1 Spring

1.1 简介

为了解决企业应用开发的复杂性.

Spring是一个轻量级控制反转（IOC）和面向切面（AOP）的容器框架.

Spring理念:使现有的技术更加容易使用.本身是一个大杂烩.

SSH Struct+Spring+Hibernate

SSM　Spring+Spring+Mybatis

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.2.0.RELEASE</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.2.0.RELEASE</version>
</dependency>
```

1.2 优点

Spring是一个开源的免费的框架(容器)

Spring是一个轻量级的 非入侵式的框架

控制反转（IOC）面前切面（AOP）

控制反转，即获取依赖对象的方式反转了

支持事务的处理，对框架整合的支持

Spring是一个轻量级控制反转（IOC）和面向切面（AOP）的容器框架.

1.3拓展

Spring Boot

快速开发的脚手架

基于SpringBoot可以快速的开发单个微服务

约定大于配置

SpringCloud基于Springboot实现

2 IOC理论推导

原来用户的需求可能会改变原有的业务代码，修改一次的成本十分昂贵。通过引入接口，使用set接口实现，已经发生革命性的变化。之前是主动创建对象，主动权在程序员手上，使用set注入的方法，程序不再具有主动性，而是变成了被动的接受对象。
从本质上解决了问题，程序员不用再去创建对象，系统的耦合性大大降低，可以更关注再业务的实现上，这就是IOC原型。（控制反转）

```
public class UserServiceImpl implements UserService {
    //之前的方式
    //private UserDao userDao = new UserDaoImpl();
    //IOC控制反转的方式，将主动权由业务层转换为用户层
    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();
    }
}
```

1.UserDao接口

```
package com.kuang.dao;

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

2.UserDaoImpl实现类

```
public class UserDaoImpl implements UserDao{
    public void getUser() {
        System.out.println("获取默认用户数据");
    }
}
```

```
package com.kuang.dao;

public class UserDaoMysqlImpl implements UserDao{
    public void getUser() {
        System.out.println("Mysql获取用户数据");
    }
}
```

3.UserService业务接口

```
package com.kuang.UserService;

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

4.UserserviceImpl业务实现类

```
package com.kuang.UserService;

import com.kuang.dao.UserDao;
import com.kuang.dao.UserDaoMysqlImpl;

public class UserServiceImpl implements UserService {
    //之前的方式
    //private UserDao userDao = new UserDaoImpl();
    //IOC控制反转的方式，将主动权由业务层转换为用户层
    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();
    }
}
```

5.用户层

```
import com.kuang.UserService.UserServiceImpl;
import com.kuang.dao.UserDaoImpl;
import com.kuang.dao.UserDaoMysqlImpl;

public class MyTest {
    public static void main(String[] args) {
        //用户实际调用的是业务层，dao层不需要接触
        //之前的方式
        //UserServiceImpl userService = new UserServiceImpl();
        //使用set之后实现控制反转
        userService.setUserDao(new UserDaoMysqlImpl());
        userService.getUser();
    }
}
```

在之前业务中，用户需求可能回影响原来的代码，需要根据用户的需求去修改代码，如果程序代码量十分大，修改一次的成本昂贵。

使用一个set接口实现；

```
private UserDao userDao;

//利用set进行动态实现值的注入
public void setUserDao(UserDao userDao) {
    this.userDao = userDao;
}
```

1.之前，程序是主动创建对象，控制权在程序员手上。

2.使用了set注入后，程序不再具有主动性，而是变成了被动的接受对象

从本质上解决了问题，不用再去管理对象的创建。系统的耦合性大大降低，可以更加专注再业务的实现上，这是IOC的原型。

2.1 IOC本质

IOC是一种设计思想，DI（依赖注入）是实现IOC的一种方法

控制反转，即获取依赖对象的方式反转了

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）

控制：谁来控制对象的创建

反转：程序本身不创建对象，而变成被动的接收对象

一句话：对象由Spring创建，管理，装配

3 HelloSpring

4 IoC创建方式

