

Spring

1、Spring框架的引言和第一个环境搭建

2、IOC和DI核心思想

3、Spring中SET方式注入语法

 SET注入

4、Spring中自动注入和构造注入的语法

 构造注入

 自动注入

5、Spring中工厂特性

 Spring工厂的相关特性

 Spring中工厂创建对象的模式

 Spring工厂创建对象的原理

 Spring工厂管理组件生命周期

 单例对象

 多例对象

 Spring工厂好处

 (第二天课程)

6、IOC和DI快速复习

7、代理概念和静态代理开发

8、动态代理原理

9、Spring中aop编程概念提取

 aop编程

10、第一个aop编程开发步骤

 Spring aop编程的编程步骤

11、aop编程之环绕通知

 aop编程之环绕通知开发

12、切入点表达式

 spring (pointcut)切入点表达式

 execution切入点表达式

 within切入点表达式

13、后置通知和异常通知

 (第三天课程)

14、IOC和AOP快速复习

 Spring框架核心

 IOC and DI

 AOP

15、Spring中创建复杂对象的方式

 Spring如何创建复杂对象

16、Spring整合mybatis思路分析

17、SM整合之DAO编程开发

 Spring整合Mybatis编程DAO层开发(整合思路)

 Spring Mybatis整合之DAO层编程步骤

18、SM整合之Service层事务控制

 SM整合之Service层事务控制思路分析

 SM整合之DAO和Service部分开发 (一)

 (第四天课程)

19、SM整合编程复习回顾

 SM 整合开发

20、SM整合Service层事务控制优化思路分析

- Spring中处理事务的两种方式
 - 编程式事务处理
 - 声明式事务处理【推荐】
- SM整合开发之Service层事务优化
 - Spring框架开发声明式事务编程

21、SM整合之最终编码

- SM整合最终编程步骤

22、log4j日志使用

- SM整合中使用log4j

23、事务的传播属性

24、事务相关属性

25、Spring整合Struts2开发

26、SSM整合之编程步骤

27、SSM整合之编程实现

28、Spring中的相关注解说明

- Spring框架的注解式开发

29、SSM整合注解式开发

- 开发步骤

Spring

框架课程安排：倒数第二个阶段

1.idea 开发工具	1天
2.ajax 异步请求技术	2天
3.spring 项目管理框架	5天
4.springmvc 控制器框架	2天
5.springboot 微框架	2天
6.bootstrap 移动端框架	4-5天
7.linux 操作系统	3天
8.redis nosql内存型数据库	2-3天
9.elasticsearch 分布式搜索引擎	3天

框架总课程时间： 26天时间

1、Spring框架的引言和第一个环境搭建

• Spring框架引言

1.spring的引言

spring 春天 2002年 罗德 约翰森 创作

定义：Spring框架是一个集众多设计模式(工厂设计模式 代理设计模式 策略设计模式 单例设计...)于一身的‘开源的’、‘轻量级’的‘项目管理’框架。致力于JAVAEE轻量级解决方案

轻量级解决方案：提供一个简单的、统一的、高效的方式构造整个应用，并且可以将单层框架以最佳的组合揉和在一起建立一个连贯的体系。

dangdang: struts2 + Spring + mybatis

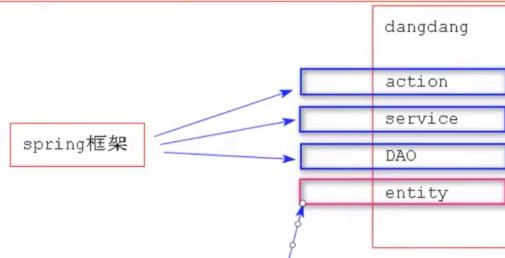
特点：Spring框架的本质不是替换项目中某个技术，而是将项目中使用单层框架进行 整合 | 管理 | 揉和

2.Spring框架具体在项目中核心使用

Spring 项目管理框架：就是用来对现有项目中组件进行管理(创建 使用 销毁)

用来负责项目中组件对象的创建 使用 销毁

```
UserAction  
UserServiceImpl  
UserDAO  
  
BookAction  
BookService  
BookDAO  
....
```



• Spring的第一个程序

- ▶ org.springframework:spring-core:4.3.2.RELEASE
- ▶ org.springframework:spring-beans:4.3.2.RELEASE
- ▶ org.springframework:spring-web:4.3.2.RELEASE
- ▶ org.springframework:spring-expression:4.3.2.RELEASE
- ▶ org.springframework:spring-aop:4.3.2.RELEASE
- ▶ org.springframework:spring-context:4.3.2.RELEASE
- ▶ org.springframework:spring-context-support:4.3.2.RELEASE
- ▶ org.springframework:spring-aspects:4.3.2.RELEASE
- ▶ org.springframework:spring-jdbc:4.3.2.RELEASE

1. 引入spring框架相关依赖

注意：引入spring核心模块以及依赖模块
spring-core 核心依赖
spring-context
spring-jdbc
spring-web
spring-expression
spring-aspects
spring-beans
spring-context-support

2. 引入spring框架配置文件

配置文件：名字随便 位置随便(本项目中resources目录里面)

applicationContext.xml | spring.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">

</beans>
```

3. 通过spring管理组件

```
<!--通过spring管理组件
  bean: 用来管理组件对象的创建
  class: 用来指定管理组件对象的全限定名 包.类
  id:    用来指定spring框架创建的当前组件对象在spring(容器|工厂)中唯一标识 全局唯一
-->
<bean class="init.UserDAOImpl" id="aa"></bean>
```

推荐：接口的首字母小写 userDAO

4. 启动工厂，获取对象进行测试

```
//启动工厂
ApplicationContext context = new ClassPathXmlApplicationContext("init/spring.xml");
//获取工厂中创建好的对象 参数：获取工厂中指定对应的唯一标识
UserDAO userDAO = (UserDAO) context.getBean("aa");
System.out.println(userDAO);
userDAO.save("xiaochen");
```

2、IOC和DI核心思想

- Spring框架中核心思想 IOC、AOP

- IOC：控制反转 | 反转控制 Inversion Of Control

- 定义：所谓控制反转其实指的是控制权利的反转
 - 通俗定义：就是将原来手动通过new关键字创建对象的权力交给Spring，通过在配置文件中配置bean标签形式创建对象，交给Spring由Spring创建对象过程

- DI：Dependency Injection 依赖注入

- 定义：为组件中成员变量完成赋值过程，这个过程称之为 依赖注入
 - 语法：
 - 1、组件对象中需要哪个组件|谁，将谁声明为成员变量并提供公开的SET方法
 - 2、在Spring的配置文件中对应的组件标签内使用property标签去完成属性的赋值操作

```

<!--管理DAO组件-->
<bean class="di.DeptDAOImpl" id="aa"></bean>

<!--管理Service组件-->
<bean class="di.DeptServiceImpl" id="deptService">
    <!--依赖的注入
        property: 用来给组件中的属性进行赋值操作
        name: 用来指定给组件中那个属性名进行赋值
        ref: 用来指定赋值对象在工厂中唯一标识 bean的id
    -->
    <property name="deptDAO" ref="aa" />
</bean>

```

- AOP：面向切面的编程
- **IOC 全部概述：**
 - **IOC 全部概述：**控制反转 就是将原来手动通过new关键字创建对象的权力交给Spring，由工厂创建对象过程，当然Spring不仅要创建对象还要在创建对象的同时通过DI的方式维护组件与组件调用关系
-



-

2.DI基本语法

DAO组件：

```
public class DeptDAOImpl implements DeptDAO {  
    @Override  
    public void save(String name) {  
        System.out.println("DAO name: " + name);  
    }  
}
```

Service组件：

```
public class DeptServiceImpl implements DeptService {  
    //需要DAO组件对象 依赖DAO组件  
    private DeptDAO deptDAO;  
    //公开的SET方法  
    public void setDeptDAO(DeptDAO deptDAO) {  
        this.deptDAO = deptDAO;  
    }  
  
    @Override  
    public void save(String name) {  
        System.out.println("deptService name: " + name);  
        deptDAO.save(name);  
    }  
}
```

配置文件赋值：

o

Service组件：

```
public class DeptServiceImpl implements DeptService {  
    //需要DAO组件对象 依赖DAO组件  
    private DeptDAO deptDAO;  
    //公开的SET方法  
    public void setDeptDAO(DeptDAO deptDAO) {  
        this.deptDAO = deptDAO;  
    }  
  
    @Override  
    public void save(String name) {  
        System.out.println("deptService name: " + name);  
        deptDAO.save(name);  
    }  
}
```

配置文件赋值：

o

配置文件赋值：

```
<!--管理DAO组件-->  
<bean class="di.DeptDAOImpl" id="aa"></bean>  
  
<!--管理Service组件-->  
<bean class="di.DeptServiceImpl" id="deptService">  
    <!--依赖的注入  
        property: 用来给组件中的属性进行赋值操作  
        name    : 用来指定给组件中那个属性名进行赋值  
        ref     : 用来指定赋值对象在工厂中唯一标识 bean的id  
    -->  
    <property name="deptDAO" ref="aa" />  
</bean>
```

启动工厂测试：

```
ApplicationContext context = new ClassPathXmlApplicationContext("di/spring.xml");  
//获取service组件  
DeptService deptService = (DeptService) context.getBean("deptService");  
deptService.save("百知教育");
```

3、Spring中SET方式注入语法

SET注入

1. Spring中注入方式

- | | |
|------------------------------|------|
| 1. SET注入：使用成员变量SET方式进行赋值 | (重点) |
| 2. 构造注入：使用构造方法形式进行属性的赋值 | (了解) |
| 3. 自动注入：就是通过在配置文件中完成类中属性自动赋值 | (了解) |

2. Spring中SET方式的注入语法

1. SET注入：使用类中属性的SET方法为属性完成赋值的过程
2. SET注入相关语法：

1. 注入八种基本类型+String类型+Date类型 统一使用value属性进行注入

```
<!-- SET注入相关语法      总结：八种基本类型+String类型注入+Date类型 使用的是value属性进行赋值-->
<property name="name" value="字符串类型"/>
<property name="price" value="2.222"/>
<property name="sex" value="true"/>
<!-- 注意：在Spring技术栈中日期格式默认为yyyy/MM/dd HH:mm:ss -->
<property name="bir" value="2021/12/20"/>
```

2. 注入对象类型（引用类型）时，统一使用 ref 属性进行注入

```
<property name="deptDAO" ref="deptDAO"/>
```

3. 注入数组类型,统一使用array标签进行注入

```
<!-- 注入数组类型array -->
<property name="qqs">
    <array>
        <value>测试一</value>
        <value>测试二</value>
        <value>测试三</value>
        <value>测试四</value>
        <value>测试五</value>
    </array>
</property>
```

