# Java EE 技术之 Spring 技术栈

讲师：刘峰吉

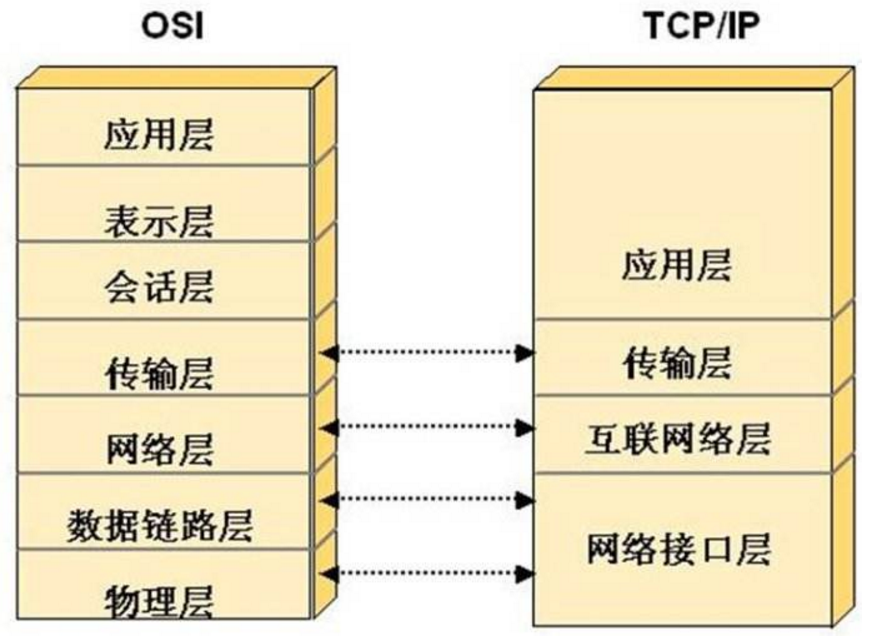
使用框架，就是让非业务功能的代码，不用让程序员写了

Spring(IOC/AOP)

IOC 让程序解耦

AOP 扩展程序功能

# 一、分层结构



## 1、网络分层

分层ISO提出的OSI（Open System Interconnection）模型将网络分为七层：

物理层（ Phisical ）、数据链路层（Data Link）、网络层（Network）、传输层（Transport）、会话层（Session）、表示层（Presentation）和应用层（Application）。

1）物理层（Physical layer）是参考模型的最低层。

该层是网络通信的数据传输介质，由连接不同结点的电缆与设备共同构成。主要功能是：利用传输介质为数据链路层提供物理连接，负责处理数据传输并监控数据出错率，以便数据流的透明传输。

2）数据链路层（Data link layer）是参考模型的第2层。

主要功能是：在物理层提供的服务基础上，在通信的实体间建立数据链路连接，传输以“帧”为单位的数据包，并采用差错控制与流量控制方法，使有差错的物理线路变成无差错的数据链路。

3）网络层（Network layer）是参考模型的第3层。

主要功能是：为数据在结点之间传输创建逻辑链路，通过路由选择算法为分组通过通信子网选择最适当的路径，以及实现拥塞控制、网络互联等功能。

4）传输层（Transport layer）是参考模型的第4层。

主要功能是向用户提供可靠的端到端（End-to-End）服务，处理数据包错误、数据包次序，以及其他一些关键传输问题。传输层向高层屏蔽了下层数据通信的细节，因此，它是计算机通信体系结构中关键的一层。

5）会话层（Session layer）是参考模型的第5层。

主要功能是：负责维扩两个结点之间的传输链接，以便确保点到点传输不中断，以及管理数据交换等功能。

6）表示层（Presentation layer）是参考模型的第6层。

主要功能是：用于处理在两个通信系统中交换信息的表示方式，主要包括数据格式变换、数据加密与解密、数据压缩与恢复等功能。

7）应用层（Application layer）是参考模型的最高层。

主要功能是：为应用软件提供了很多服务，例如文件服务器、数据库服务、电子邮件与其他网络软件服务。

## 2、物理分层



## 3、软件分层

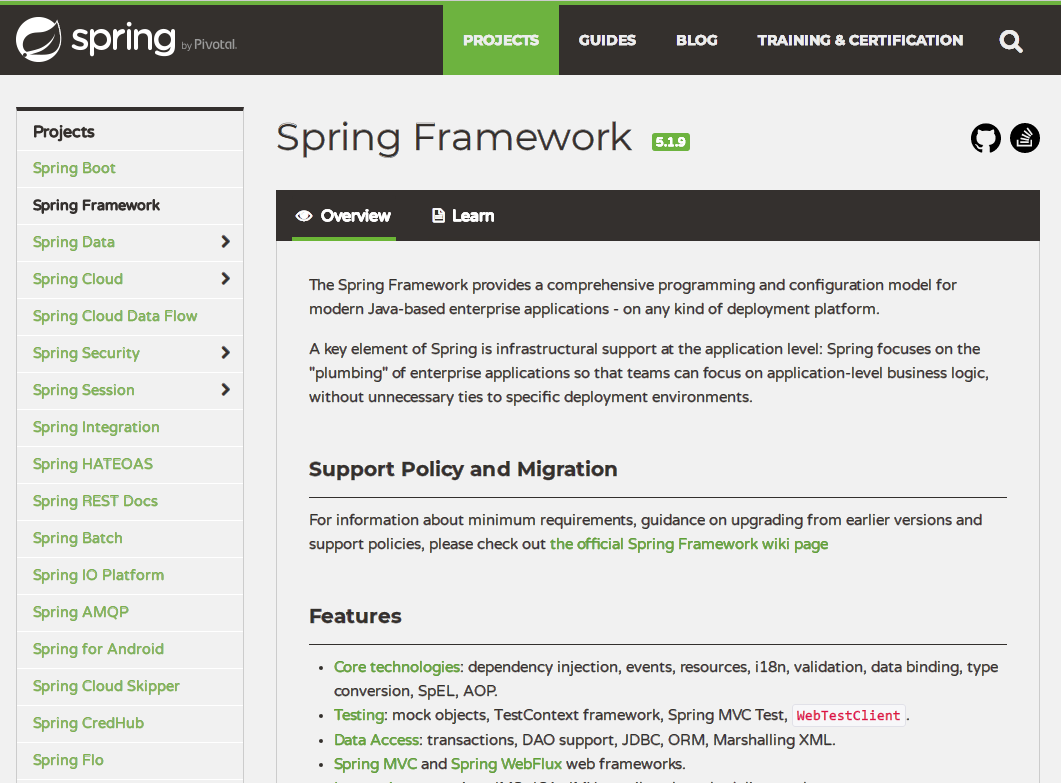


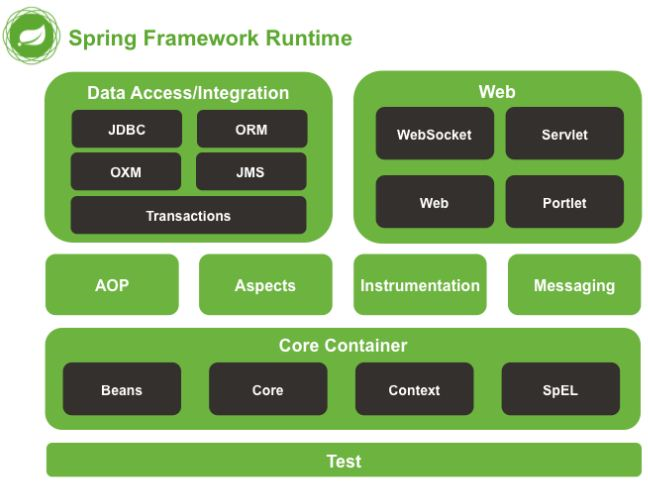
# 二、Spring 概述

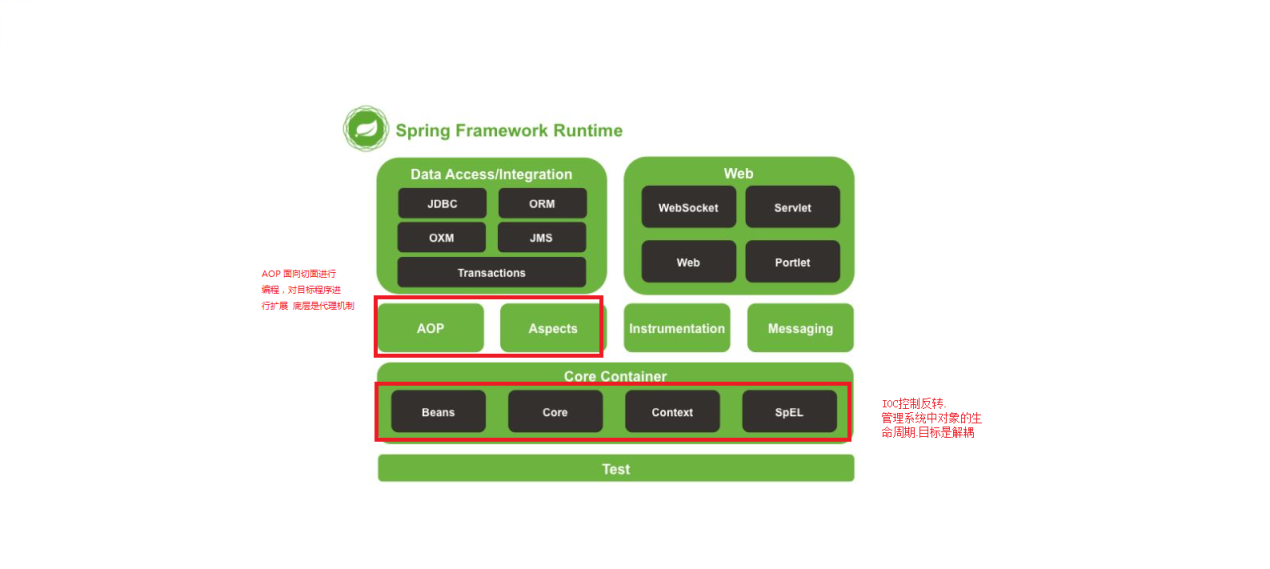
## 1、网址

https://spring.io/

https://spring.io/projects/spring-framework







## 2、Spring 简述

Spring框架是一个基于IOC和AOP的容器框架.

1）IOC

Invertion of Control 控制反转.

管理系统中对象的生命周期.目标是解耦.

2）AOP

Aspect Oritend Programming 面向切面编程

解决程序中重复性代码问题.目标是解耦.

3）容器

用于管理对象的 对象.例如:集合

管理对象的生命周期.

实现原理:工厂+XML+反射

4）框架

目的就是将非业务代码进行封装.提高开发效率.