```
    <!--1.下标赋值-->
    <bean id="user" class="com.kuang.pojo.User">
        <constructor-arg index="0" value="狂神说Java"/>
    </bean>
<!--  2.通过类型创建,不建议使用  -->
    <bean id="user" class="com.kuang.pojo.User">
        <constructor-arg type="java.lang.String" value="chen"/>
    </bean>
<!--  3.通过参数名设置  -->
    <bean id="user" class="com.kuang.pojo.User">
        <constructor-arg name="name" value="sadas"/>
    </bean>
```

总结：在配置文件加载的时候，容器中管理的对象就已经初始化了

5 Spring配置

5.1 别名

```
<alias name="user" alias="user2"/>
```

5.2 Bean的配置

```
<!--  id:bean的唯一标识符,对象名 class:包名+类型   name:也是别名,而且可以取多个别名-->
    <bean id="user" class="com.kuang.pojo.User" name="user3">
        <property name="name" value="西部开源"/>
    </bean>
```

5.3 import

一般用于团队开发使用，可以将多个配置文件，导入合并为一个

假设项目中有多个人开发，这三个人负责不同的类开发，不同的类需要注册在不同的bean中，可以利用import将所有人的bean.xml合并为一个总的，使用时，直接使用总的即可。

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

6 依赖注入

6.1构造器注入

6.2 set方式注入【重点】

依赖注入：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

【环境搭建】

1.复杂类型

```
package com.kuang.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 + '\'' +
                '}';
    }
}
```

```java
package com.kuang.pojo;

import java.util.*;

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

    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> getHobbys() {
        return hobbys;
    }

    public void setHobbys(List<String> hobbys) {
        this.hobbys = hobbys;
    }

    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 String getWife() {
        return wife;
    }

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

    public Properties getInfo() {
        return info;
    }

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

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

2.真实测试对象

```
    <bean id="address" class="com.kuang.pojo.Address">
        <property name="address" value="洛阳"/>
    </bean>


    <bean id="student" class="com.kuang.pojo.Student">
        <!--    1.普通值注入      -->
        <property name="name" value="陈恒飞"/>
        <!--    2.Bean注入 ref      -->
        <property name="address" ref="address"/>
        <!--    3.数组注入,ref      -->
        <property name="books">
            <array>
```

```
                <value>西游记</value>
                <value>水浒传</value>
                <value>三国</value>
                <value>啦啦啦</value>
            </array>
        </property>
        <!--   4.list     -->
        <property name="hobbys">
            <list>
                <value>听歌</value>
                <value>看电影</value>
                <value>掉代码</value>
            </list>
        </property>
        <!--   5.Map     -->
        <property name="card">
            <map>
                <entry key="身份证" value="12345678916"/>
                <entry key="银行卡" value="56869488731"/>
            </map>
        </property>
        <!--   6.Set     -->
        <property name="games">
            <set>
                <value>LOL</value>
                <value>COC</value>
                <value>BOB</value>
            </set>
        </property>
        <!--   7.wife     -->
        <property name="wife">
            <null/>
        </property>
        <!--   8.properties     -->
        <property name="info">
            <props>
                <prop key="学号">2020313</prop>
                <prop key="性别">男</prop>
                <prop key="姓名">陈恒飞</prop>
            </props>
        </property>
    </bean>
```

```
import com.kuang.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 student = (Student) context.getBean("student");
        System.out.println(student.toString());
    }
}
```

6.3 拓展

p命名和c命名空间不能直接使用，需要导入xml约束

```
<!--  p,c命名注入  -->
    <!--  p命名空间注入,可以注入属性的值  -->
    <bean id="user" class="com.kuang.pojo.User" p:name="陈" p:age="18"/>

    <bean id="user2" class="com.kuang.pojo.User" c:age="18" c:name="小陈"/>
```

