

Spring

整理自互联网，仅供学习交流。



目录

[一、 Spring介绍 1](#_Toc2690855)

[1.1 Spring概述 1](#_Toc2690856)

[1.2 整合框架。 1](#_Toc2690857)

[1.3 Spring好处 1](#_Toc2690858)

[1.4 Spring体系结构 2](#_Toc2690859)

[1.5 在项目中的架构 3](#_Toc2690860)

[1.6 spring ioc 的原理 3](#_Toc2690861)

[二、 Spring 快速入门 3](#_Toc2690862)

[2.1 目标： 3](#_Toc2690863)

[2.2 2.1 编写流程 4](#_Toc2690864)

[2.3 2.2 下载官网 4](#_Toc2690865)

[2.4 Spring的核心jar包 4](#_Toc2690866)

[2.5 Spring的入门案例 5](#_Toc2690867)

[2.5.1 第一步： 5](#_Toc2690868)

[2.5.2 第二步： 5](#_Toc2690869)

[2.5.3 第三步： 6](#_Toc2690870)

[2.5.4 第四步： 6](#_Toc2690871)

[2.5.5 第五步： 7](#_Toc2690872)

[2.5.6 总结：【IoC】 8](#_Toc2690873)

[2.5.7 DI解释 8](#_Toc2690874)

[2.5.8 例子： 8](#_Toc2690875)

[三、 Spring的环境搭建 9](#_Toc2690876)

[3.1 spring环境需要maven 坐标 9](#_Toc2690877)

[3.2 需要交给Spring管理注入类 10](#_Toc2690878)

[3.3 spring配置文件 10](#_Toc2690879)

[3.4 测试类 10](#_Toc2690880)

[3.5 验证问题，spring bean id重复会怎么办？ 11](#_Toc2690881)

[四、 Spring 加载过程 11](#_Toc2690882)

[4.1 4.1 spring是单例还是多例？ 11](#_Toc2690883)

[4.2 4.2 spring作用域 11](#_Toc2690884)

[4.2.1 4.2.1singleton 作用域 11](#_Toc2690885)

[4.2.2 4.2.2singleton Prototype 12](#_Toc2690886)

[4.2.3 4.2.3singleton request 12](#_Toc2690887)

[4.2.4 4.2.4singleton session 12](#_Toc2690888)

[五、 SpringIOC 容器 12](#_Toc2690889)

[5.1 5.1 IOC容器创建对象： 12](#_Toc2690890)

[5.2 依赖注入： 13](#_Toc2690891)

[六、 代理模式 15](#_Toc2690892)

[6.1 5.1 概述 15](#_Toc2690893)

[6.2 5.2 静态代理 15](#_Toc2690894)

[6.3 动态代理 16](#_Toc2690895)

[6.4 Cglib代理 16](#_Toc2690896)

[七、 AOP 17](#_Toc2690897)

[7.1 AOP概述 17](#_Toc2690898)

[7.2 概述： 18](#_Toc2690899)

[7.3 注解方式实现AOP编程 18](#_Toc2690900)

[7.4 XML方式实现AOP编程 19](#_Toc2690901)

[7.5 AOP实现原理 20](#_Toc2690902)

[7.6 AOP术语 21](#_Toc2690903)

[7.7 手动代理 21](#_Toc2690904)

[7.7.1 1.4.1 JDK动态代理 21](#_Toc2690905)

[7.7.2 1.4.2 cglib 增强字节码 23](#_Toc2690906)

[7.8 手动实现AOP编程 24](#_Toc2690907)

[7.9 AOP联盟通知类型 25](#_Toc2690908)

[7.10 Spring编写代理半自动 26](#_Toc2690909)

[7.10.1 第一步：导Jar包 26](#_Toc2690910)

[7.10.2 第二步：目标类 26](#_Toc2690911)

[7.10.3 第三步：切面类 26](#_Toc2690912)

[7.10.4 第四步：spring配置 27](#_Toc2690913)

[7.10.5 第五步：测试 27](#_Toc2690914)

[7.11 Spring AOP全自动编程 28](#_Toc2690915)

[7.11.1 第一步：导入jar包 28](#_Toc2690916)

[7.11.2 第二步：Spring 的AOP配置 28](#_Toc2690917)

[7.11.3 第三步：测试 29](#_Toc2690918)

[八、 加载Spring容器的三种方式 29](#_Toc2690919)

[8.1 类路径获得配置文件 29](#_Toc2690920)

[8.2 文件系统路径获得配置文件 29](#_Toc2690921)

[8.3 使用BeanFactory(了解) 30](#_Toc2690922)

[8.4 BeanFactory和ApplicationContext对比 30](#_Toc2690923)

[8.5 装配Bean(xml) 30](#_Toc2690924)

[8.6 实例化Bean的三种方式 30](#_Toc2690925)

[8.6.1 使用构造方法实例化 30](#_Toc2690926)

[8.6.2 使用静态工厂方法实例化 30](#_Toc2690927)

[8.6.3 使用实例工厂方法实例化 30](#_Toc2690928)

[8.7 4.2 bean的作用域 31](#_Toc2690929)

[九、 bean的生命周期【了解】 32](#_Toc2690930)

[9.1 生命周期图 32](#_Toc2690931)

[9.2 生命周期图解释 33](#_Toc2690932)

[9.3 演示 33](#_Toc2690933)

[9.4 手动装配，使用xml配置 35](#_Toc2690934)

[9.4.1 构造方法注入 35](#_Toc2690935)

[9.4.2 属性setter方法注入 36](#_Toc2690936)

[9.4.3 p命名空间注入【了解】 37](#_Toc2690937)

[9.5 6.2 SpEL表达式【了解】 37](#_Toc2690938)

[9.6 集合注入 38](#_Toc2690939)

[9.6.1 List 38](#_Toc2690940)

[9.6.2 Set 39](#_Toc2690941)

[9.6.3 Map 39](#_Toc2690942)

[9.6.4 Properties 40](#_Toc2690943)

[9.6.5 数组 40](#_Toc2690944)

[9.7 6.4 注解注入 41](#_Toc2690945)

[9.7.1 @Component 41](#_Toc2690946)

[9.7.2 @Component("id") 41](#_Toc2690947)

[9.7.3 web开发，提供3个@Component注解衍生注解（功能一样）取代<bean class=""> 41](#_Toc2690948)

[9.7.4 使用案例 41](#_Toc2690949)

[十、 AspectJ 45](#_Toc2690950)

[10.1 2.1 AspectJ简介 45](#_Toc2690951)

[10.2 2.2 切入点表达式【掌握】 46](#_Toc2690952)

[10.2.1 execution() 46](#_Toc2690953)

[10.2.2 within: 47](#_Toc2690954)

[10.2.3 this: 47](#_Toc2690955)

[10.2.4 target: 47](#_Toc2690956)

[10.2.5 args: 47](#_Toc2690957)

[10.2.6 bean(id) 47](#_Toc2690958)

[10.3 2.3 AspectJ 通知类型 47](#_Toc2690959)

[10.4 2.4 Aspect案例讲解【其于xml】 49](#_Toc2690960)

[10.4.1 第一步：导包 49](#_Toc2690961)

[10.4.2 第二步：实现类和切面类 49](#_Toc2690962)

[10.4.3 第三步：spring的xml配置 50](#_Toc2690963)

[10.4.4 第四步：测试 51](#_Toc2690964)

[10.5 2.5 Aspect案例讲解【其于xml】 51](#_Toc2690965)

[10.5.1 第1步：声明使用注解 51](#_Toc2690966)

[10.5.2 第2步：替换service和 切面 bean 52](#_Toc2690967)

[10.5.3 第3步：声明切面 52](#_Toc2690968)

[10.5.4 第4步：声明前置通知 52](#_Toc2690969)

[10.5.5 第5步：声明公共切入点 52](#_Toc2690970)

[10.5.6 第6步：声明后置通知 53](#_Toc2690971)

[10.5.7 第7步：声明环绕通知 53](#_Toc2690972)

[10.5.8 第8步：声明异常通知 53](#_Toc2690973)

[10.5.9 第9步：声明最终通知 53](#_Toc2690974)

[10.5.10 注解总结 53](#_Toc2690975)

[十一、 JdbcTemplate 54](#_Toc2690976)

[11.1 3.1 简介 54](#_Toc2690977)

[11.2 环境搭建 54](#_Toc2690978)

[11.2.1 创建数据库和表 54](#_Toc2690979)

[11.2.2 创建工程导入Jar包 54](#_Toc2690980)

[11.2.3 创建JavaBean,数据模型 55](#_Toc2690981)

[11.3 3.3 API使用【了解】 55](#_Toc2690982)

[11.4 3.4 配置DBCP 55](#_Toc2690983)

[11.5 3.5 配置c3p0 56](#_Toc2690984)

[11.6 3.6 使用JdbcDaoSupport 56](#_Toc2690985)

[11.6.1 dao层 56](#_Toc2690986)

[11.6.2 Beans.xml 57](#_Toc2690987)

[11.6.3 源码分析 57](#_Toc2690988)

[11.7 3.7 配置properties 57](#_Toc2690989)

[11.7.1 在src写个db.properties文件 58](#_Toc2690990)

[11.7.2 beans.xml 58](#_Toc2690991)

[十二、 事务管理 58](#_Toc2690992)

[12.1 事务回顾 58](#_Toc2690993)

[12.1.1 事务简介 58](#_Toc2690994)

[12.1.2 事务特性：ACID 58](#_Toc2690995)

[12.1.3 隔离问题 59](#_Toc2690996)

[12.1.4 隔离级别--解决问题 59](#_Toc2690997)

[12.1.5 mysql 事务操作--简单 60](#_Toc2690998)

[12.1.6 mysql 事务操作--Savepoint 60](#_Toc2690999)

[12.2 Spring事务管理介绍 61](#_Toc2691000)

[12.2.1 Spring提供的事务jar包 61](#_Toc2691001)

[12.2.2 1.2.2 Jar中的三个顶级接口 61](#_Toc2691002)

[12.2.3 1.2.3 PlatformTransactionManager 事务管理器 62](#_Toc2691003)

