# 1 Spring介绍

## 1.1 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。

## 1.2 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、远程调用webservice等），都提供了封装，使这些API应用难度大大降低

## 1.3 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：

|  |
| --- |
|  |

## 1.4 在项目中的架构

web层：Struts,SpringMVC

dao层：Hibernate,mybatis

|  |
| --- |
|  |

# 2 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

|  |
| --- |
|  |

## 2.3 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 |

## 2.4 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查看

|  |
| --- |
|  |

2> 离线配置文件约束提示的配置（断网时有提示，联网会自动查询）

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

3> Beans.xml文件配置

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd"*>    <!-- 配置一个bean -->  <bean id=*"us"* class=*"com.cheney.service.UserSerivce"*></bean>  </beans> |

### 第五步：

* 从beans.xml获取bean

|  |
| --- |
|  |
| //2.从spring容器获取 userSerivce对象  IUserSerivce userSerivce1 = (IUserSerivce) context.getBean("userService");  userSerivce1.add();  IUserSerivce userSerivce2 = (IUserSerivce) context.getBean("userService");  userSerivce2.add();  System.out.println(userSerivce1);  System.out.println(userSerivce2); // 相同 |
| 两次是同一个对象 |

# 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**
2. 概念：也就是说，控制反转就是把对象的实例化工作交给了spring容器来做，当我们需要某一个对象的时候，不再需要自己实例化了，而是直接问容器索取即可！
3. 作用： （问题是：为什么要使用控制反转？ 使用控制反转的好处是什么!？）
   1. 把具体类名从客户端代码中解耦出去。（完全解耦）。
   2. 我们问容器索取一个对象时，最终得到的对象，就是是哪个类的对象，不能由我们决定，而是由容器决定的！

# 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

|  |
| --- |
|  |
|  |

# 加载Spring容器的三种方式

## 3.1 类路径获得配置文件

|  |
| --- |
| ClassPath：类路径，即打包后的classes的路径  spring的配置文件路径以后就直接放在src |
| ApplicationContext context =  **new** ClassPathXmlApplicationContext("com/gyf/spring/demo01/beans.xml"); |

## 3.2 文件系统路径获得配置文件【绝对路径】

|  |
| --- |
|  |

## 3.3使用BeanFactory(了解)

|  |
| --- |
|  |

## 3.4 BeanFactory和ApplicationContext对比

* BeanFactory 采取延迟加载，第一次getBean时才会初始化Bean
* ApplicationContext 采取即时加载，加载完配置文件，对象就创建好了（调用无参）,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)

所谓的装配bean就是在xml写一个bean标签

## 4.1 实例化Bean的三种方式

### 4.1.1 使用构造方法实例化

第一种方式: new 实现类

### 4.1.2 使用静态工厂方法实例化

第二种方式：通过工厂类调用静态方法

### 4.1.3 使用实例工厂方法实例化

与静态工厂相比，就是方法不是静态的，需要对象调用

|  |
| --- |
| 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,需要使用1.7的jdk  -->* <**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(); } |

## 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接口 执行 setBeanName -- 设置bean的名字

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

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

6.如果Bean实现InitializingBean接口 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接口（后处理Bean） ，执行postProcessAfterInitialization