```
<property name="deptDAOS">
    <array>
        <ref bean="deptDAO"/>
        <ref bean="deptDAO"/>
        <ref bean="deptDAO"/>
    </array>
</property>
```

4. 注入集合类型 list set map, list集合使用list、set集合使用set、map使用map properties使用props标签

```
<!-- 注入List集合 -->
<property name="habbys">
    <list>
        <value>睡觉</value>
        <value>吃饭</value>
        <value>打豆豆</value>
        <value>再打豆豆</value>
    </list>
</property>
<property name="deptDAOList">
    <list>
        <ref bean="deptDAO"/>
        <ref bean="deptDAO"/>
        <ref bean="deptDAO"/>
    </list>
</property>
```

```

<!-- 注入map entry-->
<property name="maps">
    <map>
        <entry key="aa" value="小城"/>
        <entry key="bb" value="中城"/>
        <entry key="cc" value="大城"/>
    </map>
</property>

```

```

<!-- 注入properties-->
<property name="properties">
    <props>
        <prop key="driver">com.mysql.jdbc.Driver</prop>
        <prop key="url">jdbc:mysql://localhost:3306/test</prop>
        <prop key="username">root</prop>
        <prop key="password">root</prop>
    </props>
</property>

```

4、Spring中自动注入和构造注入的语法

构造注入

1. Spring中注入方式之构造注入

1. 构造注入：使用类中构造方法为类中成员变量赋值过程，称之为构造注入
2. 构造注入语法：
 1. 需要哪个组件属性将谁声明为成员变量，并提供公开构造方法
 2. 在配置文件中对应的组件标签内部使用 `<constructor-arg>` 标签进行注入
3. 注入语法

1	<code><!-- 管理Service组件</code>
2	1、SET方式注入 注入时使用 <code>property</code> 标签
3	2、构造方法注入 注入时使用 <code>constructor-arg</code> 标签
4	<code>--></code>
5	<code><bean class="condi.EmpServiceImpl" id="empService"></code>

```

6      <constructor-arg index="0" name="name" value="String构造方式注入测
7      试"/>
8      <constructor-arg index="1" name="age" value="22"/>
9      <constructor-arg index="2" name="empDAO" ref="empDAO"/>
10     <constructor-arg index="3" name="qqs">
11     <array>
12       <value>array1</value>
13       <value>array2</value>
14       <value>array3</value>
15     </array>
16   </constructor-arg>
</bean>

```

自动注入

- autowire="byName"

根据注入的属性名与配置文件中的bean的id匹配，一致则注入，不一致报错

- autowire="byType"

根据注入的属性类型，与配置文件中的类型匹配，类型一致注入（在多个实现类时，会产生歧义）

- 注意：无论使用以上哪种方式注入都需要为属性提供set方法

1. Spring中自动注入

1. 自动注入：在Spring工厂配置文件中通过制定自动注入方式，开启组件属性的自动赋值

注意：

1. 底层使用原理也是SET方式注入
2. 自动注入需要在对应组件标签开启才能使用
3. 只能用于引用类型|对象类型|组件类型的注入

2. 自动注入语法

1. 需要谁将谁声明为成员变量，并提供SET方法
2. 在对应组件标签上加入autowired属性并指定自动注入方式即可完成注入

3. 注入示例

```

<!-- 管理DAO组件 -->
<bean class="autodi.StudentDAOImpl" id="studentDAO"></bean>
<bean class="autodi.StudentDAONewImpl" id="studentDAONew"></bean>

<!-- 管理Service组件
      autowire: 用来给组件中成员变量完成自动赋值操作
      byType: 根据类型完成自动注入，根据成员变量类型去工厂找，找到对应类型完成赋值，找不到不赋值
      注意：如果工厂中存在多个类型一致的组件，使用类型自动注入会报错
      byName: 根据名称完成自动注入，根据成员变量名字去工厂中获取与之一致名字，找到对应的赋值，找不到不赋值
-->
<bean class="autodi.StudentServiceImpl" id="studentService" autowire="byName"></bean>

```

```

<!-- 管理DAO组件 -->
<bean class="autodi.StudentDAOImpl" id="studentDAO"></bean>
<!--      <bean class="autodi.StudentDAONewImpl" id="studentDAONew"></bean>-->

<!-- 管理Service组件
    autowire: 用来给组件中成员变量完成自动赋值操作
    byType: 根据类型完成自动注入, 根据成员变量类型去工厂找, 找到对应类型完成赋值, 找不到不赋值
        注意: 如果工厂中存在多个类型一致的组件, 使用类型自动注入会报错
    byName: 根据名称完成自动注入, 根据成员变量名字去工厂中获取与之一致名字, 找到对应的赋值, 找不到不赋值
-->
<bean class="autodi.StudentServiceImpl" id="studentService" autowire="byType"></bean>

```

```

<!-- 管理DAO组件 --> byType          byName
<bean class="autodi.StudentDAOImpl" id="studentDAO"></bean>
<bean class="autodi.StudentDAONewImpl" id="studentDAONew"></bean>

<!-- 管理Service组件
    autowire: 用来给组件中成员变量完成自动赋值操作
    byType: 根据类型完成自动注入, 根据成员变量类型去工厂找, 找到对应类型完成赋值, 找不到不赋值
        注意: 如果工厂中存在多个类型一致的组件, 使用类型自动注入会报错
    byName: 根据名称完成自动注入, 根据成员变量名字去工厂中获取与之一致名字, 找到对应的赋值, 找不到不赋值
-->
<bean class="autodi.StudentServiceImpl" id="studentService" autowire="byType"></bean>

```

5、Spring中工厂特性

Spring工厂的相关特性

Spring中工厂创建对象的模式

注意：工厂默认在管理对象都是单例方式，单例方式无论在工厂获取多少次始终获取的是同一个对象

1. 默认Spring在管理组件对象时 **单例创建 singleton**
2. 如何修改工厂创建组件对象为多例

```
1 | <bean id="" class="xxxx.userAction" scope="prototype|singleton">
```

3. 具体配置

```

<!--管理DAO组件
scope: 用来指定工厂创建对象的模式
    默认值: singleton 单例
    prototype 多例
-->
<bean class="scope.TagDAOImpl" id="tagDAO" scope="prototype"></bean>

```

Spring工厂创建对象的原理

原理：反射 + 构造方法

```
// 工厂原理  
TagDAO tt = (TagDAO) Class.forName("scope.TagDAOImpl").newInstance();  
System.out.println(tt);
```

Spring工厂管理组件生命周期

1、组件对象什么时候创建 2、组件对象什么时候销毁

单例对象

工厂启动工厂中所有单例的对象随之创建，工厂销毁工厂中所有单例对象随之销毁

```
1 //=====TagDAOImpl.java=====  
2 // init  
3 public void init0719() {  
4     System.out.println("组件对象初始化");  
5 }  
6 // destroy  
7 public void destory() {  
8     System.out.println("组件对象销毁");  
9 }
```

```
1 <!--管理DAO组件  
2 scope: 用来指定工厂创建对象的模式  
3      默认值: singleton 单例  
4          prototype 多例  
5 生命周期方法:  
6      init-method="" 在对象初始化调用  
7      destroy-method="" 在对象销毁时调用  
8 -->  
9 <bean class="scope.TagDAOImpl" id="tagDAO" init-method="init0719" destroy-  
method="destory"></bean>
```

```
1 // 启动工厂  
2 ApplicationContext context = new ClassPathXmlApplicationContext("scope/spring.xml");  
3 // 关闭工厂  
4 ((ClassPathXmlApplicationContext)context).close();
```

```
七月 19, 2021 3:51:21 下午 org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@b81eda8: startup date [Mon Jul 19 15:51:21 CST 2021]; root of context hierarchy
七月 19, 2021 3:51:21 下午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
信息: Loading XML bean definitions from class path resource [scope/spring.xml]
组件对象初始化
组件对象销毁
七月 19, 2021 3:51:21 下午 org.springframework.context.support.ClassPathXmlApplicationContext doClose
信息: Closing org.springframework.context.support.ClassPathXmlApplicationContext@b81eda8: startup date [Mon Jul 19 15:51:21 CST 2021]; root of context hierarchy
```

多例对象

每次在工厂中使用时进行创建，工厂不负责多例对象的销毁

Spring工厂好处

- 解耦合 使用配置文件管理java类，在生产环境中更换类的实现时不需要重新部署，修改文件即可
- 减少jvm内存占用 Spring默认使用单例的模式创建bean，减少内存的占用
- 通过依赖注入方式建立组件与组件|对象与对象的依赖关系，方便我们进行组件代码的维护和管理

1. bean的创建模式

singleton: 单例 默认

在工厂中 全局唯一，只创建一次

prototype: 多例

全局不唯一，每次使用都会创建一个新的对象

```
1 <bean id="" class="xxxx.userAction" scope="prototype|singleton">
2   service,dao      ----->    singleton
3   struts2 action    ----->    prototype
```

注意：在项目开发中service, dao组件单例，struts2的Action必须为多例

2. bean的生产原理

原理：反射+构造方法

```
1 userDaoImpl userDao =
2   (UserDAOImpl)Class.forName("com.baizhi.dao.UserDAOImpl").newInstance();
3   System.out.println(userDAO);
```

3. bean的生命周期

- 何时创建

随着工厂启动，所有单例bean随之创建，非单例的bean，每次使用时创建

- 何时销毁

工厂关闭，所有bean随之销毁（注意：Spring对多例bean管理松散，不会负责多例bean的销毁）

4. bean工厂创建对象的好处

1. 使用配置文件管理java类，在生产环境中更换类的实现时不需要重新部署，修改文件即可
2. Spring默认使用单例的模式创建bean，减少内存的占用
3. 通过依赖注入建立了类与类之间的关系（使java之间关系更为清晰，方便了维护与管理）

（第二天课程）

6、IOC和DI快速复习

1. Spring框架

Spring框架：项目管理框架

核心：就是用来管理项目中组件对象的创建、使用、销毁

2. Spring框架第一个环境搭建

1. 引入依赖

spring-core

spring-context

spring-beans

spring-expression

spring-jdbc

spring-web

.....

2. 引入spring.xml

名字 随便 位置：resources目录下随便

3. 通过spring管理组件的创建

```
1 | interface UserDao { void save(String name); }
```

```
1 | class UserDaoImpl implements UserDao
```

spring.xml

```
1 | <bean class="xxx.UserDaoImpl" id="userDAO" />
```

4. 启动工厂

```
1 | ApplicationContext context = new  
2 | ClassPathXmlApplicationContext("xxx/spring.xml");  
2 | context.getBean("beanid")
```

3. spring核心思想

IOC & DI

IOC: inversion of control 控制反转

就是将手动通过new关键字创建对象的权力交给Spring，由工厂创建对象过程

DI: Dependency Injection 依赖注入

Spring工厂不仅要创建对象还要在创建对象同时维护组件和组件的依赖关系

AOP: 面向切面编程

4. spring中注入

SET方式[重点] 构造方法自动注入

1. SET注入：

定义：使用set方法形式为成员变量赋值过程称之为set注入

语法：

- 1、需要谁将谁声明为成员变量并提供公开SET方法
- 2、在Spring配置文件中对应组件标签内部使用property标签完成注入

1. String + 八种基本类型 + 日期类型 使用 value 属性进行注入