[12.2.4 1.2.4 TransactionStatus 事务状态 63](#_Toc2691004)

[12.2.5 1.2.5 TransactionDefinition 63](#_Toc2691005)

[12.3 一、事物的概述 64](#_Toc2691006)

[12.3.1 原子性——不可分隔，一致性 64](#_Toc2691007)

[12.3.2 一致性——隔离性，互不影响 64](#_Toc2691008)

[12.3.3 ⑶ 隔离性（Isolation） 65](#_Toc2691009)

[12.3.4 持久性（Durability） 65](#_Toc2691010)

[12.4 程序中事务控制 65](#_Toc2691011)

[12.5 程序中事务控制 65](#_Toc2691012)

[12.5.1 事务控制概述 65](#_Toc2691013)

[12.5.2 3.2编程事物管理 66](#_Toc2691014)

[12.5.3 3.3声明式事务管理 68](#_Toc2691015)

[12.5.4 2.1 环境准备 71](#_Toc2691016)

[12.6 四、传播七种行为 72](#_Toc2691017)

[12.7 案例：转帐 74](#_Toc2691018)

[12.7.1 1.3.1 环境搭建 74](#_Toc2691019)

[12.7.2 1.3.2 手动管理事务【了解】 76](#_Toc2691020)

[12.7.3 1.3.3 工厂bean生成代理：半自动 77](#_Toc2691021)

[12.7.4 1.3.4 基本AOP的事务配置【掌握】 78](#_Toc2691022)

[12.7.5 1.3.5 基本于注解的事务 79](#_Toc2691023)

[12.7.6 1.3.5 整合Junit 79](#_Toc2691024)

[12.8 课程目标: 80](#_Toc2691025)

[十三、 SSH整合 80](#_Toc2691026)

[13.1 2.1 web 整合spring 80](#_Toc2691027)

[13.2 2.2 web整合struts+hibernate+spring 81](#_Toc2691028)

[13.2.1 整合版本 81](#_Toc2691029)

[13.2.2 第一步：jar包整合 81](#_Toc2691030)

[13.2.3 第二步：spring整合hibernate的单元测试 84](#_Toc2691031)

[13.2.4 第三步：spring整合struts 90](#_Toc2691032)

123

# Spring介绍

## Spring概述

**Spring是一个开源框架，Spring是于2003 年兴起的一个轻量级的Java 开发框架**，由Rod Johnson 在其著作Expert One-On-One J2EE Development and Design中阐述的部分理念和原型衍生而来。**它是为了解决企业应用开发的复杂性而创建的**。Spring使用基本的JavaBean来完成以前只可能由EJB完成的事情。然而，Spring的用途不仅限于服务器端的开发。从简单性、可测试性和松耦合的角度而言，任何Java应用都可以从Spring中受益

简单来说，Spring是一个轻量级的控制反转（IoC）和面向切面（AOP）的容器框架。

spring就是把每个bean（实体类）与bean（实体类）的关系全部交给第三方容器进行管理。那么这个容器就是spring，以及对象的生命周期。

spring的核心：ioc（控制反转）和di（依赖注入）、aop（面向切面）

ioc和di 的区别：

di做注入，ioc 是创建对象。

## 整合框架。

Spring提供了一站式解决方案：都是一个原理。把这个集成对象交给spring容器管理。

1） Spring Core spring的核心功能： IOC容器, 解决对象创建及依赖关系

2） Spring Web Spring对web模块的支持。

-🡪 可以与struts整合,让struts的action创建交给spring

-🡪 spring mvc模式

3） Spring DAO Spring 对jdbc操作的支持 【JdbcTemplate模板工具类】

4） Spring ORM spring对orm的支持：

🡪 既可以与hibernate整合，【session】

🡪 也可以使用spring的对hibernate操作的封装

5）Spring AOP 切面编程

6）SpringEE spring 对javaEE其他模块的支持

## Spring好处

**方便解耦，简化开发:**

* **Spring就是一个大工厂，专门负责生成Bean**，可以将所有对象创建和依赖关系维护由Spring管理

**AOP编程的支持:**

* Spring提供面向切面编程，可以方便的实现对程序进行**权限拦截**、运行监控等功能
* **声明式事务的支持:**

只需要通过配置就可以完成对事务的管理，而无需手动编程

**方便程序的测试:**

* Spring对Junit4支持，可以通过注解方便的测试Spring程序

**方便集成各种优秀框架:**

* Spring不排斥各种优秀的开源框架，其内部提供了对各种优秀框架（如：Struts、Hibernate、MyBatis、Quartz等）的支持

**降低JavaEE API的使用难度 Spring:**

* 对JavaEE开发中一些难用的API（JDBC、JavaMail、远程调webservice用等），都提供了封装，使这些API应用难度大大降低

## Spring体系结构

Spring 框架是一个分层架构,,它包含一系列的功能要素并被分为大约**20个模块**。这些模块分为Core Container、Data Access/Integration、Web、AOP（Aspect Oriented Programming)、Instrumentation和测试部分,如下图所示：

|  |
| --- |
|  |

## 在项目中的架构

web层：Struts,SpringMVC

dao层：Hibernate,mybatis

|  |
| --- |
|  |

## spring ioc 的原理

解析xml配置，获取bean class地址，

通过java的反射机制进行实例化，并返回。

# Spring 快速入门

## 目标：

掌握web中集成Spring需要哪些包

掌握IOC是什么

## 2.1 编写流程

* 下载Spring 开发包
* 导入Spring的jar包
* 配置Spring的核心xml文件
* 在程序中读取Spring的配置文件来获取Bean**【Bean其实就是一个new好的对象】**

## 2.2 下载官网

* Spring现在有很多框架，如Spring SpringMVC,Spring-Data,Sprint-Boot

|  |
| --- |
|  |

## Spring的核心jar包

|  |
| --- |
| **spring-core-3.2.2.RELEASE.jar**  包含Spring框架基本的核心工具类，Spring其它组件要都要使用到这个包里的类,是其它组件的基本核心。 |
| **spring-beans-3.2.2.RELEASE.jar**  所有应用都要用到的，它包含访问配置文件、创建和管理bean  以及进行Inversion of Control(IoC) / Dependency Injection(DI)操作相关的所有类 |
| **spring-context-3.2.2.RELEASE.jar**  Spring提供在基础IoC功能上的扩展服务，此外还提供许多企业级服务的支持,  如邮件服务、任务调度、JNDI定位、EJB集成、远程访问、缓存以及各种视图层框架的封装等。 |
| **spring-expression-3.2.2.RELEASE.jar**  Spring表达式语言 |
| **com.springsource.org.apache.commons.logging-1.1.1.jar**  第三方的主要用于处理日志 |

## Spring的入门案例

### 第一步：

* 下载Spring的包

|  |
| --- |
|  |
| spring-framework-3.2.0.RELEASE-dist.zip 【Spring的核心包】  spring-framework-3.0.2.RELEASE-dependencies.zip 【Spring的依赖包】 |

### 第二步：

* 创建Web项目，导Spring的jar包
  + **导入4核心(beans、core、context,expression) + 1个依赖(common-logging.jar)**
  + **注意导入时，不要导入带sources的源文件了**

|  |
| --- |
|  |
|  |

### 第三步：

* 写个简单的Service 并在main方法中调用

|  |  |
| --- | --- |
|  |  |

### 第四步：

* Spring IoC控制反转创建实例

1. 写了个配置文件beans.xml，配置文件的约束可以访问

spring-framework-3.2.0.RC2-docs/reference/html/xsd-config.html网页查看

|  |
| --- |
|  |

2> 离线配置文件约束提示的配置

|  |
| --- |
| **xsd可以在schema目录中找** |
|  |

3> Beans.xml文件配置

|  |
| --- |
|  |

### 第五步：

* 从beans.xml获取bean

|  |
| --- |
|  |

### 总结：【IoC】

1. **IoC Inverse of Control** 反转控制的概念，就是将原本在程序中手动创建UserService对象的控制权，交由Spring框架管理，简单说，就是**创建UserService对象控制权被反转到了Spring框架**

### DI解释

* **Dependency Injection 依赖注入**，在Spring框架负责创建Bean对象时，动态的将依赖对象注入到Bean组件。

### 例子：

在UserService中提供一个get/set的name方法，在beans.xml中通过property去注入

|  |
| --- |
|  |
|  |

**蚂蚁课堂**

# Spring的环境搭建

## spring环境需要maven 坐标

|  |
| --- |
| <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>com.itmayiedu</groupId>  <artifactId>itmayiedu-spring</artifactId>  <version>0.0.1-SNAPSHOT</version>  <dependencies>  <!-- 引入Spring-AOP等相关Jar -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-core</artifactId>  <version>3.0.6.RELEASE</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>  <version>3.0.6.RELEASE</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aop</artifactId>  <version>3.0.6.RELEASE</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-orm</artifactId>  <version>3.0.6.RELEASE</version>  </dependency>  <dependency>  <groupId>org.aspectj</groupId>  <artifactId>aspectjrt</artifactId>  <version>1.6.1</version>  </dependency>  <dependency>  <groupId>aspectj</groupId>  <artifactId>aspectjweaver</artifactId>  <version>1.5.3</version>  </dependency>  <dependency>  <groupId>cglib</groupId>  <artifactId>cglib</artifactId>  <version>2.1\_2</version>  </dependency>  </dependencies>  </project> |

## 需要交给Spring管理注入类

|  |
| --- |
| public class UserEntity {  private String name;  private Integer age;  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public Integer getAge() {  return age;  }  public void setAge(Integer age) {  this.age = age;  }  } |

## spring配置文件

|  |
| --- |
| <?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:p=*"http://www.springframework.org/schema/p"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:aop=*"http://www.springframework.org/schema/aop"*  xsi:schemaLocation=*"*  *http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop.xsd"*>  <bean id=*"userEntity"* class=*"com.itmayiedu.entity.UserEntity"* />  </beans> |

## 测试类

|  |
| --- |
| public class SpringTest {  public static void main(String[] args) {  ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext(  "applicationContext.xml");  UserEntity userEntity = (UserEntity) applicationContext.getBean("userEntity");  System.*out*.println(userEntity);  }  } |