**-- 执行业务处理**

9.如果Bean实现 DisposableBean接口 执行 destroy

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

## 演示

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

# 依赖注入Bean属性

## 6.1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】

xmlns xml nanespace xml命名空间

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

## 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**

|  |
| --- |
|  |
|  |

一个对象引用另外一个对象两写法

1.ref: 引用<property name="address" ref="address"></property>

2.SpEL:<property name="address" value="#{address}"></property>

## 6.3 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

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

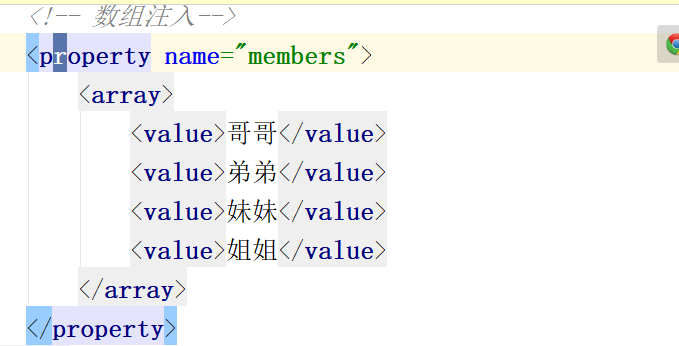
### Map

|  |
| --- |
|  |
|  |

### 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。**
* **使用注解要开启，做如下配置**

|  |
| --- |
|  |

### 6.4.1 @Component

@component取代<bean class="">

1.如果@Component没配置id,通过类型获取

2.这个类型可以是接口，还可以是实现类

IUserService service = (IUserService) context.getBean(IUserServiceImpl.class);

### 6.4.2 @Component("id")

取代 <bean id="" class="">

### 6.4.4 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

Spring会根据注解,按照顺序生成dao,service,action对象

### 6.4.5 使用案例

#### 第一个案例

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

#### 第二个案例

|  |
| --- |
|  |
|  |

#### 第三个案例

|  |
| --- |
| 使用注解：不需要提供getter和setter方法，spring会根据类型自动赋值 |
|  |
|  |
| 使用xml：属性须提供getter和setter方法 |
| <!-- 1.配置dao-->  <bean id="userDao" class="com.gyf.dao.UserDaoImpl"></bean>  <!-- 2.配置service -->  <bean id="userService" class="com.gyf.service.UserServiceImpl">  <property name="userDao" ref="userDao"></property>  </bean>  <!--3.配置action-->  <bean id="userAction" class="com.gyf.web.action.UserAction">  <property name="userService" ref="userService"></property>  </bean> |

#### 第四个案例

|  |
| --- |
|  |
|  |
| 推荐使用类型自动注入 |

Autowired是根据类型注入值【用的多】

\* 如果你是一个接口，从容器找接口实现类

\* 如果你就是一个类，就查找类

@Resource等效于@Autowired+@Qualifier【用的比较少】

#### 第五个案例

|  |
| --- |
|  |

#### 第六个案例

|  |
| --- |
| @Repository  public class UserDaoImpl implements IUserDao{  // <bean init-method="" destroy-method=""></bean>  @PostConstruct //相当于init-method=""  public void myInit(){  System.out.println("UserDaoImpl自定义的初始化方法...");  }  @PreDestroy//相当于destroy-method=""  public void myDestroy(){  System.out.println("UserDaoImpl自定义的销毁方法...");  }  @Override  public void add(User user) {  System.out.println("dao 添加用户:" + user);  }  } |
|  |
| //关闭容器 -- 通过反射  context.getClass().getMethod("close").invoke(context); |

# AOP【重要】

面试题：aop是什么？在项目中有什么用？

Aop：面向切面编程

作用：解耦，提高了开发效率

## 7.1 AOP概述

1. 在软件业，**AOP为Aspect Oriented Programming**的缩写，意为：**面向切面编程**，通过预编译方式和运行期动态代理 实现程序功能的统一维护的一种技术。
2. 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。
3. 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。
4. 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
5. **经典应用：事务管理、性能监视、安全检查、缓存 、日志等【画图】**
6. 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
7. **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

|  |
| --- |
| OOP -- 通过纵向继承 -- 业务耦合度高，需要关联 |
|  |
| AOP -- 通过代理拦截 -- 业务耦合度低，不需要关联 |
|  |

