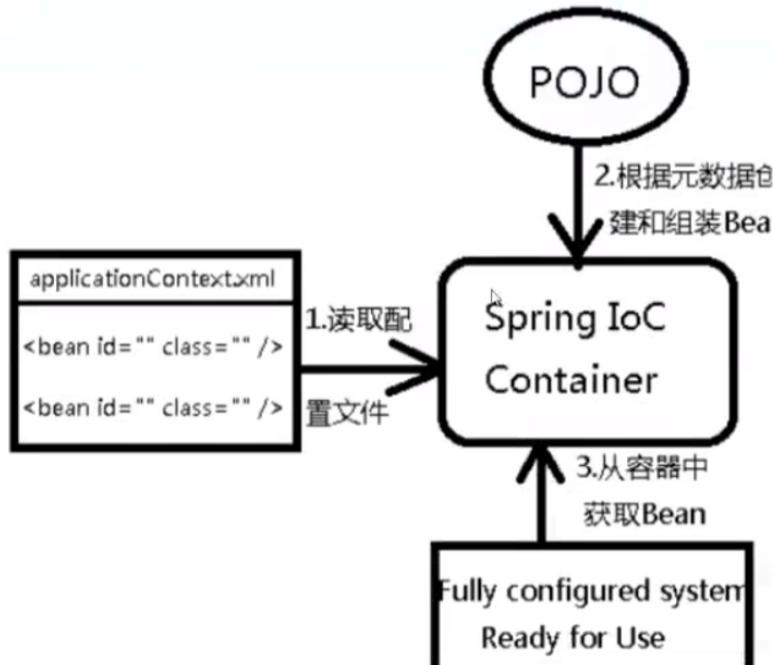
**控制反转IoC(Inversion of Control)**, 是一种设计思想, DI(依赖注入)是实现IoC的一种方法, 也有人认为DI只是IoC的另一种说法。没有IoC的程序中, 我们使用面向对象编程, 对象的创建与对象间的依赖关系完全硬编码在程序中, 对象的创建由程序自己控制, 控制反转后将对象的创建转移给第三方, 个人认为所谓控制反转就是: 获得依赖对象的方式反转了。



**IoC是Spring框架的核心内容**, 使用多种方式完美的实现了IoC, 可以使用XML配置, 也可以使用注解, 新版本的Spring也可以零配置实现IoC。

**IoC是Spring框架的核心内容**, 使用多种方式完美的实现了IoC, 可以使用XML配置, 也可以使用注解, 新版本的Spring也可以零配置实现IoC。

Spring容器在初始化时先读取配置文件, 根据配置文件或元数据创建与组织对象存入容器中, 程序使用时再从IoC容器中取出需要的对象。



采用XML方式配置Bean的时候, Bean的定义信息是和实现分离的, 而采用注解的方式可以把两者合为于一体, Bean的定义信息直接以注解的形式定义在实现类中, 从而达到了零配置的目的。

**控制反转是一种通过描述 (XML或注解) 并通过第三方去生产或获取特定对象的方式。在Spring中实现控制反转的是IoC容器, 其实现方法是依赖注入 (Dependency Injection,DI) 。**

```
<?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
                           https://www.springframework.org/schema/beans/spring-beans.xsd">
    <!--使用Spring来创建对象, 在Spring这些对象都称为Bean-->
    <bean id="hello" class="com.xiang.pojo.Hello">
        <property name="str" value="Spring"/>
    </bean>
</beans>
```

```
import com.xiang.pojo.Hello;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MyTest {
    public static void main(String[] args) {
        //获取Spring的上下文对象!
        ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");
        //我们的对象现在都在Spring中管理了, 我们要使用, 直接去里面取出来就可以了
        Hello hello = (Hello) context.getBean("hello");
        System.out.println(hello.toString());
    }
}
```

- Hello 对象是谁创建的 ?

hello 对象是由Spring创建的

- Hello 对象的属性是怎么设置的 ?

hello 对象的属性是由Spring容器设置的 ,

这个过程就叫控制反转 :



控制 : 谁来控制对象的创建 , 传统应用程序的对象是由程序本身控制创建的 , 使用Spring后 , 对象是由Spring来创建的 .

反转 : 程序本身不创建对象 , 而变成被动的接收对象 .

依赖注入 : 就是利用set方法来进行注入的 .

IOC是一种编程思想 , 由主动的编程变成被动的接收 .

可以通过newClassPathXmlApplicationContext去浏览一下底层源码 .

**OK , 到了现在 , 我们彻底不用再程序中去改动了 , 要实现不同的操作 , 只需要在xml配置文件中进行修改 , 所谓的IoC,一句话搞定 : 对象由Spring 来创建 , 管理 , 装配 !**

```
<?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="mysqlImpl" class="com.xiang.dao.UserDaoMysqlImpl"/>
    <bean id="oracleImpl" class="com.xiang.dao.UserDaoOracleImpl"/>

    <bean id="UserServiceImp1" class="com.xiang.service.UserServiceImp1">
        <!--
            ref: 引用Spring容器中创建好的对象
            value: 具体的值, 基本数据类型!
        -->
        <property name="userDao" ref="oracleImpl"/>
    </bean>
</beans>
```

```
import com.xiang.dao.UserDaoImpl;
import com.xiang.dao.UserDaoMysqlImpl;
import com.xiang.service.UserService;
import com.xiang.service.UserServiceImp1;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MyTest {
    public static void main(String[] args) {

        //用户实际调用的是业务层, dao层他们不需要接触
        // UserService userService = new UserServiceImpl();
        // UserService userService = new UserServiceImpl();
        // ((UserServiceImp1)userService).setUserDao(new UserDaoMysqlImpl());
        // userService.getUser();

        //设置Spring容器后
        //获取ApplicationContext:拿到Spring的容器
        ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");
        //拿到容器之后就可以使用容器中的所有对象了, 需要什么, 就直接get什么
        UserServiceImpl userServiceImpl = (UserServiceImp1)
context.getBean("UserServiceImp1");
        userServiceImpl.getUser();
```

```
}
```

## 4. IOC创建对象的方式

### (1) 默认使用无参构造方法创建对象。

```
<!--无参构造方法对象配置-->
<!--    <bean id="user" class="com.xiang.pojo.User">-->
<!--        <property name="name" value="xiang"/>-->
<!--    </bean>-->
```

### (2) 若要使用有参构造方法创建对象。

#### A. 下标赋值

```
<!--方法一：下标赋值-->
<bean id="user" class="com.xiang.pojo.User">
    <constructor-arg index="0" value="wang"/>
</bean>
```

#### B. 类型赋值

```
<!--方法二：类型赋值，不建议使用-->
<bean id="user" class="com.xiang.pojo.User">
    <!--type为基本类型直接用，引用类型写全名-->
    <constructor-arg type="java.lang.String" value="王祥太"/>
</bean>
```

#### C. 直接通过参数名设置

```
<!--方法三：直接通过参数名设置-->
<bean id="user" class="com.xiang.pojo.User">
    <constructor-arg name="name" value="王祥"/>
    <!--若变量是引用类型-->
    <!--若变量是引用类型-->
    <!--    <constructor-arg ref="beanone"/>-->
</bean>
<!--若变量是引用类型-->
<!--    <bean id="beanone" class="x.y.ThingOne"/>-->
```

总结：在配置文件加载的时候，容器中管理的对象就已经被初始化了。

## 5、Spring配置

### 5.1 别名

```
<!--别名，可以通过别名获取到对象-->
<alias name="user" alias="别名"/>
```

### 5.2、bean配置

```
<!--
id:bean的唯一表示，相当于对象名
class: bean对象所对应的全限定名：包名+类型
name: 也是别名，而且name可以同时取多个别名
-->
<bean id="userT" class="com.xiang.pojo.UserT" name="t1, t2">
    <property name="name" value="540"/>
</bean>
```

### 5.3、import

Import一般用于团队开发使用，它可以将多个配置文件，导入合并为一个applicationContext.xml文件

若项目中有几个人开发，这些人负责不同的类开发，不同的类需要注册在不同的bean中，我们可以利用import将所有的beans.xml文件合并为一个总的，使用的时候，直接使用总的配置就可以了。

```
<import resource="beans.xml"/>
<import resource="beans2.xml"/>
<import resource="beans3.xml"/>
```

## 6、依赖注入 (Dependency Injection, DI)

### 6.1、构造器注入

```
<!--依赖注入方式一：构造器注入-->
<bean id="exampleBean" class="example.ExampleBean">
    <constructor-arg>
        <ref bean="anotherExampleBean"/>
    </constructor-arg>
    <constructor-arg ref="yetAnotherBean"/>
    <constructor-arg type="int" value="1"/>    <property name="id">
        <value>
            </value>
    </property>
</bean>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
<bean id="anotherExampleBean" class="examples.AnotherBean"/>
```

### 6.2、set方式注入【重点】

依赖注入的本质是set注入

依赖：bean对象的创建依赖于容器

注入：bean对象中的所有属性，由容器来注入

主类

```
package com.xiang.pojo;

import java.util.*;

public class Student {
    //不同类型的set依赖注入
    private String name; //value
    private Address address; //ref
    private String[] books; //idref
    private List<String> hobbys; //list
    private Map<String, String> card; //map
    private Set<String> games; //set
    private Properties info; //配置类 props
    private String wife; //null

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }
}
```

```

}

public String[] getBooks() {
    return books;
}

public void setBooks(String[] books) {
    this.books = books;
}

public List<String> getHobbies() {
    return hobbies;
}

public void setHobbies(List<String> hobbies) {
    this.hobbies = hobbies;
}

public Map<String, String> getCard() {
    return card;
}

public void setCard(Map<String, String> card) {
    this.card = card;
}

public Set<String> getGames() {
    return games;
}

public void setGames(Set<String> games) {
    this.games = games;
}

public Properties getInfo() {
    return info;
}

public void setInfo(Properties info) {
    this.info = info;
}

public String getWife() {
    return wife;
}

public void setWife(String wife) {
    this.wife = wife;
}

@Override
public String toString() {
    return "student{" +
        "name=" + name + '\'' + +
        ", address=" + address.toString() +
        ", books=" + Arrays.toString(books) +
        ", hobbies=" + hobbies +
        ", card=" + card +
        ", games=" + games +
        ", info=" + info +
        ", wife=" + wife + '\'' +
        '}';
}
}