怎么证明springbean是单例的？

通过构造函数

|  |
| --- |
| System.out.println("Spring容器被加载");  //获取bean的id查找对象。  UserEetity userEntity = (UserEetity) applicationContext.getBean("userEntity");  UserEetity userEntity1 = (UserEetity) applicationContext.getBean("userEntity");  System.out.println(userEntity==userEntity1); |

True

默认是单例，会发生什么问题？

线程安全问题。

面试官问：单例在哪里使用的呢。

我们用的spring是单例的。线程不安全。怎么解决

尽量不要使用，共享全局变量。

## 验证问题，spring bean id重复会怎么办？

在xml中不能存在两个相同的beanid，会报错。id值出现多次。

如果同时使用xml和注解。不会报错。

# Spring 加载过程

## spring是单例还是多例？

## spring作用域

### singleton 作用域

当一个bean的 作用域设置为singleton, 那么Spring IOC容器中只会存在一个共享的bean实例，并且所有对bean的请求，只要id与该bean定义相匹配，则只会返回bean的同一实例。换言之，当把 一个bean定义设置为singleton作用域时，Spring IOC容器只会创建该bean定义的唯一实例。这个单一实例会被存储到单例缓存（singleton cache）中，并且所有针对该bean的后续请求和引用都 将返回被缓存的对象实例，这里要注意的是singleton作用域和GOF设计模式中的单例是完全不同的，单例设计模式表示一个ClassLoader中 只有一个class存在，而这里的singleton则表示一个容器对应一个bean，也就是说当一个bean被标识为singleton时 候，spring的IOC容器中只会存在一个该bean。

### 4.2.2singleton Prototype

prototype作用域部署的bean，每一次请求（将其注入到另一个bean中，或者以程序的方式调用容器的 getBean()方法）都会产生一个新的bean实例，相当与一个new的操作，对于prototype作用域的bean，有一点非常重要，那就是Spring不能对一个prototype bean的整个生命周期负责，容器在初始化、配置、装饰或者是装配完一个prototype实例后，将它交给客户端，随后就对该prototype实例不闻不问了。不管何种作用域，容器都会调用所有对象的初始化生命周期回调方法，而对prototype而言，任何配置好的析构生命周期回调方法都将不会被调用。 清除prototype作用域的对象并释放任何prototype bean所持有的昂贵资源，都是客户端代码的职责。（让Spring容器释放被singleton作用域bean占用资源的一种可行方式是，通过使用 bean的后置处理器，该处理器持有要被清除的bean的引用。）

### 4.2.3singleton request

request表示该针对每一次HTTP请求都会产生一个新的bean，同时该bean仅在当前HTTP request内有效，配置实例：

request、session、global session使用的时候首先要在初始化web的web.xml中做如下配置：

### 4.2.4singleton session

session作用域表示该针对每一次HTTP请求都会产生一个新的bean，同时该bean仅在当前HTTP session内有效

# SpringIOC 容器

SpringIOC容器，是spring核心内容。

作用： 创建对象 & 处理对象的依赖关系

## IOC容器创建对象：

创建对象, 有几种方式：

1） 调用无参数构造器

2） 带参数构造器

3） 工厂创建对象

工厂类，静态方法创建对象

工厂类，非静态方法创建对象

|  |
| --- |
| <!-- 无参构造函数 -->  <bean id=*"user1"* class=*"com.itmayiedu.entity.UserEntity"* scope=*"prototype"* />  <!-- 有参构造函数 -->  <bean id=*"user2"* class=*"com.itmayiedu.entity.UserEntity"*>  <constructor-arg name=*"name"* type=*"java.lang.String"*  value=*"张三"*></constructor-arg>  <constructor-arg name=*"age"* type=*"java.lang.Integer"*  value=*"18"*></constructor-arg>  </bean>  <bean id=*"factory"* class=*"com.itmayiedu.entity.ObjectFactory"*></bean>  <!-- 通过实例工厂方法创建 -->  <bean id=*"user3"* factory-bean=*"factory"* factory-method=*"getInstance"*></bean>  <!-- 通过静态工厂方法创建 -->  <bean id=*"user4"* class=*"com.itmayiedu.entity.ObjectFactory"*  factory-method=*"getStaticInstance"*></bean> |

## 依赖注入：

Spring中，如何给对象的属性赋值? 【DI, 依赖注入】

1) 通过构造函数

2) 通过set方法给属性注入值

3) p名称空间

4) 注解

依赖注入的原理：

根据反射生成一个bean，通过setter方法注入。

#### # (常用)Set方法注入值

|  |
| --- |
| <!-- dao instance -->  <bean id=*"userDao"* class=*"cn.itmayiedu.UserDao"*></bean>  <!-- service instance -->  <bean id=*"userService"* class=*"cn.itmayiedu.UserService"*>  <property name=*"userDao"* ref=*"userDao"*></property>  </bean>    <!-- action instance -->  <bean id=*"userAction"* class=*"cn.itmayiedu.UserAction"*>  <property name=*"userService"* ref=*"userService"*></property>  </bean> |

**同学们 思考下，为什么需要生成set方法才能赋值？**

#### # p 名称空间注入属性值 (优化)

|  |
| --- |
| <?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:p=*"http://www.springframework.org/schema/p"*  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"*>    <!-- ###############对象属性赋值############### -->    <!--  给对象属性注入值：  # p 名称空间给对象的属性注入值  (spring3.0以上版本才支持)  -->  <bean id=*"userDao"* class=*"cn.itmayiedu.UserDao"*></bean>    <bean id=*"userService"* class=*"cn.itmayiedu.UserService"* p:userDao-ref=*"userDao"*></bean>    <bean id=*"userAction"* class=*"cn.itmayiedu.UserAction"* p:userService-ref=*"userService"*></bean>      <!-- 传统的注入：  <bean id="user" class="cn.itmayiedu.User" >  <property name="name" value="xxx"></property>  </bean>  -->  <!-- p名称空间优化后 -->  <bean id=*"user"* class=*"cn.itmayiedu.User"* p:name=*"Jack0001"*></bean>    </beans> |

#### # 注解版本使用

注解方式可以简化spring的IOC容器的配置!

使用注解步骤：

1）先引入context名称空间

xmlns:context="http://www.springframework.org/schema/context"

2）开启注解扫描

<context:component-scan base-package="cn.itcast.e\_anno2"></context:component-scan>

3）使用注解

通过注解的方式，把对象加入ioc容器。

创建对象以及处理对象依赖关系，相关的注解：

@Component 指定把一个对象加入IOC容器

@Repository 作用同@Component； 在持久层使用

@Service 作用同@Component； 在业务逻辑层使用

@Controller 作用同@Component； 在控制层使用

@Resource 属性注入

总结：

1） 使用注解，可以简化配置，且可以把对象加入IOC容器,及处理依赖关系(DI)  
 2） 注解可以和XML配置一起使用。

@Resource与@autiwer

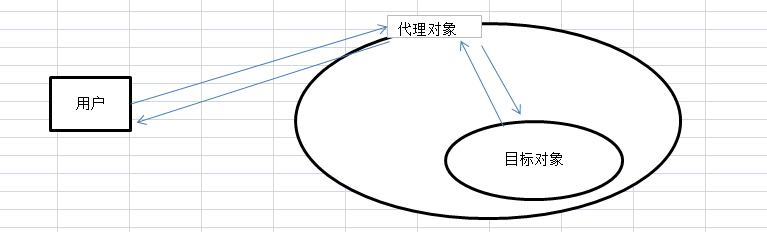
# 代理模式

## 概述

代理（Proxy）是一种设计模式， 提供了对目标对象另外的访问方式；即通过代理访问目标对象。 这样好处： 可以在目标对象实现的基础上，增强额外的功能操作。(扩展目标对象的功能)。

举例：明星（邓紫棋）🡨---经纪人<-------用户

目标 （代理）



代理模式的关键点： 代理对象与目标对象。

## 静态代理

静态代理，

1） 代理对象，要实现与目标对象一样的接口；

2） 举例:

保存用户(模拟)

Dao , 直接保存

DaoProxy, 给保存方法添加事务处理

总结静态代理：

1）可以做到在不修改目标对象的功能前提下，对目标对象功能扩展。

2）缺点：

--》 因为代理对象，需要与目标对象实现一样的接口。所以会有很多代理类，类太多。

--》 一旦接口增加方法，目标对象与代理对象都要维护。

解决：

代理工厂？ 可以使用动态代理。

## 动态代理

动态代理，

1）代理对象，不需要实现接口；

2）代理对象的生成，是利用JDKAPI， 动态的在内存中构建代理对象(需要我们指定创建 代理对象/目标对象 实现的接口的类型；);

3) 动态代理， JDK代理， 接口代理；

JDK中生成代理对象的API：

|-- Proxy

static Object newProxyInstance(

ClassLoader loader, 指定当前目标对象使用类加载器

Class<?>[] interfaces, 目标对象实现的接口的类型

InvocationHandler h 事件处理器

)

动态代理总结：

代理对象不需要实现接口，但是目标对象一定要实现接口；否则不能用动态代理！  
 (class $Proxy0 implements IuserDao)

思考：

有一个目标对象，想要功能扩展，但目标对象没有实现接口，怎样功能扩展？

Class UserDao{}

// 子类的方式

Class subclass extends UserDao{}

以子类的方式实现(cglib代理)

## Cglib代理

Cglib代理，也叫做子类代理。在内存中构建一个子类对象从而实现对目标对象功能的扩展。

* **JDK的动态代理有一个限制，就是使用动态代理的对象必须实现一个或多个接口。如果想代理没有实现接口的类，就可以使用CGLIB实现。**
* **CGLIB是一个强大的高性能的代码生成包，它可以在运行期扩展Java类与实现Java接口。它广泛的被许多AOP的框架使用，例如Spring AOP和dynaop，为他们提供方法的interception（拦截）。**
* **CGLIB包的底层是通过使用一个小而快的字节码处理框架ASM，来转换字节码并生成新的类。不鼓励直接使用ASM，因为它要求你必须对JVM内部结构包括class文件的格式和指令集都很熟悉。**