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

6.4 Bean的作用域

Scope	Description
singleton	(默认)将每个 Spring IoC 容器的单个 bean 定义范围限定为单个对象实例。
prototype	将单个 bean 定义的作用域限定为任意数量的对象实例。
request	将单个 bean 定义的范围限定为单个 HTTP 请求的生命周期。也就是说，每个 HTTP 请求都有一个在单个 bean 定义后面创建的 bean 实例。仅在可感知网络的 Spring `ApplicationContext` 中有效。
session	将单个 bean 定义的范围限定为 HTTP `Session` 的生命周期。仅在可感知网络的 Spring `ApplicationContext` 上下文中有效。
application	将单个 bean 定义的范围限定为 `ServletContext` 的生命周期。仅在可感知网络的 Spring `ApplicationContext` 上下文中有效。
websocket	将单个 bean 定义的范围限定为 `WebSocket` 的生命周期。仅在可感知网络的 Spring `ApplicationContext` 上下文中有效。

1.单例模式（Spring默认机制）

```
<bean id="user2" class="com.kuang.pojo.User" c:age="18" c:name="小陈"
scope="singleton"/>
```

2.原型模式

每次从容器中get时，都会产生一个新对象

```
<bean id="user2" class="com.kuang.pojo.User" c:age="18" c:name="小陈"
scope="prototype"/>
```

3.其余的request、session、application这些只能在web开发中用到

7 Bean的自动装配

自动装配是Spring满足bean依赖一种方式

Spring回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】

7.1 测试

1.搭建环境：一个人有两只宠物

```
package com.kuang.pojo;

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

    public Cat getCat() {
        return cat;
    }

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

    public Dog getDog() {
        return dog;
    }

    public void setDog(Dog dog) {
        this.dog = dog;
    }

    public String getName() {
        return name;
    }

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

    @Override
    public String toString() {
        return "People{" +
                "cat=" + cat +
```

```
                ", dog=" + dog +
                ", name='" + name + '\'' +
                '}';
    }
}
```

7.2 byName自动装配

```
<bean id="cat" class="com.kuang.pojo.Cat"/>
<bean id="dog" class="com.kuang.pojo.Dog"/>
<!--
       byName:会自动在容器上下文中查找,和自己对象set方法后面的值对应的beanid
 -->
<bean id="people" class="com.kuang.pojo.People" autowire="byName">
    <property name="name" value="xass"/>
</bean>
```

7.3 byType自动装配

```
<bean id="cat" class="com.kuang.pojo.Cat"/>
<bean id="dog123" class="com.kuang.pojo.Dog"/>
<!--
       byName:会自动在容器上下文中查找,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,和自己对象属性类型相同的bean
 -->

<bean id="people" class="com.kuang.pojo.People" autowire="byType">
    <property name="name" value="xass"/>
</bean>
```

小结：byName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set方法的值一致

byType时，需要保证所有bean的class唯一，并且bean需要和自动注入的属性的类型一致

7.4 使用注解实现自动装配

Autowired可以省去`<property name="args" ref="class">`

要使用注解须知：

1.导入约束context约束

2.配置注解的支持 `<context:annotation-config/>`

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

    <context:annotation-config/>

</beans>
```

@Autowired

直接在属性上添加即可也可以在set方式上使用

使用Autowired可以不用编写set方法，前提是这个自动装配的属性在IOC中存在，且符合byName

Autowired测试代码

```java
public class People {
    //如果显示定义了Autowired属性为false,说明这个对象可以为null,否则不允许为空
    @Autowired(required = false)
    private Cat cat;
    @Autowired
    private Dog dog;
    private String name;

    public Cat getCat() {
        return cat;
    }

    public Dog getDog() {
        return dog;
    }
}
```

```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/context/spring-aop.xsd">

    <context:annotation-config/>

    <bean id="cat" class="com.kuang.pojo.Cat"/>
    <bean id="dog" class="com.kuang.pojo.Dog"/>

    <bean id="people" class="com.kuang.pojo.People"/>
```

```
</beans>
```

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

@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

执行顺序不同：

8 使用注解开发

在Spring4之后，要使用注解开发，必须要保证aop的包导入了

1.bean

2.属性如何注入