```

## Address

```
package com.xiang.pojo;

public class Address {
    private String address;

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    @Override
    public String toString() {
        return "Address{" +
            "address='" + address + '\'' +
            '}';
    }
}
```

## 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="add" class="com.xiang.pojo.Address">
        <property name="address" value="赣州"/>
    </bean>

    <bean id="student" class="com.xiang.pojo.Student">
        <!--依赖注入方式-->
        <!--第一种：基本类型注入 value-->
        <property name="name" value="望乡台"/>

        <!--第二种： bean注入， ref-->
        <property name="address" ref="add"/>

        <!--第三种： 数组注入， -->
        <property name="books">
            <array>
                <value>红楼梦</value>
                <value>西游记</value>
                <value>水浒传</value>
                <value>三国演义</value>
            </array>
        </property>

        <!--第四种： List注入-->
        <property name="hobbys">
            <list>
                <value>听歌</value>
                <value>敲代码</value>
                <value>写论文</value>
            </list>
        </property>

        <!--第五种： map注入-->
        <property name="card">
            <map>
                <entry key="a" value="啊"/>
                <entry key="b" value="吧"/>
            </map>
        </property>
    </bean>

```

```

        <entry key="c" value="撮"/>
    </map>
</property>

<!--第六种：set注入-->
<property name="games">
    <set>
        <value>LOL</value>
        <value>WOW</value>
    </set>
</property>

<!--第七种：null注入-->
<property name="wife">
    <null/>
</property>

<!--第八种：Properties注入-->
<property name="info">
    <props>
        <prop key="driver">com.mysql</prop>
        <prop key="url">localhost</prop>
        <prop key="username">root</prop>
        <prop key="password">123456</prop>
    </props>
</property>
</bean>
</beans>

```

## 测试类

```

import com.xiang.pojo.Student;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MyTest {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");
        Student stu = (Student) context.getBean("student");
        System.out.println(stu.toString());
    }
}

```

## 6.3、其它方式注入

### P命名空间注入

在pom.xml文件中导入单元测试依赖

```

<!-- https://mvnrepository.com/artifact/junit/junit -->
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>

```

导入xml约束，注入依赖，测试

```

xmlns:p="http://www.springframework.org/schema/p"
<!--p命名空间注入，可以直接注入属性的值：property-->
<bean id="user" class="com.xiang.pojo.User" p:name="望乡台" p:age="18"/>

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

```

}

## C命名空间注入

```
xmlns:c="http://www.springframework.org/schema/c
<!--命名空间注入，可以直接注入属性的值：property-->
<bean id="user" class="com.xiang.pojo.User" c:name="望乡台" c:age="18"/>

@Test
public void test2() {
    ApplicationContext context = new
ClassPathXmlApplicationContext("userbeans.xml");
    User user = context.getBean("user2", User.class);
    System.out.println(user);
}
```

## 6.4、

### Bean的作用域：

Scope	Description
singleton	(Default) Scopes a single bean definition to a single object instance for each Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
request	Scopes a single bean definition to the lifecycle of a single HTTP request. That is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.
session	Scopes a single bean definition to the lifecycle of an HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.
application	Scopes a single bean definition to the lifecycle of a ServletContext. Only valid in the context of a web-aware Spring ApplicationContext.
websocket	Scopes a single bean definition to the lifecycle of a WebSocket. Only valid in the context of a web-aware Spring ApplicationContext.

#### 1. Singeton: 单例模式，spring的默认模式

```
<bean id="user" class="com.xiang.pojo.User" p:name="望乡台" p:age="18"
scope="singleton"/>
User user = context.getBean("user", User.class); //原型模型每次从容器中getBean
的时候都会产生新的对象
User user2 = context.getBean("user", User.class); //单例模型每次从容器中
getBean的时候得到的是同一对象 System.out.println(user ==
user2); //scope="singteton" 得到true 所有实例共享一个对象
```

#### 2. Prototype

```
<bean id="user" class="com.xiang.pojo.User" p:name="望乡台" p:age="18"
scope="prototype"/> System.out.println(user == user2); //scope="prototype"
得到false //原型模型每次从容器中getBean的时候都会产生新的对象
```

#### 3. request

#### 4. session

#### 5. application

#### 6. websocket

其余的request、session、application，这些只能在web开发中使用到。

## 7、Bean的自动装配

自动装配是Spring满足bean依赖的一种方式

Spring会在上下文中自动寻找，并自动给bean装配属性

在Spring中有三种装配的方式：

#### 1. 在xml中显示的配置

## 2. 在java中显示配置

### 3. 隐式的自动装配bean【重要】

byName自动装配： byName会自动在容器上下文中查找，找到和自己引用对象属性的set方法后面的值对应的beanid。

byType自动装配： byType会自动在容器上下文中查找，找到和自己对象属性类型相同的bean的class。

```
<bean id="cat" class="com.xiang.pojo.Cat"/>
  <bean id="dog11" class="com.xiang.pojo.Dog"/>

  <!--byName会自动在容器上下文中查找，找到和自己对象set方法后面的值对应的beanid。-->
  <!--byType会自动在容器上下文中查找，找到和自己对象属性类型相同的bean的class。-->
<!--  <bean id="people" class="com.xiang.pojo.People" autowire="byName" -->
  <bean id="people" class="com.xiang.pojo.People" autowire="byType">
    <property name="name" value="王者荣耀"/>
<!--    <property name="dog" ref="dog"/> 手动装配-->
<!--    <property name="cat" ref="cat"/>-->
  </bean>
```

小结： byName的时候，需要保证所有bean的id唯一，并且这个bean需要和自动注入的属性的set方法后面的值一致。

byType的时候，需要保证所有bean的class唯一，并且这个bean需要和自动注入的对象的类型一致。

使用注解实现自动装配： jdk1.5支持注解，Spring2.5就支持注解了

使用注解须知：

- 导入约束： context约束
- 配置注解的支持

```
<?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
  https://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/context
  https://www.springframework.org/schema/context/spring-context.xsd">
  <context:annotation-config/>  </beans>
```

@Autowired

直接在属性上使用即可，也可以在set方法上使用

使用Autowired之后，可以不用编写set方法了，前提是自动装配的属性在IOC容器中存在，且符合名字byname

扩展：

```
//如果显示定义了Autowired的required属性为false，说明这个对象可以为null，否则不允许为
空
@.Autowired(required = false)
```

@Nullable 字段标记了这个注解，说明这个字段可以为null

自动装配时name和type都不同时如：

```
<bean id="cat11" class="com.xiang.pojo.Cat"/>
<bean id="cat111" class="com.xiang.pojo.Cat"/>
```

```
<bean id="dog22" class="com.xiang.pojo.Dog"/>
<bean id="dog222" class="com.xiang.pojo.Dog"/>
```

如果@Autowired自动装配的环境比较复杂，自动装配无法通过一个注解【@Autowired】完成的时候，我们可以使用@Qualifier (value=xxx) 配合@Autowired，指定一个唯一的bean对象注入。

```
@Autowired
@Qualifier(value = "cat11")
private Cat cat;
@Autowired
@Qualifier(value = "dog222")
private Dog dog;
private String name;
```

@Resource注解：Java自动装配@Resource，先找id和class，如果都无法确定装配对象，则使用@Resource(name="xxx")指定id装配。

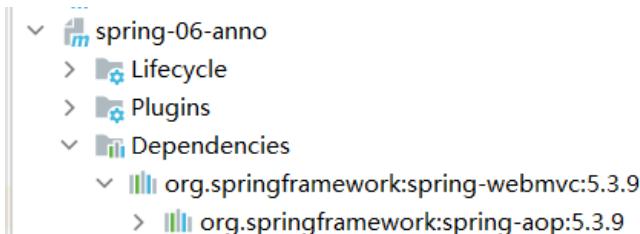
小结：

@Resource 和@ Autowired 的区别：

- 都是用来自动装配的，都可以放在属性字段上
- @ Autowired 通过byType的方式实现，而且必须要求这个对象存在！ 【常用】
- @ Resource 默认通过byname的方式实现，如果找不到名字，则通过byType实现！如果两个都找不到的情况下，就报错！ 【常用】
- 执行顺序不同：@ Autowired 通过byType的方式实现。@ Resource 默认通过byname的方式实现。

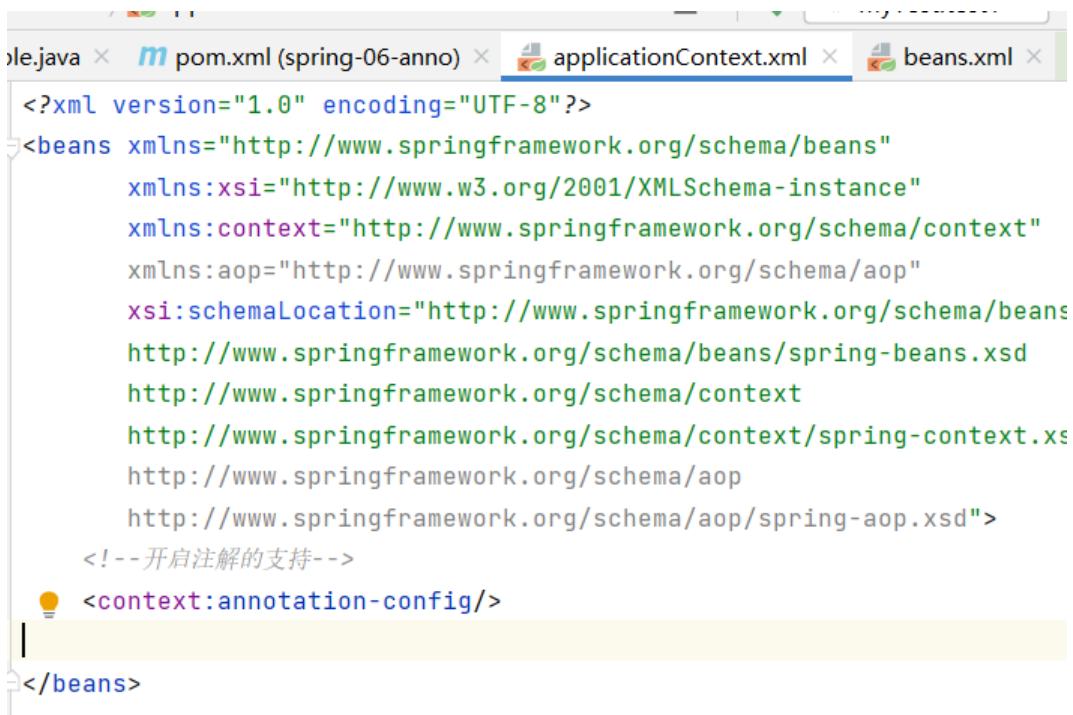
## 8、使用注解开发

在Spring4之后，要使用注解开发，必须保证aop的包导入了。