```
<property name="" value="xx">
```

注意：Spring中日期格式默认为 yyyy/MM/dd HH:mm:ss

2. 对象|引用类型

```
<property name="" ref="beanid">
```

3. 数组类型

```
1 | <!-- 非引用类型 -->  
2 | <property name="">  
3 |   <array>  
4 |     <value>xxx</value>  
5 |   </array>  
6 | </property>  
7 | <!-- 引用类型 -->  
8 | <property name="">  
9 |   <array>  
10 |     <ref bean="beanid"/>  
11 |   </array>  
12 | </property>
```

4. list set map

list

```

1 <!-- 非引用类型 -->
2 <property name="">
3   <list>
4     <value>xx</value>
5     <value>xxx</value>
6     <value>xxxx</value>
7     <value>xx</value>
8   </list>
9 </property>
10 <!-- 引用类型 -->
11 <property name="">
12   <list>
13     <ref bean="beanid"/>
14     <ref bean="beanid"/>
15     <ref bean="beanid"/>
16   </list>
17 </property>
```

map

```

1 <property name="maps">
2   <map>
3     <entry key="aa" value="小城"/>
4     <entry key="bb" value="中城"/>
5     <entry key="cc" value="大城"/>
6   </map>
7 </property>
8 <!-- 引用类型 -->
9 <property name="maps">
10  <map>
11    <entry key-ref="beanid" value-ref="beanid"/>
12    <entry key-ref="beanid" value-ref="beanid"/>
13  </map>
14 </property>
```

5. properties

```

1 <!-- 注入properties 集合使用props标签-->
2 <property name="properties">
3   <props>
4     <prop key="driver">com.mysql.jdbc.Driver</prop>
5     <prop key="url">jdbc:mysql://localhost:3306/test</prop>
6     <prop key="username">root</prop>
7     <prop key="password">root</prop>
8   </props>
9 </property>
```

5. spring工厂细节

1. 创建方式 —— 默认使用单例

```
<bean class="" id="" scope="singleton(service dao)|prototype(struts2action)">
```

2. 工厂原理

反射 + 构造方法

```
Class.forName("xxxx").newInstance();
```

3. 工厂生命周期

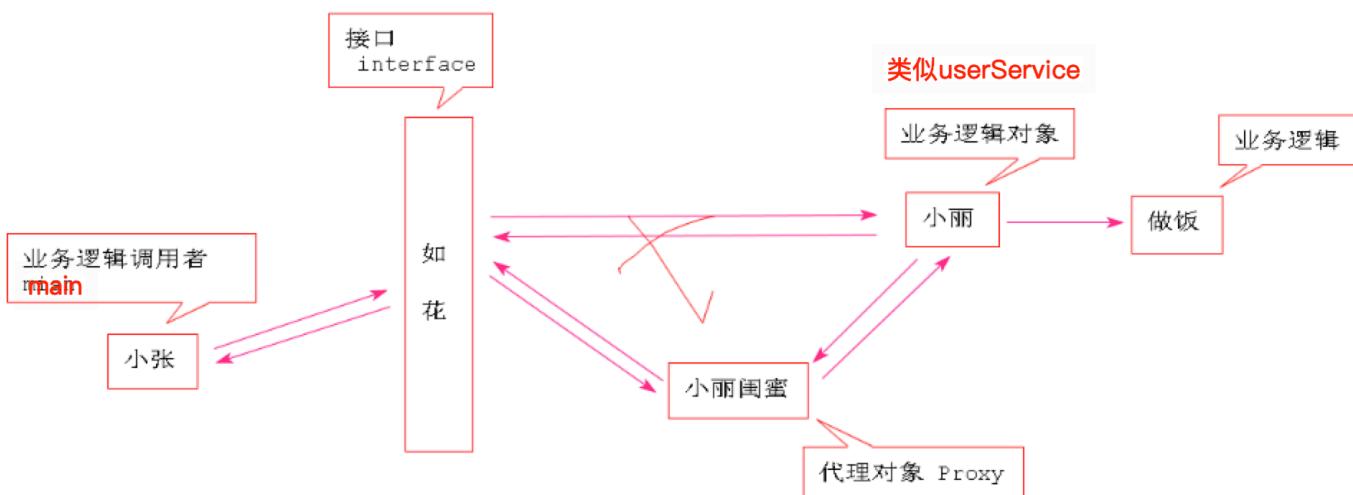
单例：工厂启动工厂中单例对象随之创建；工厂正常关闭，工厂中单例对象随之销毁

多例：每次使用时工厂进行多例对象的创建，Spring工厂不负责多例对象的销毁

7、代理概念和静态代理开发

(Proxy) 代理的引言

(Proxy) 代理的引言



1. 代理对象 (Proxy)

作用：在整个业务逻辑中完成传话(附加操作)的作用，同时也可以中断整个业务

好处：既可以保证原始业务逻辑 原始业务功能不变的同时还可以完成一些附加的操作

2. 如何开发一个代理对象？

1. 代理对象和业务逻辑对象(真正业务逻辑对象) 实现相同 接口

2. 代理对象中 依赖 真正业务逻辑对象

(Target Object)：目标对象 被代理的对象称之为 目标对象 原始业务逻辑对象称之为 目标对象

https://blog.csdn.net/wuque_perfect

8、动态代理原理

通过jdk提供的Proxy这个类,动态为现有的业务生成代理类

参数一:当前线程类加载器

参数二:生成代理类的接口类型

参数三:通过代理类对象调用方法时会优先进入参数三中的invoke方法Proxy.newProxyInstance(loader, interfaces, h); //返回值就是动态代理对象

```
public class TestDynamicProxy {
    public static void main(String[] args) {
        UserService userService = new UserServiceImpl(); // 目标类
        System.out.println("userService class: " + userService.getClass()); // userService class: class staticproxy.UserServiceImpl
        // 使用动态代理对象: 指的是在程序运行过程中动态通过代码的方式为指定的类生成动态代理对象
        // proxy: 用来生成动态代理对象的类
        /**
         * 参数1: ClassLoader 类加载器
         * 参数2: Class[] 目标对象的接口类型的数组
         * 参数3: InvocationHandler接口类型 invoke方法, 用来书写额外功能 附加操作
         * 返回值: 创建好的动态代理对象
         */
        //参数1
        ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
        //参数2
        Class[] classes = {UserService.class};
        System.out.println(classes); // [Ljava.lang.Class;@2503dbd3
        //...
        UserService userServiceDynamicProxy = (UserService) Proxy.newProxyInstance(classLoader, classes, new InvocationHandler() {
            /**
             * 通过动态代理对象自己里面代理方法时会优先指定InvocationHandler类中的invoke方法
             * @param proxy 当前创建好的代理对象
             * @param method 当前代理对象执行的方法对象
             * @param args 当前代理对象执行方法的参数
             */
            @Override
            public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                System.out.println("当前执行的方法: " + method.getName());
                // System.out.println("当前执行的方法" + method); // public abstract void staticproxy.UserService.save(java.lang.String)
                System.out.println("当前执行的方法的参数: " + args[0]);
                try {
                    System.out.println("开启事务");
                    // 调用目标类中业务方法, 通过反射机制, 调用目标类中当前方法
                    Object invoke = method.invoke(new UserServiceImpl(), args);
                    System.out.println("测试返回值: "+invoke);
                    System.out.println("提交事务");
                    return invoke; //如果有返回值, 就返回
                } catch (Exception e) {
                    e.printStackTrace();
                    System.out.println("回滚事务");
                }
                return null;
            }
        });
        System.out.println("userServiceDynamicProxy: " + userServiceDynamicProxy.getClass());
        System.out.println("\n=====动态代理测试=====");
        userServiceDynamicProxy.findAll("动态代理测试"); // 通过动态代理对象调用代理中的方法
    }
}
```

9、Spring中aop编程概念提取

aop编程

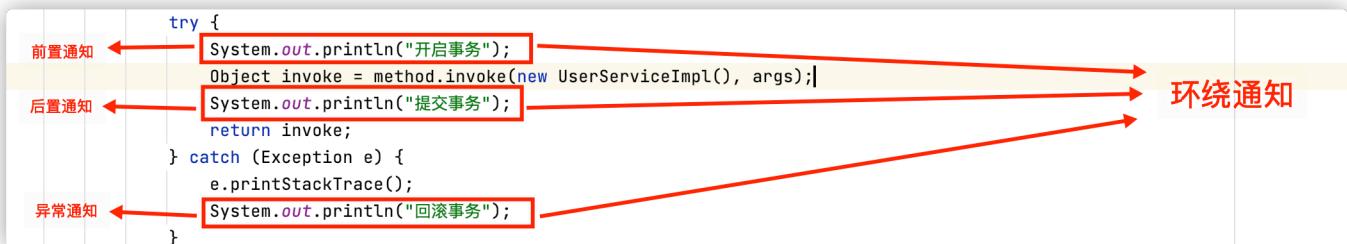
- AOP: Aspect (切面) Oriented (面向) Programming 面向切面编程
 - 底层原理: Java代理设计模式 动态代理
 - 好处: 在保证原始业务功能不变情况下, 通过代理对象完成业务中附加操作 (事务), 将业务中核心操作放在目标对象中执行, 实现了附加操作和核心操作解耦
- 通知 (Advice) : 除了目标方法以外的操作都称之为通知 (附加功能/事务通知/日志通知/性能通知.....)

- 切入点 (Pointcut) : 指定开发好的通知应用于项目中哪些组件中哪些方法, 一般通知多用于业务层
- 切面 (Aspect) = 通知 (advice) + 切入点 (pointcut)

面向切面的编程有两步: 1、开发通知 2、配置切入点

开发通知的过程就是告诉开发人员在通知中要实现具体的哪些额外功能

配置切入点的过程就是告诉开发人员这些通知应该用于哪些类 (为哪些类创建动态代理对象), 然后在动态代理对象内部去实现



- 总结: AOP (面向切面编程) : 1. 开发通知 (附加功能) 2. 配置切面点 3. 组装切面 (Aspect)

10、第一个aop编程开发步骤

Spring aop编程的编程步骤

1. 引入aop编程相关依赖

```

spring-aop
spring-expression
spring-aspect

```

2. 项目开发额外功能通知

环绕通知 MethodIntercept

前置通知 MethodBeforeAdvice