Cglib子类代理：

1) 需要引入cglib – jar文件， 但是spring的核心包中已经包括了cglib功能，所以直接引入spring-core-3.2.5.jar即可。

2）引入功能包后，就可以在内存中动态构建子类

3）代理的类不能为final， 否则报错。

4） 目标对象的方法如果为final/static, 那么就不会被拦截，即不会执行目标对象额外的业务方法。

在Spring的AOP编程中，

如果加入容器的目标对象有实现接口，用JDK代理；

如果目标对象没有实现接口，用Cglib代理；

# AOP

## AOP概述

**Aop，** aspect object programming 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。

功能： 让关注点代码与业务代码分离！

**关注点,**

重复代码就叫做关注点；

**切面，**

关注点形成的类，就叫切面(类)！

面向切面编程，就是指 对很多功能都有的重复的代码抽取，再在运行的时候网业务方法上动态植入“切面类代码”。

**切入点，**

执行目标对象方法，动态植入切面代码。

可以通过切入点表达式，指定拦截哪些类的哪些方法； 给指定的类在运行的时候植入切面类代码。

1. AOP是OOP（面向对象编程）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。
2. 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。
3. AOP采取横向抽取机制，取代了传统纵向继承体系重复性代码
4. **经典应用：事务管理、性能监视、安全检查、缓存 、日志等【画图】**
5. Spring AOP使用纯Java实现，不需要专门的编译过程和类加载器，在运行期通过代理方式向目标类织入增强代码
6. **AspectJ是一个基于Java语言的AOP框架**，Spring2.0开始，Spring AOP引入对Aspect的支持，AspectJ扩展了Java语言，提供了一个专门的编译器，在编译时提供横向代码的织入

## 概述：

## 注解方式实现AOP编程

步骤：

1） 先引入aop相关jar文件 （aspectj aop优秀组件）

spring-aop-3.2.5.RELEASE.jar 【spring3.2源码】

aopalliance.jar 【spring2.5源码/lib/aopalliance】

aspectjweaver.jar 【spring2.5源码/lib/aspectj】或【aspectj-1.8.2\lib】

aspectjrt.jar 【spring2.5源码/lib/aspectj】或【aspectj-1.8.2\lib】

注意： 用到spring2.5版本的jar文件，如果用jdk1.7可能会有问题。

需要升级aspectj组件，即使用aspectj-1.8.2版本中提供jar文件提供。

2） bean.xml中引入aop名称空间

3） 开启aop注解

<aop:aspectj-autoproxy></aop:aspectj-autoproxy>

4) 使用注解

@Aspect 指定一个类为切面类

@Pointcut("execution(\* com.itmayiedu.service.UserService.add(..))") 指定切入点表达式

@Before("pointCut\_()") 前置通知: 目标方法之前执行

@After("pointCut\_()") 后置通知：目标方法之后执行（始终执行）

@AfterReturning("pointCut\_()") 返回后通知： 执行方法结束前执行(异常不执行)

@AfterThrowing("pointCut\_()") 异常通知: 出现异常时候执行

@Around("pointCut\_()") 环绕通知： 环绕目标方法执行

|  |
| --- |
| @Component  @Aspect  public class Aop {  @Before("execution(\* com.itmayiedu.service.UserService.add(..))")  public void begin() {  System.*out*.println("前置通知");  }  @After("execution(\* com.itmayiedu.service.UserService.add(..))")  public void commit() {  System.*out*.println("后置通知");  }  @AfterReturning("execution(\* com.itmayiedu.service.UserService.add(..))")  public void afterReturning() {  System.*out*.println("运行通知");  }  @AfterThrowing("execution(\* com.itmayiedu.service.UserService.add(..))")  public void afterThrowing() {  System.*out*.println("异常通知");  }  @Around("execution(\* com.itmayiedu.service.UserService.add(..))")  public void around(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {  System.*out*.println("我是环绕通知-前");  proceedingJoinPoint.proceed();  System.*out*.println("我是环绕通知-后");  }  } |

## XML方式实现AOP编程

Xml实现aop编程：

1） 引入jar文件 【aop 相关jar， 4个】

2） 引入aop名称空间

3）aop 配置

\* 配置切面类 （重复执行代码形成的类）

\* aop配置

拦截哪些方法 / 拦截到方法后应用通知代码

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:p=*"http://www.springframework.org/schema/p"*  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"*>    <!-- dao 实例 -->  <bean id=*"userDao"* class=*"com.itmayiedu.UserDao"*></bean>  <bean id=*"orderDao"* class=*"com.itmayiedu.OrderDao"*></bean>    <!-- 切面类 -->  <bean id=*"aop"* class=*"com.itmayiedu.Aop"*></bean>    <!-- Aop配置 -->  <aop:config>  <!-- 定义一个切入点表达式： 拦截哪些方法 -->  <aop:pointcut expression=*"execution(\* com.itmayiedu.\*.\*(..))"* id=*"pt"*/>  <!-- 切面 -->  <aop:aspect ref=*"aop"*>  <!-- 环绕通知 -->  <aop:around method=*"around"* pointcut-ref=*"pt"*/>  <!-- 前置通知： 在目标方法调用前执行 -->  <aop:before method=*"begin"* pointcut-ref=*"pt"*/>  <!-- 后置通知： -->  <aop:after method=*"after"* pointcut-ref=*"pt"*/>  <!-- 返回后通知 -->  <aop:after-returning method=*"afterReturning"* pointcut-ref=*"pt"*/>  <!-- 异常通知 -->  <aop:after-throwing method=*"afterThrowing"* pointcut-ref=*"pt"*/>    </aop:aspect>  </aop:config>  </beans> |

## AOP实现原理

1. aop底层将采用代理机制进行实现。
2. 接口 + 实现类 ：spring采用 jdk 的**动态代理Proxy**。
3. 实现类：spring 采用 **cglib字节码**增强。

## AOP术语

**1.target：**目标类，需要被代理的类。例如：UserService

**2.Joinpoint**(连接点):所谓连接点是指那些可能被拦截到的方法。例如：所有的方法

**3.PointCut** 切入点：已经被增强的连接点。例如：addUser()

4.**advice 通知/增强**，**增强代码**。例如：after、before

**5. Weaving**(织入):是指把增强advice应用到目标对象target来创建新的代理对象proxy的过程.

**6.proxy** 代理类

**7. Aspect**(切面): 是切入点pointcut和通知advice的结合

一个线是一个特殊的面。

一个切入点和一个通知，组成成一个特殊的面。

|  |
| --- |
|  |

## 手动代理

### 1.4.1 JDK动态代理

#### 目标类

|  |  |
| --- | --- |
|  |  |

#### 切面类

|  |
| --- |
|  |

#### 工厂类

|  |
| --- |
|  |

#### 测试类

|  |
| --- |
|  |

### 1.4.2 cglib 增强字节码

* 没有接口，只有实现类。
* 采用字节码增强框架 cglib，在运行时 创建目标类的子类，从而对目标类进行增强。
* 导入jar包：

自己导包（了解）：

核心：hibernate-distribution-3.6.10.Final\lib\bytecode\cglib\cglib-2.2.jar

依赖：struts-2.3.15.3\apps\struts2-blank\WEB-INF\lib\asm-3.3.jar

**spring-core.jar 已经整合以上两个内容**

|  |
| --- |
|  |

#### 工厂类

|  |
| --- |
|  |

#### 测试结果

|  |
| --- |
|  |

## 手动实现AOP编程

AOP 面向切面的编程，

AOP可以实现“业务代码”与“关注点代码”分离

|  |
| --- |
| // 保存一个用户  public void add(User user) {  Session session = null;  Transaction trans = null;  try {  session = HibernateSessionFactoryUtils.getSession(); // 【关注点代码】  trans = session.beginTransaction(); // 【关注点代码】    session.save(user); // 核心业务代码    trans.commit(); //…【关注点代码】  } catch (Exception e) {  e.printStackTrace();  if(trans != null){  trans.rollback(); //..【关注点代码】  }  } finally{  HibernateSessionFactoryUtils.closeSession(session); ////..【关注点代码】  }  } |

分析总结：

关注点代码，就是指重复执行的代码。

**业务代码与关注点代码分离，好处？**

--🡪 关注点代码写一次即可；

-🡪开发者只需要关注核心业务；

-🡪运行时期，执行核心业务代码时候动态植入关注点代码； 【代理】

如何分离？

过程式/对象式/代理模式分离

## AOP联盟通知类型

|  |
| --- |
| AOP联盟为通知Advice定义了org.aopalliance.aop.Advice  Spring按照通知Advice在目标类方法的连接点位置，可以分为5类  •**前置通知** org.springframework.aop.MethodBeforeAdvice  •在目标方法执行前实施增强  •**后置通知** org.springframework.aop.AfterReturningAdvice  •在目标方法执行后实施增强  •**环绕通知** org.aopalliance.intercept.MethodInterceptor  •在目标方法执行前后实施增强  •**异常抛出通知** org.springframework.aop.ThrowsAdvice  •在方法抛出异常后实施增强  •**引介通知** org.springframework.aop.IntroductionInterceptor  在目标类中添加一些新的方法和属性 |

|  |
| --- |
| **环绕通知**，**必须手动执行目标方法**  try{  //前置通知  //执行目标方法  //后置通知  } catch(){  //抛出异常通知  } |

## Spring编写代理半自动

目标：掌握让spring 创建代理对象，从spring容器中手动的获取代理对象。

### 第一步：导Jar包

【核心4+1 、AOP联盟（规范）、spring-aop （实现）】

|  |
| --- |
|  |

### 第二步：目标类

|  |  |
| --- | --- |
|  |  |

### 第三步：切面类

|  |
| --- |
|  |

### 第四步：spring配置

|  |
| --- |
|  |

### 第五步：测试

|  |
| --- |
|  |

## Spring AOP全自动编程

目录：明白什么是全自动织入

### 第一步：导入jar包

spring-framework-3.0.2.RELEASE-dependencies\org.aspectj\com.springsource.org.aspectj.weaver\1.6.8.RELEASE

|  |
| --- |
|  |

### 第二步：Spring 的AOP配置

|  |
| --- |
|  |

### 第三步：测试

|  |
| --- |
|  |

# 加载Spring容器的三种方式

## 类路径获得配置文件

|  |
| --- |
| ApplicationContext context =  **new** ClassPathXmlApplicationContext("com/gyf/spring/demo01/beans.xml"); |