```
spring-06-anno
  > Lifecycle
  > Plugins
  <!-- Dependencies -->
    > org.springframework:spring-webmvc:5.3.9
      > org.springframework:spring-aop:5.3.9
```

使用注解需要导入context约束，增加注解支持



```
<?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">
    <!-- 开启注解的支持-->
    <context:annotation-config/>
</beans>
```

@Component: 组件，放在类上，说明这个类被Spring管理了，相当于bean。

@Value(“xxx”)实现属性注入的注解，可以放在字段声明上，也可以放在set方法上。

### (1) bean

```
<!--指定要扫描的包，扫描组件，这个包下的注解就会生效-->
<context:component-scan base-package="com.xiang.pojo"/>
<!--开启注解的支持-->
<context:annotation-config/>

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

@Component //等价于<bean id="user" class="com.xiang.pojo.User"/>
public class User {
```

### (2) 属性注入

```
@Value("婵娟")
public String name;
//也可以在set方法注入，重复注入会覆盖。
@Value("明月")
public void setName(String name) {
    this.name = name;
}
```

## 3. 衍生的注解

@component有几个衍生注解，我们在web开发中，会按照mvc三层架构分层

dao 【@Repository】

service 【@Service】

controller 【@Controller】

这四个注解，功能相同，都是将某个类注册到Spring中，装配bean

## 4. 自动装配

@Autowired: 自动装配，通过类型，名字，如果Autowired不能唯一自动装配属性，则需要通过

@Qualifier(value="xxx")

@Nullable 字段标记了这个注解，说明这个字段可以为null

@Resource: 自动装配，通过名字，类型。

## 5. 作用域

```
@Scope("prototype")
public class User {
```

## 6. 小结

XML与注解：

Xml更加万能，适用于任何场合，维护简单方便

注解 不是自己的类使用不了，维护相对复杂

Xml与注解最佳实践：

Xml用来管理bean

注解只负责完成属性的注入

我们在使用的过过程中，只需要注意一个问题，要想让注解生效，就需要开启注解的支持：

```
<!--指定要扫描的包，扫描组件，这个包下的注解就会生效-->
<context:component-scan base-package="com.xiang"/>
<!--开启注解的支持-->
```

```
<context:annotation-config/>
```

纯Java的配置方式：

实体类

```
package com.xiang.pojo;

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

public class User {
    private String name;

    public String getName() {
        return name;
    }
    @Value("查尔斯")
    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "User{" +
            "name='" + name + '\'' +
            '}';
    }
}
```

配置文件

```
package com.xiang.config;

import com.xiang.pojo.User;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;

//这个也会被Spring容器托管，注册到容器中，因为它本身就是一个@Component
//@Configuration代表这是一个配置类，和我们之前看的beans.xml一样
@Configuration
@ComponentScan("com.xiang.pojo")
@Import(MyConfig2.class) //引入配置类
public class MyConfig {
    /*
    注册一个bean，就相当于我们之前写的一个bean标签<bean>
    这个方法的名字，就相当于bean标签中的id属性
    这个方法的返回值，就相当于bean标签中的class属性
    */
    @Bean
    public User user() {
        return new User(); //就是返回要注入到bean的对象
    }
}
```

用@Configuration代替xml方式启动容器

```
/*
 * Jdk代理：基于接口的代理，一定是基于接口，会生成目标对象的接口的子对象。
 * Cglib代理：基于类的代理，不需要基于接口，会生成目标对象的子对象。
 * 1. 注解@EnableAspectJAutoProxy开启代理；
 * 2. 如果属性proxyTargetClass默认为false，表示使用jdk动态代理织入增强；
 * 3. 如果属性proxyTargetClass设置为true，表示使用Cglib动态代理技术织入增强；
```

```
* 4. 如果属性proxyTargetClass设置为false，但是目标类没有声明接口。  
* Spring aop还是会使用Cglib动态代理，也就是说非接口的类要生成代理都用Cglib。  
*/
```

## 测试类

```
import com.xiang.config.MyConfig;  
import com.xiang.pojo.User;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
public class MyTest {  
    public static void main(String[] args) {  
        //如果完全使用了配置类方式去做，我们就只能通过AnnotationConfig上下文来获取容器，通过  
        //配置类的Class对象加载  
        ApplicationContext context = new  
        AnnotationConfigApplicationContext(MyConfig.class);  
        User getUser = (User) context.getBean("user");  
        System.out.println(getUser.getName());  
    }  
}
```

这种纯Java的配置方式，在SpringBoot中随处可见

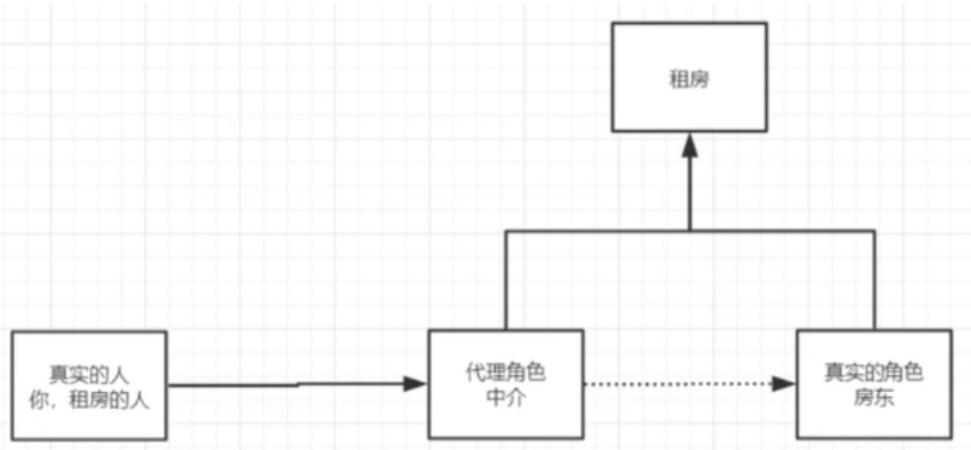
## 10、代理模式（23种设计模式之一）

为什么要学习代理模式，因为这是SpringAOP的底层 【SpringAOP和SpringMVC】面试必问

代理模式的分类：

1. 静态代理

2. 动态代理



### 10.1 静态代理

角色分析：

抽象角色：一般会使用接口或者抽象类来解决

真实角色：被代理的角色

代理角色：代理真实角色，代理真实角色后，我们一般会做一些附属操作

客户：访问代理对象的人

代理的好处：

可以使真实角色的操作更加纯粹，不用去关注一些公共的业务

公共业务交给代理角色，实现了业务的分工

公共业务发生扩展的时候，方便集中管理

缺点：

一个真实角色就会产生一个代理角色，代码量翻倍，开发效率会变低

代码步骤：

### 1. 接口

```
package com.xiang.demo02;

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

### 2. 真实角色

```
package com.xiang.demo02;

//真实对象
public class UserServiceImpl implements UserService{
    @Override
    public void add() {
        System.out.println("增加了一个用户");
    }

    @Override
    public void delete() {
        System.out.println("删除了一个用户");
    }

    @Override
    public void update() {
        System.out.println("修改了一个用户");
    }

    @Override
    public void query() {
        System.out.println("查询了一个用户");
    }
    //新增功能去改源代码是开发中的大忌
}
```

### 3. 代理角色

```
package com.xiang.demo02;

public class UserServiceProxy implements UserService{
    private UserService userService;

    public void setUserService(UserService userService) {
        this.userService = userService;
    }

    @Override
    public void add() {
```

```

        log("add");
        userService.add();
    }

@Override
public void delete() {
    log("delete");
    userService.delete();
}

@Override
public void update() {
    log("update");
    userService.update();
}

@Override
public void query() {
    log("query");
    userService.query();
}

//在不改变UserServiceIml类的基础上，使用代理新增日志功能
public void log(String msg) {
    System.out.printf("使用" + msg + "方法");
}
}

```

#### 4. 客户端访问代理角色

```

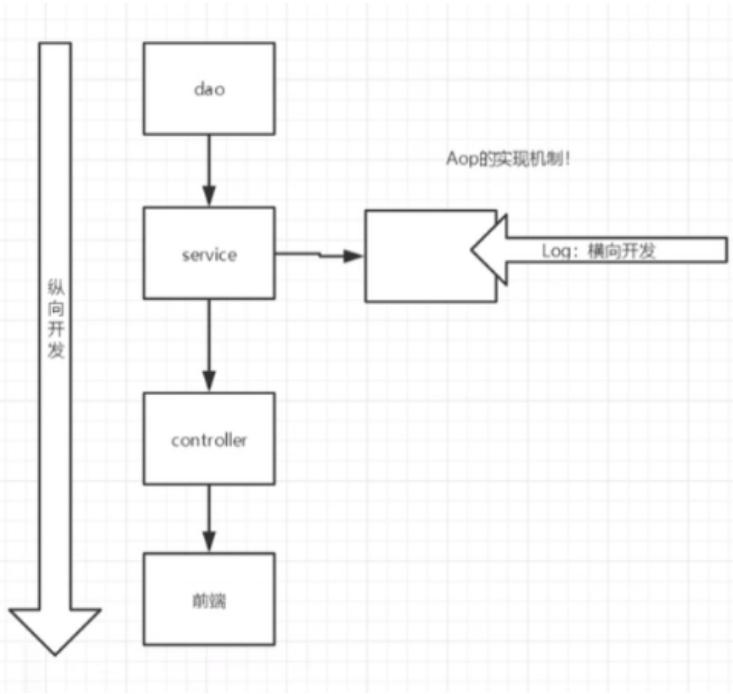
package com.xiang.demo02;

public class Client {
    public static void main(String[] args) {
        UserServiceImpl userService = new UserServiceImpl();

        UserServiceProxy proxy = new UserServiceProxy();
        proxy.setUserService(userService);
        proxy.delete();
    }
}