```

1 // 自定义记录业务方法名称的前置通知 前置通知: 目标方法执行之前先执行的额外操作
2 public class MyBeforeAdvice implements MethodBeforeAdvice {
3     // before 参数1: 当前执行方法对象 参数2: 当前执行方法的参数 参数3: 目标对象
4     @Override
5     public void before(Method method, Object[] objects, Object o) throws
6         Throwable {
7         System.out.println("当前执行的方法: " + method.getName());
8         System.out.println("当前执行的方法的参数: " + objects);
9         System.out.println("目标对象: " + o);
10    }

```

后置通知 AfterReturningAdvice

异常通知 ThrowsAdvice

3. 配置切面 spring.xml

3. 配置切面 spring.xml

a). 注册通知类

```
<bean id="myBeforeAdvice" class="aop.MyBeforeAdvice"/>
```

b). 组装切面 aspect = advice + pointcut

```
<!-- 组装切面 -->
<aop:config>
    <!-- 配置切入点 pointcut
        id: 切入点在工厂中唯一标识
        expression: 用来指定切入项目中哪些组件中哪些方法
                    execution(返回值 包.类名.*(..))
    -->
    <aop:pointcut id="pc" expression="execution(* aop.EmpServiceImpl.*(..))"/>
    <!-- 配置切面 advice-ref: 工厂中通知id pointcut-ref: 工厂切入点唯一标识 -->
    <aop:advisor advice-ref="myBeforeAdvice" pointcut-ref="pc"/>
</aop:config>
```

1. 注册通知类

```
<bean id="" class="xxxAdvice"></bean>
```

```
<bean id="myBeforeAdvice" class="aop.MyBeforeAdvice"/>
```

2. 组装切面

```
1 <aop:config>
2     <aop:pointcut></aop:pointcut>
3     <aop:advisor></aop:advisor>
4 </aop:config>
```

```
1 <aop:config>
2     <!-- 配置切入点 pointcut
3         id: 切入点在工厂中的唯一标识
4         expression: 用来指定切入项目中哪些组件中哪些方法
                    execution(返回值 包名.类名.方法名(参数类型))
5     -->
6     <aop:pointcut id="pc" expression="execution(* aop.EmpServiceImpl.*(..))"/>
7     <!-- 配置切面 advice-ref: 工厂中通知id pointcut-ref: 工厂切入点唯一标识 -->
8     <aop:advisor advice-ref="myBeforeAdvice" pointcut-ref="pc"/>
9
10 </aop:config>
```

4. 创建service组件

```
1 // 原始业务对象 目标对象
2 public class EmpServiceImpl implements EmpService{
3     @Override
4     public void save(String name) {
5         System.out.println("处理业务逻辑 调用save DAO name : " + name);
6     }
7
8     @Override
9     public String find(String name) {
10        System.out.println("处理业务逻辑 调用find DAO name : " + name);
11        return name;
12    }
13 }
```

5. 测试

```
1 // 启动工厂
2 ApplicationContext context = new
3 ClassPathXmlApplicationContext("aop/spring.xml");
4 // 获取对象
5 EmpService empService = (EmpService) context.getBean("empService");
6 System.out.println(empService.getClass());
7 // 代理对象
8 empService.find(" aop");
```

11、aop编程之环绕通知

aop编程之环绕通知开发

1. 引入aop相关依赖

spring-aop
spring-aspect
spring-expression (切入点表达式)

2. 开发业务组件

DeptDAO
DeptDAOImpl
DeptService
DeptServiceImpl

3. 开发环绕通知

```
public class MethodInvokeTimeAdvice implements MethodInterceptor {  
    //参数1: invocation 获取当前执行方法 获取当前执行方法参数 获取目标对象  
    @Override  
    public Object invoke(MethodInvocation invocation) throws Throwable {  
        System.out.println("=====进入环绕通知=====");  
        System.out.println("当前执行方法: " + invocation.getMethod().getName());  
        System.out.println("方法的参数: " + invocation.getArguments()[0]);  
        System.out.println("获取当前的目标对象: " + invocation.getThis());  
        try{  
            long start = new Date().getTime();  
            //放行目标方法  
            Object proceed = invocation.proceed(); //继续处理  
            long end = new Date().getTime();  
            System.out.println("方法: " + invocation.getMethod().getName() + ", 本次执行了 [" + (end-start) + "] ms!");  
            return proceed;  
        }catch (Exception e){  
            e.printStackTrace();  
            System.out.println("出现异常时业务处理");  
        }  
        return null;  
    }  
}
```

性能通知：计算业务层目标方法执行时长

```
1 // 参数: methodInvocation 获取当前执行方法 获取当前执行方法的参数 获取目标对象  
2 @Override  
3 public Object invoke(MethodInvocation methodInvocation) throws Throwable {  
4     System.out.println("=====进入环绕通知  
=====");  
5     System.out.println("当前执行方法: " + methodInvocation.getMethod().getName());  
6     System.out.println("方法的参数: " + methodInvocation.getArguments()[0]);  
7     System.out.println("获取当前的目标对象: " + methodInvocation.getThis());  
8     try {  
9         long start = new Date().getTime();  
10        // 放行目标方法  
11        System.out.println("=====放行目标方法  
=====");  
12        Object proceed = methodInvocation.proceed();  
13        long end = new Date().getTime();  
14        System.out.println(methodInvocation.getMethod().getName() + "方法本次执行了[" +  
15        (end-start) + "]ms !");  
16        return proceed;  
17    } catch (Exception e) {  
18        e.printStackTrace();  
19        System.out.println("环绕通知---出现异常时业务处理");  
20    }  
21    return null;  
}
```

4. 配置spring.xml

1. 管理 业务、DAO组件
2. 配置切面

b) 配置切面

- 1). 注册通知组件


```
<!--注册通知类-->
<bean id="methodInvokeTimeAdvice" class="com.baizhi.advices.MethodInvokeTimeAdvice"></bean>
```
- 2). 配置切面


```
<!--配置切面-->
<aop:config>
  <!--配置切入点-->
  <aop:pointcut id="pc" expression="execution(* com.baizhi.service.*ServiceImpl.*(..))"/>
  <!--组装切面-->
  <aop:advisor advice-ref="methodInvokeTimeAdvice" pointcut-ref="pc"/>
</aop:config>
```

1. 注册通知组件

```
1 <!-- 注册通知 -->
2 <bean id="methodInvokeTimeAdvice"
      class="com.adviceStudy.advices.MethodInvokeTimeAdvice"/>
```

2. 配置切面

```
1 <!-- 配置切面 -->
2 <aop:config>
3   <!-- 配置切入点pointcut -->
4   <aop:pointcut id="pc" expression="execution(*
      com.adviceStudy.service.*Service.*(..))"/>
5   <!-- 配置切面 -->
6   <aop:advisor advice-ref="methodInvokeTimeAdvice" pointcut-ref="pc"/>
7 </aop:config>
```

12、切入点表达式

spring (pointcut)切入点表达式

主要用来决定项目中哪些组件中哪些方法需要加入通知

expression="切入点表达式"

execution切入点表达式

方法级别的切入点表达式

控制粒度：方法级别

效率低

完整语法：

```
1 1. execution (访问权限修饰符 返回值 包名.类名.方法名 (参数类型) )  
2 2. execution (返回值 包名.类名.方法名 (参数类型) )
```

*: 任意多个字符

```
1 1. execution(* com.baizhi.service.*.*(..)) 【使用较多】  
2   包:      com.baizhi.service  
3   类:      任意类  
4   方法:    任意方法  
5   参数:    任意参数  
6   返回值:  任意返回值类型  
7 2. execution(String com.baizhi.service.*ServiceImpl.*(..))  
8   包:      com.baizhi.service  
9   类:      以ServiceImpl结尾的类  
10  方法:    方法名任意  
11  参数:    任意参数  
12  返回值:  返回值必须是String类型  
13 3. execution(String com.baizhi.service.*Service*.*(String))  
14  包:      com.baizhi.service  
15  类:      类名中包含Service关键字  
16  方法:    方法名任意  
17  参数:    参数只有一个 类型必须是String  
18  返回值:  返回值必须是String类型  
19 4. execution(String com.baizhi.service...*.*(..)) 【使用较多】  
20  包:      com.baizhi.service及这个包中子包的子包  
21  类:      任意类  
22  方法:    方法名任意  
23  参数:    任意参数  
24  返回值:  任意类型  
25 5. execution(String com.baizhi.service.*ServiceImpl.*(..)) 【使用较多】  
26  包:      com.baizhi.service  
27  类:      以ServiceImpl结尾的类  
28  方法:    方法名任意  
29  参数:    任意参数  
30  返回值:  任意类型  
31 6. execution(* .*.*(..)) 【全部方法, 避免使用这种方式】  
32  包:      项目中所有包  
33  类:      项目中所有类  
34  方法:    所有方法  
35  参数:    所有参数  
36  返回值:  任意类型  
37  
38 # 注意: 尽可能精准切入, 避免不必要的切入
```

注意: 方法级别的切入点表达式尽可能精准, 否则程序运行可能出现异常

within切入点表达式

类级别的切入点表达式

控制粒度：类级别

效率高

```
1 expression="within()"  
2 完整语法：  
3     within(包.类名)  
4 # within(com.baizhi.service.*ServiceImpl)  
5 # within(com.baizhi.service.*)  
6 # within(com.baizhi.service.UserServiceImpl)
```

注意：within的效率高于execution表达式，推荐使用within表达式

13、后置通知和异常通知

```
1 public interface AfterThrowsService {  
2     void save(String name);  
3     String find(String name);  
4 }
```

```
1 public class AfterThrowsServiceImpl implements AfterThrowsService{  
2     @Override  
3     public void save(String name) {  
4         System.out.println("业务层Service save " + name);  
5         // throw new RuntimeException("用来测试异常通知---save方法调用出错");  
6     }  
7     @Override  
8     public String find(String name) {  
9         System.out.println("业务层Service find " + name);  
10        throw new RuntimeException("用来测试异常通知---find方法调用出错");  
11        //return name;  
12    }  
13 }
```

```
1 // 自定义后置和异常通知  
2 public class MyAfterAdvice implements AfterReturningAdvice, ThrowsAdvice {  
3     // 参数1：目标方法返回值  参数2：当前执行方法对象  参数3：执行方法参数  参数4：目标对象  
4     @Override  
5     public void afterReturning(Object o, Method method, Object[] objects, Object  
o1) throws Throwable {
```

```

6     System.out.println("=====开始后置通知");
7     System.out.println("返回值: " + o);
8     System.out.println("方法名: " + method.getName());
9     System.out.println("方法参数: " + objects[0]);
10    System.out.println("目标对象: " + o1 + "\n");
11 }
12 // Spring中不要求必须处理异常通知,但如果实现该接口那么必须要有对应方法
13 // 出现异常时执行通知处理
14 public void afterThrowing(Method method, Object[] args, Object target,
Exception ex) {
15     System.out.println("=====进入异常通知");
16 }
17 }
```