## 7.2 AOP实现原理【重要】

aop底层采用代理机制进行实现。

1. 接口 + 实现类 ：可以采用 **jdk 的Proxy**来实现代理。

2. 有接口的实现类和没有接口的实现类：都可以采用 **cglib字节码**增强实现代理。

共同点：都是动态代理，在运行中生成。

## 7.3 AOP术语

1. **横切关注点：**非核心业务
2. **核心业务关注点：**核心业务
3. **target：**目标类，需要被代理的类。被加入切面的对象，例如：UserService
4. **Joinpoint**(连接点):所谓连接点是指那些可能被拦截到的方法。可以执行通知地点的全集，例如：所有的方法
5. **PointCut** 切入点：已经被增强的连接点。执行通知的地点，例如：addUser()
6. **advice 通知/增强：增强代码**。首先是一个方法，例如：after、before
7. **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（把目标对象的切面结合在一起的过程）
8. **proxy** 代理类 ：目标对象和切面织入在一起后，得到的对象
9. **Aspect**(切面): 是切入点pointcut和通知advice的结合（通知的集合，切面中描述了在什么地点，什么时间，干什么事）

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。

|  |
| --- |
|  |

## 7.4 手动代理

### 7.4.1 JDK动态代理 Proxy

#### 目标类

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

#### 切面类

|  |
| --- |
|  |

#### 工厂类

|  |
| --- |
|  |
| 步骤：  1.创建目标对象target  2.声明切面类对象  3.把切面类和目标类结合，即将要执行的代码插入  3.1 创建JDK代理,拦截方法  ClassLoader loader, 类加载器，写当类  Class<?>[] interfaces, 接口,接口的方法会被拦截  InvocationHandler h) 处理 |

#### 测试类

|  |
| --- |
|  |
| 代理类具有目标类的功能，并可以强转目标类 |

### 7.4.2 cglib 增强字节码

问题：如果没有接口，只有实现类，则不能使用动态代理。

* 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。
* 这种方式可以达到解耦的效果

1. JDK代理最后还是要目标类对象调用方法
2. cglib可以使用目标类对象调用方法，也可以使用代理类对象调用父类的方法，这样就解耦了

* 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

|  |
| --- |
|  |
| 步骤：  1.创建目标对象target  2.声明切面类对象  3.创建增强对象  Enhancer enhancer = new Enhancer();  //设置父类  enhancer.setSuperclass(xxx.getClass());  设置回调【拦截】：里面传入接口 |

#### 测试结果

|  |
| --- |
|  |

## 7.5 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  } |

## 7.6 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

|  |
| --- |
|  |

### 总结：

* 思想：由spring的工厂生成代理对象，比较自己手动代理，所以需要给工厂的属性赋值，即目标类对象、目标类接口、切面类对象。
* 默认使用jdk动态代理，可以配置使用cglib实现代理
* 切面类要实现MethodInterceptor接口

## 7.7 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

|  |
| --- |
|  |

### 第三步：切面类

|  |
| --- |
|  |

### 第三步：Spring 的AOP配置

|  |
| --- |
|  |

### 第四步：测试

|  |
| --- |
|  |

### 总结：

* 所谓全自动就是不需要自己直接使用代理将通知和切入点结合，而是通过一种方式实现自动织入
* 步骤

1. 在bean中配置aop约束
2. 配置aop:conifg内容，把切入点和通知结合
   1. proxy-target-class: 为true时使用cglib实现代理
   2. 配置切入点
   3. 通知 关联 切入点

* 获取的对象不再是真实对象，而是spring生成的代理对象
* AOP:用于事务配置&日志记录
* 切面类要实现MethodInterceptor接口

# 8、AspectJ

## 8.1 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**

## 8.2 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') |

## 8.3 AspectJ 通知类型

|  |
| --- |
| aop联盟定义通知类型，aop联盟的jar都是接口，必须要有实现类。  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种**。  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，放行**  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

|  |
| --- |
|  |
|  |

## 8.4 Aspect案例讲解【基于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

|  |
| --- |
|  |

### 总结：

* 切面类不需要实现MethodIntercetor接口
* 配置aop与spring全自动aop有区别，配置了切面aspect
* 配置步骤：

1. 配置业务类
2. 配置切面类
3. 配置 aop

aop:指定切面

1. 定义切入点
2. 配置通知，引用切入点

* 好处

1. 这样可以将方法和增强代码（通知）解耦
2. 可以单独设置需要的通知

* 通知方法参数：连接点 即拦截的方法，spring会自动注入。如果是环绕通知需要放行
* 后置通知可以获取service方法执行后的返回值
* 最终通知：不管有没有异常都会执行，一般用于释放资源