```

所谓AOP



### 10.3、动态代理

动态代理和静态代理角色一样

动态代理的代理类是动态生成的，不是我们直接写好的

动态代理分为两大类：基于接口的动态代理 基于类的动态代理

    基于接口的动态代理 如JDK动态代理

    基于类的动态代理 如cglib, cglib (Code Generation Library) 是一个高性能开源的代码生成包，它采用非常底层的字节码技术，对指定的目标类生成一个子类，并对子类进行增强。

    Java字节码实现：javassist

需要了解两个类：Proxy（代理）， InvocationHandler（调用处理器）：InvocationHandler是由代理实例的调用处理器实现的接口，每个代理实例都有一个关联的调用处理器，**当在代理实例上调用方法时，方法调用将被编码并分派到其调用处理器(InvocationHandler)的invoke方法**

## Spring AOP和动态代理

### 1. Spring AOP简介

#### 1.1 什么是AOP

AOP的全称是Aspect-Oriented Programming，即面向切面编程（也称面向方面编程）。它是面向对象编程（OOP）的一种补充，目前已成为一种比较成熟的编程方式。

在传统的业务处理代码中，通常都会进行事务处理、日志记录等操作。虽然使用OOP可以通过组合或者继承的方式来达到代码的重用，但如果要实现某个功能（如日志记录），同样的代码仍然会分散到各个方法中。这样，如果想要关闭某个功能，或者对其进行修改，就必须要修改所有的相关方法。这不但增加了开发人员的工作量，而且提高了代码的出错率。

为了解决这一问题，AOP思想随之产生。AOP采取横向抽取机制，将分散在各个方法中的重复代码提取出来，然后在程序编译或运行时，再将这些提取出来的代码应用到需要执行的地方。这种采用横向抽取机

制的方式，采用传统的OOP思想显然是无法办到的，因为OOP只能实现父子关系的纵向的重用。虽然AOP是一种新的编程思想，但却不是OOP的替代品，它只是OOP的延伸和补充。

AOP的使用，使开发人员在编写业务逻辑时可以专心于核心业务，而不用过多的关注于其他业务逻辑的实现，这不但提高了开发效率，而且增强了代码的可维护性。

## 1.2 AOP术语

**Aspect（切面）：**封装的用于横向插入系统功能（如事务、日志等）的类。该类要被Spring容器识别为切面，需要在配置文件中通过元素指定。

**Joinpoint（连接点）：**在程序执行过程中的某个阶段点，它实际上是一个对象的操作，例如方法的调用或者异常的抛出。在Spring AOP中，连接点就是指方法的调用。

**Pointcut（切入点）：**切面与程序流程的交叉点，即那些需要处理的连接点。通常在程序中，切入点指的是类或者方法名，如某个通知要应用到所有以add开头的方法中，那么所有满足这一规则的方法都是切入点。

**Advice(通知/增强处理)**：AOP框架在特定的切入点执行的增强处理，即在定义好的切入点处所要执行的程序代码。可以将其理解为切面类中的方法，它是切面的具体实现。

**Target Object(目标对象)**：指所有被通知的对象，也称为被增强对象。如果AOP框架采用的是动态的AOP实现，那么该对象就是一个被代理对象。

**Proxy（代理）**：将通知应用到目标对象之后，被动态创建的对象

**Weaving（织入）**：将切面代码插入到目标对象上，从而生成代理对象的过程。

## 2. 动态代理

AOP中的代理说是由AOP框架动态生成的一个对象，该对象可以作为目标对象使用。Spring 中的AOP有两种，JDK动态代理，CGLIB代理

### 2.1 JDK动态代理

JDK动态代理是通过java.lang.reflect.Proxy类来实现的，我们可以调用Proxy类的newProxyInstance()方法来创建代理对象。对于使用业务接口的类，Spring默认会使用JDK动态代理来实现AOP。

案例演示：

1. 创建cn.ssm.jdk包，在该包下创建接口UserDao，并在该接口中编写添加和删除的方法。

```
public interface UserDao { public void addUser(); public void deleteUser(); }
```

2. 创建UserDao接口的实现类UserDaoImpl，分别实现接口中的方法

```
//目标类 public class UserDaoImpl implements UserDao { @Override public void addUser() { System.out.println("添加用户"); } @Override public void deleteUser() { System.out.println("删除用户"); } }
```

3. 创建com.ssm.aspect包，在该包下创建切面类MyAspect，在该类中定义一个模拟权限检查的方法和一个模拟记录日志的方法，这两个方法就表示切面中的通知。

```
//切面类：可能存在多个通知Advice(即增强的方法)    public class MyAspect {  
    public void check_Permissions() {           System.out.println("模拟检查权限....");  
    }      public void log() {           System.out.println("模拟记录日志.....");  
    } }
```

4. 在cn.ssm.jdk包下创建代理类JdkProxy，该类需要实现InvocationHandler接口，并编写代理方法。在代理方法中，需要通过Proxy类实现动态代理。

```
/** * 代理类 */ public class JdkProxy implements InvocationHandler{      //  
声明目标类接口     private UserDao userDao;          //创建代理方法     public Object  
createProxy(UserDao userDao) {           this.userDao=userDao;          //1.类加载器  
ClassLoader classLoader=JdkProxy.class.getClassLoader();          //2.被代理对象  
实现的所有接口     Class[] clazz=userDao.getClass().getInterfaces();  
//3.使用代理类进行增强，返回的是代理后的对象     return  
Proxy.newProxyInstance(classLoader, clazz, this);      }      /*     所有动态  
代理类的方法调用，都会交由invoke()方法处理     proxy 被代理后的对象     method 将  
要被执行的方法信息（反射）     args 执行方法时需要的参数     */     @Override  
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
//声明切面     MyAspect myAspect=new MyAspect();          //前增强  
myAspect.check_Permissions();          //在目标类上调用方法，并传入参数  
Object obj=method.invoke(userDao, args);          //后增强     myAspect.log();  
return obj;      } }
```

JdkProxy类实现了InvocationHandler接口，并实现了接口中的invoke()方法，所有动态代理类所调用的方法都会交由该方法处理。在创建的代理方法createProxy()中，使用了Proxy类的newProxyInstance()方法来创建代理对象。newProxyInstance()方法中包含3个参数，其中第1个参数是当前类的类加载器，第2个参数表示的是被代理对象实现的所有接口，第3个参数this代表的就是代理类JdkProxy本身。在invoke()方法中，目标类方法执行的前后，会分别执行切面类中的check\_Permissions()方法和log()方法。

5. 在cn.ssm.jdk包中，创建测试类JdkTest。在该类中的main()方法中创建代理对象和目标对象，然后从代理对象中获得对目标对象userDao增强后的对象，最后调用该对象中的添加和删除方法。

```
public class JdkTest {      public static void main(String[] args) {          //创  
建代理对象     JdkProxy jdkProxy=new JdkProxy();          //创建目标对象  
UserDao userDao=new UserDaoImpl();          //从代理对象中获取增强后的目标对象  
UserDao userDao1=(UserDao) jdkProxy.createProxy(userDao);          //执行方法  
userDao1.addUser();          userDao1.deleteUser();      }      }      执行结果：  
模拟检查权限... 添加用户 模拟记录日志... 模拟检查权限... 删除用户 模拟记录日志...
```

## 2.2 CGLIB代理

通过前面的学习可知，JDK的动态代理用起来非常简单，但它是有局限性的，使用动态代理的对象必须实现一个或多个接口（如UserDaoImpl实现了UserDao接口）。如果想代理没有实现接口的类，那么可以使用CGLIB代理。

CGLIB (Code Generation Library) 是一个高性能开源的代码生成包，它采用非常底层的字节码技术，对指定的目标类生成一个子类，并对子类进行增强。

在Spring的核心包中已经集成了CGLIB所需的包，所以开发中不需要导包。

1. 创建cn.ssm.cglib包，在包中创建一个目标类UserDao， UserDao 需要实现任何接口，只需定义一个添加用户的方法和一个删除用户的方法。

```
public class UserDao {     public void addUser() {  
    System.out.println("添加用户"); }     public void deleteUser() {  
    System.out.println("删除用户"); } }
```

2. 创建代理类CglibProxy，该代理类需要实现MethodInterceptor接口，并实现接口中的intercept()方法。

```
//代理类  public class CglibProxy implements MethodInterceptor {      //代理方法  
public Object createProxy(Object target) {          //创建一个动态类对象  
Enhancer enhancer=new Enhancer();          //确定要增强的类，设置其父类  
enhancer.setSuperclass(target.getClass());          //添加回调函数  
enhancer.setCallback(this);          //返回创建的代理类          return  
enhancer.create(); }      /** * proxy Cglib根据父类生成的代理对象  
* method 拦截的方法 * args 拦截方法的参数组 * methodProxy 方法的代理对  
象，用于执行父类的方法 */ @Override public Object  
intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy)  
throws Throwable {          //创建切面类对象 MyAspect myAspect=new  
MyAspect();          //前增强 myAspect.check_Permissions();  
//目标方法执行 Object obj=methodProxy.invokeSuper(proxy, args);          //后  
增强 myAspect.log();          return obj; }
```

首先创建了一个动态类对象Enhancer，它是CGLIB的核心类；然后调用了Enhancer类的setSuperclass()方法来确定目标对象；接下来调用了setCallback()方法添加回调函数，其中的this代表的就是代理类CglibProxy本身。最后通过return语句将创建的代理类对象返回。intercept()方法会在程序执行目标方法时被调用，方法运行时将会执行切面类中的增强方法。

3. 创建测试类 CglibTest.java

```
public class CglibTest {     public static void main(String[] args) {          //创  
建代理类对象 CglibProxy cglibProxy=new CglibProxy();          //创建目标  
对象 UserDao userDao=new UserDao();          //获取增强后的目标对象  
UserDao userDao=(UserDao) cglibProxy.createProxy(userDao);          //执行方法  
userDao.addUser();     userDao.deleteUser(); } }
```

目标类UserDao中的方法被成功调用并增强了。这种没有实现接口的代理方式就是CGLIB代理。原文链接：<https://blog.csdn.net/lcachang/article/details/86595470>