Spring是一个开源框架

Spring为简化企业级开发而生，使用Spring，JavaBean就可以实现很多以前要靠EJB才能实现的功能。同样的功能，在EJB中要通过繁琐的配置和复杂的代码才能够实现，而在Spring中却非常的优雅和简洁。

Spring是一个IOC(DI)和AOP容器框架。

Spring的优良特性

[1]非侵入式：基于Spring开发的应用中的对象可以不依赖于Spring的API

[2]依赖注入：DI——Dependency Injection，反转控制(IOC)最经典的实现。

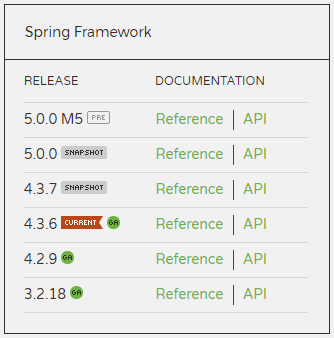
[3]面向切面编程：Aspect Oriented Programming——AOP

[4]容器：Spring是一个容器，因为它包含并且管理应用对象的生命周期

[5]组件化：Spring实现了使用简单的组件配置组合成一个复杂的应用。在 Spring 中可以使用XML和Java注解组合这些对象。

[6]一站式：在IOC和AOP的基础上可以整合各种企业应用的开源框架和优秀的第三方类库（实际上Spring 自身也提供了表述层的SpringMVC和持久层的Spring JDBC）。

Spring目前的版本http://spring.io/projects



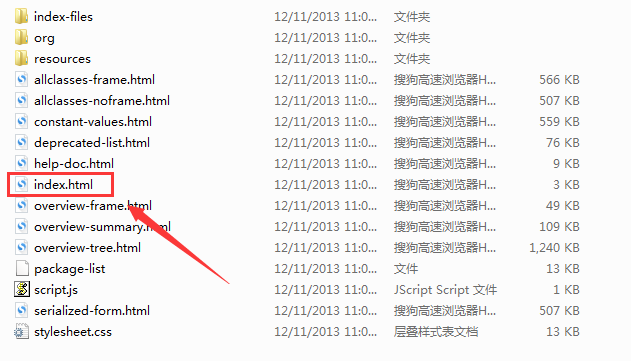
## 3、Spring 安装包

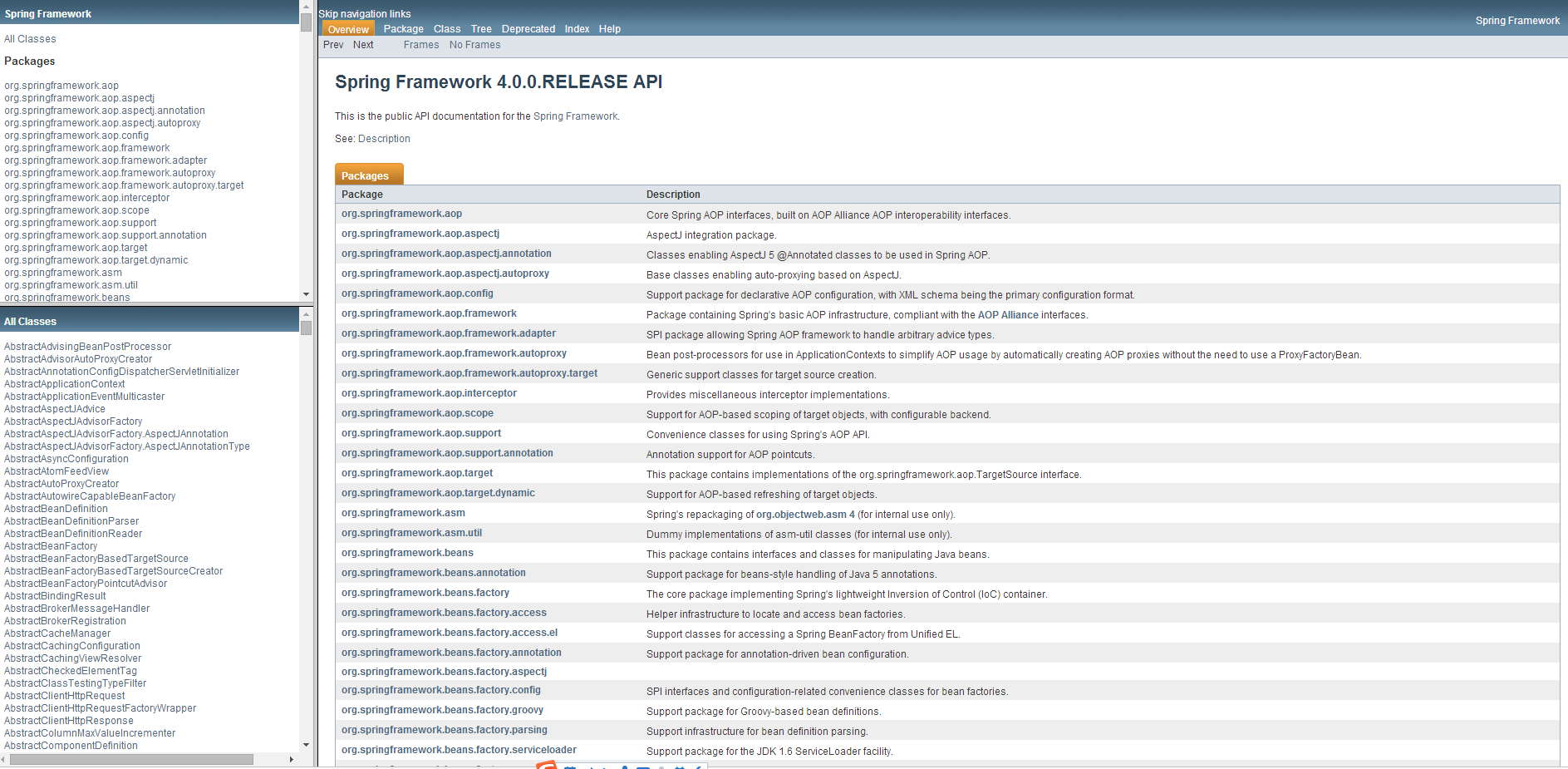
spring-framework-4.0.0.RELEASE-dist.zip

## 4、Spring API

### 1）Javadoc-api

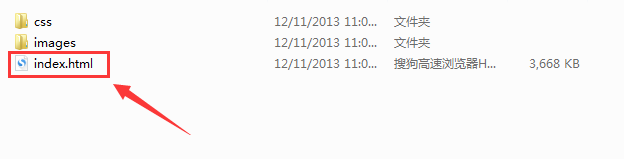
spring-framework-4.0.0.RELEASE\docs\javadoc-api