```
@Component
public class User {
    @Value("陈恒飞")        //相当于<property name = "name" value = "陈恒飞"/>
    public String name;
}
```

3.衍生的注解

@componet有几个衍生注解，在web开发中按照mvc三层架构分层

@dao【@Repository】

@controller【@Service】

** @service【@Controller】**

这四个注解功能一样，都是代表将某个类注册到Spring中，装配Bean

4.自动装配

```
## 注解说明
@Autowired 自动装配通过类型。 名字
    如果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

6.小结

> 注册bean-->@component代替 注入值----->@Value代替 作用域---->@scope代替

```java
package com.kuang.pojo;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;
//等价于<bean id="user" class="com.kuang.pojo.User"/>
@Component
@Scope("singleton")
public class User {
    @Value("陈恒飞")       //相当于<property name = "name" value = "陈恒飞"/>
    public String name;
}
```

xml更加万能,适用于任何场合,为何简单方便

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

注解只负责属性的注入

在使用过程中,只需要注意一个问题,必须让注解生效,就需要开启注解的支持

```xml
<!--  2.指定要扫描的包，这个包下的注解就会生效  -->
<context:component-scan base-package="com.kuang"/>
<context:annotation-config/>
```

9 完全使用Java的方式配置Spring

完全不使用Spring的xml配置，全权交给Java来做

JavaConfig是Spring的一个子项目，在Spring4之后变成了核心功能

`AnnotationConfigApplicationContext`

1.pojo类

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

    public String getName() {
        return name;
    }

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

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

```
                '}';
    }
}
```

2.JavaConfig配置类

```
@Configuration  //也会被Spring容器托管，注册到容器中，因为本身就是一个@Component
@ComponentScan("com.kuang.pojo")
@Import(MyConfig2.class)
public class MyConfig {

    //注册一个Bean相当于之前写的一个Bean标签
    //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 getUser(){
        return new User();  //返回要注入到Bean的对象
    }
}
```

```
@Configuration
public class MyConfig2 {
}
```

3.容器：测试类

```
public class MyTest {
    public static void main(String[] args) {
        //如果完全使用了配置类方式去做，只能通过AnnotationConfigApplicationContext来获取容器，通过配置类的class对象加载
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(MyConfig.class);
        User getUser = (User) context.getBean("getUser");
        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

代理模式就是SpringAOP的底层【SpringAOP和SpringMVC】

代理模式的分类：

静态代理

动态代理

10.1 静态代理

角色分析：

抽象的角色：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，一般会做一些附属操作

客户：访问代理对象的人

```
//房东
public class Host implements Rent{
    public void rent() {
        System.out.println("房东要出租房子");
    }
}
```

```
//代理
public class Proxy implements Rent{
    private Host host;

    public Proxy() {
    }

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

    public void rent() {
        seeHouse();
        hetong();
        host.rent();
        free();
    }