## 文件系统路径获得配置文件

|  |
| --- |
|  |

## 使用BeanFactory(了解)

|  |
| --- |
|  |

## BeanFactory和ApplicationContext对比

* BeanFactory 采取延迟加载，第一次getBean时才会初始化Bean
* ApplicationContext是对BeanFactory扩展，提供了更多功能
* 国际化处理
* 事件传递
* **Bean自动装配**
* 各种不同应用层的Context实现

## 装配Bean(xml)

## 实例化Bean的三种方式

### 使用构造方法实例化

### 使用静态工厂方法实例化

|  |
| --- |
| Bean.xml |
| *<!--装配bean的三种方式，所谓的装配bean就是在xml写一个bean标签-->   <!-- 第一种方式: new 实现类-->* <**bean id="userService1" class="com.gyf.service.UserServiceImpl"**></**bean**>   *<!-- 第二种方式：通过静态工厂方法  spring的版本过低，3.0版本,把jdk改成1.7  -->* <**bean id="userService2" class="com.gyf.service.UserSereviceFactory1" factory-method="createUserService"**></**bean**>   *<!--第三种方式：通过实例工厂方法 -->  <!-- 创建实例factory2 bean-->* <**bean id="factory2" class="com.gyf.service.UserSereviceFactory2"**></**bean**>   <**bean id="userService3" factory-bean="factory2" factory-method="createUserService"**></**bean**> |
| test |
| @Test **public void** test1(){  ApplicationContext context = **new** ClassPathXmlApplicationContext(**"beans3.xml"**);   *//new 对象  //IUserService userService1 = (IUserService) context.getBean("userService1");  //userService1.add();    //静态工厂  //IUserService userService2 = UserSereviceFactory1.createUserService();  //IUserService userService2 = (IUserService) context.getBean("userService2");  //userService2.add();    //实例工厂  //1.创建工厂  // UserSereviceFactory2 factory2 = new UserSereviceFactory2();  //IUserService userService3 = factory2.createUserService();* IUserService userService3 = (IUserService) context.getBean(**"userService3"**);  userService3.add(); } |

### 使用实例工厂方法实例化

## 4.2 bean的作用域

**掌握红色两个常用的即可**

|  |  |
| --- | --- |
| 类别 | 说明 |
| **singleton** | 在Spring IoC容器中仅存在一个Bean实例，Bean以单例方式存在，默认值 |
| **prototype** | 每次从容器中调用Bean时，都返回一个新的实例，即每次调用getBean()时 ，相当于执行new XxxBean() |
| **request** | 每次HTTP请求都会创建一个新的Bean，该作用域仅适用于WebApplicationContext环境 |
| **session** | 同一个HTTP Session 共享一个Bean，不同Session使用不同Bean，仅适用于WebApplicationContext 环境 |
| **globalSession** | 一般用于Portlet应用环境，该作用域仅适用于WebApplicationContext 环境 |

案例：

|  |
| --- |
|  |
|  |

# bean的生命周期【了解】

## 生命周期图

|  |
| --- |
|  |

## 生命周期图解释

1.instantiate bean对象实例化

2.populate properties 封装属性

3.如果Bean实现BeanNameAware 执行 setBeanName

4.如果Bean实现BeanFactoryAware 执行setBeanFactory ，**获取Spring容器**

5.如果存在类实现 BeanPostProcessor（后处理Bean） ，执行postProcessBeforeInitialization

6.如果Bean实现InitializingBean 执行 afterPropertiesSet

7.调用<bean init-method="init"> 指定初始化方法 init

8.如果存在类实现 BeanPostProcessor（处理Bean） ，执行postProcessAfterInitialization

**执行业务处理**

9.如果Bean实现 DisposableBean 执行 destroy

101.调用<bean destroy-method="customerDestroy"> 指定销毁方法 customerDestroy

## 演示

|  |
| --- |
|  |
|  |
|  |
|  |
|  |

依赖注入Bean属性(xml)

## 手动装配，使用xml配置

### 构造方法注入

|  |
| --- |
|  |
|  |

### 属性setter方法注入

**setter方法有两种注入，一般使用第一种直观**

|  |
| --- |
| <bean id=*"user"* class=*"com.gyf.spring.demo04.User"*>  <property name=*"username"* value=*"zhangsan"*></property>  <property name=*"password"* value=*"123456"*></property>  </bean> |
|  |

### p命名空间注入【了解】

|  |
| --- |
|  |
|  |
|  |

## 6.2 SpEL表达式【了解】

Spring 表达式

* **对<property>进行统一编程，所有的内容都使用value**
* **<property name="" value="#{表达式}">**

**#{123}、#{'jack'} ： 数字、字符串**

**#{beanId} ：另一个bean引用**

**#{beanId.propName} ：操作数据**

**#{beanId.toString()} ：执行方法**

**#{T(类).字段|方法} ：静态方法或字段**

|  |
| --- |
|  |
|  |

## 集合注入

集合的注入都是给<property>添加子标签

数组：<array>

List：<list>

Set：<set>

Map：<map> ，map存放k/v 键值对，使用<entry>描述

Properties：<props> <prop key=""></prop> 【】

普通数据：<value>

引用数据：<ref>

### List

|  |
| --- |
|  |
|  |

### Set

|  |
| --- |
|  |
|  |
|  |

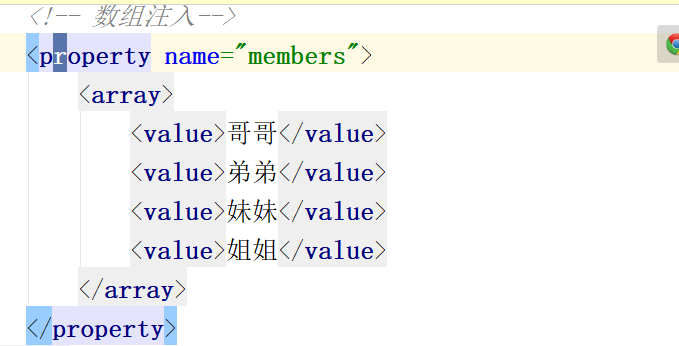
### Map

|  |
| --- |
|  |
|  |

### Properties

|  |
| --- |
|  |
|  |

### 数组



## 6.4 注解注入

* **注解：就是一个类，使用@注解名称**
* **开发中：使用注解 取代 xml配置文件。**

### @Component

@component取代<bean class="">

### @Component("id")

取代 <bean id="" class="">

### web开发，提供3个@Component注解衍生注解（功能一样）取代<bean class="">

@Repository(“名称”)：dao层

@Service(“名称”)：service层

@Controller(“名称”)：web层

@Autowired：自动根据类型注入

@Qualifier(“名称”):指定自动注入的id名称

@Resource(“名称”)

@ PostConstruct 自定义初始化

@ PreDestroy 自定义销毁

### 使用案例

#### 第一个案例

|  |
| --- |
|  |
|  |
|  |
|  |

#### 第二个案例

|  |
| --- |
|  |
|  |

#### 第三个案例

|  |
| --- |
|  |
|  |

#### 第四个案例

|  |
| --- |
|  |
|  |

#### 第五个案例

|  |
| --- |
|  |

#### 第六个案例

|  |
| --- |
|  |
|  |

# AspectJ

## 2.1 AspectJ简介

* AspectJ是一个基于Java语言的AOP框架
* Spring2.0以后新增了对AspectJ切点表达式支持
* @AspectJ 是AspectJ1.5新增功能，通过JDK5注解技术，允许直接在Bean类中定义切面
* **新版本Spring框架，建议使用AspectJ方式来开发AOP**
* **主要用途：自定义开发**

## 2.2 切入点表达式【掌握】

### execution()

用于描述方法 【掌握】

|  |
| --- |
| 语法：execution(修饰符 返回值 包.类.方法名(参数) throws异常) |

**修饰符，一般省略**

public 公共方法

\* 任意

**返回值，不能省略**

void 返回没有值

String 返回值字符串

\* 任意

**包，[省略]**

com.gyf.crm 固定包

com.gyf.crm.\*.service crm包下面子包任意 （例如：com.gyf.crm.staff.service）

com.gyf.crm.. crm包下面的所有子包（含自己）

com.gyf.crm.\*.service.. crm包下面任意子包，固定目录service，service目录任意包

**类，[省略]**

UserServiceImpl 指定类

\*Impl 以Impl结尾

User\* 以User开头

\* 任意

**方法名，不能省略**

addUser 固定方法

add\* 以add开头

\*Do 以Do结尾

\* 任意

**(参数)**

() 无参

(int) 一个整型

(int ,int) 两个

(..) 参数任意

**throws ,可省略，一般不写**。

**案例1：**

**execution(\* com.gyf.crm.\*.service..\*.\*(..))**

**案例2：或**

**<aop:pointcut expression="execution(\* com.gyf.crm.service.\*.\*(..)) ||**

**execution(\* com.gyf.\*Do.\*(..))" id="myPointCut"/>**

### within:

匹配包或子包中的方法(了解)

|  |
| --- |
| within(com.gyf.aop..\*) |

### this:

匹配实现接口的代理对象中的方法(了解)

|  |
| --- |
| this(com.gyf.aop.user.UserDAO) |

### target:

匹配实现接口的目标对象中的方法(了解)

|  |
| --- |
| target(com.gyf.aop.user.UserDAO) |

### args:

匹配参数格式符合标准的方法(了解)

|  |
| --- |
| args(int,int) |

### bean(id)

对指定的bean所有的方法(了解)

|  |
| --- |
| bean('userServiceId') |

## 2.3 AspectJ 通知类型

|  |
| --- |
| aop联盟定义通知类型，具有特性接口，必须实现，从而确定方法名称。  aspectj 通知类型，只定义类型名称，以及方法格式。  个数：6种，**知道5种，掌握1**中。  before:前置通知(应用：各种校验)  在方法执行前执行，如果通知抛出异常，阻止方法运行  afterReturning:后置通知(应用：常规数据处理)  方法正常返回后执行，如果方法中抛出异常，通知无法执行  必须在方法执行后才执行，所以可以获得方法的返回值。  around:环绕通知(应用：十分强大，可以做任何事情)  方法执行前后分别执行，可以阻止方法的执行  **必须手动执行目标方法**  afterThrowing:抛出异常通知(应用：包装异常信息)  方法抛出异常后执行，如果方法没有抛出异常，无法执行  after:最终通知(应用：清理现场)  方法执行完毕后执行，无论方法中是否出现异常 |