真实角色：

```
package com.xiang.demo04;  
  
//真实对象  
public class UserServiceImpl implements UserService{  
    @Override  
    public void add() {
```

```

        System.out.println("增加了一位用户");
    }

@Override
public void delete() {
    System.out.println("删除了一位用户");
}

@Override
public void update() {
    System.out.println("修改了一位用户");
}

@Override
public void query() {
    System.out.println("查询了一位用户");
}
//新增功能去改源代码是开发中的大忌
}

```

## 调用处理程序

```

package com.xiang.demo04;

import com.xiang.demo03.Rent;

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

//调用处理程序，等下我们会用这个类，自动生成代理类
public class ProxyInvocationHandler implements InvocationHandler {
    //被代理的接口
    private Object target;

    public void setTarget(Object target) {
        this.target = target;
    }
    //Proxy提供了创建动态代理类和实例的静态方法，它也是由这些方法创建的所有动态代理
    //类的超类
    //得到代理类（本类加载，代理接口类加载，ProxyInvocationHandler对象）
    public Object getProxy() {
        return Proxy.newProxyInstance(this.getClass().getClassLoader(),
target.getClass().getInterfaces(), this);
    }
    /*
    InvocationHandler（接口）是由代理实例的调用处理程序实现的接口
    每个代理实例都有一个关联的调用处理程序，当在代理实例上调用方法时，
    方法（真实角色的方法）调用将被编码并分派到其 调用处理程序 的invoke方法。
    */
    @Override //处理代理实例，并返回结果
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        log(method.getName());
        //动态代理的本质，就是使用反射机制实现。
        Object result = method.invoke(target, args); //invoke--执行什么方法
        return result;
    }

    public void log(String msg) {
        System.out.printf("执行" + msg + "方法");
    }
}

```

## 客户测试：

```
package com.xiang.demo04;
```

```

import com.xiang.demo02.UserService;
import com.xiang.demo02.UserServiceImpl;

public class Client {
    public static void main(String[] args) {
        //真实角色
        UserServiceImpl userService = new UserServiceImpl();
        //代理角色, 不存在
        ProxyInvocationHandler pih = new ProxyInvocationHandler();

        pih.setTarget(userService); //设置需要代理的对象
        //生成动态代理类
        UserService proxy = (UserService) pih.getProxy();
        proxy.query();
        proxy.add();
        proxy.delete();
        proxy.update();
    }
}

```

动态代理的好处：

可以使真实角色的操作更加纯粹，不用去关注一些公共的业务

公共业务交给代理角色，实现了业务的分工

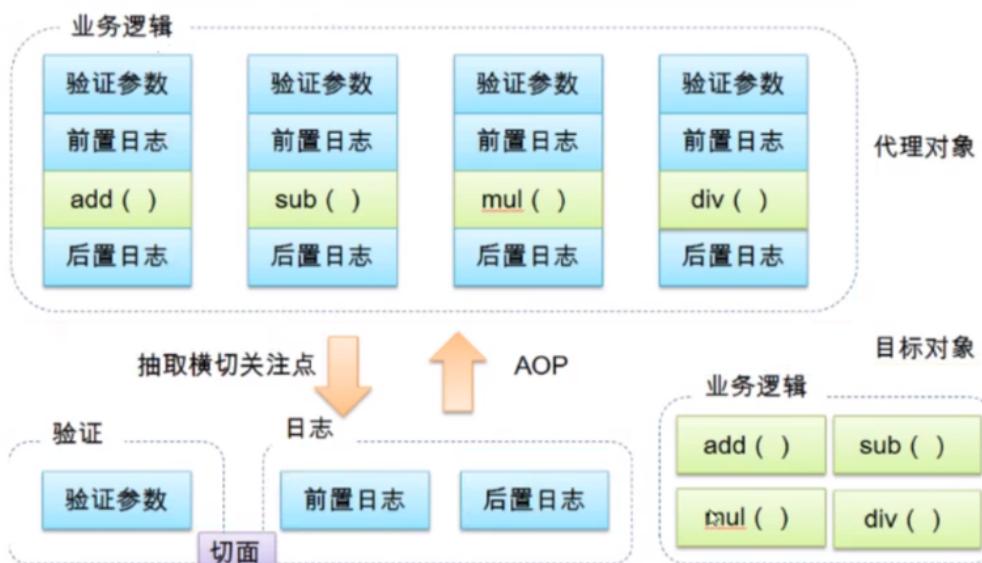
公共业务发生扩展的时候，方便集中管理

一个动态代理类处理的是一个接口，一般就是对应的一类业务

一个动态代理类可以处理多个类，只要是实现了同一个接口即可。

JDK的动态代理用起来非常简单，但它是有局限性的，使用动态代理的对象必须实现一个或多个接口（如UserDaoImpl实现了UserDao接口）。如果想代理没有实现接口的类，那么可以使用CGLIB代理

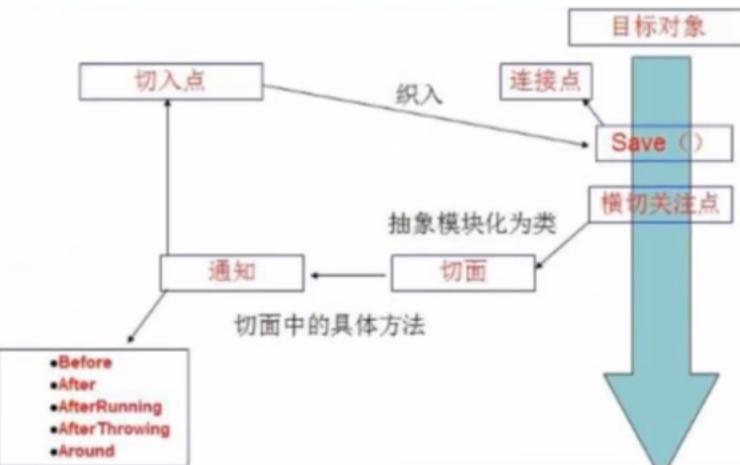
AOP (Aspect Oriented Programming) 意为：面向切面编程，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。AOP是OOP的延续，是软件开发中的一个热点，也是Spring框架中的一个重要内容，是函数式编程的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。



## 11.2 Aop在Spring中的作用

提供声明式事务；允许用户自定义切面

- 横切关注点：跨越应用程序多个模块的方法或功能。即是，与我们业务逻辑无关的，但是我们需要关注的部分，就是横切关注点。如日志，安全，缓存，事务等等 ....
- 切面 (ASPECT)：横切关注点被模块化的特殊对象。即，它是一个类。
- 通知 (Advice)：切面必须要完成的工作。即，它是类中的一个方法。
- 目标 (Target)：被通知对象。
- 代理 (Proxy)：向目标对象应用通知之后创建的对象。
- 切入点 (PointCut)：切面通知执行的“地点”的定义。
- 连接点 (JointPoint)：与切入点匹配的执行点。



想象有一栋写字楼，写字楼里有10间公司，每间公司的业务各不相同，但可能都存在安保服务、后勤服务的需求。这些需求都不是业务关注点，但是公司业务的开展需要这些服务的支撑。如果你是公司的boss，你希望花很大精力在这些关注点上吗？还是只需要关注你的核心业务领域？写字楼提供了一种理想的组织模式，它把安保、后勤等服务从每间公司的需求清单上抽取出来，外包给其他组织，公司只需要在需要的时候，在它业务的某个点上使用这些服务就可以了。

代码中常常弥漫着这样一些关注点，这些关注点关心事务、安全等非业务特性，比如：一次数据库操作需要包裹在事务中；执行应用的某个操作之前需要检查是否具有权限。

对比写字楼模式，我们应该如何实现上述的关注点？（1）需要把这些关注点剥离出来，在单独的模块（oo中是class）中实现；（2）在需要这些关注点的位置，能自由的织入。

如何命名这些关注点？这些关注点是在你的业务主线上根据需要被织入的，所以“横切关注点”是一个很贴切的名称。

**横切关注点可以被模块化为特殊的类，这些类被称为切面(aspect)。这样做有两个优点：**

- 1) 每个关注点都集中于一个地方，而不是分散到多处代码中；
- 2) 服务模块更简洁，因为它们只包含主要的关注点的代码（核心业务逻辑），

而次要关注点的代码（日志，事务，安全等）都被转移到切面中。

**切面类有自己要完成的工作，切面类的工作就称为通知。通知定义了切面是做什么以及何时使用。**

"做什么"，即切面类中定义的方法是干什么的；

"何时使用"，即5种通知类型，是在目标方法执行前，还是目标方法执行后等等；

"何处做"，即通知定义了做什么，何时使用，但是不知道用在何处，而切点定义的就是告诉通知应该用在哪个类的那个目标方法上，从而完美的完成横切点功能

**Spring切面定义了5种类型通知：**

- 1) 前置通知(Before): 在目标方法被调用之前调用通知功能。
- 2) 后置通知(After): 在目标方法完成之后调用通知，不会关心方法的输出是什么。
- 3) 返回通知(After-returning): 在目标方法成功执行之后调用通知。
- 4) 异常通知(After-throwing): 在目标方法抛出异常后调用通知。
- 5) 环绕通知(Around): 通知包裹了被通知的方法，在被通知的方法调用之前和之后执行自定义的行为。

## 切面(Aspect)

切面是通知和切点的结合，通知和切点共同定义了切面的全部内容。因为通知定义的是切面的"要做什么"和"在何时做"，而切点定义的是切面的"在何地做"。将两者结合在一起，就可以完美的展现切面在何时，何地，做什么(功能)。

## 连接点(Join point)

即被通知的类中的方法都可能成为切点，所以这些都是连接点，定义成切点之后，这个连接点就变成了切点，通知的类可能是一个类，也有可能是一个包底下的所有类，所以连接点可以成千上万来记，是一个虚概念，可以把连接点看成是切点的集合。

## 切点(Poincut)

在被通知的类上，连接点谈的是一个飘渺的大范围，而切点是一个具体的位置，用于缩小切面所通知的连接点的范围。

前面说过，通知定义的是切面的"要做什么"和"在何时做"，是不是没有去哪里做，而切点就定义了"去何处做"。切点的定义会匹配通知所要织入的一个或多个连接点。我们通常使用明确的类和方法名称，或者是使用正则表达式定义所匹配的类和方法名称来指定切点。说白了，切点就是让通知找到"作用的地方"。