```

1 <!-- 管理Service组件 -->
2 <bean id="afterThrowsService" class="afterthrows.AfterThrowsServiceImpl"/>
3 <!-- 注册通知类 -->
4 <bean id="myAfterAdvice" class="afterthrows.MyAfterAdvice"/>
5 <!-- 配置切面 -->
6 <aop:config>
7   <aop:pointcut id="pc" expression="within(afterthrows.*ServiceImpl)"/>
8   <aop:advisor advice-ref="myAfterAdvice" pointcut-ref="pc"/>
9 </aop:config>
```

```

1 //Test.java
2
3 public static void main(String[] args) {
4     // 启动工厂
5     ApplicationContext context = new
6     ClassPathXmlApplicationContext("afterthrows/spring.xml");
7     // 获取对象
8     AfterThrowsService afterThrowsService = (AfterThrowsService)
9     context.getBean("afterThrowsService");
10    System.out.println(afterThrowsService.getClass());
11    afterThrowsService.save("XXX");
12    afterThrowsService.find("XXXfind");
13 }
```

(第三天课程)

```
1 ## 前两天是Spring核心，从第三天课程开始进入框架整合核心。  
2 ## 框架整合核心就是通过Spring框架去整合现有的Mybatis、Structs2等框架的适用。
```

14、IOC和AOP快速复习

Spring框架核心

IOC and DI

```
1 Inversion of Control 控制反转  
2 Dependency Injection 依赖注入  
3 # 定义：就是将手动通过new关键字创建对象的权力交给Spring，交给工厂由工厂创建对象，Spring不仅要创建对象还要在创建对象的同时维护组件与组件的依赖关系
```

AOP

```
1 Aspects(切面) Oriented(面向) Programming(编程)  
2 # 底层原理：java中动态代理对象的封装  
3 # 定义：通过在程序运行的过程中动态的为项目中某些组件生成动态代理对象，通过在生成的动态代理对象中执行相应的附加操作|额外功能，减少项目中通用代码的冗余问题  
4 # 通知（advice）：除类核心业务方法以外的附加操作都称之为通知 额外功能 事务通知.....性能 日志.....  
5 # 切入点（pointcut）：用来指定项目中哪些组件中哪些方法需要加入 通知（附加操作，额外功能）  
6 # 切面（aspect）： = 通知 + 切入点  
7  
8 举例：  
9 myadvice1 性能  
10 myadvice2 日志  
11 1) 工厂管理通知  
12     myadvice1  
13     myadvice2  
14 2) 配置切面  
15 <aop:config>  
16     <aop:pointcut id="pc" expression="execution()|within()" />  
17     <aop:pointcut id="pc1" expression="execution()|within()" />  
18     ...  
19     <aop:advisor advice-ref="myadvice1" pointcut-ref="pc" />  
20     <aop:advisor advice-ref="myadvice2" pointcut-ref="pc" />  
21     ...  
22 </aop:config>
```

15、Spring中创建复杂对象的方式

Spring如何创建复杂对象

```
1 # 1. spring 项目管理框架
2     定义：Spring项目管理框架 Spring框架主要负责项目中 组件对象的创建、使用、销毁
3         Spring 工厂 容器 ==> 对象 ==> 对象唯一标识
4 # 2. Spring框架如何管理组件对象的创建
5     a) 组件对象 UserDAO UserDaoImpl UserService UserServiceImpl
6     b) 工厂创建
7         <bean id="userDAO" class="com.baizhi.dao.UserDAOImpl"/>
8     c) 获取组件对象
9         UserDao userDao = (UserDAO) new
10        ClassPathXmlApplicationContext("spring.xml").getBean("userDAO");
11        d) 工厂原理
12            Class.forName("com.baizhi.dao.UserDAOImpl").newInstance(); // ==> 调用类中构造
方法进行对象创建
13 # 3. 通过工厂创建简单对象
14     简单对象：可以直接通过new关键字创建的对象，统一称为简单对象
15     工厂创建时：<bean id="" class="XXX" scope="singleton|prototype">
16 # 4. 通过工厂创建复杂对象
17     复杂对象：不能直接通过new关键字进行创建对象 接口类型 Connection | 抽象类类型 Calendar
MessageDigest
18     工厂创建复杂对象：
19     a) 类 implements FactoryBean<>{}
20         public class CalendarFactoryBean implements FactoryBean<Calendar> {
21             // 用来书写复杂对象的创建方式
22             @Override
23             public Calendar getObject() throws Exception {
24                 return Calendar.getInstance();
25             }
26             // 指定创建的复杂对象的类型
27             @Override
28             public Class<?> getObjectType() {
29                 return Calendar.class;
30             }
31             // 用来指定创建的对象模式 true 单例 false 多例
32             @Override
33             public boolean isSingleton() {
34                 return false;
35             }
36         b) 通过工厂配置创建复杂对象
37             <bean id="calendar" class="factorybean.CalendarFactoryBean"/>
38             <bean id="conn" class="factorybean.ConnectionFactoryBean"/>
39         c) 在工厂中获取
40             ApplicationContext context = new
ClassPathXmlApplicationContext("factorybean/spring.xml");
```

```
41     Calendar calendar = (Calendar) context.getBean("calendar");
42     Connection connection = (Connection) context.getBean("conn");
```

16、Spring整合mybatis思路分析

```
1 1. 引入相关依赖
2     spring mybatis mysql ... druid
3 2. 如何整合
4     Spring 项目管理框架 主要是用来负责项目中组件对象的创建、使用、销毁      对象创建
5     mybatis 持久层框架 主要是用来简化原始jdbc技术对数据库访问操作      操作数据库      mybatis
6 中核心对象 --- 数据库
7 # 整合思路：通过Spring框架接管Mybatis框架中核心对象的创建
8 3. Mybatis框架中核心对象是谁
9     SqlSessionFactory---Mybatis中核心对象 读取Mybatis-Config.xml[数据源配置 mapper文件配
10    置]
11    sqlSession
12    dao
13 4. SM整合 Spring整合Mybatis框架
14 # 整合思路：通过Spring框架接管Mybatis中核心的SqlSessionFactory对象的创建
15    a) Spring如何管理SqlSessionFactory对象的创建
16        SqlSessionFactory: 通过查看源码得知是一个接口类型的复杂对象
17        如何创建：
18            is = Resources.getResourceAsStream("mybatis-config.xml");
19            sqlSessionFactory = new SqlSessionFactoryBuilder().build(is);
20        1) SqlSessionFactoryBean implements FactoryBean<SqlSessionFactory> {
21            SqlSessionFactory getObject() {
22                is = Resources.getResourceAsStream("mybatis-config.xml");
23                return new SqlSessionFactoryBuilder().build(is);
24            }
25            Class getClass(){return SqlSessionFactory.class;}
26            boolean isSingleton(){return true;}
27        }
28    2) 工厂管理sqlSessionFactory
29        <bean id="sqlSessionFactory" class="xxx.SqlSessionFactoryBean"/>
30    3) 工厂获取
31        SqlSessionFactory sf = context.getBean("sqlSessionFactory");
32 5. Mybatis官方
33     Mybatis-spring jar包, 封装SqlSessionFactory对象的创建  SqlSessionFactoryBean
34         <dependency>
35             <groupId>org.mybatis</groupId>
36             <artifactId>mybatis-spring</artifactId>
37             <version>2.0.4</version>
38         </dependency>
39 # 注意：mybatis官方提供sqlSessionFactoryBean，不能再使用mybatis-config.xml
40 6. 创建数据源对象 druid
```

```

39    引入依赖
40    <!--druid-->
41      <dependency>
42        <groupId>com.alibaba</groupId>
43        <artifactId>druid</artifactId>
44        <version>1.1.22</version>
45      </dependency>
46
47    <!-- 创建数据源对象 druid C3p0 dbcp-->
48    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
49      <property name="driverClassName" value="com.mysql.jdbc.Driver" />
50      <property name="url" value="jdbc:mysql://localhost:3306/mybatis?
characterEncoding=utf-8"/>
51      <property name="username" value="root" />
52      <property name="password" value="root" />
53    </bean>

```

● spring整合mybatis的编程步骤

- 整合思路：通过Spring接管mybatis中核心对象SqlSessionFactory的创建

SqlSessionFactory复杂对象 ==> SqlSessionFactoryBean创建 ==> Mybatis mybatis-spring
SqlSessionFactoryBean

1. 引入依赖

1. spring
2. mybatis
3. mysql
4. mybatis-spring
5. druid

2. 配置spring.xml

a. 创建sqlSessionFactory

```

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
  <!--依赖数据源对象-->
  <property name="dataSource" ref="dataSource"/>
  <!--依赖mapper文件注册-->
</bean>

```

b. 创建数据源对象

```

<!--创建数据源对象 druid C3p0 dbcp-->
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
  <property name="driverClassName" value="com.mysql.jdbc.Driver" />
  <property name="url" value="jdbc:mysql://localhost:3306/2001" />
  <property name="username" value="root" />
  <property name="password" value="root" />
</bean>

```

1. 创建sqlSessionFactory

```

1 <bean id="sqlSessionFactory"
2   class="org.mybatis.spring.SqlSessionFactoryBean">
3     <!--依赖数据源对象-->
4     <property name="dataSource" ref="dataSource"/>
5     <!-- 依赖mapper文件注册 -->
6   </bean>

```

2. 创建数据源对象

```

1 <!-- 创建数据源对象 druid C3p0 dbcp-->
2 <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
3   <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
4   <property name="url" value="jdbc:mysql://localhost:3306/mybatis?
5   characterEncoding=utf-8"/>
6   <property name="username" value="root"/>
7   <property name="password" value="root"/>
8 </bean>

```

3. 从工厂中获取sqlSessionFactory对象

```

1 ApplicationContext context = new
2 ClassPathXmlApplicationContext("factorybean/spring.xml");
3 SqlSessionFactory sqlSessionFactoryBean = (SqlSessionFactory)
4 context.getBean("sqlSessionFactory");
5 SqlSession sqlSession = sqlSessionFactoryBean.openSession();
6 System.out.println(sqlSession);

```

17、SM整合之DAO编程开发

Spring整合Mybatis编程DAO层开发(整合思路)

1. 项目引入相关依赖

spring mybatis mysql mybatis-spring druid

2. 编写spring.xml

整合：Spring接管mybatis中SqlSessionFactory对象的创建

```

1 <!--spring.xml-->
2 <!-- 创建数据源对象DataSource druid C3p0 dbcp-->
3 <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
4   <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
5   <property name="url" value="jdbc:mysql://localhost:3306/mybatis?
6   characterEncoding=utf-8"/>

```