|  |  |
| --- | --- |
|  |  |

查看源码

|  |
| --- |
|  |
|  |

## 2.4 Aspect案例讲解【其于xml】

### 第一步：导包

|  |  |
| --- | --- |
|  | AOP联盟规范 |
|  | AOP实现 |
|  | 规范 |
|  | aspectj实现 |

|  |
| --- |
|  |

### 第二步：实现类和切面类

|  |
| --- |
|  |
|  |

### 第三步：spring的xml配置

|  |
| --- |
|  |

### 第四步：测试

|  |
| --- |
|  |

## 2.5 Aspect案例讲解【其于xml】

### 第1步：声明使用注解

|  |
| --- |
|  |

### 第2步：替换service和 切面 bean

|  |
| --- |
|  |
|  |
|  |

### 第3步：声明切面

|  |
| --- |
|  |
|  |

### 第4步：声明前置通知

|  |
| --- |
|  |

### 第5步：声明公共切入点

|  |
| --- |
|  |

### 第6步：声明后置通知

|  |
| --- |
|  |

### 第7步：声明环绕通知

|  |
| --- |
|  |

### 第8步：声明异常通知

|  |
| --- |
|  |

### 第9步：声明最终通知

|  |
| --- |
|  |

### 注解总结

@Aspect 声明切面，修饰切面类，从而获得 通知。

通知

@Before 前置

@AfterReturning 后置

@Around 环绕

@AfterThrowing 抛出异常

@After 最终

切入点

@PointCut ，修饰方法 private void xxx(){} 之后通过“方法名”获得切入点引用

# JdbcTemplate

## 3.1 简介

* jdbcTemplate类似人DBUtils,用于操作Jdbc的工具类，它需要依赖于连接池DataSource(数据源)
* JDBC（Java DataBase Connectivity,java数据库连接）是一种用于执行SQL语句的Java API
* ODBC（Open Database Connectivity，ODBC）开放数据库连接,是微软公司开提供了一组对数据库访问的标准API（应用程序编程接口）
* DBCP（DataBase Connection Pool）数据库连接池，是java数据库连接池的一种，由Apache开发
* C3P0是一个开源的JDBC连接池，它实现了数据源和JNDI绑定，支持JDBC3规范和JDBC2的标准扩展。目前使用它的开源项目有Hibernate，Spring等。

**c3p0与dbcp区别**

dbcp没有自动回收空闲连接的功能

c3p0有自动回收空闲连接功能

## 环境搭建

### 创建数据库和表

|  |
| --- |
| create database spring\_day02;  use spring\_day02;  create table t\_user(  id int primary key auto\_increment,  username varchar(50),  password varchar(32)  );  insert into t\_user(username,password) values('jack','520');  insert into t\_user(username,password) values('rose','521'); |

### 创建工程导入Jar包

|  |
| --- |
|  |

### 创建JavaBean,数据模型

|  |
| --- |
|  |

## 3.3 API使用【了解】

|  |
| --- |
|  |

## 3.4 配置DBCP

|  |
| --- |
| beans.xml |
|  |
| 测试 |
|  |

## 3.5 配置c3p0

|  |
| --- |
|  |

## 3.6 使用JdbcDaoSupport

### dao层

|  |
| --- |
|  |

### Beans.xml

|  |
| --- |
|  |

### 源码分析

**通过数据源创建模板**

|  |
| --- |
|  |

## 3.7 配置properties

目数据库的连接信息配置到一个独立的文件中

### 在src写个db.properties文件

|  |
| --- |
|  |

### beans.xml

|  |
| --- |
|  |

# 事务管理

## 事务回顾

### 事务简介

一组业务ABCD操作，要么全部成功，要么全部不成功。

### 事务特性：ACID

* 原子性：整体 【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】
* 一致性：数据 【一个事务执行之前和执行之后都必须处于一致性状态】
* 隔离性：并发 【对于任意两个并发的事务T1和T2，在事务T1看来，T2要么在T1开始之前就已经结束，要么在T1结束之后才开始，这样每个事务都感觉不到有其他事务在并发地执行。】
* 持久性：结果 【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】

### 隔离问题

* 脏读：一个事务读到另一个事务未提交的内容**【读取未提交内容】**

在该隔离级别，所有事务都可以看到其他未提交事务的执行结果。**本隔离级别很少用于实际应用，因为它的性能也不比其他级别好多少。**

* 不可重复读：一个事务读到另一个事务已提交的内容（insert）**【读取提交内容】**

这是大多数数据库系统的默认隔离级别（但不是MySQL默认的）。它满足了隔离的简单定义：一个事务只能看见已经提交事务所做的改变。

* 虚读（幻读）：一个事务读到另一个事务已提交的内容（update）

这是MySQL的默认事务隔离级别，它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行。不过理论上，这会导致另一个棘手的问题：幻读 （Phantom Read）。简单的说，幻读指当用户读取某一范围的数据行时，另一个事务又在该范围内插入了新行，当用户再读取该范围的数据行时，会发现有新的“幻影” 行。

* Serializable（可串行化）

这是最高的隔离级别，它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之，它是在每个读的数据行上加上共享锁。在这个级别，可能导致大量的超时现象和锁竞争。

### 隔离级别--解决问题

* read uncommittd，读未提交。存在3个问题。
* read committed，读已提交。解决：脏读。存在2个问题。
* repeatable read ，可重复读。解决：脏读、不可重复读。存在1个问题。
* serializable，串行化。单事务。没有问题。

|  |
| --- |
|  |

### mysql 事务操作--简单

|  |
| --- |
| ABCD 一个事务  Connection conn = null;  try{  //1 获得连接  conn = ...;  //2 开启事务  conn.setAutoCommit(false);  A  B  C  D  //3 提交事务  conn.commit();  } catche(){  //4 回滚事务  conn.rollback();  } |

### mysql 事务操作--Savepoint

|  |
| --- |
| 需求：AB（必须），CD（可选）  Connection conn = null;  Savepoint savepoint = null; //保存点，记录操作的当前位置，之后可以回滚到指定的位置。（可以回滚一部分）  try{  //1 获得连接  conn = ...;  //2 开启事务  conn.setAutoCommit(false);  A  B  savepoint = conn.setSavepoint();  C  D  //3 提交事务  conn.commit();  } catche(){  if(savepoint != null){ //CD异常  // 回滚到CD之前  conn.rollback(savepoint);  // 提交AB  conn.commit();  } else{ //AB异常  // 回滚AB  conn.rollback();  }  } |

## Spring事务管理介绍

### Spring提供的事务jar包

transaction = tx

|  |
| --- |
|  |

### 1.2.2 Jar中的三个顶级接口

|  |
| --- |
|  |
| **PlatformTransactionManager：**  平台事务管理器，spring要管理事务，必须使用事务管理器,进行事务配置时，必须**配置事务管理器** |
| **TransactionDefinition：**  事务详情（事务定义、事务属性），spring用于确定事务具体详情，  例如：隔离级别、是否只读、超时时间 等  进行事务配置时，**必须配置详情**。spring将配置项封装到该对象实例。 |
| **TransactionStatus：**  事务状态，spring用于记录当前事务运行状态。例如：是否有保存点，事务是否完成。  spring底层根据状态进行相应操作。 |

### 1.2.3 PlatformTransactionManager 事务管理器

#### 先导入两个包

|  |
| --- |
|  |

#### 常用的两个事务管理器

|  |
| --- |
|  |

### 1.2.4 TransactionStatus 事务状态

|  |
| --- |
|  |

### 1.2.5 TransactionDefinition

|  |
| --- |
|  |

**传播行为：在两个业务之间如何共享事务**

|  |  |
| --- | --- |
| **PROPAGATION\_REQUIRED**  required , 必须 【默认值】 | 支持当前事务，A如果有事务，B将使用该事务。  如果A没有事务，B将创建一个新的事务。 |
| **PROPAGATION\_SUPPORTS**  supports ，支持 | 支持当前事务，A如果有事务，B将使用该事务。  如果A没有事务，B将以非事务执行。 |
| **PROPAGATION\_MANDATORY**  mandatory ，强制 | 支持当前事务，A如果有事务，B将使用该事务。  如果A没有事务，B将抛异常。 |
| **PROPAGATION\_REQUIRES\_NEW**  requires\_new ，必须新的 | 如果A有事务，将A的事务挂起，B创建一个新的事务  如果A没有事务，B创建一个新的事务 |
| **PROPAGATION\_NOT\_SUPPORTED**  not\_supported ,不支持 | 如果A有事务，将A的事务挂起，B将以非事务执行  如果A没有事务，B将以非事务执行 |
| **PROPAGATION\_NEVER**  never，从不 | 如果A有事务，B将抛异常  如果A没有事务，B将以非事务执行 |
| **PROPAGATION\_NESTED**  nested ，嵌套 | A和B底层采用保存点机制，形成嵌套事务。 |

掌握：PROPAGATION\_REQUIRED、PROPAGATION\_REQUIRES\_NEW、PROPAGATION\_NESTED

## 事物的概述

### 原子性——不可分隔，一致性

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要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

### 一致性——隔离性，互不影响

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。

拿转账来说，假设用户A和用户B两者的钱加起来一共是5000，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5000，这就是事务的一致性。

### ⑶ 隔离性（Isolation）

隔离性是当多个用户并发访问数据库时，比如操作同一张表时，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。

即要达到这么一种效果：对于任意两个并发的事务T1和T2，在事务T1看来，T2要么在T1开始之前就已经结束，要么在T1结束之后才开始，这样每个事务都感觉不到有其他事务在并发地执行。

关于事务的隔离性数据库提供了多种隔离级别，稍后会介绍到。

### 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，就可以认定事务以及正确提交，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，否则就会造成我们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。

## 程序中事务控制

## 程序中事务控制

### 事务控制概述

**编程式事务控制**

自己手动控制事务，就叫做编程式事务控制。

Jdbc代码：

Conn.setAutoCommite(false); // 设置手动控制事务

Hibernate代码：

Session.beginTransaction(); // 开启一个事务

【细粒度的事务控制： 可以对指定的方法、指定的方法的某几行添加事务控制】

(比较灵活，但开发起来比较繁琐： 每次都要开启、提交、回滚.)