## 引入(Introduction)

引入这个概念就比较高大尚，引入允许我们向现有的类添加新方法或属性。

主要目的是想在无需修改A的情况下，引入B的行为和状态。

## 织入(Weaving)

织入是把切面应用到目标对象并创建新的代理对象的过程。切面在指定的连接点被织入到目标对象中。

在目标对象的生命周期里有多个点可以进行织入：

### 编译期:

切面在目标类编译时被织入。需要特殊的编译器，是AspectJ的方式，不是spring的菜。

### 类加载期:

切面在目标类加载到JVM时被织入。这种方式需要特殊的类加载器，它可以在目标类被引入应用之前增强该目标类的字节码。AspectJ5支持这种方式。

### 运行期:

切面在应用运行的某个时刻被织入。一般情况下，在织入切面时，AOP容器会为目标对象动态的创建一个代理对象。而这正是Spring AOP的织入切面的方式。

# AOP实战（注解版）

## 1、创建切面类

切面类包含通知和切入点，在创建切面类之前，我们需要了解下AspectJ的切点表达式，因为我们需要通过切点表达式定义切点，用于准确的定位应该在什么地方应用切面的通知。

直接上一个图，解释下切点表达式的元素：

Spring借助AspectJ的切点表达式语言来定义Spring的切面	
AspectJ指示器	描述
arg()	限制连接点匹配参数为指定类型的执行方法
@args()	限制连接点匹配参数有指定注解标注的执行方法
execution()	用于匹配连接点的执行方法
this()	限制连接点匹配AOP代理的bean引用为指定类型的类
target()	限制连接点匹配目标对象为指定类型的类
@target()	限制连接点匹配特定的执行对象，这些对象对应的类要具有指定类型的注解
within()	限制连接点匹配指定的类型
@within()	限制连接点匹配指定主机机的类型
@annotation()	限制匹配带有指定注解的连接点

咱们解释一下切点表达式的含义：



通过execution指示器，选择ConferenceServiceImpl类中的conference()方法。

方法表达式以“\*”号开始，表明了我们不关心方法的返回值是神马鬼。

对于方法参数列表通过两个点表示，表示我们不在乎conference的参数。

在执行表达式的时候，我们可以通过逻辑运算符&&(and), ||(or), !(not)对表达式进行搭配。比如：

```
execution(* com.lanhui.spring.ConferenceServiceImpl.conference(..)
           && within(com.lanhui.spring.*))
```

增加了一个限制就是我们只管com.lanhui.spring下的包，这里的&&可以使用and来替代，

同理||, !都是一样的用法，灵活多变，只能根据实际情况看着办。

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

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

即 Aop 在 不改变原有代码的情况下，去增加新的功能 .

## 11.3 使用Spring实现Aop

【重点】使用AOP织入，需要导入一个依赖包！

```
1 <!-- https://mvnrepository.com/artifact/org.aspectj/aspectjweaver -->
2 <dependency>
3   <groupId>org.aspectj</groupId>
4   <artifactId>aspectjweaver</artifactId>
5   <version>1.9.4</version>
6 </dependency>
```

方式一：使用Spring的API接口实现Aop

```
package com.xiang.service;

public interface UserService {
    public int add();
    public void delete();
    public void update();
    public void select();
}

package com.xiang.service;

public class UserServiceImpl implements UserService{
    @Override
    public int add() {
        System.out.println("增加了一个用户");
        return 11;
    }

    @Override
    public void delete() {
        System.out.println("删除了一个用户！");
    }

    @Override
    public void update() {
        System.out.println("修改了一个用户！");
    }

    @Override
```

```

public void select() {
    System.out.println("查询了一个用户！");
}

package com.xiang.log;

import org.springframework.aop.MethodBeforeAdvice;

import java.lang.reflect.Method;

public class Log implements MethodBeforeAdvice {
    @Override//(要执行的目标对象的方法, 参数args, 目标对象:target)
    public void before(Method method, Object[] args, Object target) throws Throwable
    {
        System.out.println(target.getClass().getName()+"的"+method.getName()+"方法被执行了。");
    }
}

package com.xiang.log;

import org.springframework.aop.AfterAdvice;
import org.springframework.aop.AfterReturningAdvice;

import java.lang.reflect.Method;

public class AfterLog implements AfterReturningAdvice {
    @Override//执行method之后得到返回值, returnValue:返回值。
    public void afterReturning(Object returnValue, Method method, Object[] objects,
Object o1) throws Throwable {
        System.out.println("执行了"+method.getName()+"方法之后, 返回结果
为: "+returnValue);
    }
}

```

```

package com.xiang.log;

import org.springframework.aop.AfterAdvice;
import org.springframework.aop.AfterReturningAdvice;

import java.lang.reflect.Method;

public class AfterLog implements AfterReturningAdvice {
    @Override//执行method之后得到返回值, returnValue:返回值。
    public void afterReturning(Object returnValue, Method method, Object[] objects,
Object o1) throws Throwable {
        System.out.println("执行了"+method.getName()+"方法之后, 返回结果
为: "+returnValue);
    }
}

```

```

import com.xiang.service.UserService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MyTest {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
        //动态代理, 代理的是"接口", 而不是具体的类
        UserService userService = (UserService) context.getBean("userService");
        userService.select();
    }
}

```

## 方式二：自定义实现AOP（主要是切面定义）

```
package com.xiang.diy;
//自定义pointcut切入点
public class DiyPointCut {

    public void before() {
        System.out.println("=====方法执行前=====");
    }

    public void after() {
        System.out.println("=====方法执行后=====");
    }
}
```

配置切面：

```
<?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">

    <!--注册bean-->
    <bean id="userService" class="com.xiang.service.UserServiceImpl"/>
    <bean id="log" class="com.xiang.log.Log"/>
    <bean id="afterLog" class="com.xiang.log.AfterLog"/>

    <!--方式一：使用原生的Spring API接口-->
    <!--配置aop：需要导入aop的约束-->
    <!--<aop:config>-->
    <!--      &lt;!&ndash;切入点：expression:表达式，execution(public修饰词 返回值 类名 方法名 参数)-定位到要执行的位置&ndash;&gt;-->
    <!--      <aop:pointcut id="pointcut" expression="execution(*
com.xiang.service.UserServiceImpl.*(..))"/>-->
    <!--      &lt;!&ndash;执行环绕增强&ndash;&gt;-->
    <!--      <aop:advisor advice-ref="log" pointcut-ref="pointcut"/>-->
    <!--      <aop:advisor advice-ref="afterLog" pointcut-ref="pointcut"/>-->
    </aop:config>-->

    <!--自定义类-->
    <bean id="diy" class="com.xiang.diy.DiyPointCut"/>
    <aop:config>
        <!--自定义切面，ref:要引用的类-->
        <aop:aspect ref="diy">
            <!--定义切入点-->
            <aop:pointcut id="point" expression="execution(*
com.xiang.service.UserServiceImpl.*(..))"/>
            <!--通知环绕-->
            <aop:before method="before" pointcut-ref="point"/>
            <aop:after method="after" pointcut-ref="point"/>
        </aop:aspect>
    </aop:config>
</beans>
```

测试：

```
import com.xiang.service.UserService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

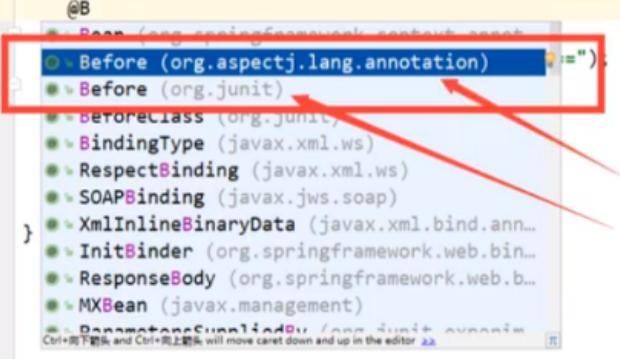
public class MyTest {
    public static void main(String[] args) {
```

```

    ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
    //动态代理，代理的是“接口”，而不是具体的类
    UserService userService = (UserService) context.getBean("userService");
    userService.add();
}
}   输出： =====方法执行前===== 增加了一个用户 =====方法执行后
=====

```

### 方式三：使用注解实现aop



```

3 //方式三：使用注解方式实现AOP
4
5 import org.aspectj.lang.annotation.Aspect;
6
7 @Aspect //标注这个类是一个切面
8 public class AnnotationPointCut {
9
10     @B
11     @Before("org.springframework.context.annotation:=")
12     @Before("org.junit")
13     @BeforeClass("org.junit")
14     @BindingType(javax.xml.ws)
15     @RespectBinding(javax.xml.ws)
16     @SOAPBinding(javax.jws.soap)
17     @XmlElementBinaryData(javax.xml.bind.annotation)
18     @InitBinder(org.springframework.web.bind)
19     @ResponseBody(org.springframework.web.bind.annotation)
20     @MXBean(javax.management)
21 }

```

```

package com.xiang.diy;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.Signature;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

//使用注解方式实现AOP
@Aspect //标注这个类是一个切面
public class AnnotationAspect {
    @Before("execution(* com.xiang.service.UserServiceImpl.*(..))")
    public void before() {
        System.out.println("++++++方法执行前++++++");
    }
    @After("execution(* com.xiang.service.UserServiceImpl.*(..))")
    public void after() {
        System.out.println("++++++方法执行后+++++");
    }
}

//在环绕增强中，我们可以给定一个参数，代表我们要获取切入的点（与切入点匹配的连接点）
@Around("execution(* com.xiang.service.UserServiceImpl.*(..))")
public void around(ProceedingJoinPoint joinPoint) throws Throwable {
    System.out.println("----环绕前----");
    Signature signature = joinPoint.getSignature(); //获得签名
    System.out.println("signature: " + signature);
    //执行方法
    Object proceed = joinPoint.proceed();
    System.out.println("----环绕后----");
    System.out.println(proceed);
}

```

```
}
```

咱们解释一下切点表达式的含义:



通过execution指示器，选择ConferenceServiceImpl类中的conference()方法。

方法表达式以“\*”号开始，表明了我们不关心方法的返回值是神马鬼。

对于方法参数列表通过两个点表示，表示我们不在乎conference的参数。

在执行表达式的时候，我们可以通过逻辑运算符&&(and), ||(or), !(not)对表达式进行搭配。比如：

```
execution(* com.lanhuiгу.spring.ConferenceServiceImpl.conference(..)
          && within(com.lanhuiгу.spring.*))
```