```
6   <property name="username" value="root"/>
7   <property name="password" value="root"/>
8 </bean>
9
10 <!-- 创建SqlSessionFactory -->
11 <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
12   <!-- 依赖数据源对象-->
13   <property name="dataSource" ref="dataSource"/>
14   <!-- 依赖mapper文件注册 -->
15 </bean>
```

3. 建表

4. 实体类

5. DAO接口

6. 开发mapper

```
1 <select id="findAll" resultType="com.springdao.entity.User">
2   select id,name,age,bir from t_user
3 </select>
```

7. 启动工厂获取SqlSessionFactory

```
1 // 启动工厂
2 ApplicationContext context = new
3 ClassPathXmlApplicationContext("com/springdao/spring.xml");
4 // 获取数据
5 SqlSessionFactory sqlSessionFactory = (SqlSessionFactory)
6 context.getBean("sqlSessionFactory");
7 SqlSession sqlSession = sqlSessionFactory.openSession();
8 System.out.println(sqlSession);
9 UserDAO userDAO = sqlSession.getMapper(UserDAO.class);
10 userDAO.findAll().forEach(user -> System.out.println(user));
```

随着业务逻辑的复杂 DAO会越来越多

mybatis-spring中对

```
SqlSession sqlSession = sqlSessionFactory.openSession();
UserDAO userDAO = sqlSession.getMapper(UserDAO.class);
```

做进一步封装 MapperFactoryBean ——创建DAO对象的一个类

1. 依赖于SqlSessionFactory
2. 依赖创建DAO全限定名

```
1 <!-- spring.xml -->
2
3 <!-- 创建DAO组件类 -->
4 <bean id="userDAO" class="org.mybatis.spring.mapper.MapperFactoryBean">
5   <!-- 注入SqlSessionFactory -->
6   <property name="sqlSessionFactory" ref="sqlSessionFactory"/>
7   <!-- 注入创建DAO接口类型 注入接口的全限定名 包.接口名 -->
8   <property name="mapperInterface" value="com.springdao.dao.UserDAO"/>
9 </bean>
```

```
1 public static void main(String[] args) {
2   ApplicationContext context = new
3   ClassPathXmlApplicationContext("com/springdao/spring.xml");
4   UserDAO userDAO = (UserDAO) context.getBean("userDAO");
5   userDAO.findAll().forEach(user -> System.out.println(user));
}
```

Spring Mybatis整合之DAO层编程步骤

1. 引入依赖

spring mybatis mysql mybatis-spring druid

2. 建表

3. 实体类

4. DAO接口

5. Mapper配置文件

6. 编写spring.xml整合mybatis

6. 编写spring.xml整合mybatis

```
a. 创建DataSource 注入driverClassName url username password
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/2001"/>
    <property name="username" value="root"/>
    <property name="password" value="root"/>
</bean>

b. 创建SqlSessionFactory SqlSessionFactoryBean 注入 dataSource mapperLocations 配置文件位置
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <!-- 依赖数据源-->
    <property name="dataSource" ref="dataSource"/>
    <!-- 注入mapper配置文件-->
    <property name="mapperLocations">
        <array>
            <value>classpath:com/baizhi/mapper/UserDAOMapper.xml</value>
        </array>
    </property>
</bean>

c. 创建DAO MapperFactoryBean 注入sqlSessionFactory DAO接口类型(包.接口名)
<bean id="userDAO" class="org.mybatis.spring.mapper.MapperFactoryBean">
    <!-- 注入sqlSessionFactory-->
    <property name="sqlSessionFactory" ref="sqlSessionFactory"/>
    <!-- 注入创建DAO接口类型 注入接口的全限定名 包.接口名-->
    <property name="mapperInterface" value="com.baizhi.dao.UserDAO"/>
</bean>
```

7. 启动工厂获取DAO调用方法测试

注意：黄色部分为框架源码定义固定写死，不能更换

7. 启动工厂获取DAO调用方法测试

18、SM整合之Service层事务控制

SM整合之Service层事务控制思路分析

1. Mybatis框架中事务控制

SqlSession 提交：sqlSession.commit(); 回滚：sqlSession.rollback();

mybatis对原始jdbc技术封装 ==> **Connection对象** java.sql.Connection commit() rollback()

项目中真正负责数据库事务控制的对象：**Connection对象**

2. 用来实现事务控制的核心对象是

Connection连接对象的commit() rollback()

3. 如何在现有的项目中获取Connection对象

注意：现有项目中DruidDataSource连接池 Connection连接

Connection conn = DruidDataSource().getConnection();

4. 在Spring和Mybatis框架中提供了一个类

DataSourceTransactionManager 数据源事务管理器

作用：在全局创建一个事务管理器，用来统一调度业务层当前线程使用连接对象和DAO层实现连接对象一致
类似JDBCUtils中的ThreadLocal

创建

```
1 <!-- 数据源事务管理 -->
2 <bean id="transactionManager"
3   class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
4   <!-- 注入数据源对象 -->
5   <property name="dataSource" ref="dataSource"/>
6 </bean>
```

5. 给现有的业务层加入事务控制

```
1 public class UserServiceImpl implements UserService{
2     private UserDAO userDAO;
3     //声明事务管理器
4     private PlatformTransactionManager platformTransactionManager;
5
6     public void setPlatformTransactionManager(PlatformTransactionManager
platformTransactionManager) {
7         this.platformTransactionManager = platformTransactionManager;
8     }
9     public void setUserDAO(UserDAO userDAO) {
10         this.userDAO = userDAO;
11     }
12     @Override
13     public List<User> findAll() {
14         return userDAO.findAll();
15     }
16     @Override
17     public void save(User user) {
18         //创建事务配置对象
19         TransactionDefinition transactionDefinition = new
DefaultTransactionDefinition();
20         //获取事务状态
21         TransactionStatus transaction =
platformTransactionManager.getTransaction(transactionDefinition);
22         try {
23             //处理业务
24             user.setId(UUID.randomUUID().toString());
25             //调用业务
26             userDAO.save(user); // spring为了方便执行，在DAO层方法上做了一个小事务，方
便测试DAO。当外部存在事务时，小事务自动失效
27             int i = 1/0;
28             platformTransactionManager.commit(transaction);
29         } catch (Exception e) {
30             e.printStackTrace();
31             platformTransactionManager.rollback(transaction);
32         }
33     }
34 }
```

SM整合之DAO和Service部分开发（一）

1. 引入依赖

```
spring mybatis mysql druid mybatis-spring
```

2. 建表

3. 实体类

4. DAO接口

5. mapper配置文件

6. 编写spring.xml

1. 创建数据源 DataSource 注入driverClassname url username password

```
1 <!-- 创建数据源对象DataSource druid C3p0 dbcp-->
2 <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
3   <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
4   <property name="url" value="jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8"/>
5   <property name="username" value="root"/>
6   <property name="password" value="root"/>
7 </bean>
```

2. 创建SqlSessionFactory 注入dataSource 注入mapperLocations 注入typeAliasesPackage别名设置

```
1 <!-- 创建SqlSessionFactoty -->
2 <bean id="sqlSessionFactory"
3   class="org.mybatis.spring.SqlSessionFactoryBean">
4     <!-- 依赖数据源对象-->
5     <property name="dataSource" ref="dataSource"/>
6     <!-- 注入mapper文件通用方式 -->
7     <property name="mapperLocations"
8       value="classpath:com/springdao/mapper/*.xml"/>
9     <!-- 注入别名相关配置 typeAliasesPackage: 用来给指定包中所有类起别名 默认的别名: 类名/类名首字母小写 -->
10    <property name="typeAliasesPackage" value="com.springdao.entity"/>
11 </bean>
```

3. 创建DAO MapperScannerConfigurer 注入SqlSessionFactory 注入basePackage DAO接口所在包

```

1      <!-- 一次创建项目中所有DAO对象 MapperScannerConfigurer
2          MapperScannerConfigurer:
3              默认创建对象在工厂中唯一标识：接口的首字母小写
4                  UserDAO ===> userDao  Userdao ===> userdao
5                  OrderDAO ==> orderDAO Orderdao ==> orderdao
6                  EmpDAO ==> empDAO
7          -->
8      <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
9          <!-- 注入sqlSessionFactory -->
10         <property name="sqlSessionFactoryBeanName"
11             value="sqlSessionFactory" />
12         <!-- 扫描DAO接口所在的包 -->
13         <property name="basePackage" value="com.springdao.dao" />
14     </bean>

```

7. 测试DAO

8. 开发Service接口

9. 开发Service实现类 注入：依赖DAO组件 依赖事务管理器 在对应增删改查方法中使用事务管理器控制事务

```

1  public class UserServiceImpl implements UserService{
2      private UserDao userDao;
3      //声明事务管理器
4      private PlatformTransactionManager platformTransactionManager;
5
6      public void setPlatformTransactionManager(PlatformTransactionManager
7          platformTransactionManager) {
8          this.platformTransactionManager = platformTransactionManager;
9      }
10     public void setUserDAO(UserDao userDao) {
11         this.userDAO = userDao;
12     }
13     @Override
14     public List<User> findAll() {
15         return userDao.findAll();
16     }
17     @Override
18     public void save(User user) {
19         //创建事务配置对象
20         TransactionDefinition transactionDefinition = new
DefaultTransactionDefinition();
21         //获取事务状态
22         TransactionStatus transaction =
platformTransactionManager.getTransaction(transactionDefinition);
23         try {
24             //处理业务
25             user.setId(UUID.randomUUID().toString());
//调用业务

```

```
26         userDao.save(user); // spring为了方便执行，在DAO层方法上做了一个小事务，方便测试DAO。当外部存在事务时，小事务自动失效
27         int i = 1/0;
28         platformTransactionManager.commit(transaction);
29     } catch (Exception e) {
30         e.printStackTrace();
31         platformTransactionManager.rollback(transaction);
32     }
33 }
34 }
```

10. 编写spring.xml

1. 配置数据源事务管理器DataSourceTransactionManager 注入dataSource

```
1 <!-- 数据源事务管理 -->
2 <bean id="transactionManager"
3   class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
4   <!-- 注入数据源对象 -->
5   <property name="dataSource" ref="dataSource" />
6 </bean>
```

2. 管理Service组件 注入DAO 以及 事务管理器

```
1 <!-- 管理Service组件 -->
2 <bean id="userService" class="com.springdao.service.UserServiceImpl">
3   <property name="userDAO" ref="userDAO" />
4   <property name="platformTransactionManager"
5     ref="transactionManager" />
6 </bean>
```

11. 启动工厂测试