## 8.5 Aspect案例讲解【基于注解】

### 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

## 8.6 xml和注解通知执行顺序区别

* service方法有返回值 + 无异常 + XML

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1.前置通知...deleteUser

2.环绕通知...开启事务...deleteUser

通过id删除用户

3.后置通知...deleteUser

返回值:1

4.环绕通知....提交事务...

最终通知...deleteUser

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

* service方法有返回值 + 无异常 + 注解

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2.环绕通知...开启事务...deleteUser

1.前置通知...deleteUser

通过id删除用户

4.环绕通知....提交事务...

最终通知...deleteUser

3.后置通知...deleteUser

返回值:1

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

* service方法没有返回值 + 无异常 + XML

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1.前置通知...deleteUser

2.环绕通知...开启事务...deleteUser

删除用户。。。。

3.后置通知...deleteUser

返回值:null

4.环绕通知....提交事务...

最终通知...deleteUser

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

* service方法没有返回值 + 无异常 + 注解

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2.环绕通知...开启事务...deleteUser

1.前置通知...deleteUser

删除用户。。。。

4.环绕通知....提交事务...

最终通知...deleteUser

3.后置通知...deleteUser

返回值:null

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# 9、JdbcTemplate

## 9.1 简介

* jdbcTemplate类似于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

|  |
| --- |
| 1 |

## 9.2 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

|  |
| --- |
|  |

## 9.3 API使用【了解】

|  |
| --- |
|  |

## 9.4 配置DBCP

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

## 9.5 配置c3p0

掌握一种思想：数据源不自己创建，由spring创建。

|  |
| --- |
|  |

## 9.6 使用JdbcDaoSupport

### dao层

|  |
| --- |
|  |

### Beans.xml

|  |
| --- |
|  |

### 源码分析

**通过数据源创建模板**

|  |
| --- |
|  |

## 9.7 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

|  |
| --- |
|  |

# 10、事务管理

## 10.1 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（保存点）

|  |
| --- |
| 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();  }  } |

## 10.2 Spring事务管理介绍

### 10.2.1 Spring提供的事务jar包

transaction = tx

|  |
| --- |
|  |

### 10.2.2 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。 |

### 10.2.3 PlatformTransactionManager 事务管理器

#### 先导入两个包

|  |
| --- |
|  |

#### 常用的两个事务管理器

|  |
| --- |
|  |

### 10.2.4 TransactionStatus 事务状态

|  |
| --- |
|  |

### 10.2.5 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

## 10.3 案例：转帐

### 10.3.1 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（core,context,beans,expression）+logging
* 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

|  |
| --- |
|  |

### 10.3.2 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

|  |
| --- |
|  |

### 10.3.3 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：

|  |
| --- |
|  |

### 10.3.4 基本AOP的事务配置【掌握】

#### Spring的配置文件

|  |
| --- |
|  |
|  |

#### 测试同上

### 10.3.5 基本于注解的事务

#### spring配置

|  |
| --- |
|  |

|  |
| --- |
|  |

### 10.3.5 整合Junit

目的：少写一些代码

导入一个spring-test包

|  |
| --- |
|  |

# 11、SSH整合

## 11.1 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

|  |
| --- |
|  |

## 11.2 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>  <!-- 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>  <!-- 顺序 -->  <listener>  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  </listener> |

#### 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