增加了一个限制就是我们只管com.lanhuiгу.spring下的包，这里的&&可以使用and来替代，

同理||, !都是一样的用法，灵活多变，只能根据实际情况看着办。

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: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">

    <!--注册bean-->
    <bean id="userService" class="com.xiang.service.UserServiceImpl"/> <bean
        id="annotationAspect" class="com.xiang.diy.AnnotationAspect"/>
    <!--开启注解支持 基于接口JDK(默认proxy-target-class="false")和基于类cglib(proxy-
        target-class="true")两种代理模式-->
    <aop:aspectj-autoproxy proxy-target-class="true"/> </beans>
```

测试：

```
import com.xiang.service.UserService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MyTest {
    public static void main(String[] args) {
        ApplicationContext context = new
        ClassPathXmlApplicationContext("applicationContext.xml");
        //动态代理，代理的是"接口"，而不是具体的类
        UserService userService = (UserService) context.getBean("userService");
        userService.delete();
    }
}
```

## 12、整合Mybatis

步骤：

1. 导入jar包

Junit

Mybatis

Mysql数据库 导入的mysql包要和电脑中的mysql版本一致。

Spring相关的

Aop织入aspectj

Mybatis-spring

2. 编写配置文件

3. 测试

### 12.1 回忆mybatis

- A. 编写实体类
- B. 编写核心配置文件
- C. 编写接口
- D. 编写Mapper.xml
- E. 测试

### 12.2 Mybatis-Spring

#### (1) 编写数据源配置

```
<!--DataSource: 使用Spring的数据源替换Mybatis的配置 c3p0 dbcp druid 这里使用Spring提供的JDBC-->
<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/mybatis?useSSL=true&useUnicode=false&characterEncoding=UTF8&autoReconnect=true&failOverReadOnly=false"/>
    <property name="username" value="root"/>
    <property name="password" value="123456"/>
</bean>
```

## (2) SqlSessionFactory

```
<!--SqlSessionFactory-->
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <!--绑定Mybatis配置文件-->
    <property name="configLocation" value="classpath:mybatis-config.xml"/>
    <property name="mapperLocations" value="classpath:com/xiang/mapper/*.xml"/><!--
这里是UserMapper.xml-->
</bean>
```

## 3. sqlSessionTemple

```
<!--SqlSessionTemplate: 就是我们使用的sqlSession-->
<bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">
    <!--只能使用构造器注入sqlSessionFactory, 因为它没有set方法-->
    <constructor-arg index="0" ref="sqlSessionFactory"/>
</bean>
```

## 4. 需要给接口加实现类

```
package com.xiang.mapper;

import com.xiang.pojo.User;
import org.mybatis.spring.SqlSessionTemplate;

import java.util.List;

public class UserMapperImpl implements UserMapper{
    //以前, 和数据库相关的操作都使用sqlSession来执行, 现在都使用SqlSessionTemplate来
    //执行
    private SqlSessionTemplate sqlSession;

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

    @Override
    public List<User> selectUser() {
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        return mapper.selectUser();
    }
}
```

## 5. 将自己写的实现类, 注入到Spring中

```
<bean id="userMapper" class="com.xiang.mapper.UserMapperImpl">
    <property name="sqlSession" ref="sqlSession"/>
</bean>
```

## 6. 测试使用即可

```
@Test
public void test() {

    ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml");

    UserMapper userMapper = context.getBean("userMapper", UserMapper.class);
    for (User user : userMapper.selectUser()) {
        System.out.println(user);
    }
}
```

```
}
```

## SqlSessionTemplate

`SqlSessionTemplate` 是 MyBatis-Spring 的核心。作为 `SqlSession` 的一个实现，这意味着可以使用它无缝代替你代码中已经在使用的 `SqlSession`。`SqlSessionTemplate` 是线程安全的，可以被多个 DAO 或映射器所共享使用。

当调用 SQL 方法时（包括由 `getMapper()` 方法返回的映射器中的方法），`SqlSessionTemplate` 将会保证使用的 `SqlSession` 与当前 Spring 的事务相关。此外，它管理 session 的生命周期，包含必要的关闭、提交或回滚操作。另外，它也负责将 MyBatis 的异常翻译成 Spring 中的 `DataAccessExceptions`。

由于模板可以参与到 Spring 的事务管理中，并且由于其是线程安全的，可以供多个映射器类使用，你应该总是用 `SqlSessionTemplate` 来替换 MyBatis 默认的 `DefaultSqlSession` 实现。在同一应用程序中的不同类之间混杂使用可能会引起数据一致性的问题。

可以使用 `SqlSessionFactory` 作为构造方法的参数来创建 `SqlSessionTemplate` 对象。

```
<bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">
  <constructor-arg index="0" ref="sqlSessionFactory" />
</bean>
```

```
@Bean
public SqlSessionTemplate sqlSession() throws Exception {
    return new SqlSessionTemplate(sqlSessionFactory());
}
```

现在，这个 `bean` 就可以直接注入到你的 DAO bean 中了。你需要在你的 `bean` 中添加一个 `SqlSession` 属性，就像下面这样：

```
public class UserDaoImpl implements UserDao {
    private SqlSession sqlSession;

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

    public User getUser(String userId) {
        return sqlSession.selectOne("org.mybatis.spring.sample.mapper.UserMapper.getUser", userId);
    }
}
```

## SqlSessionDaoSupport

`SqlSessionDaoSupport` 是一个抽象的支持类，用来为你提供 `SqlSession`。调用 `getSqlSession()` 方法你会得到一个 `SqlSessionTemplate`，之后可以用于执行 SQL 方法，就像下面这样：

```
public class UserDaoImpl extends SqlSessionDaoSupport implements UserDao {
    public User getUser(String userId) {
        return getSqlSession().selectOne("org.mybatis.spring.sample.mapper.UserMapper.getUser", userId);
    }
}
```

在这个类里面，通常更倾向于使用 `MapperFactoryBean`，因为它不需要额外的代码。但是，如果你需要在 DAO 中做其它非 MyBatis 的工作或需要一个非抽象的实现类，那么这个类就很有用了。

`SqlSessionDaoSupport` 需要通过属性设置一个 `sqlSessionFactory` 或 `SqlSessionTemplate`。如果两个属性都被设置了，那么 `SqlSessionFactory` 将被忽略。

假设类 `UserMapperImpl` 是 `SqlSessionDaoSupport` 的子类，可以编写如下的 Spring 配置来执行设置：

```
<bean id="userDao" class="org.mybatis.spring.sample.dao.UserDaoImpl">
  <property name="sqlSessionFactory" ref="sqlSessionFactory" />
</bean>
```

## 实现类

```
package com.xiang.mapper;

import com.xiang.pojo.User;
import org.apache.ibatis.session.SqlSession;
import org.mybatis.spring.support.SqlSessionDaoSupport;

import java.util.List;

public class UserMapperImpl2 extends SqlSessionDaoSupport implements UserMapper {
    @Override
    public List<User> selectUser() {
        //     SqlSession sqlSession = getSqlSession();
        //     UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        //     return mapper.selectUser();
        return getSqlSession().getMapper(UserMapper.class).selectUser();
    }
}
```

## 注入spring

```
<!--方式二-->
<bean id="userMapper2" class="com.xiang.mapper.UserMapperImpl2">
```

```
<!--这个属性由父类SqlSessionDaoSupport接收-->
<property name="sqlSessionFactory" ref="sqlSessionFactory"/>
</bean>
```

测试

```
@Test
public void test2() {
    ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");

    UserMapper userMapper = context.getBean("userMapper2", UserMapper.class);
    for (User user : userMapper.selectUser()) {
        System.out.println(user);
    }
}
```

## 13、声明式事务

### (1) 回顾事务

把一组业务当成一个业务来做，要么都成功，要么都失败

事务在项目开发中，十分的重要，涉及到事务的一致性

确保完整性和一致性

事务的ACID原则：

原子性(Atomicity)：原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，因此事务的操作如果成功就必须完全应用到数据库，如果操作失败则不能对数据库有任何影响。

一致性(Consistency)：一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。举例来说，假设用户A和用户B两者的钱加起来一共是1000，那么不管A和B之间如何转账、转几次账，事务结束后两个用户的钱相加起来应该还得是1000，这就是事务的一致性。

隔离性(Isolation)：隔离性是当多个用户并发访问数据库时，比如同时操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。关于事务的隔离性数据库提供了多种隔离级别

持久性(Durability)：持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。例如我们在使用JDBC操作数据库时，在提交事务方法后，提示用户事务操作完成，当我们程序执行完成直到看到提示后，就可以认定事务已经正确提交，即使这时候数据库出现了问题，也必须要将我们的事务完全执行完成。否则的话就会造成我们虽然看到提示事务处理完毕，但是数据库因为故障而没有执行事务的重大错误。这是不允许的。

## 事务

一个使用 MyBatis-Spring 的其中一个主要原因是它允许 MyBatis 参与到 Spring 的事务管理中。而不是给 MyBatis 创建一个新的专用事务管理器，MyBatis-Spring 借助了 Spring 中的 `DataSourceTransactionManager` 来实现事务管理。

一旦配置好了 Spring 的事务管理器，你就可以在 Spring 中按你平时的方式来配置事务。并且支持 `@Transactional` 注解和 AOP 风格的配置。在事务处理期间，一个单独的 `SqlSession` 对象将会被创建和使用。当事务完成时，这个 `session` 会以合适的方式提交或回滚。

事务配置好了以后，MyBatis-Spring 将会透明地管理事务。这样在你的 DAO 类中就不需要额外的代码了。

## 标准配置

要开启 Spring 的事务处理功能，在 Spring 的配置文件中创建一个 `DataSourceTransactionManager` 对象：

```
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <constructor-arg ref="dataSource" />
</bean>
```