```
1 public static void main(String[] args) {
2     ApplicationContext context = new
3     ClassPathXmlApplicationContext("com/springdao/spring.xml");
4     UserService userService = (UserService) context.getBean("userService");
5     //save
6     userService.save(new User(null, "service", 23, new Date()));
7     //findAll
8     userService.findAll().forEach(user -> System.out.println(user));
9 }
```

SM 整合之 DAO 和 Service 部分开发 (一)

Courier New

1. 引入依赖

```
spring mybatis mysql druid mybatis-spring
```

2. 建表

3. 实体类

4. DAO 接口

5. Mapper 配置文件

6. 编写 spring.xml

- a. 创建数据源 DataSource 注入 driverClassName url username password
- b. 创建 SqlSessionFactory 注入 dataSource 注入 mapperLocations 注入 typeA...Pakage 别名设置
- c. 创建 DAO MapperScannerConfigurer 注入 SqlSessionFactory 注入 basePackage DAO 接口所在包

7. 测试 DAO

8. 开发 Service 接口

9. 开发 Service 实现类 注入：依赖 DAO 组件 依赖事务管理器 在对应增删改查方法中使用事务管理器控制事务

10. 编写 spring.xml

- a. 配置数据源事务管理器 DataSourceTransactionManager 注入 dataSource
- b. 管理 Service 组件 注入 DAO 以及 事务管理器

11. 启动工厂测试

(第四天课程)

19、SM整合编程复习回顾

SM 整合开发

SM

整合思路：通过 spring 接管 mybatis 核心对象 SqlSessionFactory 对象创建

1. 引入依赖

```
spring mybatis mysql mybatis-spring druid
```

2. 建表

3. 实体类

4. DAO 接口

5. Mapper 配置文件

6. 整合配置 spring.xml

- 1. 创建数据源对象 DruidDataSource 注入 driverClassName url username password

2. 创建SqlSessionFactory SqlSessionFactoryBean 注入datasource mapperLocations typeAliasesPackage
3. 创建DAO MapperScannerConfigurer 注入: sqlSessionFactory basePackage
7. 测试DAO
8. service接口
9. service实现 注入DAO 注入事务管理器
10. 配置spring.xml 加入事务管理
 4. 创建事务管理类 **DataSourceTransactionManager** 用来保证同一个线程在调用同一个service方法和DAO方法时使用连接对象一致
注入: dataSource
 5. 管理业务层组件 注入DAO和事务管理器对象

20、SM整合Service层事务控制优化思路分析

Spring中处理事务的两种方式

编程式事务处理

- 定义: 通过在业务层中注入事务管理器对象, 然后通过编码的方式进行事务控制
- 缺点:
 - 代码冗余
 - 不够通用
 - 不便于维护

声明式事务处理【推荐】

- 定义: 通过利用aop切面编程进行事务控制, 并对事务属性在配置文件中完成细粒度配置, 这种方式称之为声明式事务
- **好处: 通用, 减少代码冗余, 更加专注于业务逻辑开发, 无需重复编码**

SM整合开发之Service层事务优化

不使用Spring封装

自定义完成事务管理

1. 开发基于事务通知 环绕通知

```
1 public class UserServiceImpl implements UserService{
2     private UserDAO userDAO;
3     public void setUserDAO(UserDAO userDAO) {
4         this.userDAO = userDAO;
5     }
6 }
```

```

7     @Override
8     public List<User> findAll() {
9         return userDAO.findAll();
10    }
11    @Override
12    public void save(User user) {
13        //处理业务
14        user.setId(UUID.randomUUID().toString());
15        //调用业务
16        userDAO.save(user); // spring为了方便执行，在DAO层方法上做了一个小事务，方便测试DAO。当外部存在事务时，小事务自动失效
17    }
18 }

```

```

1 public class MethodInvokeAdvice implements MethodInterceptor {
2     private PlatformTransactionManager platformTransactionManager;
3     public void setPlatformTransactionManager(PlatformTransactionManager
platformTransactionManager) {
4         this.platformTransactionManager = platformTransactionManager;
5     }
6     @Override
7     public Object invoke(MethodInvocation invocation) throws Throwable {
8         System.out.println("=====进入环绕通知
=====");
9         //创建事务配置对象
10        TransactionDefinition transactionDefinition = new
DefaultTransactionDefinition();
11        //获取事务状态
12        TransactionStatus transaction =
platformTransactionManager.getTransaction(transactionDefinition);
13        try {
14            Object proceed = invocation.proceed(); // 放行
15            System.out.println("OK");
16            platformTransactionManager.commit(transaction); // 提交事务
17            return proceed;
18        } catch (Exception e) {
19            e.printStackTrace();
20            System.out.println("进入catch");
21            platformTransactionManager.rollback(transaction); // 回滚事务
22        }
23        return null;
24    }
25 }

```

2. 配置切面

```

1      <!-- 配置通知对象 -->
2      <bean id="methodInvokeAdvice"
3          class="com.springdao.adviceSpringdao.MethodInvokeAdvice">
4          <property name="platformTransactionManager"
5              ref="transactionManager" />
6      </bean>
7
8      <!-- 配置切面 -->
9      <aop:config>
10         <aop:pointcut id="pc"
11             expression="within(com.springdao.service.*ServiceImpl)" />
12             <aop:advisor advice-ref="methodInvokeAdvice" pointcut-ref="pc" />
13         </aop:config>

```

Spring框架开发声明式事务编程

1. Spring框架提供 tx:advice 标签

作用：1.可以根据事务管理器创建一个基于事务环绕通知对象 2.**tx:advice**标签可以对事务进行细粒度控制

```
<tx:advice id="创建出来通知对象在工厂中唯一标识" transactionManager="事务管理器是谁">
</tx:advice>
```

```

1      <!-- tx:advice标签
2          id: 基于事务管理器创建的环绕通知对象在工厂中的唯一标识
3          作用:
4              1.根据指定的事务管理器在工厂中创建一个事务的环绕通知对象
5              2.对业务层方法进行细粒度事务控制
6          -->
7      <tx:advice id="txAdvice" transaction-manager="transactionManager">
8          <!-- 事务细粒度配置 -->
9          <tx:attributes>
10             <tx:method name="save" />
11             <tx:method name="delete*" />
12             <tx:method name="update" />
13         </tx:attributes>
14     </tx:advice>

```

2. 配置切面

```

1      <!-- 配置切面 -->
2      <aop:config>
3          <aop:pointcut id="pc"
4              expression="within(com.springdao.service.*ServiceImpl)" />
5              <aop:advisor advice-ref="txAdvice" pointcut-ref="pc" />
6      </aop:config>

```

21、SM整合之最终编码

SM整合最终编程步骤

1. 引入依赖 spring mybatis mysql mybatis-spring druid
2. 建表
3. 实体类
4. DAO接口
5. Mapper配置文件
6. Service接口
7. Service实现类
8. 编写SM整合配置 spring.xml

1. 创建数据源对象

```
1 <bean id="dataSource"
2   class="com.alibaba.druid.pool.DruidDataSource">
3     <property name="driverClassName"
4       value="com.mysql.jdbc.Driver"/>
5     <property name="url"
6       value="jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8"/>
7     <property name="username" value="root"/>
8     <property name="password" value="root"/>
9   </bean>
```

2. 创建sqlSessionFactory对象

```
1 <bean id="sqlSessionFactory"
2   class="org.mybatis.spring.SqlSessionFactoryBean">
3     <!-- 依赖数据源对象-->
4     <property name="dataSource" ref="dataSource" />
5     <!-- 注入mapper文件通用方式 -->
6     <property name="mapperLocations"
7       value="classpath:com/springdao/mapper/*.xml"/>
8     <!-- 注入别名相关配置 typeAliasesPackage: 用来给指定包中所有类起别名 默认的别名: 类名/类名首字母小写 -->
9     <property name="typeAliasesPackage"
10    value="com.springdao.entity"/>
11  </bean>
```

3. 创建DAO对象

```
1 <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
2   <!-- 注入sqlSessionFactory -->
3   <property name="sqlSessionFactoryBeanName"
4     value="sqlSessionFactory"/>
5   <!-- 扫描DAO接口所在的包 -->
6   <property name="basePackage" value="com.springdao.dao"/>
7 </bean>
```

4. 创建事务管理器（帮助解决现有业务层调用和DAO调用中的链接安全问题）

```
1 <bean id="transactionManager"
2   class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
3   >
4     <!-- 注入数据源对象 -->
5     <property name="dataSource" ref="dataSource"/>
6   </bean>
```

5. 根据事务管理器创建事务环绕通知 tx:advice

```
1 <tx:advice id="txAdvice" transaction-manager="transactionManager">
2   <!-- 事务细粒度配置 -->
3   <tx:attributes>
4     <tx:method name="save"/>
5     <tx:method name="delete*"/>
6     <tx:method name="update"/>
7   </tx:attributes>
8 </tx:advice>
```

6. 配置事务切面

```
1 <aop:config>
2   <aop:pointcut id="pc"
3     expression="within(com.springdao.service.*ServiceImpl)"/>
4   <aop:advisor advice-ref="txAdvice" pointcut-ref="pc"/>
5 </aop:config>
```

9. 启动工厂 测试Service对象

22、log4j日志使用

SM整合中使用log4j

1. log4j

作用：用来展示项目中的运行日志

日志分类：1.项目根日志（全局日志） 2.项目子日志（指定包级别日志）

日志级别：ERROR（高） > WARN > INFO > DEBUG（低）

2. 如何使用

1. 引入依赖

```
1      <!-- log4j -->
2      <dependency>
3          <groupId>org.apache.logging.log4j</groupId>
4          <artifactId>log4j-api</artifactId>
5          <version>2.10.0</version>
6      </dependency>
7      <dependency>
8          <groupId>org.slf4j</groupId>
9          <artifactId>slf4j-log4j12</artifactId>
10         <version>1.6.1</version>
11     </dependency>
```

2. 引入log4j.properties配置文件

注意：必须防止在resources根目录下，名字

```
1 log4j.rootLogger=ERROR,bb
2 log4j.appender.bb=org.apache.log4j.ConsoleAppender
3 log4j.appender.bb.layout=org.apache.log4j.PatternLayout
4 log4j.appender.bb.layout.conversionPattern=[%p] %d{yyyy-MM-dd} %m%n
5 log4j.logger.com.springmybatis.dao=DEBUG
6 log4j.logger.org.springframework=DEBUG
```

```
1 https://www.cnblogs.com/zhangguangxiang/p/12007924.html
2 #####
3 # 输出到控制台
4 #####
5
6 # log4j.rootLogger日志输出类别和级别：只输出不低于该级别的日志信息DEBUG < INFO
< WARN < ERROR < FATAL
7 # WARN: 日志级别      CONSOLE: 输出位置自己定义的一个名字      logfile: 输出位
置自己定义的一个名字
8 log4j.rootLogger=WARN,CONSOLE,logfile
9 # 配置CONSOLE输出到控制台
10 log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
11 # 配置CONSOLE设置为自定义布局模式
12 log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
13 # 配置CONSOLE日志的输出格式 [frame] 2019-08-22 22:52:12,000 %r耗费毫秒数
%p日志的优先级 %t线程名 %c所属类名通常为全类名 %L代码中的行号 %x线程相关联的NDC
%m日志 %n换行
```