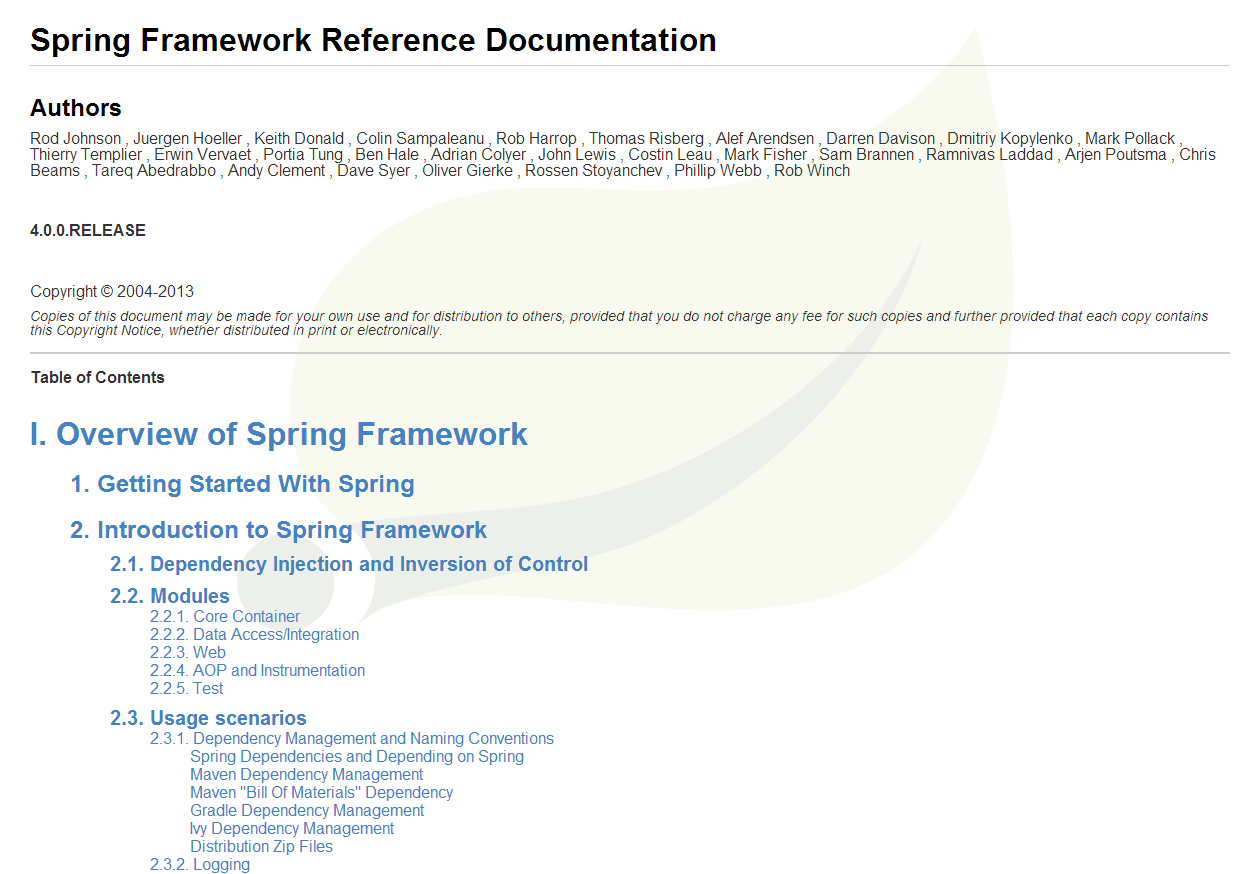




### 2）Documentation

spring-framework-4.0.0.RELEASE\docs\spring-framework-reference\htmlsingle

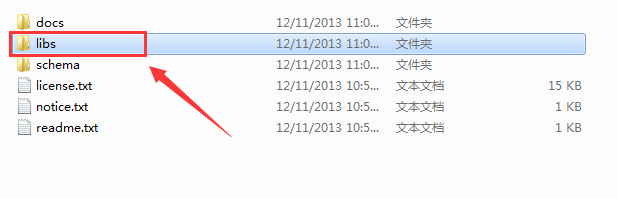




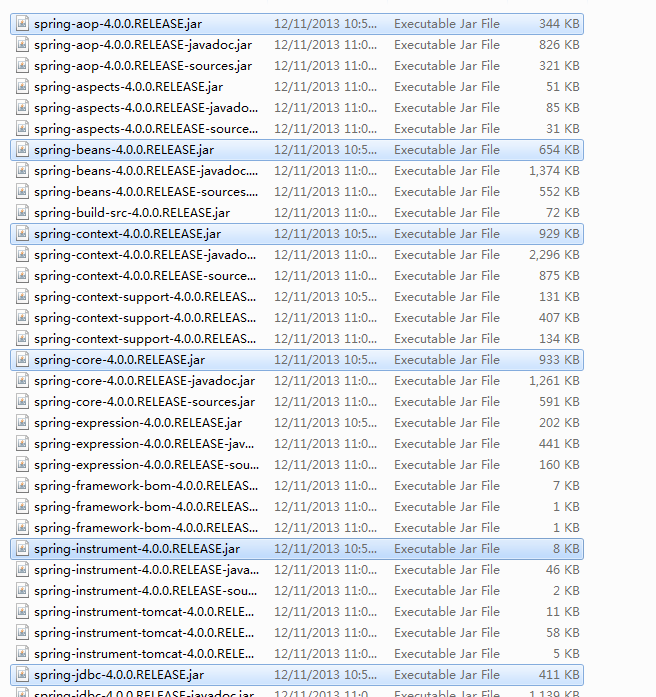
## 5、Spring 核心包

spring-framework-4.0.0.RELEASE\libs

### 1）图一



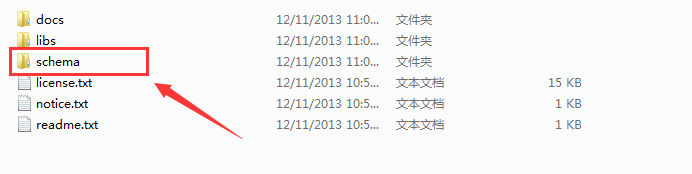
### 2）图二

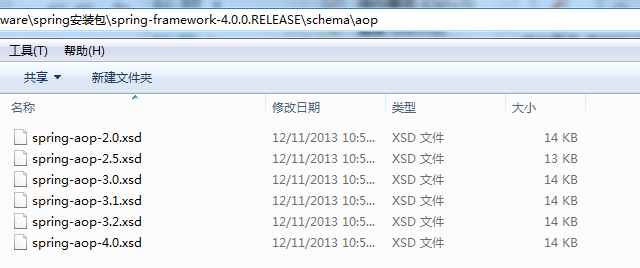


## 6、Spring 约束文件

框架的约束文件有两种，主要是Schema和DTD，Mybatis使用的约束文件就是DTD，现在大部分框架的约束文件使用的都是Schema

spring-framework-4.0.0.RELEASE\schema\aop

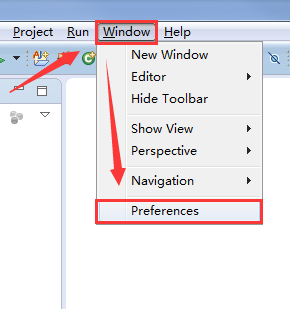




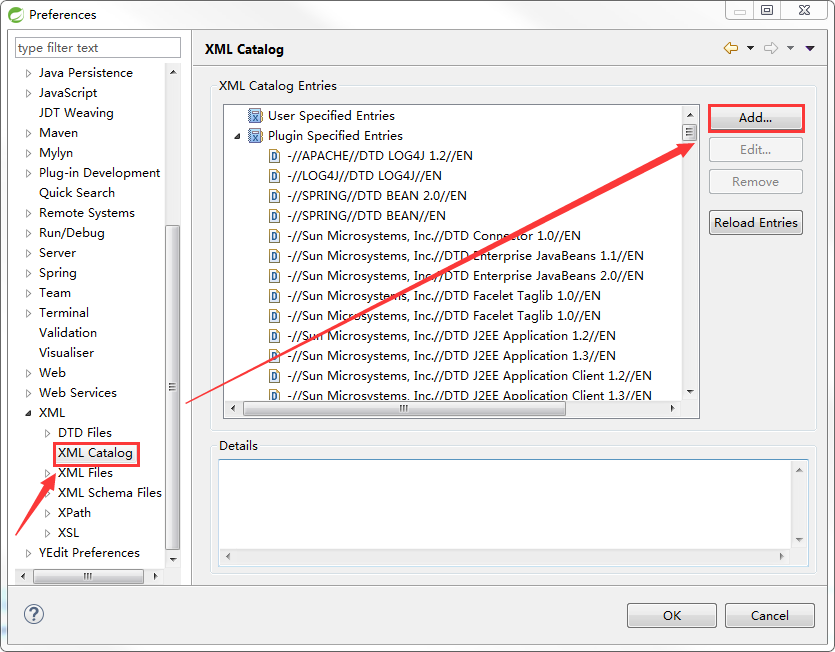
## 7、spring xsd约束文件配置

这里以beans.xsd 为例进行讲解

### 1）window -> Preferences

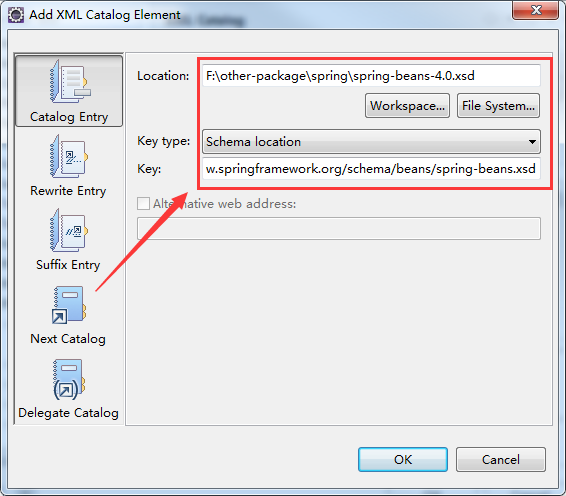


### 2）XML -> XML Catalog -> Add



### 3）配置spring-beans.xsd

Key: http://www.springframework.org/schema/beans/spring-beans.xsd



### 4）扩展

#### （1）DTD类型约束文件

   Window->Preferences->XML->XML Catalog->UserSpecified Entries窗口中,选择Add 按纽

   在Add XML Catalog Entry对话框中选择或输入以下内容:

       Location:D:\soft\dtd\Ibatis\sql-map-config-2.dtd

       Key Type: URI

       KEY:http://ibatis.apache.org/dtd/sql-map-config-2.dtd

#### （2）XSD类型约束文件

   Window->Preferences->XML->XML Catalog->UserSpecified Entries窗口中,选择Add 按纽

   在Add XML Catalog Entry对话框中选择或输入以下内容:

       Location:D:\soft\xsd\spring\spring-beans-2.5.xsd

       Key Type: Schema Location

       KEY:http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

# 三、Spring Hello World

## 1、思路

1）创建Java项目（Java,Web项目都可以）

2）拷贝框架常用jar包,增加到classpath路径,让项目能够识别这些jar包

3）创建Spring框架的核心配置文件,存放到src目录下.也就是编译后的classpath路径.