```
@Bean
public DataSourceTransactionManager transactionManager() {
  return new DataSourceTransactionManager(dataSource());
}
```

传入的 `DataSource` 可以是任何能够与 Spring 兼容的 JDBC `DataSource`，包括连接池和通过 JNDI 查找获得的 `DataSource`。

注意：为事务管理器指定的 `DataSource` 必须和用来创建 `SqlSessionFactoryBean` 的是同一个数据源，否则事务管理器就无法工作了。

## 交由容器管理事务

如果你正使用一个 JEE 容器而且想让 Spring 参与到容器管理事务（Container managed transactions, CMT）的过程中，那么 Spring 应该被设置为使用 `JtaTransactionManager` 或由容器指定的一个子类作为事务管理器。最简单的方式是使用 Spring 的事务命名空间或使用 `JtaTransactionManagerFactoryBean`：

```
<tx:jta-transaction-manager />
```

```
@Bean
public JtaTransactionManager transactionManager() {
  return new JtaTransactionManagerFactoryBean().getObject();
}
```

在这个配置中，MyBatis 将会和其它由容器管理事务配置的 Spring 事务资源一样。Spring 会自动使用任何一个存在的容器事务管理器，并注入一个 `SqlSession`。如果没有正在进行的事务，而基于事务配置需要一个新的事务的时候，Spring 会开启一个新的由容器管理的事务。

注意，如果你想使用由容器管理的事务，而不想使用 Spring 的事务管理，你就不能配置任何的 Spring 事务管理器。并必须配置 `SqlSessionFactoryBean` 以使用基本的 MyBatis 的 `ManagedTransactionFactory`：

```
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="transactionFactory">
    <bean class="org.apache.ibatis.transaction.managed.ManagedTransactionFactory" />
  </property>
</bean>
```

```
@Bean
public SqlSessionFactory sqlSessionFactory() {
  SqlSessionFactoryBean factoryBean = new SqlSessionFactoryBean();
  factoryBean.setDataSource(dataSource());
  factoryBean.setTransactionFactory(new ManagedTransactionFactory());
  return factoryBean.getObject();
}
```

## 编程式事务管理

MyBatis 的 `SqlSession` 提供几个方法来在代码中处理事务。但是当使用 MyBatis-Spring 时，你的 bean 将会注入由 Spring 管理的 `SqlSession` 或映射器。也就是说，Spring 总是为你处理了事务。

你不能在 Spring 管理的 `SqlSession` 上调用 `SqlSession.commit()`, `SqlSession.rollback()` 或 `SqlSession.close()` 方法。如果这样做了，就会抛出 `UnsupportedOperationException` 异常。在使用注入的映射器时，这些方法也不会暴露出来。

无论 JDBC 连接是否设置为自动提交，调用 `SqlSession` 数据方法或在 Spring 事务之外调用任何在映射器中方法，事务都将会自动被提交。

如果你想编程式地控制事务，请参考 [the Spring reference document\(Data Access -Programmatic transaction management\)](#)。下面的代码展示了如何使用 `PlatformTransactionManager` 手工管理事务。

```
TransactionStatus txStatus =
  transactionManager.getTransaction(new DefaultTransactionDefinition());
try {
  userMapper.insertUser(user);
} catch (Exception e) {
  transactionManager.rollback(txStatus);
  throw e;
}
transactionManager.commit(txStatus);
```

在使用 `TransactionTemplate` 的时候，可以省略对 `commit` 和 `rollback` 方法的调用。

```
TransactionTemplate transactionTemplate = new TransactionTemplate(transactionManager);
transactionTemplate.execute(txStatus -> {
  userMapper.insertUser(user);
  return null;
});
```

注意：虽然这段代码使用的是一个映射器，但换成 `SqlSession` 也是可以工作的。

## Spring中的事务管理

1. 声明式事务管理(交由容器管理事务): 以AOP的方式
2. 编程式事务管理: 需要在代码中, 进行事务的管理

**一、**在声明式的事务处理中, 要配置一个切面, 其中就用到了propagation, 表示打算对这些方法怎么使用事务, 是用还是不用, 其中propagation有七种配置, REQUIRED、SUPPORTS、MANDATORY、REQUIRES\_NEW、NOT\_SUPPORTED、NEVER、NESTED。默认是REQUIRED。

**二、**Spring中七种Propagation类的事务属性详解:

**REQUIRED:** 支持当前事务, 如果当前没有事务, 就新建一个事务。这是最常见的选择。

**SUPPORTS:** 支持当前事务, 如果当前没有事务, 就以非事务方式执行。

**MANDATORY:** 支持当前事务, 如果当前没有事务, 就抛出异常。

**REQUIRES\_NEW:** 新建事务, 如果当前存在事务, 把当前事务挂起。

**NOT\_SUPPORTED:** 以非事务方式执行操作, 如果当前存在事务, 就把当前事务挂起。

**NEVER:** 以非事务方式执行, 如果当前存在事务, 则抛出异常。

**NESTED:** 支持当前事务, 如果当前事务存在, 则执行一个嵌套事务, 如果当前没有事务, 就新建一个事务。

UserMapper接口:

```
package com.xiang.mapper;

import com.xiang.pojo.User;

import java.util.List;

public interface UserMapper {
    public List<User> selectUser();

    //添加一个用户
    public int addUser(User user);

    //删除一个用户
    public int deleteUser(int id);
}
```

User实体类

```
package com.xiang.pojo;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {

    private int id;
    private String name;
    private String pwd;
```

```
}
```

### UserMapper.xml

```
<?xml version="1.0" encoding="UTF8" ?>
<!DOCTYPE mapper
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="com.xiang.mapper.UserMapper">
  <select id="selectUser" resultType="user">
    select *
    from mybatis.user;
  </select>

  <insert id="addUser" parameterType="user">
    insert into mybatis.user (id, name, pwd) values (#{}id, #{}name, #{}pwd);
  </insert>

  <delete id="deleteUser" parameterType="_int">
    delete from mybatis.user where id=#{id};
  </delete>
</mapper>
```

### Mybatis-config.xml

```
<?xml version="1.0" encoding="UTF8" ?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-config.dtd">

<!--configuration核心配置文件-->
<configuration>

  <typeAliases>
    <package name="com.xiang.pojo"/>
  </typeAliases>

</configuration>
```

### spring-dao.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"
       xmlns:tx="http://www.springframework.org/schema/tx"
       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
                           http://www.springframework.org/schema/tx
                           http://www.springframework.org/schema/tx/spring-tx.xsd">

  <!--DataSource: 使用Spring的数据源替换Mybatis的配置 c3p0 dbcp druid 这里使用
       Spring提供的JDBC-->
  <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/mybatis?
useSSL=true&useUnicode=false&characterEncoding=UTF8&autoReconnect=true&
mp;failOverReadOnly=false"/>
    <property name="username" value="root"/>
    <property name="password" value="123456"/>
  </bean>

  <!--SqlSessionFactory-->
  <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
```

```

<property name="dataSource" ref="dataSource"/>
<!--绑定Mybatis配置文件-->
<property name="configLocation" value="classpath:mybatis-config.xml"/>
<property name="mapperLocations" value="classpath:com/xiang/mapper/*.xml"/>
<!--这里是UserMapper.xml-->
</bean>

<!--SqlSessionTemplate: 就是我们使用的sqlSession-->
<bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">
    <!--只能使用构造器注入sqlSessionFactory, 因为它没有set方法-->
    <constructor-arg index="0" ref="sqlSessionFactory"/>
</bean>

<!--配置声明式事务-->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <constructor-arg ref="dataSource"/>
</bean>

<!--结合AOP实现事务的织入-->
<!--配置事务通知: -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <!--指定要配置事务的方法-->
    <tx:attributes>
        <!--配置事务的传播特性propagation = [常用REQUIRED, NESTED]-->
        <tx:method name="add" propagation="REQUIRED"/>
        <tx:method name="delete" propagation="REQUIRED"/>
        <tx:method name="update" propagation="REQUIRED"/>
        <tx:method name="query" read-only="true"/>
        <tx:method name="*" propagation="REQUIRED"/> <!--所有方法-->
    </tx:attributes>
</tx:advice>

<!--配置事务切入-->
<aop:config>
    <aop:pointcut id="txPointCut" expression="execution(* com.xiang.mapper.*.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointCut"/>
</aop:config>

</beans>

```

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">

    <import resource="spring-dao.xml"/>

    <!--bean-->
    <bean id="userMapper" class="com.xiang.mapper.UserMapperImpl">
        <property name="sqlSessionFactory" ref="sqlSessionFactory"/>
    </bean>

</beans>

```

UserMapperImpl.class

```

package com.xiang.mapper;

import com.xiang.pojo.User;
import org.mybatis.spring.SqlSessionTemplate;

```

```

import org.mybatis.spring.support.SqlSessionDaoSupport;

import java.util.List;

public class UserMapperImpl extends SqlSessionDaoSupport implements UserMapper{

    @Override
    public List<User> selectUser() {
        User user = new User(8, "小王", "888");

        UserMapper mapper = getSqlSession().getMapper(UserMapper.class);

        mapper.addUser(user);
        mapper.deleteUser(2);

        return mapper.selectUser();
    }

    @Override
    public int addUser(User user) {
        return getSqlSession().getMapper(UserMapper.class).addUser(user);
    }

    @Override
    public int deleteUser(int id) {
        return getSqlSession().getMapper(UserMapper.class).deleteUser(id);
    }
}

```

### Spring-dao.xml配置声明式事务部分

```

<!--配置声明式事务-->
<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <constructor-arg ref="dataSource"/>
</bean>

<!--结合AOP实现事务的织入-->
<!--配置事务通知: -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <!--指定要配置事务的方法-->
    <tx:attributes>
        <!--配置事务的传播特性propagation = [常用REQUIRED, NESTED]-->
        <tx:method name="add" propagation="REQUIRED"/>
        <tx:method name="delete" propagation="REQUIRED"/>
        <tx:method name="update" propagation="REQUIRED"/>
        <tx:method name="query" read-only="true"/>
        <tx:method name="*" propagation="REQUIRED"/> <!--所有方法-->
    </tx:attributes>
</tx:advice>

<!--配置事务切入-->
<aop:config>
    <aop:pointcut id="txPointCut" expression="execution(* com.xiang.mapper.*.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointCut"/>
</aop:config>

```

### 为什么需要事务？

如果不配置事务，可能存在数据提交不一致的情况

如果我们不在spring中配置声明式事务，我们就需要在代码中手动配置事务  
事务在项目的开发中十分重要，涉及到数据的一致性和完整性问题，不容马虎！

SSH: Struct2 + Spring + Hibernate

SSM: SpringMVC + Spring + Mybatis