    //看房
    public void seeHouse(){
        System.out.println("中介带看房");
    }

    public void free(){
        System.out.println("收中介费");
    }

    public void hetong(){
        System.out.println("签合同");
    }
}
```

```java
//客户
public class Client {
    public static void main(String[] args) {
        Host host = new Host();
        //代理
        Proxy proxy = new Proxy(host);
        proxy.rent();
    }
}
```

使用代理的优势：

可以使真实角色的操作更加纯粹！不用关注一些公共业务

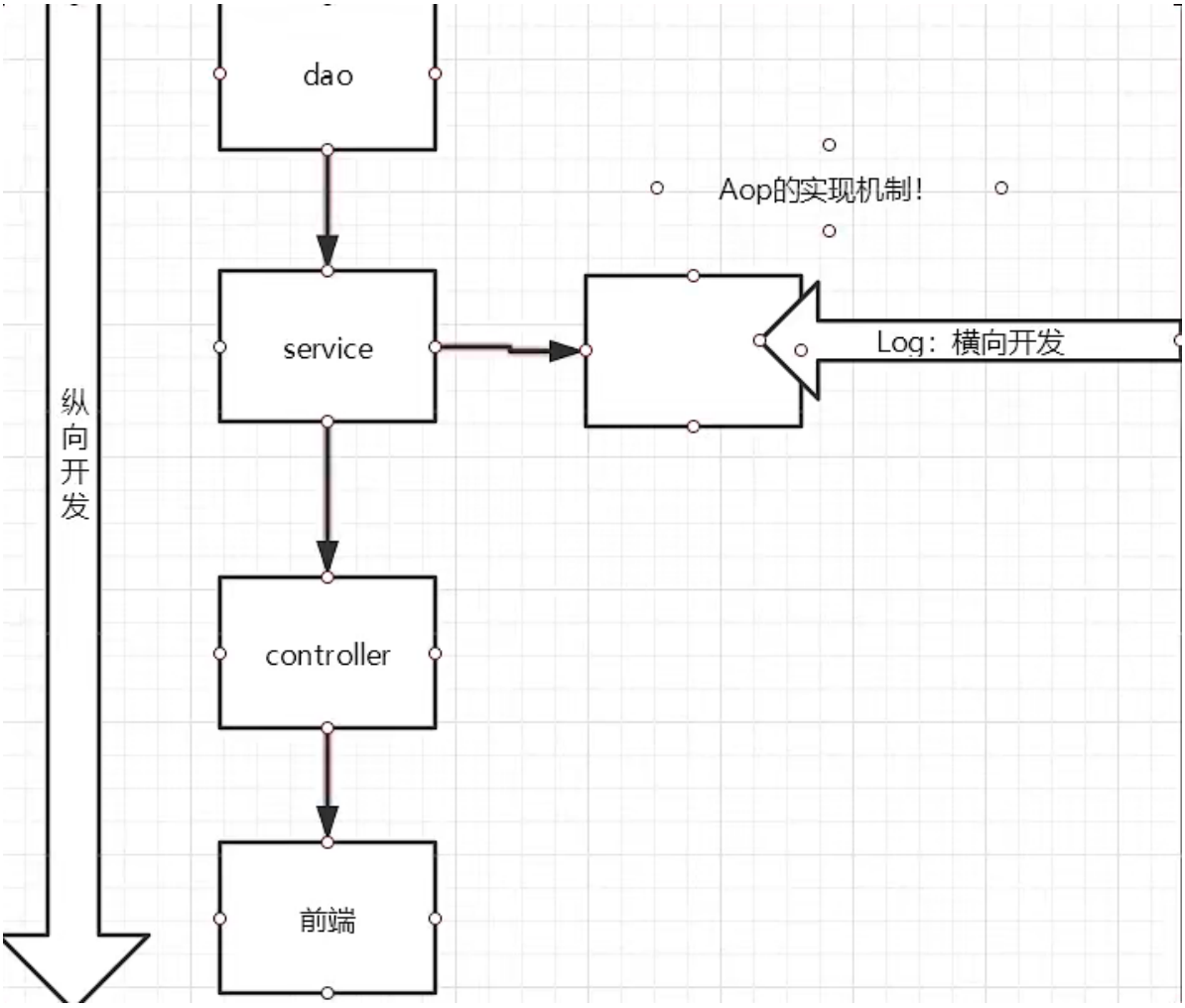
公务业务交给代理角色，实现业务的分工

公共业务发生扩展时，方便集中管理

10.2 静态代理再理解

spring-08-proxy两个例子好好理解

> 聊聊AOP，代码对应spring-08-proxy



10.3 动态代理

一个动态代理类代理的是一个接口,一般就是对应的一类业务

动态代理的底层都是反射

> 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----JDK动态代理
> >
> > 基于类:cglib
> >
> > Java字节码实现:javasist

需要了解两个类:

> ## Proxy:生成动态代理
>
> ## InvocationHandler:调用处理程序,并返回结果

```
//使用这个类，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;
    }

    //生成得到代理类
    public Object getProxy(){
        return
Proxy.newProxyInstance(this.getClass().getClassLoader(),target.getClass().getInt
erfaces(),this);
    }

    //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());
        Object result = method.invoke(target, args);
        return result;
    }

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

```
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();

        //设置要代理的对象
```

```
        pih.setTarget(userService);

        //动态生成代理类
        UserService proxy = (UserService) pih.getProxy();

        proxy.add();
    }
}
```

11 AOP

11.1 AOP

AOP: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技术,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.