常用名称

spring-context.xml

applicationContext.xml

beans.xml

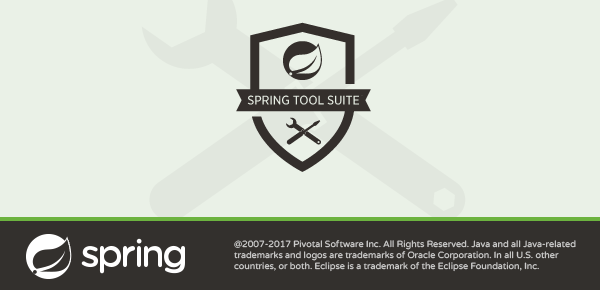
4）创建IOC容器对象

5）从IOC容器对象中获取bean对象

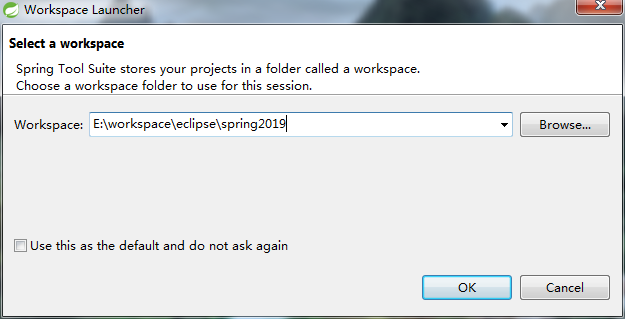
6）关闭IOC容器对象

## 2、Hello World实操（spring\_helloworld01）

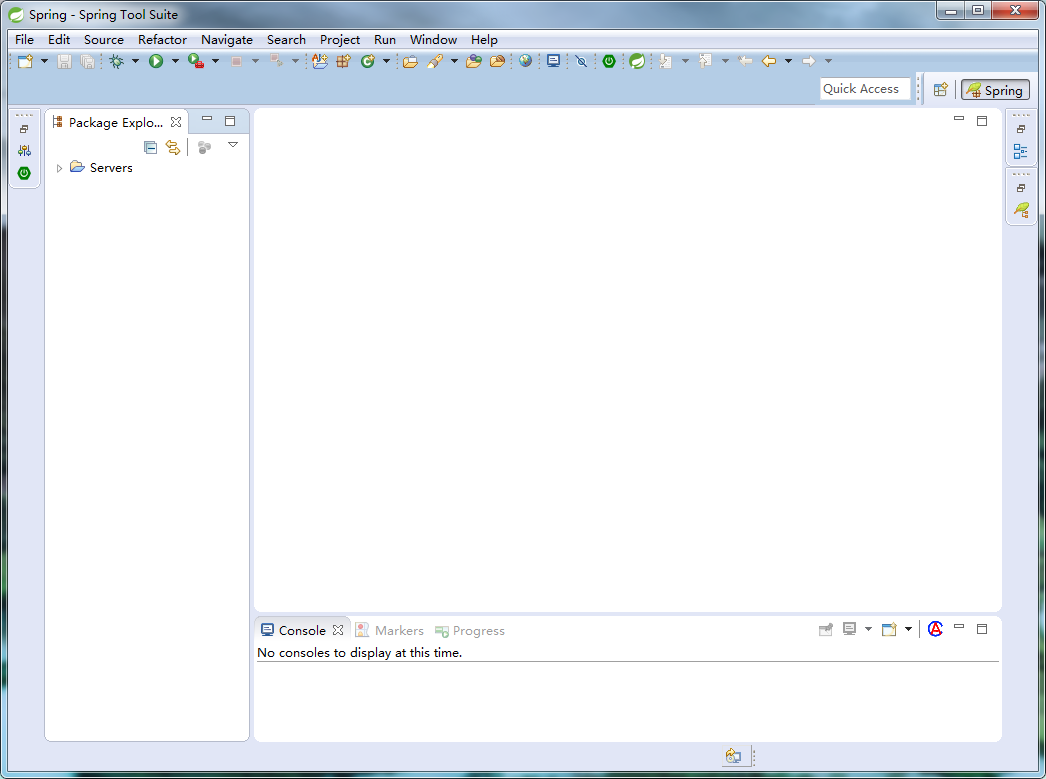
### 1）双击Eclipse



### 2）选择工作空间

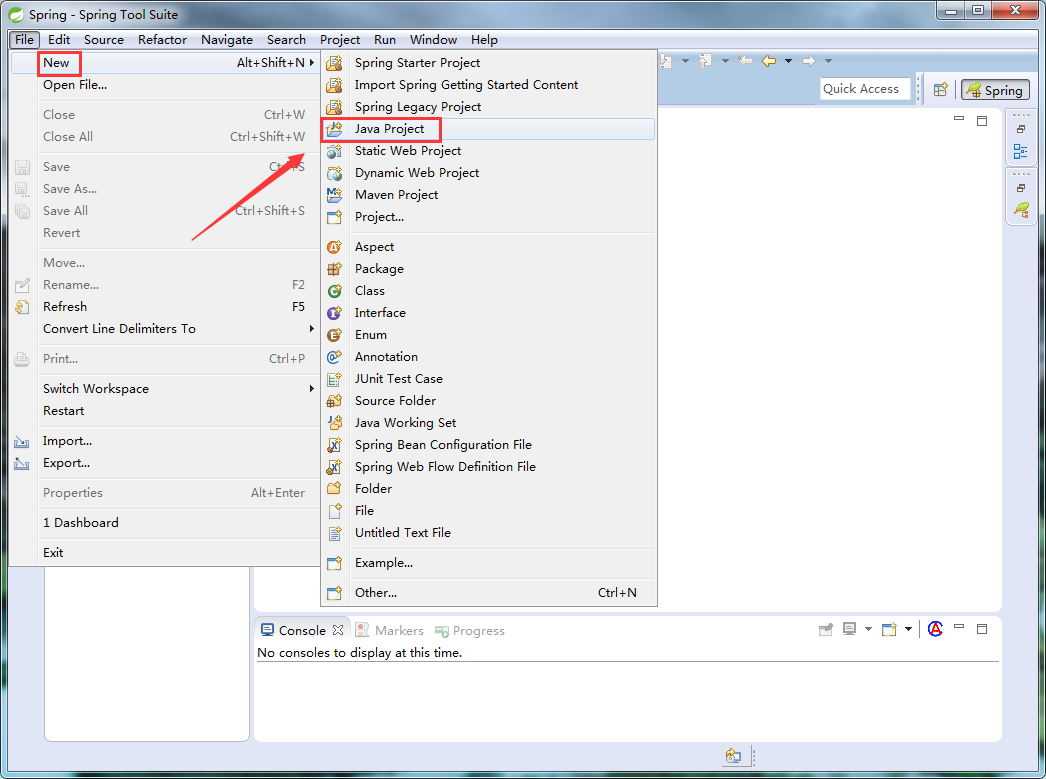


### 3）最小化不用的选项

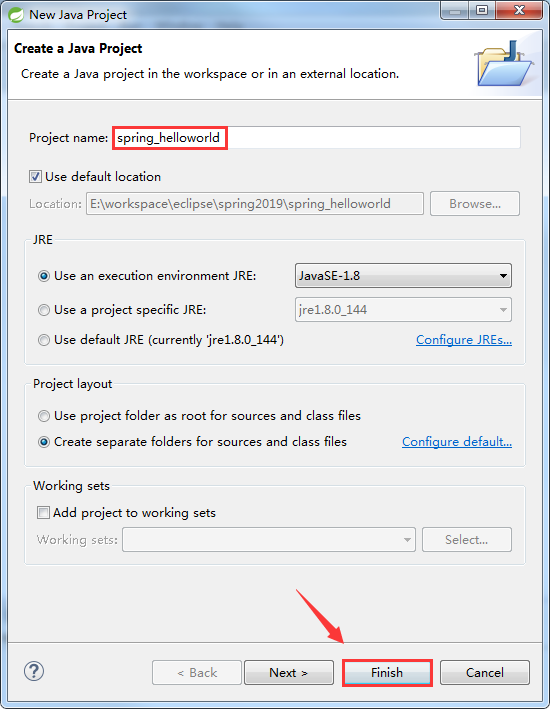


### 4）创建Java Project

File -> New -> Java Project

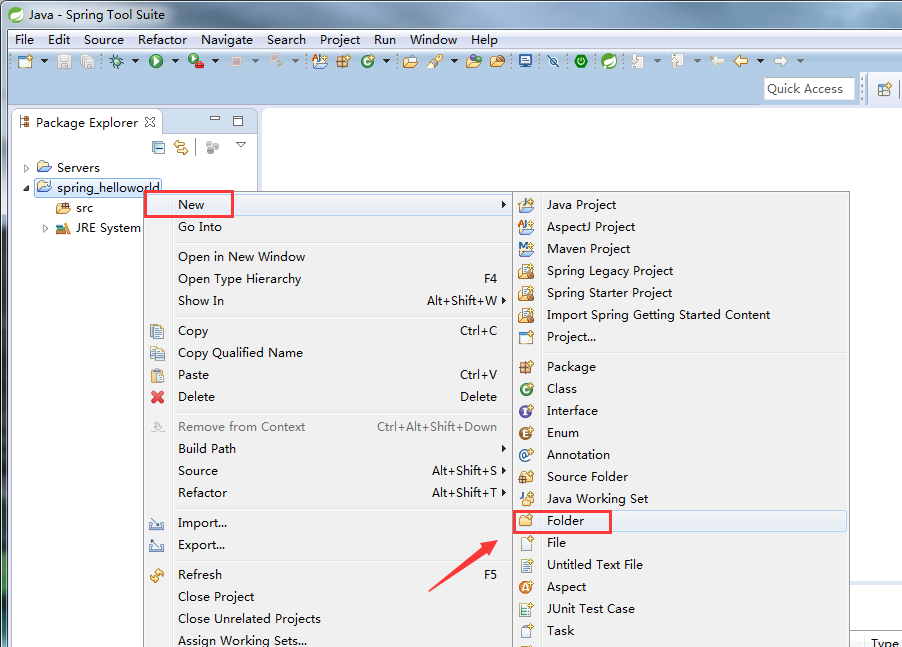


### 5） Project Name

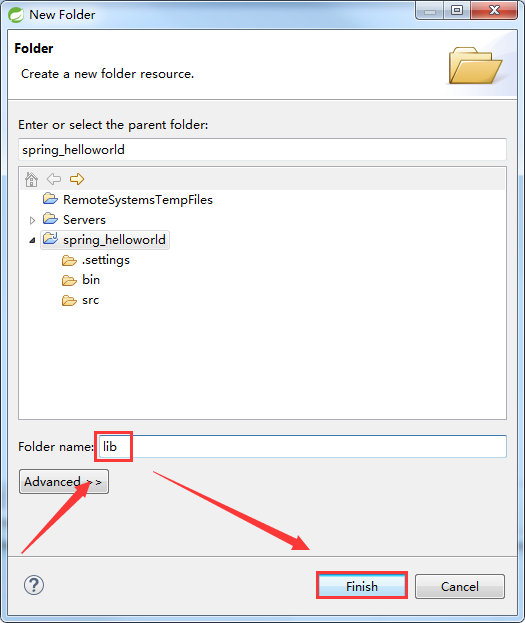


### 6）创建Lib文件夹

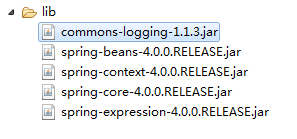
#### （1）项目右键 –> new -> Folder



#### （2）lib -> Finish

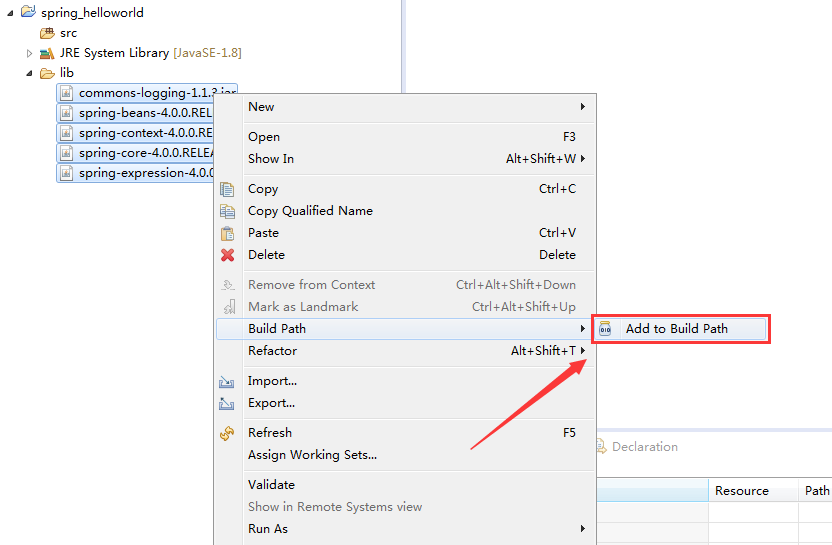


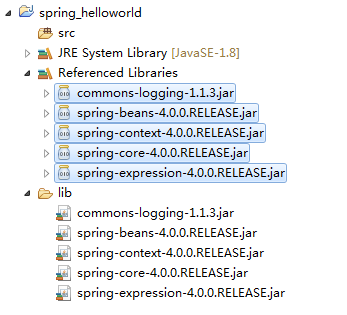
### 7）复制Jar包



### 8） Build Path

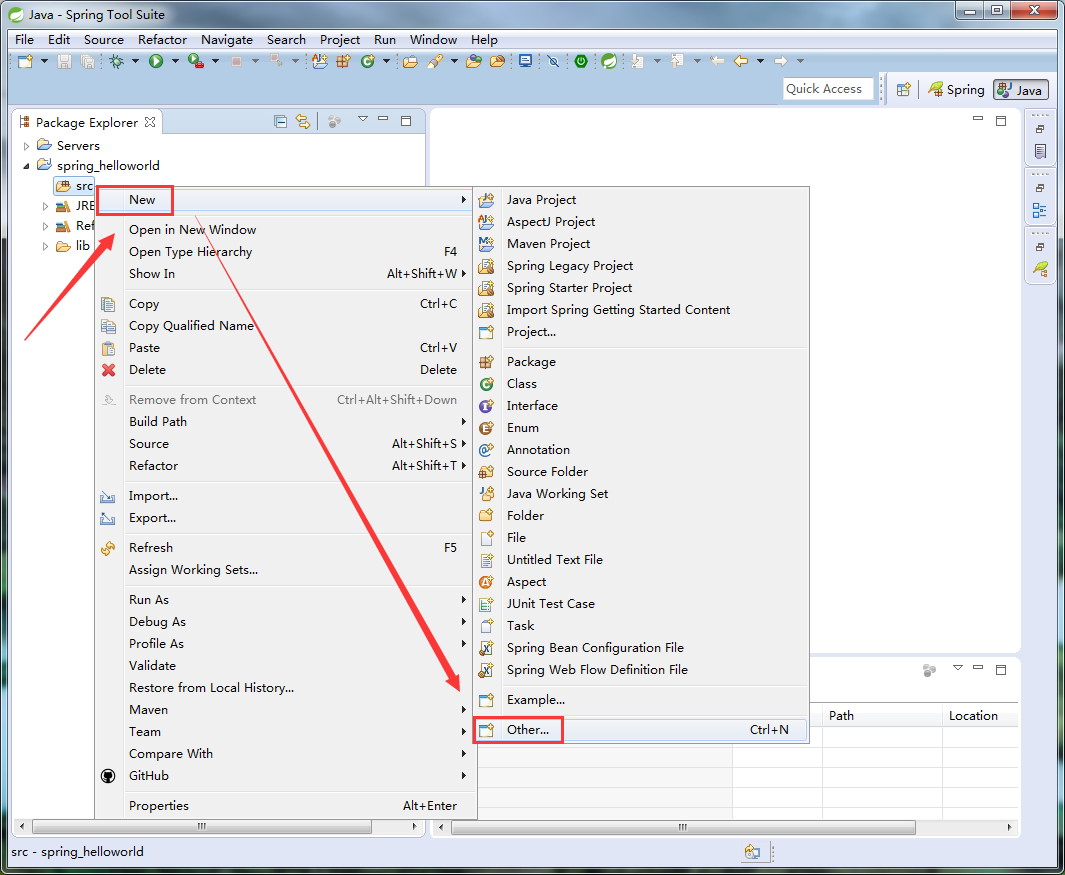
把Jar包添加到类路径



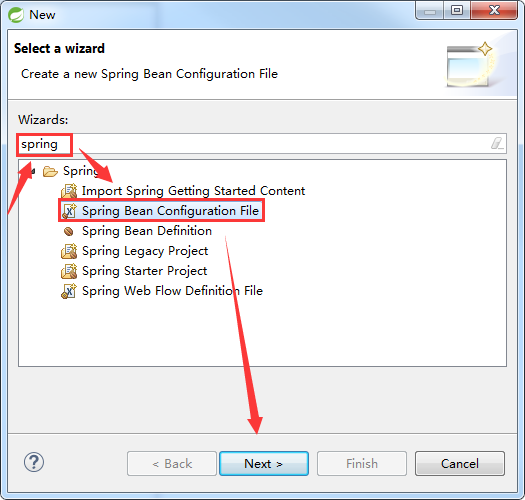


### 9）创建 spring-context.xml

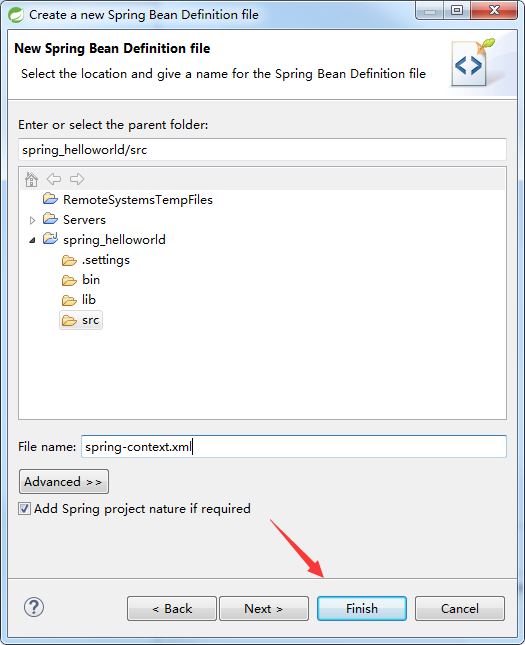
#### （1）src右键 –> New -> Other



#### （2）Spring -> Configuration File -> Next

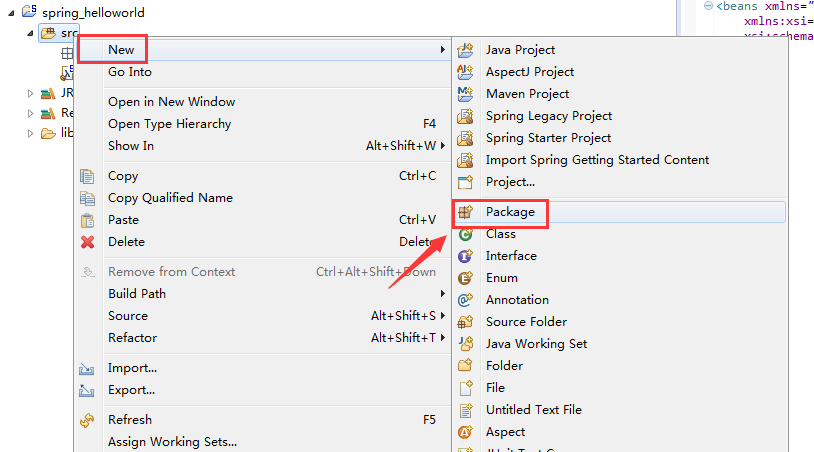


#### （3）spring-context.xml -> Finish

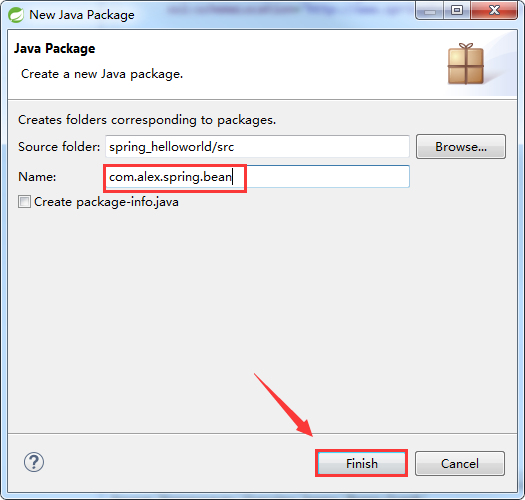


### 10）创建包

#### （1） src右键 –> new -> Package

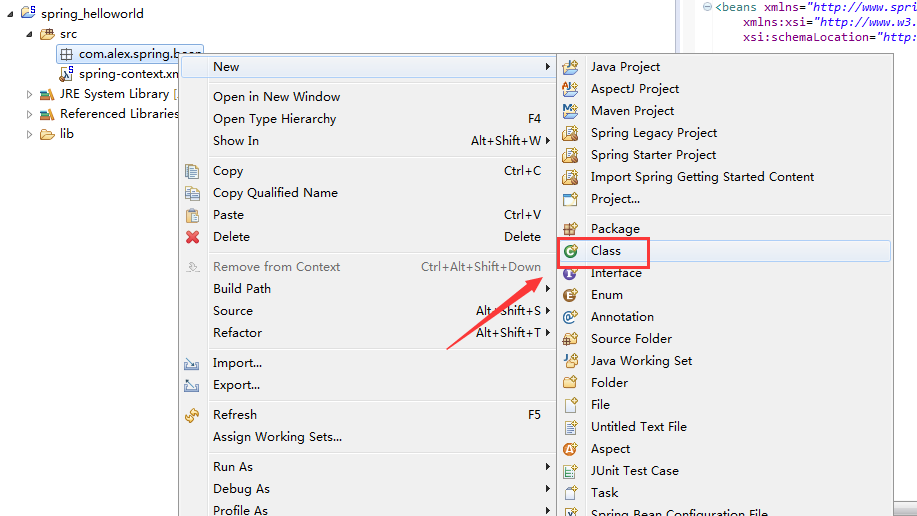