**声明式事务控制**

Spring提供了对事务的管理, 这个就叫声明式事务管理。

Spring提供了对事务控制的实现。用户如果想用Spring的声明式事务管理，只需要在配置文件中配置即可； 不想使用时直接移除配置。这个实现了对事务控制的最大程度的解耦。

Spring声明式事务管理，核心实现就是基于Aop。

【粗粒度的事务控制： 只能给整个方法应用事务，不可以对方法的某几行应用事务。】

(因为aop拦截的是方法。)

Spring声明式事务管理器类：

Jdbc技术：**DataSourceTransactionManager**

Hibernate技术：**HibernateTransactionManager**

### 编程事物管理

#### 3.2.1手动管理事物

|  |
| --- |
| 1. UserDao.java |
| @Repository  public class UserDao {  @Autowired  private JdbcTemplate jdbcTemplate;  public void add(String name, Integer age) {  String sql = "INSERT INTO users(NAME, age) VALUES(?,?);";  int update = jdbcTemplate.update(sql, name, age);  System.*out*.println("updateResult:" + update);  }  } |
| 2. UserService |
| @Service  public class UserService {  @Autowired  private UserDao userDao;  public void add() {  userDao.add("lisi", 18);  int i=1/0;//可能会发生异常  userDao.add("yushengjun", 19);  }  } |
| 3. App 测试类 |
| public class UserTest {  public static void main(String[] args) {  ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext("bean.xml");  UserService userService = (UserService) applicationContext.getBean("userService");  userService.add();  }  } |
| 4.手动事物管理类 |
| @Component  **public** **class** TransactionUtils {  // 事物管理器  @Autowired  **private** DataSourceTransactionManager dataSourceTransactionManager;  **public** TransactionStatus begin() {  TransactionStatus transaction = dataSourceTransactionManager.getTransaction(**new** DefaultTransactionDefinition());  **return** transaction;  }  **public** **void** commit(TransactionStatus transaction) {  dataSourceTransactionManager.commit(transaction);  }  **public** **void** rollback(TransactionStatus transaction) {  dataSourceTransactionManager.rollback(transaction);  }  } |
| 5. bean.xml (Spring务管理配置) |
| <?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:p=*"http://www.springframework.org/schema/p"*  xmlns:context=*"http://www.springframework.org/schema/context"*  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/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*  *http://www.springframework.org/schema/tx*  *http://www.springframework.org/schema/tx/spring-tx.xsd"*>  <!-- 开启注解 -->  <context:component-scan base-package=*"com.itmayiedu"*></context:component-scan>  <!-- 1. 数据源对象: C3P0连接池 -->  <bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>  <property name=*"driverClass"* value=*"com.mysql.jdbc.Driver"*></property>  <property name=*"jdbcUrl"* value=*"jdbc:mysql://localhost:3306/test"*></property>  <property name=*"user"* value=*"root"*></property>  <property name=*"password"* value=*"root"*></property>  </bean>  <!-- 2. JdbcTemplate工具类实例 -->  <bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>  <property name=*"dataSource"* ref=*"dataSource"*></property>  </bean>  <!-- 配置事物 -->  <bean id=*"DataSourceTransactionManager"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>  <property name=*"dataSource"* ref=*"dataSource"*></property>  </bean>  </beans> |

### 声明式事务管理

方法结束后提交。当抛出异常的时候才会回滚。

#### XML方式实现

|  |
| --- |
| 1. UserDao.java |
| @Repository  public class UserDao {  @Autowired  private JdbcTemplate jdbcTemplate;  public void add(String name, Integer age) {  String sql = "INSERT INTO users(NAME, age) VALUES(?,?);";  int update = jdbcTemplate.update(sql, name, age);  System.*out*.println("updateResult:" + update);  }  } |
| 2. UserService |
| @Service  public class UserService {  @Autowired  private UserDao userDao;  public void add() {  userDao.add("lisi", 18);  int i=1/0;//可能会发生异常  userDao.add("yushengjun", 19);  }  } |
| 3. App 测试类 |
| **public** **class** UserTest {  **public** **static** **void** main(String[] args) {  ClassPathXmlApplicationContext applicationContext = **new** ClassPathXmlApplicationContext("bean.xml");  UserService userService = (UserService) applicationContext.getBean("userService");  userService.add();  }  } |
| 4.手动事物管理类 |
| @Component  **public** **class** TransactionUtils {  // 事物管理器  @Autowired  **private** DataSourceTransactionManager dataSourceTransactionManager;  **public** TransactionStatus begin() {  TransactionStatus transaction = dataSourceTransactionManager.getTransaction(**new** DefaultTransactionDefinition());  **return** transaction;  }  **public** **void** commit(TransactionStatus transaction) {  dataSourceTransactionManager.commit(transaction);  }  **public** **void** rollback(TransactionStatus transaction) {  dataSourceTransactionManager.rollback(transaction);  }  } |
| 5. bean.xml (Spring务管理配置) |
| <!-- 开启注解 -->  <context:component-scan base-package=*"com.itmayiedu"*></context:component-scan>  <!-- 1. 数据源对象: C3P0连接池 -->  <bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>  <property name=*"driverClass"* value=*"com.mysql.jdbc.Driver"*></property>  <property name=*"jdbcUrl"* value=*"jdbc:mysql://localhost:3306/test"*></property>  <property name=*"user"* value=*"root"*></property>  <property name=*"password"* value=*"root"*></property>  </bean>  <!-- 2. JdbcTemplate工具类实例 -->  <bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>  <property name=*"dataSource"* ref=*"dataSource"*></property>  </bean>  <!-- 配置事物 -->  <bean id=*"dataSourceTransactionManager"*  class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>  <property name=*"dataSource"* ref=*"dataSource"*></property>  </bean>  <!—配置事物增强-->  <tx:advice id=*"txAdvice"* transaction-manager=*"dataSourceTransactionManager"*>  <tx:attributes>  <tx:method name=*"get\*"* read-only=*"true"* />  <tx:method name=*"find\*"* read-only=*"true"* />  <tx:method name=*"\*"* read-only=*"false"* />  </tx:attributes>  </tx:advice>  <!-- Aop配置： 拦截哪些方法(切入点表表达式) + 应用上面的事务增强配置 -->  <aop:config>  <aop:pointcut expression=*"execution(\* com.itmayiedu.service.\*.\*(..))"*  id=*"pt"* />  <aop:advisor advice-ref=*"txAdvice"* pointcut-ref=*"pt"* />  </aop:config> |

##### 使用事物注意事项

事物是程序运行如果没有错误,会自动提交事物,如果程序运行发生异常,则会自动回滚。

如果使用了try捕获异常时.一定要在catch里面手动回滚。

事物手动回滚代码

TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();

#### 3.3.2 注解方式实现

使用注解实现Spring的声明式事务管理，更加简单！

步骤：

1） 必须引入Aop相关的jar文件

2） bean.xml中指定注解方式实现声明式事务管理以及应用的事务管理器类

3）在需要添加事务控制的地方，写上: @Transactional

@Transactional注解：

1）应用事务的注解

2）定义到方法上： 当前方法应用spring的声明式事务

3）定义到类上： 当前类的所有的方法都应用Spring声明式事务管理;

4）定义到父类上： 当执行父类的方法时候应用事务。

|  |
| --- |
| Bean.xm |
| <?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:p=*"http://www.springframework.org/schema/p"*  xmlns:context=*"http://www.springframework.org/schema/context"*  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/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*  *http://www.springframework.org/schema/tx*  *http://www.springframework.org/schema/tx/spring-tx.xsd"*>  <!-- 开启注解 -->  <context:component-scan base-package=*"com.itmayiedu"*></context:component-scan>  <!-- 1. 数据源对象: C3P0连接池 -->  <bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>  <property name=*"driverClass"* value=*"com.mysql.jdbc.Driver"*></property>  <property name=*"jdbcUrl"* value=*"jdbc:mysql://localhost:3306/test"*></property>  <property name=*"user"* value=*"root"*></property>  <property name=*"password"* value=*"root"*></property>  </bean>  <!-- 2. JdbcTemplate工具类实例 -->  <bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>  <property name=*"dataSource"* ref=*"dataSource"*></property>  </bean>  <!-- 配置事物 -->  <bean id=*"dataSourceTransactionManager"*  class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>  <property name=*"dataSource"* ref=*"dataSource"*></property>  </bean>  <!-- 开启注解事物 -->  <tx:annotation-driven transaction-manager=*"dataSourceTransactionManager"*/>  </beans> |
| UserService |
| **@Transactional**  **public void add() {**  **try {**  **userDao.add("lisi", 18);**  **int i = 1 / 0;**  **userDao.add("yushengjun", 19);**  **} catch (Exception e) {**  **TransactionAspectSupport.*currentTransactionStatus*().setRollbackOnly();**  **}**    **}** |
|  |

### 环境准备

用户访问—》C--》 Service---》Dao

一个业务的成功： 调用的service是执行成功的，意味着service中调用的所有的dao是执行成功的。  **事务应该在Service层统一控制。**

1）没有应用事务的代码：

2）模拟：

在service中调用2次dao， 希望其中一个dao执行失败，整个操作要回滚。

开发步骤：

1. 后台环境准备

数据库、表/entity/dao/service

2. dao 的实现用JdbcTemplate

3. 对象创建都有Spring容器完成

|  |
| --- |
|  |

## 传播七种行为

Spring中事务的定义：

Propagation（key属性确定代理应该给哪个方法增加事务行为。这样的属性最重要的部份是传播行为。）有以下选项可供使用：

PROPAGATION\_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。

PROPAGATION\_SUPPORTS--支持当前事务，如果当前没有事务，就以非事务方式执行。

PROPAGATION\_MANDATORY--支持当前事务，如果当前没有事务，就抛出异常。

PROPAGATION\_REQUIRES\_NEW--新建事务，如果当前存在事务，把当前事务挂起。

PROPAGATION\_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。

PROPAGATION\_NEVER--以非事务方式执行，如果当前存在事务，则抛出异常。

|  |
| --- |
| @Transactional(  readOnly = false, // 读写事务  timeout = -1, // 事务的超时时间不限制  noRollbackFor = ArithmeticException.class, // 遇到数学异常不回滚  isolation = Isolation.*DEFAULT*, // 事务的隔离级别，数据库的默认  propagation = Propagation.*REQUIRED* // 事务的传播行为  ) |