切面:横切关注点要完成的工作,即类中的一个方法.

11.2 在Spring中实现AOP

方式一:使用Spring的API接口

```
public class UserServiceImpl implements UserService {
    public void add() {
        System.out.println("增加了一个用户");
    }

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

    public void update() {
        System.out.println("更新了一个用户");
    }

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

```
public class Log implements MethodBeforeAdvice {

    //method要执行的目标对象方法
    //args: 参数
    //target: 目标对象
    public void before(Method method, Object[] args, Object target) throws
Throwable {
        System.out.println(target.getClass().getName()+"的"+method.getName()+"被
执行了");

    }
}
```

```java
public class AfterLog implements AfterReturningAdvice {
    //返回后returnValue
    public void afterReturning(Object returnValue, Method method, Object[] args,
Object target) throws Throwable {

        System.out.println("执行了"+method.getName()+"返回结果为："+returnValue);
    }
}
```

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

    <bean id="userService" class="com.kuang.service.UserServiceImpl"/>
    <bean id="log" class="com.kuang.log.Log"/>
    <bean id="afterLog" class="com.kuang.log.AfterLog"/>

    <!--  配置AOP 需要导入aop的约束 -->
    <aop:config>
        <!--   切入点  expression:表达式   -->
        <aop:pointcut id="pointcut" expression="execution(*
com.kuang.service.UserServiceImpl.*(..))"/>
        <!--   执行环绕增加,把log类切入到这个上面  -->
        <aop:advisor advice-ref="log" pointcut-ref="pointcut"/>
        <aop:advisor advice-ref="afterLog" pointcut-ref="pointcut"/>
    </aop:config>
</beans>
```

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

        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
        //动态代理的是接口
        UserService userService = (UserService) context.getBean("userService");

        userService.add();
    }
}
```

方式二:自定义实现AOP

```xml
    <bean id="userService" class="com.kuang.service.UserServiceImpl"/>
    <bean id="log" class="com.kuang.log.Log"/>
    <bean id="afterLog" class="com.kuang.log.AfterLog"/>


<!--  方式二:自定义类  -->
```

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

```
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=========");
    }
}
```

方式三:使用注解实现

```
//方式三:使用注解方式实现AOP
@Aspect //标注这个类是一个切面
public class AnnotationPointCut {
    @Before("execution(* com.kuang.service.UserServiceImpl.*(..))")
    public void before(){
        System.out.println("==========方法执行前============");
    }

    @After("execution(* com.kuang.service.UserServiceImpl.*(..))")
    public void after(){
        System.out.println("==========方法执行后============");
    }

    //在环绕增强中可以给定一个参数,代表要获取处理切入的点
    @Around("execution(* com.kuang.service.UserServiceImpl.*(..))")
    public void around(ProceedingJoinPoint jp) throws Throwable {
        System.out.println("环绕前");

        //Signature signature = jp.getSignature();
        //System.out.println("signature:"+signature);
        //执行方法
        Object proceed = jp.proceed();

        System.out.println("环绕后");
    }

}
```

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

        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
        //动态代理的是接口
        UserService userService = (UserService) context.getBean("userService");

        userService.add();
    }
}
```

```
    <bean id="userService" class="com.kuang.service.UserServiceImpl"/>
    <bean id="log" class="com.kuang.log.Log"/>
    <bean id="afterLog" class="com.kuang.log.AfterLog"/>

    <!-- 方式三   -->
    <bean id="annotationPointCut" class="com.kuang.diy.AnnotationPointCut"/>
    <!--  开启注解支持  -->
    <aop:aspectj-autoproxy/>
```