#### （2） Name -> Finish

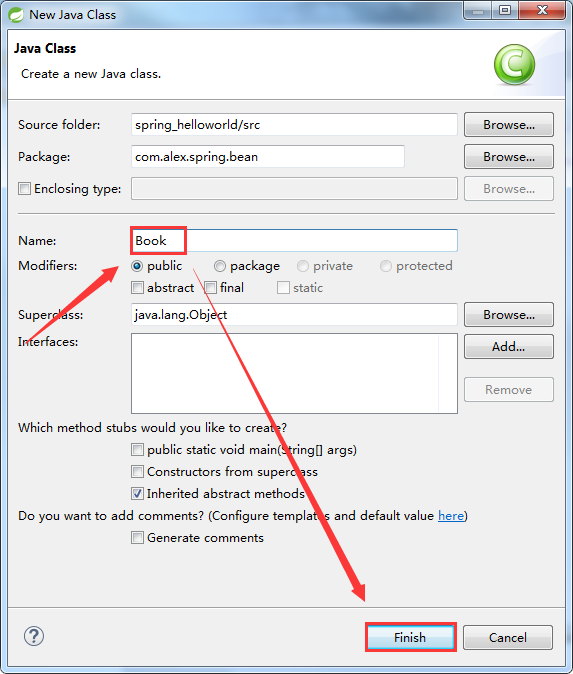


### 11）创建类

#### （1）包右键 –> New -> Class



#### （2）Book -> Finish



#### （3）Book.java

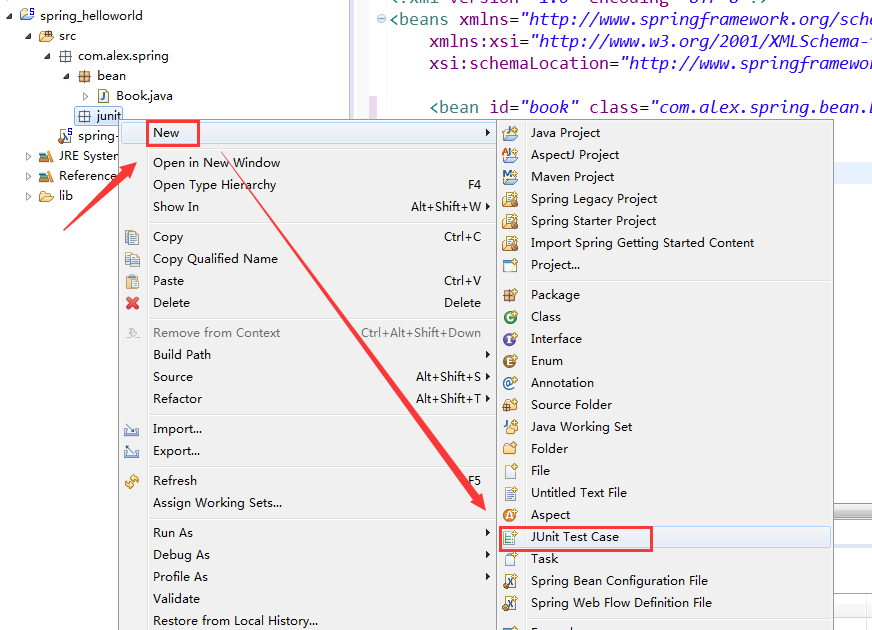
|  |
| --- |
| package com.alex.spring.bean;  public class Book {  private Integer id;  private String title;  private double price;  private String author;  public Book() {  }  public Book(Integer id, String title, double price, String author) {  this.id = id;  this.title = title;  this.price = price;  this.author = author;  }  public Integer getId() {  return id;  }  public void setId(Integer id) {  this.id = id;  }  public String getTitle() {  return title;  }  public void setTitle(String title) {  this.title = title;  }  public double getPrice() {  return price;  }  public void setPrice(double price) {  this.price = price;  }  public String getAuthor() {  return author;  }  public void setAuthor(String author) {  this.author = author;  }  @Override  public String toString() {  return "Book [id=" + id + ", title=" + title + ", price=" + price + ", author=" + author + "]";  }  } |

### 12）配置spring-context.xml

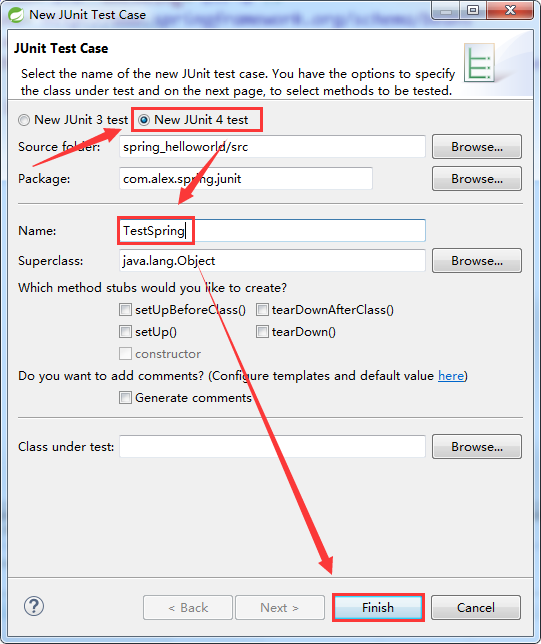
|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">  <!-- 声明Bean对象，默认管理对象是单例的 -->  <bean id="book" class="com.alex.spring.bean.Book"></bean>  </beans> |