事务传播行为:

Propagation.REQUIRED

指定当前的方法必须在事务的环境下执行；

如果当前运行的方法，已经存在事务， 就会加入当前的事务；

Propagation.REQUIRED\_NEW

指定当前的方法必须在事务的环境下执行；

如果当前运行的方法，已经存在事务： 事务会挂起； 会始终开启一个新的事务，执行完后； 刚才挂起的事务才继续运行。

举例：

Class Log{

Propagation.REQUIRED

insertLog();

}

Propagation.REQUIRED

Void saveDept(){

insertLog(); // 加入当前事务

**.. 异常, 会回滚**

saveDept();

}

Class Log{

Propagation.REQUIRED\_NEW

insertLog();

}

Propagation.REQUIRED

Void saveDept(){

insertLog(); // 始终开启事务

.. 异常, 日志不会回滚

saveDept();

}

测试步骤：

1）日志表Log\_

2）LogService.java

insertLog();

## 案例：转帐

### 环境搭建

#### 创建数据库表

|  |
| --- |
| create database spring\_day3;  use spring\_day3;  create table account(  id int primary key auto\_increment,  username varchar(50),  money int  );  insert into account(username,money) values('jack','10000');  insert into account(username,money) values('rose','10000'); |

#### 导入jar包

* 核心：4+1
* aop ： 4 (aop联盟、spring aop、aspectj规范、spring aspect)
* 数据库：2 （jdbc/tx）
* 驱动：mysql
* 连接池：c3p0

|  |
| --- |
|  |

#### Dao层

|  |
| --- |
|  |

#### Service层

|  |
| --- |
|  |

#### Spring的配置

配置c3p0数据源->dao -> service

|  |
| --- |
|  |

#### 测试转帐

|  |
| --- |
|  |

### 手动管理事务【了解】

**spring底层使用 TransactionTemplate 事务模板进行操作。**

**操作**

1.service 需要获得 TransactionTemplate

2.spring 配置模板，并注入给service

3.模板需要注入事务管理器

4.配置事务管理器：DataSourceTransactionManager ，需要注入DataSource

**了解底层即可，因为以后都是通过aop来配置事务**

#### 修改Service

|  |
| --- |
|  |

#### 修改spring的配置文件

|  |
| --- |
|  |

### 工厂bean生成代理：半自动

Spring提供 管理事务的代理工厂bean **TransactionProxyFactoryBean**

#### 修改spring配置文件

|  |
| --- |
| transactionAttributes:事务详情  prop.key ：确定哪些方法使用当前事务配置  prop.text:用于配置事务详情  格式：PROPAGATION,ISOLATION,readOnly,-Exception,+Exception  传播行为 隔离级别 是否只读 异常回滚 异常提交 |

|  |
| --- |
|  |

#### 测试：

|  |
| --- |
|  |

### 基本AOP的事务配置【掌握】

#### Spring的配置文件

|  |
| --- |
|  |
|  |

#### 测试同上

### 基本于注解的事务

#### spring配置

|  |
| --- |
|  |

|  |
| --- |
|  |

### 1.3.5 整合Junit

目的：少写一些代码

导入一个spring-test包

|  |
| --- |
|  |

**Spring事物与传播行为**

## 课程目标:

Spring事物概念、理解事物的传播行为

# SSH整合

## 2.1 web 整合spring

配置tomcat加载spring的配置文件

第一步：需要添加

第二步：在web.xml配置spring的监听

|  |
| --- |
|  |
| 出现下面的错误是配置文件加载位置不对，在web.xml改成classpath目录下 |
|  |
|  |

第三步：创建Servlet获取Spring的应用上下文件ApplicationContext

|  |
| --- |
|  |

## 2.2 web整合struts+hibernate+spring

### 整合版本

**struts-2.3.33-all**

**spring-framework-3.0.2.RELEASE**

**hibernate-distribution-3.6.10.Final-dist**

### 第一步：jar包整合

#### Struts的jar包

|  |
| --- |
|  |

#### Spring的jar包

|  |
| --- |
| 基础：4+1 ， beans、core、context、expression ， commons-logging (struts已经导入)  AOP：aop联盟(aopalliance)、spring aop 、aspect规范（aspect.weaver）、spring aspect  db：jdbc、tx  测试：test  web开发：spring web  驱动：mysql  连接池：c3p0  整合hibernate：spring orm |

#### Hibernate的jar包

|  |
| --- |
| 核心包 |
|  |
| required包下的介绍    **jpa用于注解开发@Entity @Id** |

#### 整合log4j

导入 log4j...jar (struts已经导入)



整合（过渡）：slf4j-log4j12-1.7.2.jar

|  |
| --- |
|  |

#### 二级缓存

Commons-loggin.jar已经存在

|  |
| --- |
|  |

#### 整合包

spring整合hibernate： spring orm

struts 整合spring：struts2-spring-plugin-2.3.15.3.jar

#### 删除重复jar包



### 第二步：spring整合hibernate的单元测试

#### 创建表

|  |
| --- |
| create table t\_user(  id int primary key auto\_increment,  username varchar(50),  password varchar(32),  age int  ); |

#### po类和映射文件

|  |
| --- |
| **public** **class** User {  **private** Integer id;  **private** String username;  **private** String password;  **private** Integer age; |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">  <hibernate-mapping>  <class name=*"com.gyf.borrowsys.domain.User"* table=*"t\_user"*>  <id name=*"id"*>  <generator class=*"native"*></generator>  </id>    <property name=*"username"*></property>  <property name=*"password"*></property>  <property name=*"age"*></property>  </class>  </hibernate-mapping> |

#### dao

|  |
| --- |
|  |

#### Service

|  |
| --- |
|  |

#### hibernate.cfg.xml

|  |
| --- |
| <!DOCTYPE hibernate-configuration PUBLIC  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">  <hibernate-configuration>  <session-factory>  <!-- 1、配置数据库连接的4个参数 -->  <property name=*"hibernate.connection.driver\_class"*>com.mysql.jdbc.Driver</property>  <property name=*"hibernate.connection.url"*>jdbc:mysql://localhost:3306/web\_ssh</property>  <property name=*"hibernate.connection.username"*>root</property>  <property name=*"hibernate.connection.password"*>123456</property>    <!-- 2、是否显示sql语句 -->  <property name=*"show\_sql"*>true</property>    <!-- 3、是否格式化sql语句 -->  <property name=*"format\_sql"*>true</property>    <!-- 4、Hiberante映射与DDl语句的策略 update【常用】: 如果数据库没有表，会创建表  hibernate.-->  <property name=*"hibernate.hbm2ddl.auto"*>update</property>    <property name=*"hibernate.dialect"*>org.hibernate.dialect.MySQL5Dialect</property>    <!-- 配置JavaBean与表的映射文件 -->  <mapping resource=*"com/gyf/borrowsys/domain/User.hbm.xml"*/>    </session-factory>  </hibernate-configuration> |

#### applicationContext.xml

|  |
| --- |
|  |

#### 单元测试

|  |
| --- |
|  |

#### 配置Hibrenate的事务

|  |
| --- |
|  |

#### 简化：去除hibernate.cfg.xml文件

|  |
| --- |
| <!-- dataSource -->  <bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>  <property name=*"driverClass"* value=*"com.mysql.jdbc.Driver"*></property>  <property name=*"jdbcUrl"* value=*"jdbc:mysql:///web\_ssh"*></property>  <property name=*"user"* value=*"root"*></property>  <property name=*"password"* value=*"123456"*></property>  </bean>    <!-- sessionFactory -->  <bean id=*"sessionFactory"* class=*"org.springframework.orm.hibernate3.LocalSessionFactoryBean"*>  <!--configLocation:hibernate配置文件的位置 -->  <!-- <property name="configLocation" value="classpath:hibernate.cfg.xml"></property> -->  <property name=*"dataSource"* ref=*"dataSource"*></property>  <property name=*"hibernateProperties"*>  <props>  <prop key=*"hibernate.dialect"*>org.hibernate.dialect.MySQL5Dialect</prop>  <prop key=*"hibernate.show\_sql"*>true</prop>  <prop key=*"hibernate.format\_sql"*>true</prop>  <prop key=*"hibernate.hbm2ddl.auto"*>update</prop>  <prop key=*"hibernate.current\_session\_context\_class"*>thread</prop>  </props>  </property>  <!-- 映射文件位置 -->  <property name=*"mappingLocations"* value=*"classpath:com/gyf/borrowsys/domain/\*.hbm.xml"*></property>  </bean> |

### 第三步：spring整合struts

编写action类，并将其配置给spring ，spring可以注入service

编写struts.xml

表单jsp页面

#### web.xml 配置

1.确定配置文件contextConfigLocation

2.配置监听器 ContextLoaderListener

3.配置前端控制器 StrutsPrepareAndExecuteFitler

|  |
| --- |
| <!-- 加载spring的配置文件，初始化相关的bean -->  <context-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:applicationContext.xml</param-value>  </context-param>  <listener>  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  </listener>    <!-- struts的过滤器 -->  <filter>  <filter-name>struts2</filter-name>  <filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>  </filter>    <filter-mapping>  <filter-name>struts2</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |

#### action和spring配置文件

|  |
| --- |
| **action中service默认会根据名称注入**  **默认情况下框架使用的自动装配策略是name，也就是说框架会去 Spring中寻找与action属性名字相同的bean** |
|  |

#### actoin对象由spring创建