12 整合Mybatis

1.导入相关jar包

```
<dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.11</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.47</version>
    </dependency>
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.6</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.3.16</version>
    </dependency>
<!--    连接spring-->
    <!-- https://mvnrepository.com/artifact/org.springframework/spring-jdbc -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>5.3.16</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.mybatis/mybatis-spring -->
    <dependency>
        <groupId>org.mybatis</groupId>
```

```
        <artifactId>mybatis-spring</artifactId>
        <version>2.0.7</version>
    </dependency>

    <dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjweaver</artifactId>
        <version>1.9.4</version>
    </dependency>
```

2.编写配置文件

3.测试

12.1 回顾mybatis

1.编写实体类

2.编写核心配置文件

3.编写接口

4.编写mapper，在mybatis-config.xml中注册mapper

5.添加build，配置resources,来防止我们资源导出失败的问题

12.2 mybatis-spring

1.编写数据源

2.sqlSessionFactory

3.sqlSessionTemplate

```
<?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">

    <!--DataSource:使用Spring的数据源替换mybatis c3p0 dbcp druid
        使用Spring提供的JDBC
org.springframework.jdbc.datasource.DriverManagerDataSource
        -->
    <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&amp;useUnicode=true&amp;characterEncoding=utf-8"/>
        <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/kuang/mapper/*.xml"/>
    </bean>

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

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

</beans>
```

4.需要给接口添加实现类添加set方法，自己写的实现类注入到spring中

```
public class UserMapperImpl implements UserMapper {
    //原来所有操作都使用sqlSession执行,现在都使用SqlSessionTemplate

    private SqlSessionTemplate sqlSession;

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

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

5.测试

```
@Test
public void test1() throws IOException {
    //String resources = "mybatis-config.xml";
    //InputStream in = Resources.getResourceAsStream(resources);
    //SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(in);
    //SqlSession sqlSession = sqlSessionFactory.openSession(true);
    //
    //UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    //List<User> users = mapper.selectUser();
    //for (User user : users) {
    //    System.out.println(user);
    //}
    //sqlSession.close();
    ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
    UserMapper userMapper = context.getBean("userMapper", UserMapper.class);
```

```
    for (User user : userMapper.selectUser()) {
        System.out.println(user);
    }
}
```

13 声明式事务

13.1 回顾

把一组业务当成一个业务来做，要么都成功，要么都失败

在项目开发中，十分重要，涉及到数据的一致性，不能马虎

确保完整性和一致性

事务的ACID原则

原子性：要么成功，要么失败

一致性：资源和状态保持一致

隔离性：多个业务可能操作同一个资源，防止数据损坏

持久性：事务一旦提交，无论系统发生什么问题，结果都不会被影响，被持久化的写入存储器中

四个原则合起来就是要么都成功，要么都失败

13.2 Spring中的事务管理

声明式事务：AOP

编程式事务：需要在代码中，进行事务的管理（try catch捕获处理）

配置声明式事务：在spring-dao中

```
<!--插入事务-->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <constructor-arg ref="dataSource" />
</bean>
<!--结合AOP实现事务的植入-->
<!--配置事务的类-->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <!--给哪些方法配置事务-->
    <!--配置事务的传播特性:new-->
    <tx:attributes>
        <tx:method name="add" propagation="REQUIRED"/>
        <tx:method name="delete" propagation="REQUIRED"/>
        <tx:method name="update" propagation="REQUIRED"/>
        <tx:method name="query"/>
        <tx:method name="*" propagation="REQUIRED"/>
    </tx:attributes>
</tx:advice>
<!--配置事务切入-->
<aop:config>
    <aop:pointcut id="txPointCut" expression="execution(* com.kuang.mapper.*.*
(..))"/>
```

```
    <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointCut"/>
</aop:config>
```

为什么要配置事务？（支持事务需要导入aspectjweavor）

如果不配置事务，可能存在数据提交不一致的问题；

如果不在spring中配置声明式事务，需要在代码中手动配置事务

事务在项目开发中十分重要，涉及到数据一致性和完整性问题，不容马虎