### 13）创建单元测试

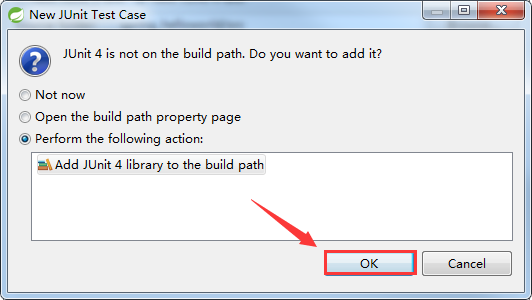
#### （1）创建 JUnit Test Case



#### （2）Name -> Finish



#### （3）Add JUnit4 -> OK

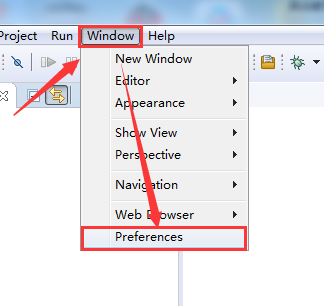


#### （4）TestSpring.java

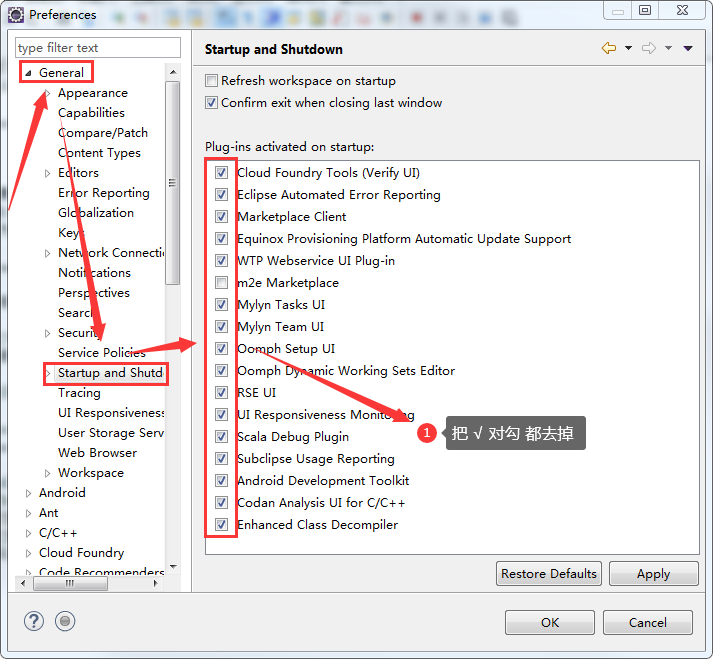
|  |
| --- |
| package com.alex.spring.junit;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.ConfigurableApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.alex.spring.bean.Book;  public class TestSpring {  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");    @Test  public void testIOC() {  //从 IOC 容器中获取对象  Book book1 = (Book) ioc.getBean("book");  System.out.println(book1);    Book book2 = (Book) ioc.getBean("book");  //伪单例，Spring容器帮助管理对象  System.out.println(book1 == book2);    //关闭 IOC 容器对象，一般是不需要关闭的，因为它是单例的  ConfigurableApplicationContext cac = (ConfigurableApplicationContext)ioc;  cac.close();  }  } |

## 3、TST 卡顿现象解决

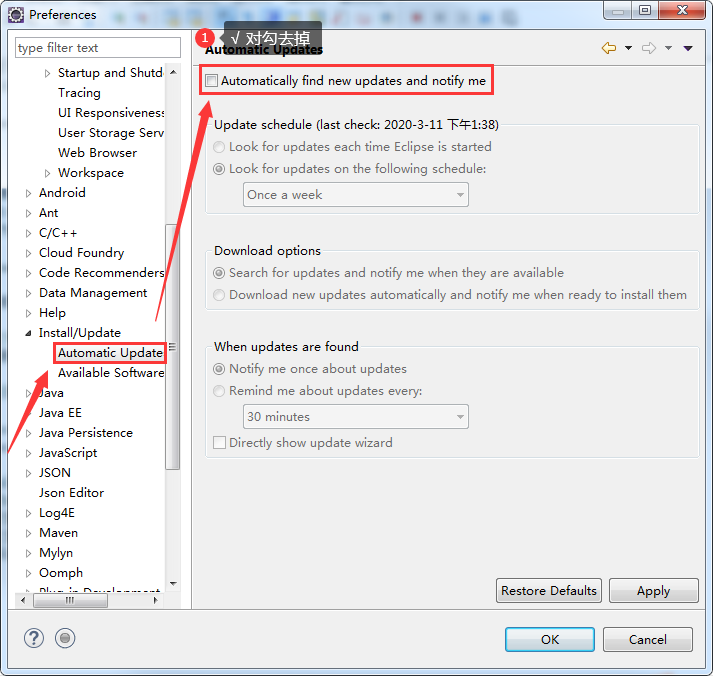
### 1）window -> Preferences



### 2）start and shutdown

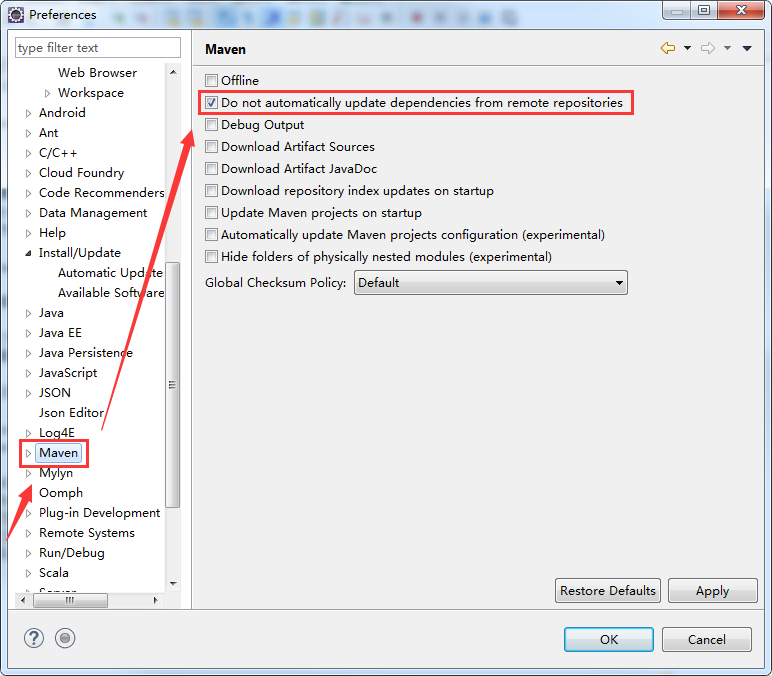


### 3）Automatic Update

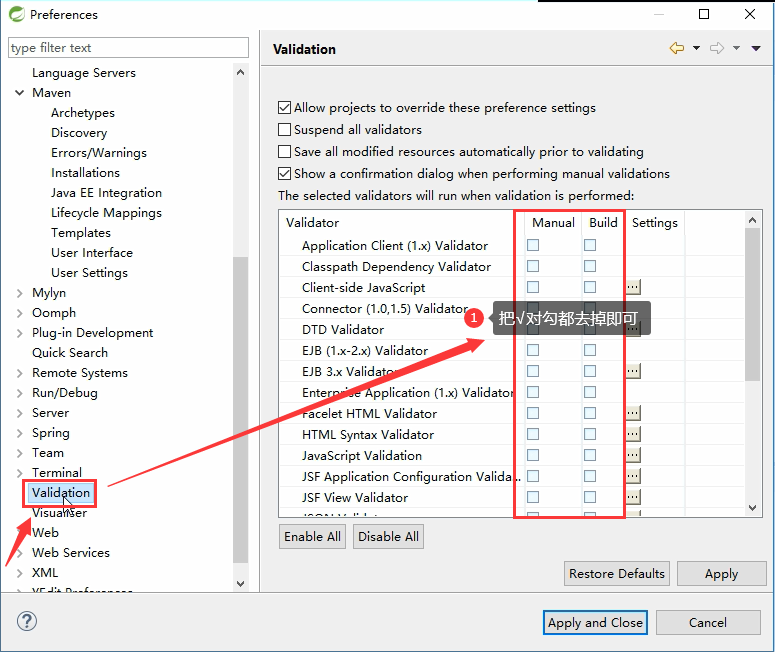


### 4）maven

保持如图即可



### 5）Validation



# 四、Spring IOC 初始化认识

## 1、IOC 和 DI

### 1）IOC(Inversion of Control)：反转控制

反转的是对Bean生命周期管理的控制

在应用程序中的组件需要获取资源时，传统的方式是组件主动的从容器中获取所需要的资源，在这样的模式下开发人员往往需要知道在具体容器中特定资源的获取方式，增加了学习成本，同时降低了开发效率。

反转控制的思想完全颠覆了应用程序组件获取资源的传统方式：反转了资源的获取方向——改由容器主动的将资源推送给需要的组件，开发人员不需要知道容器是如何创建资源对象的，只需要提供接收资源的方式即可，极大的降低了学习成本，提高了开发的效率。这种行为也称为查找的被动形式。

### 2）DI(Dependency Injection)：依赖注入

IOC的另一种表述方式：即组件以一些预先定义好的方式(例如：setter 方法)接受来自于容器的资源注入。相对于IOC而言，这种表述更直接。

对象有容器管理,需要对象时,由容器来提供.

容器可以组合 对象之间的关系.

依赖注入方式:

（1）set方法注入（常用方式）

通过当前对象的set方法将关联对象传递给当前对象来使用.（推荐使用）

（2）构造方法注入

通过当前对象的构造器将关联对象传递给当前对象来使用.

（3）接口注入（不建议使用）

通过当前对象所实现的接口的方法将关联对象传递给当前对象来使用.（不建议使用，依赖性太强）

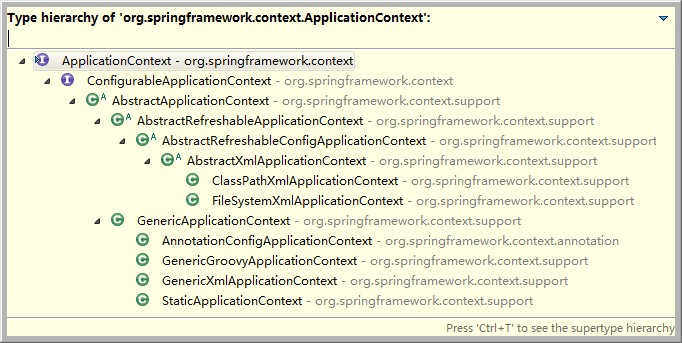
### 3）IOC容器在Spring中的实现

（1）在通过IOC容器读取Bean的实例之前，需要先将IOC容器本身实例化。

（2）Spring提供了IOC容器的两种实现方式

[1]BeanFactory：IOC容器的基本实现，是Spring内部的基础设施，是面向Spring本身的，不是提供给开发人员使用的。

[2]ApplicationContext：BeanFactory的子接口，提供了更多高级特性。面向Spring的使用者，几乎所有场合都使用ApplicationContext而不是底层的BeanFactory。



### 4）ApplicationContext的主要实现类

（1）ClassPathXmlApplicationContext：对应类路径下的XML格式的配置文件

（2）FileSystemXmlApplicationContext：对应文件系统中的XML格式的配置文件

（3）在初始化时就创建单例的bean，也可以通过配置的方式指定创建的Bean是多实例的。

### 5）ConfigurableApplicationContext

（1）是ApplicationContext的子接口，包含一些扩展方法

（2）refresh()和close()让ApplicationContext具有启动、关闭和刷新上下文的能力。

### 6）WebApplicationContext

专门为WEB应用而准备的，它允许从相对于WEB根目录的路径中完成初始化工作

## 2、如何创建IOC容器

### 1）本地JUnit创建

三种方式

|  |
| --- |
| ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");  ApplicationContext ioc = new FileSystemXmlApplicationContext("D:/temp/spring-context.xml");  ApplicationContext ioc = new FileSystemXmlApplicationContext("src/spring-context.xml"); |

### 2）Web运行环境

框架提供了监听器来创建IOC容器对象.

|  |
| --- |
| WebApplicationContext ico = new XmlWebApplicationContext("spring-context.xml"); |

### 3）了解

ApplicationContext接口是在org.springframework.beans.factory.BeanFactory接口的基础上做了扩展.

扩展了国际化,加载外部资源,以及消息等功能.