```
14 log4j.appender.CONSOLE.layout.ConversionPattern=[frame] %d{yyyy-MM-dd  
HH:mm:ss,SSS} - %-4r %-5p [%t] %C:%L %x - %m%n  
15 #####  
16 # 输出到日志文件中  
17 #####  
18  
19 # 配置logfile输出到文件中 文件大小到达指定尺寸的时候产生新的日志文件  
20 log4j.appender.logfile=org.apache.log4j.RollingFileAppender  
21 # 保存编码格式  
22 log4j.appender.logfile.Encoding=UTF-8  
23 # 输出文件位置此为项目根目录下的logs文件夹中  
24 log4j.appender.logfile.File=logs/root.log  
25 # 后缀可以是KB,MB,GB达到该大小后创建新的日志文件  
26 log4j.appender.logfile.MaxFileSize=10MB  
27 # 设置滚动文件的最大值3 指可以产生root.log.1、root.log.2、root.log.3和  
28 root.log四个日志文件  
29 log4j.appender.logfile.MaxBackupIndex=3  
30 # 配置logfile为自定义布局模式  
31 log4j.appender.logfile.layout=org.apache.log4j.PatternLayout  
32 log4j.appender.logfile.layout.ConversionPattern=%d{yyyy-MM-dd  
HH:mm:ss} %F %p %m%n  
33 #####  
34 #####  
35 # 对不同的类输出不同的日志文件  
36 #####  
37  
38 # club.bagedate包下的日志单独输出  
39 log4j.logger.club.bagedate=DEBUG,bagedate  
40 # 设置为false该日志信息就不会加入到rootLogger中了  
41 log4j.additivity.club.bagedate=false  
42 # 下面就和上面配置一样了  
43 log4j.appender.bagedate=org.apache.log4j.RollingFileAppender  
44 log4j.appender.bagedate.Encoding=UTF-8  
45 log4j.appender.bagedate.File=logs/bagedate.log  
46 log4j.appender.bagedate.MaxFileSize=10MB  
47 log4j.appender.bagedate.MaxBackupIndex=3  
48 log4j.appender.bagedate.layout=org.apache.log4j.PatternLayout  
49 log4j.appender.bagedate.layout.ConversionPattern=%d{yyyy-MM-dd  
HH:mm:ss} %F %p %m%n  
50
```

23、事务的传播属性

事务传播：在**多个业务层之间相互调用**时传递事务的过程称之为事务传播

将事务对象在业务层之间进行传递的过程

```
1 # propagation: 事务传播属性
2   # REQUIRED (默认) : 需要事务。          如果外层没有事务，则开启新的事务；如果外层存
3   在事务，则融入当前事务。
4   # SUPPORTS:           支持事务。          如果外层没有事务，不会开启新的事务；如果外层
5   存在事务，则融入当前事务。（一般用于查询）
6   # REQUIRES_NEW:        每次开启新的事务。    如果外层存在事务，外层事务挂起，自己开启新的
7   事务执行，执行完成，恢复外层事务继续执行。（一般用于日志）
8   # NOT_SUPPORTED:       不支持事务。        如果外层存在事务，外层事务挂起，自己以非事务
9   方式执行，执行完成，恢复外部事务执行。
10  # NEVER:              不能有事务。        存在事务报错。
11  # MANDATORY:          强制事务。        没有事务报错。
12  # NESTED:             嵌套事务。        事务之间可以嵌套运行。数据库oracle, mysql不
13  支持。
```

24、事务相关属性

```
1 # isolation: 事务隔离级别
2   # DEFAULT:          使用数据库默认的隔离级别【推荐】
3   # READ_UNCOMMITTED: 读未提交。一个客户端读到另一个客户端没有提交的数据，脏读现象。
4   # READ_COMMITTED:   读提交。一个客户端只能读到另一个客户端提交的数据，避免脏读现象。
5   (oracle)
6   # REPEATABLE_READ:  可重复读。主要用来避免不可重复读现象出现，行锁。(mysql)
7   # SERIALIZABLE:     序列化读。主要用来避免幻影读现象出现，表锁。
8   注意：隔离级别越高，查询效率越低。一般推荐使用数据库默认隔离级别。
9
10 # read-only: 事务读写性 (mysql支持 oracle不支持)
11   # true  只读，不能执行增删改操作
12   # false 可读可写
13
14 # rollback-for: 出现什么类型异常回滚。默认出现RuntimeException及其子类异常回滚
15 # no-rollback-for: 出现什么类型异常不回滚 (eg: java.lang.RuntimeException)
16
17 # timeout: 事务超时性
18   # -1: 用不超时
19   # 0 : 不等待
20   # >0: 代表设置超时时间，单位 秒。（正整数）
```

25、Spring整合Struts2开发

26、SSM整合之编程步骤

27、SSM整合之编程实现

28、Spring中的相关注解说明

Spring框架的注解式开发

1. 注解（Annotation）式开发

定义：通过Spring框架提供一系列**相关注解**完成项目中快速开发

注解：Annotation是java中一种特殊类，类似于interface。使用时：@注解类名（属性=参数）

2. spring中注解

前置条件：必须在工厂配置文件中完成注解扫描

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

1. 创建对象相关注解

@Component注解 通用组件创建注解

作用：用来负责对象的创建

修饰范围：只能用在类上

注意：默认使用这个注解在工厂中创建的对象的唯一标识为 类名首字母小写

UserDAOImpl==>userDAOImpl

value属性：用来指定创建的对象在工厂中唯一标识。推荐：存在接口-接口首字母小写 | 不存在-使用默认

@Repository 作用：一般用来创建DAO中组件的注解

@Service 作用：一般用来创建Service中组件的注解

@Controller 作用：一般用来创建Action中组件的注解

2. 控制对象在工厂中创建次数

1. 配置文件修改 `<bean id="" class="" scope="singleton|prototype" />`

2. 注解如何控制：

@Scope

作用：用来指定对象的创建次数，默认单例

修饰范围：只能加在类上

value属性：singleton|prototype

3. 属性注入的相关注解

1. Spring框架提供的 @Autowired 注意：默认根据类型自动注入
2. javaEE本身就有的 @Resource 注意：默认根据名字注入，名称找不到时自动根据类型注入

修饰范围：用在类中的成员变量，或者是类中成员变量的SET方法上

作用：用来完成成员变量的赋值|注入操作

注意：使用注解进行注入时，日后注入的成员变量可以不再提供SET方法

4. 控制事务注解

@Transactional

作用：用来给类中方法加入事务控制，简化配置文件中两段配置。

1、事务通知及事务细粒度配置

2、简化事务切面配置

修饰范围：类|方法上。

1、加在类上：代表类中所有方法加入事务控制

2、加在方法上：代表当前方法加入事务控制

3、类和方法上同时存在：方法优先。**局部优先原则**

注解属性：

1、propagation 用来控制传播属性

2、isolation 用来控制隔离级别

3、timeout 用来设置超时性

4、rollbackFor 用来设置什么异常回滚

5、noRollbackFor 用来设置什么异常不回滚

6、readOnly 用来设置事务读写性

注意：如果想要让@Transactional这个注解生效，配置在配置文件中加入如下配置

```
<!--开启注解式事务生效-->  
<tx:annotation-driven transaction-manager="transactionManager">  
</tx:annotation-driven>
```

29、SSM整合注解式开发

开发步骤

1. 引入依赖

spring mybatis mybatis-spring mysql druid (struts2 struts2-spring-plugin) fastjson log4j

2. SM = spring + mybatis

1. 建表
2. 实体类
3. DAO接口
4. Mapper文件
5. Service接口
6. Service实现类 @Service @Transactional 控制事务 注入DAO
7. 编写spring.xml

1. 开启注解扫描

```
<context:component-scan base-package="包名"/>
```

2. 创建数据源对象

```
1 <bean id="dataSource"
2   class="com.alibaba.druid.pool.DruidDataSource">
3     <property name="driverClassName"
4       value="com.mysql.jdbc.Driver"/>
5     <property name="url"
6       value="jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-
8"/>
7     <property name="username" value="root"/>
8     <property name="password" value="root"/>
9   </bean>
```

3. 创建SqlSessionFactory

```
1 <bean id="sqlSessionFactory"
2   class="org.mybatis.spring.SqlSessionFactoryBean">
3     <!-- 依赖数据源对象 -->
4     <property name="dataSource" ref="dataSource"/>
5     <!-- 注入Mapper文件通用方式 -->
6     <property name="mapperLocations"
7       value="classpath:com/springAnnotation/mapper/*.xml"/>
8     <!-- 注入别名相关配置 typeAliasesPackage: 用来给指定包中所有类起别
名 默认的别名: 类名/类名首字母小写 -->
9     <property name="typeAliasesPackage"
10    value="com.springAnnotation.entity"/>
11  </bean>
```

4. 创建DAO对象

```
1 <bean  
2   class="org.mybatis.spring.mapper.MapperScannerConfigurer">  
3     <!-- 注入SqlSessionFactory -->  
4     <property name="sqlSessionFactoryBeanName"  
5       value="sqlSessionFactory" />  
6     <!-- 扫描DAO接口所在的包 -->  
7     <property name="basePackage"  
8       value="com.springAnnotation.dao" />  
9   </bean>
```

5. 创建事务管理器

```
1 <bean id="transactionManager"  
2   class="org.springframework.jdbc.datasource.DataSourceTransactionMan  
3   ager">  
4     <!-- 注入数据源 -->  
5     <property name="dataSource" ref="dataSource" />  
6   </bean>
```

6. 开启注解式事务生效

```
1 <tx:annotation-driven transaction-manager="transactionManager">  
2 </tx:annotation-driven>
```

3. SS = spring + struts2 (类比Spring+springmvc)

1. 配置web.xml

1. 配置工厂监听器

```
1 <!--配置spring的监听器-->  
2 <listener>  
3   <listener-  
4     class>org.springframework.web.context.ContextLoaderListener</listen  
5     er-class>  
6   </listener>
```

2. 配置工厂配置文件

```
1 <!--配置spring配置文件位置-->  
2 <context-param>  
3   <param-name>contextConfigLocation</param-name>  
4   <param-value>classpath:spring.xml</param-value>  
5 </context-param>
```

3. 配置struts2核心filter

2. 引入struts.xml

3. 开发Action对象，加入@Controller @Scope("prototype")
 4. 配置struts.xml
4. 启动服务器部署测试