## 3、如何从IOC容器中获得对象

1）通过Bean的id来获取

通过ID 来取是唯一的

|  |
| --- |
| Book book1 = (Book)ioc.getBean("book"); |

2）通过Bean的类型来获取

当泛型使用，你传什么类型，返回就是什么类型

通过类型来取，当spring-context.xml配置文件出现两个相同的类，它会报异常

通过类型获取对象，一定要保证整个IOC容器中，只有一个这样的对象存储，否则会报错

|  |
| --- |
| Book book3 = ioc.getBean(Book.class); |

比如说这种情况

|  |
| --- |
| <bean id="book" class="com.alex.spring.bean.Book"></bean>  <bean id="book2" class="com.alex.spring.bean.Book"></bean> |

## 4、IOC和DI案例实操（spring\_helloworld02）

### 1）TestSpring

ioc 获取对象

|  |
| --- |
| package com.alex.spring.junit;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.alex.spring.bean.Book;  public class TestSpring {  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");    @Test  public void testIOC() {  //从 IOC 容器中获取对象  Book book1 = (Book) ioc.getBean("book");  System.out.println(book1);    Book book2 = ioc.getBean(Book.class);  System.out.println(book2);    System.out.println(book1 == book2);  }  } |

### 2）spring-context.xml

di set 依赖注入

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">  <bean id="book" class="com.alex.spring.bean.Book">  <property name="id" value="101"></property>  <property name="title" value="三国演义"></property>  <property name="price" value="100.5"></property>  <property name="author" value="罗贯中"></property>  </bean>  </beans> |

**尖叫提示：set注入，name值变大写，前面加set，跟实体类里面的成员变量没有任何关系，只是长得有点像而已，只跟set方法有关系**

# 五、Spring IOC 实验

## 1、IOC实验项目预览

实验1：通过IOC容器创建对象，并为属性赋值★

实验2：根据bean的类型从IOC容器中获取bean的实例★

实验3：通过构造器为bean的属性赋值

实验4：引用其他bean★

实验5：使用List类型的集合属性

实验6：使用Map类型的集合属性

实验7：使用prop子元素为Properties类型的属性赋值

实验8：配置FactoryBean★

实验9：测试bean的作用域，分别创建单实例和多实例的bean★

实验10：引用外部属性文件★

实验11：基于XML的自动装配

实验12：通过注解分别创建Dao、Service、Controller★

实验13：使用context:include-filter指定扫描包时要包含的类★

实验14：使用context:exclude-filter指定扫描包时不包含的类★

实验15：使用@Autowired注解实现根据类型实现自动装配★

实验16：如果资源类型的bean不止一个，默认根据@Autowired注解标记的成员变量名作为id查找bean，进行装配★

实验17：如果根据成员变量名作为id还是找不到bean，可以使用@Qualifier注解明确指定目标bean的id★

实验18：在方法的形参位置使用@Qualifier注解

实验19：@Autowired注解的required属性指定某个属性允许不被设置★

## 2、实验1（spring\_ioc01）

通过IOC容器创建对象，并为属性赋值★

创建对象并赋值

|  |
| --- |
| <bean id="book" class="com.alex.spring.bean.Book">  <property name="id" value="101"></property>  <property name="title" value="三国演义"></property>  <property name="price" value="100.5"></property>  <property name="author" value="罗贯中"></property>  </bean> |

**尖叫提示：我们赋予的值，被框架作为String字符串来使用了，但是它会根据属性的类型，帮我们做类型的转换，前提是必须能转换成功，转换不成功，它会报异常**

## 3、实验2（spring\_ioc02）

根据bean的类型从IOC容器中获取bean的实例★

|  |
| --- |
| //实验2：根据bean的类型从IOC容器中获取bean的实例★  Book book2 = ioc.getBean(Book.class);  System.out.println(book2); |

**尖叫提示：好处是不需要强制转换了**

## 4、实验3（spring\_ioc03）

通过构造器为bean的属性赋值

### 1）创建构造器

|  |
| --- |
| public Book() {  }  public Book(Integer id, String title, double price, String author) {  this.id = id;  this.title = title;  this.price = price;  this.author = author;  } |

**尖叫提示：通过构造器为bean属性赋值，需要创建有参构造器，创建了有参构造器，如果不主动创建无参构造器，就不能 <bean id="book" class="com.alex.spring.bean.Book">**

**</bean> 这样声明了**

### 2）spring-context.xml

|  |
| --- |
| <!-- 实验3：通过构造器为bean的属性赋值 -->  <bean id="book2" class="com.alex.spring.bean.Book">  <constructor-arg value="102"></constructor-arg>  <constructor-arg value="西游记"></constructor-arg>  <constructor-arg value="103.9"></constructor-arg>  <constructor-arg value="吴承恩"></constructor-arg>  </bean> |

### 3）TestSpring.java

|  |
| --- |
| @Test  public void testIOC2() {  //从 IOC 容器中获取对象  /\*实验3：通过构造器为bean的属性赋值\*/  Book book = (Book) ioc.getBean("book2");  System.out.println(book);  } |

### 4）error

|  |
| --- |
| <!-- 实验3：通过构造器为bean的属性赋值 -->  <bean id="book2" class="com.alex.spring.bean.Book">  <constructor-arg value="102" type="java.lang.Integer" index="0"></constructor-arg>  <constructor-arg value="西游记"></constructor-arg>  <constructor-arg value="103.9"></constructor-arg>  <constructor-arg value="吴承恩"></constructor-arg>  </bean> |

**尖叫提示：当有多个相同参数的构造器的时候，可以指定参数类型，下表索引是从0开始的**

## 5、实验4（非常重要spring\_ioc04）

引用其他bean★



### 1）spring-context.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">    <!-- 实验4：引用其他bean★ -->  <bean id="userController" class="com.alex.spring.controller.UserController" >  <property name="userServiceImpl" ref="userServiceImpl"></property>  </bean>    <bean id="userServiceImpl" class="com.alex.spring.service.UserServiceImpl"></bean>  </beans> |

### 2）UserServiceImpl

|  |
| --- |
| package com.alex.spring.service;  public class UserServiceImpl {  } |

### 3）UserController

|  |
| --- |
| package com.alex.spring.controller;  import com.alex.spring.service.UserServiceImpl;  public class UserController {  //以前  //UserServiceImpl userServiceImpl = new UserServiceImpl();    UserServiceImpl userServiceImpl;  public UserServiceImpl getUserServiceImpl() {  return userServiceImpl;  }  public void setUserServiceImpl(UserServiceImpl userServiceImpl) {  this.userServiceImpl = userServiceImpl;  }  } |

### 4）TestSpring

|  |
| --- |
| package com.alex.spring.junit;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.alex.spring.controller.UserController;  public class TestSpring {  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");    @Test  public void testIOC() {  UserController userController = (UserController) ioc.getBean("userController");  System.out.println(userController.getUserServiceImpl());  }  } |

## 6、实验 5/6/7（spring\_ioc05）

实验5

使用List类型的集合属性

实验6

使用Map类型的集合属性

实验7

使用prop子元素为Properties类型的属性赋值

### 1）Book

|  |
| --- |
| package com.alex.spring.bean;  public class Book {  private Integer id;  private String title;  private double price;  private String author;  public Book() {  }  public Book(Integer id, String title, double price, String author) {  this.id = id;  this.title = title;  this.price = price;  this.author = author;  }  public Integer getId() {  return id;  }  public void setId(Integer id) {  this.id = id;  }  public String getTitle() {  return title;  }  public void setTitle(String title) {  this.title = title;  }  public double getPrice() {  return price;  }  public void setPrice(double price) {  this.price = price;  }  public String getAuthor() {  return author;  }  public void setAuthor(String author) {  this.author = author;  }  @Override  public String toString() {  return "Book [id=" + id + ", title=" + title + ", price=" + price + ", author=" + author + "]";  }  } |

### 2）BookShop

|  |
| --- |
| package com.alex.spring.bean;  import java.util.List;  import java.util.Map;  import java.util.Properties;  public class BookShop {    private List<Book> bookList;  private Map<Integer, Book> bookMaps;  private Properties propertis; //继承了Hashtable  //Hashtable 是线程安全的，虽然慢点，HashMap是线程不安全的， 在并发的时候会有体现  public List<Book> getBookList() {  return bookList;  }  public void setBookList(List<Book> bookList) {  this.bookList = bookList;  }  public Map<Integer, Book> getBookMaps() {  return bookMaps;  }  public void setBookMaps(Map<Integer, Book> bookMaps) {  this.bookMaps = bookMaps;  }  public Properties getPropertis() {  return propertis;  }  public void setPropertis(Properties propertis) {  this.propertis = propertis;  }  @Override  public String toString() {  return "BookShop [bookList=" + bookList + ", bookMaps=" + bookMaps + ", propertis=" + propertis + "]";  }  } |

### 3）TestSpring

|  |
| --- |
| package com.alex.spring.junit;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.alex.spring.bean.BookShop;  public class TestSpring {  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");    /\* 实验5：使用List类型的集合属性  实验6：使用Map类型的集合属性  实验7：使用prop子元素为Properties类型的属性赋值\*/  @Test  public void testIOC() {  BookShop bookShop = (BookShop) ioc.getBean("bookShop");  System.out.println(bookShop);  }  } |

### 4）spring-context.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">  <!-- 实验5：使用List类型的集合属性  实验6：使用Map类型的集合属性  实验7：使用prop子元素为Properties类型的属性赋值 -->  <bean id="bookShop" class="com.alex.spring.bean.BookShop">  <property name="bookList">  <list>  <ref bean="book" />  <ref bean="book2" />  </list>  </property>  <property name="bookMaps">  <map>  <entry key="111" value-ref="book"></entry>  <entry key="222" value-ref="book2"></entry>  </map>  </property>  <property name="propertis">  <props>  <prop key="username">root</prop>  <prop key="password">123</prop>  </props>  </property>  </bean>  <bean id="book" class="com.alex.spring.bean.Book">  <property name="id" value="101"></property>  <property name="title" value="三国演义"></property>  <property name="price" value="100.5"></property>  <property name="author" value="罗贯中"></property>  </bean>  <bean id="book2" class="com.alex.spring.bean.Book">  <constructor-arg value="102" type="java.lang.Integer"  index="0"></constructor-arg>  <constructor-arg value="西游记"></constructor-arg>  <constructor-arg value="103.9"></constructor-arg>  <constructor-arg value="吴承恩"></constructor-arg>  </bean>  </beans> |

## 7、实验8（spring\_ioc06）

配置FactoryBean ★

### 1）Book

|  |
| --- |
| package com.alex.spring.bean;  public class Book {  private Integer id;  private String title;  private double price;  private String author;  public Book() {  }  public Book(Integer id, String title, double price, String author) {  this.id = id;  this.title = title;  this.price = price;  this.author = author;  }  public Integer getId() {  return id;  }  public void setId(Integer id) {  this.id = id;  }  public String getTitle() {  return title;  }  public void setTitle(String title) {  this.title = title;  }  public double getPrice() {  return price;  }  public void setPrice(double price) {  this.price = price;  }  public String getAuthor() {  return author;  }  public void setAuthor(String author) {  this.author = author;  }  @Override  public String toString() {  return "Book [id=" + id + ", title=" + title + ", price=" + price + ", author=" + author + "]";  }  } |

### 2）MyFactoryBean

|  |
| --- |
| package com.alex.spring.factory;  import org.springframework.beans.factory.FactoryBean;  import com.alex.spring.bean.Book;  /\*\*  \* FactoryBean 用于Spring框架与第三方框架做集成的  \* FactoryBean 提供了一种集成的能力，就是可以将任何第三方的东西，往Spring里面插入  \*/  public class MyFactoryBean implements FactoryBean<Book>{  @Override  public Book getObject() throws Exception {  //集成的时候，对象我们自己来创建了，但是创建后的对象，教给容器来处理了  //spring 框架集成第三方框架时，可以通过getObject（）方法，来创建对象，并将这个对象返回，交给容器进行管理  Book book = new Book();  book.setId(105);  book.setTitle("红楼梦");  book.setAuthor("曹雪芹");  book.setPrice(123.4);  return book;  }  //getObject()方法返回什么，这里就返回什么类型  @Override  public Class<?> getObjectType() {  return Book.class;  }  //设置为 true，就是单例的，从容器里面取  //设置为false 就是从getObject（）方法里面取  @Override  public boolean isSingleton() {  return true;  }  } |

### 3）TestSpring

|  |
| --- |
| package com.alex.spring.junit;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.alex.spring.bean.Book;  public class TestSpring {  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");    /\*实验8：配置FactoryBean★\*/  @Test  public void testIOC() {  Book book = (Book) ioc.getBean("book");  System.out.println(book);  }  } |

### 4）spring-context.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">  <!-- 实验8：配置FactoryBean★ -->  <bean id="book" class="com.alex.spring.factory.MyFactoryBean"></bean>  </beans> |

**尖叫提示：通过这个案例，我们可以知道，声明bean的时候，可以声明什么得到什么，也可以声明A得到B**

## 8、实验9（spring\_ioc07）

测试bean的作用域，分别创建单实例和多实例的bean★

### 1）Book

|  |
| --- |
| package com.alex.spring.bean;  public class Book {  private Integer id;  private String title;  private double price;  private String author;  public Book() {  }  public Book(Integer id, String title, double price, String author) {  this.id = id;  this.title = title;  this.price = price;  this.author = author;  }  public Integer getId() {  return id;  }  public void setId(Integer id) {  this.id = id;  }  public String getTitle() {  return title;  }  public void setTitle(String title) {  this.title = title;  }  public double getPrice() {  return price;  }  public void setPrice(double price) {  this.price = price;  }  public String getAuthor() {  return author;  }  public void setAuthor(String author) {  this.author = author;  }  @Override  public String toString() {  return "Book [id=" + id + ", title=" + title + ", price=" + price + ", author=" + author + "]";  }  } |

### 2）TestSpring

|  |
| --- |
| package com.alex.spring.junit;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.alex.spring.bean.Book;  public class TestSpring {  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");    /\*实验9：测试bean的作用域，分别创建单实例和多实例的bean★\*/  @Test  public void testIOC() {  //多例  Book book1 = (Book) ioc.getBean("book1");  Book book2 = (Book) ioc.getBean("book1");  System.out.println(book1 == book2);    //单例  Book book3 = (Book) ioc.getBean("book2");  Book book4 = (Book) ioc.getBean("book2");  System.out.println(book3 == book4);  }  } |

### 3）spring-context.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">  <!-- 实验9：测试bean的作用域，分别创建单实例和多实例的bean★ -->  <!--  scope="singleton" 默认单例管理对象  scope="prototype" 表示多列创建对象，每一次调用getBean()方法都会创建一个新对象    那些对象需要单例创建：如果对象没有成员变量，只是拥有方法。  例如 service、Dao、...    那些对象需要多例创建：如果对象有成员变量，而且每一次使用对象时，变量值都会发生变化。  例如 Page、User、Book    对象什么时候被创建：  单例情况下，在服务器启动，即IOC容器被创建时，就将所有的Bean对象创建好  多例情况下，每一次调用getBean()方法时才进行创建  -->  <bean id="book1" class="com.alex.spring.bean.Book" scope="prototype"></bean>    <bean id="book2" class="com.alex.spring.bean.Book" scope="singleton"></bean>    </beans> |

## 9、实验10（spring\_ioc08）

引用外部属性文件★

### 1）TestSpring

|  |
| --- |
| package com.alex.spring.junit;  import java.sql.Connection;  import java.sql.SQLException;  import javax.sql.DataSource;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class TestSpring {  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");    /\*实验10：引用外部属性文件★\*/  @Test  public void testIOC() throws SQLException {  DataSource dataSource = (DataSource) ioc.getBean("dataSource");  Connection connection = dataSource.getConnection();  //拿到的是代理对象  //com.mchange.v2.c3p0.impl.NewProxyConnection@2d2e5f00  System.out.println(connection);  //并不是销毁，而是归还  connection.close();  }  } |

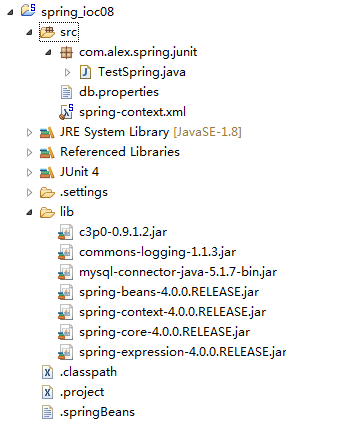
### 2）db.properties

|  |
| --- |
| driverClass=com.mysql.jdbc.Driver  jdbcUrl=jdbc:mysql://localhost:3306/test  user=root  password=root  initialPoolSize=2  minPoolSize=4  maxPoolSize=10 |

### 3）spring-context.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd">  <!-- 加载外部属性资源文件 -->  <context:property-placeholder location="classpath:/db.properties" />  <!-- 配置C3P0数据源 -->  <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">  <property name="driverClass" value="${driverClass}"></property>  <property name="jdbcUrl" value="${jdbcUrl}"></property>  <property name="user" value="${user}"></property>  <property name="password" value="${password}"></property>  <property name="initialPoolSize" value="${initialPoolSize}"></property>  <property name="minPoolSize" value="${minPoolSize}"></property>  <property name="maxPoolSize" value="${maxPoolSize}"></property>  </bean>  </beans> |

### 4）导入Jar包



### 5）c3p0连接池配置

|  |
| --- |
| <!-- c3p0连接池配置 -->  <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">  <property name="user" value="${username}"/>  <property name="password" value="${password}"/>  <property name="driverClass" value="${driver\_class}"/>  <property name="jdbcUrl" value="${url}"/>  <!--连接池中保留的最大连接数。默认值: 15 -->  <property name="maxPoolSize" value="20"/>  <!-- 连接池中保留的最小连接数，默认为：3-->  <property name="minPoolSize" value="2"/>  <!-- 初始化连接池中的连接数，取值应在minPoolSize与maxPoolSize之间，默认为3-->  <property name="initialPoolSize" value="2"/>  <!--最大空闲时间，60秒内未使用则连接被丢弃。若为0则永不丢弃。默认值: 0 -->  <property name="maxIdleTime">60</property>  <!-- 当连接池连接耗尽时，客户端调用getConnection()后等待获取新连接的时间，超时后将抛出SQLException，如设为0则无限期等待。单位毫秒。默认: 0 -->  <property name="checkoutTimeout" value="3000"/>  <!--当连接池中的连接耗尽的时候c3p0一次同时获取的连接数。默认值: 3 -->  <property name="acquireIncrement" value="2"/>  <!--定义在从数据库获取新连接失败后重复尝试的次数。默认值: 30 ；小于等于0表示无限次-->  <property name="acquireRetryAttempts" value="0"/>  <!--重新尝试的时间间隔，默认为：1000毫秒-->  <property name="acquireRetryDelay" value="1000" />  <!--关闭连接时，是否提交未提交的事务，默认为false，即关闭连接，回滚未提交的事务 -->  <property name="autoCommitOnClose">false</property>  <!--c3p0将建一张名为Test的空表，并使用其自带的查询语句进行测试。如果定义了这个参数那么属性preferredTestQuery将被忽略。你不能在这张Test表上进行任何操作，它将只供c3p0测试使用。默认值: null -->  <property name="automaticTestTable">Test</property>  <!--如果为false，则获取连接失败将会引起所有等待连接池来获取连接的线程抛出异常，但是数据源仍有效保留，并在下次调用getConnection()的时候继续尝试获取连接。如果设为true，那么在尝试获取连接失败后该数据源将申明已断开并永久关闭。默认: false-->  <property name="breakAfterAcquireFailure">false</property>  <!--每60秒检查所有连接池中的空闲连接。默认值: 0，不检查 -->  <property name="idleConnectionTestPeriod">60</property>  <!--c3p0全局的PreparedStatements缓存的大小。如果maxStatements与maxStatementsPerConnection均为0，则缓存不生效，只要有一个不为0，则语句的缓存就能生效。如果默认值: 0-->  <property name="maxStatements">100</property>  <!--maxStatementsPerConnection定义了连接池内单个连接所拥有的最大缓存statements数。默认值: 0 -->  <property name="maxStatementsPerConnection"></property>  </bean> |

### 6）c3p0 图



## 10、实验11（spring\_ioc09）

基于XML的自动装配，真实的开发的时候，不使用，使用注解的方式

### 1）UserController

|  |
| --- |
| package com.alex.spring.controller;  import com.alex.spring.service.UserServiceImpl;  public class UserController {  //以前  //UserServiceImpl userServiceImpl = new UserServiceImpl();    UserServiceImpl userServiceImpl;  public UserServiceImpl getUserServiceImpl() {  return userServiceImpl;  }  public void setUserServiceImpl(UserServiceImpl userServiceImpl) {  this.userServiceImpl = userServiceImpl;  }  } |

### 2）UserServiceImpl

|  |
| --- |
| package com.alex.spring.service;  public class UserServiceImpl {  } |

### 3）TestSpring

|  |
| --- |
| package com.alex.spring.junit;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.alex.spring.controller.UserController;  public class TestSpring {  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");    /\* 实验11：基于XML的自动装配 \*/  @Test  public void testIOC() {  UserController userController = (UserController) ioc.getBean("userController");  System.out.println(userController.getUserServiceImpl());  }  } |

### 4）spring-context.xml

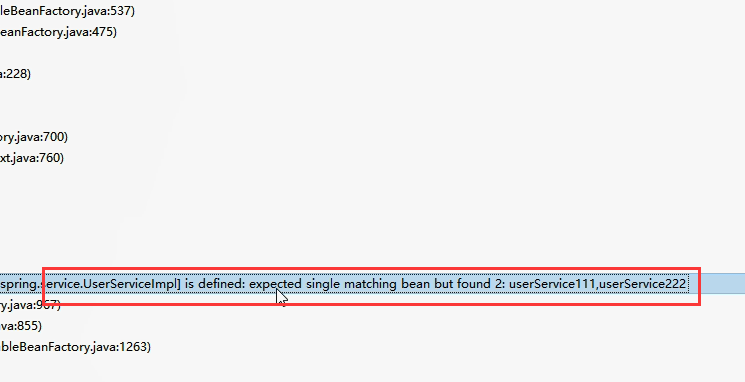
|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"  default-autowire="default"  >  <!-- 实验11：基于XML的自动装配 -->  <!--  autowire="byName"  表示根据属性的名称从IOC容器中查找对象,如果存在,就自动注入.如果不存在,则为null  autowire="byType"  表示根据属性的类型从IOC容器中查找对象,如果存在则进行装配  -->  <bean id="userController" class="com.alex.spring.controller.UserController" autowire="byName">  <!-- 手动装配:如果需要装配的属性值特别多,比较麻烦,可以采用自动装配. -->  <!-- <property name="userServiceImpl" ref="userServiceImpl"></property> -->  </bean>  <bean id="userServiceImpl" class="com.alex.spring.service.UserServiceImpl"></bean>  </beans> |

### 5）autowire="byType"

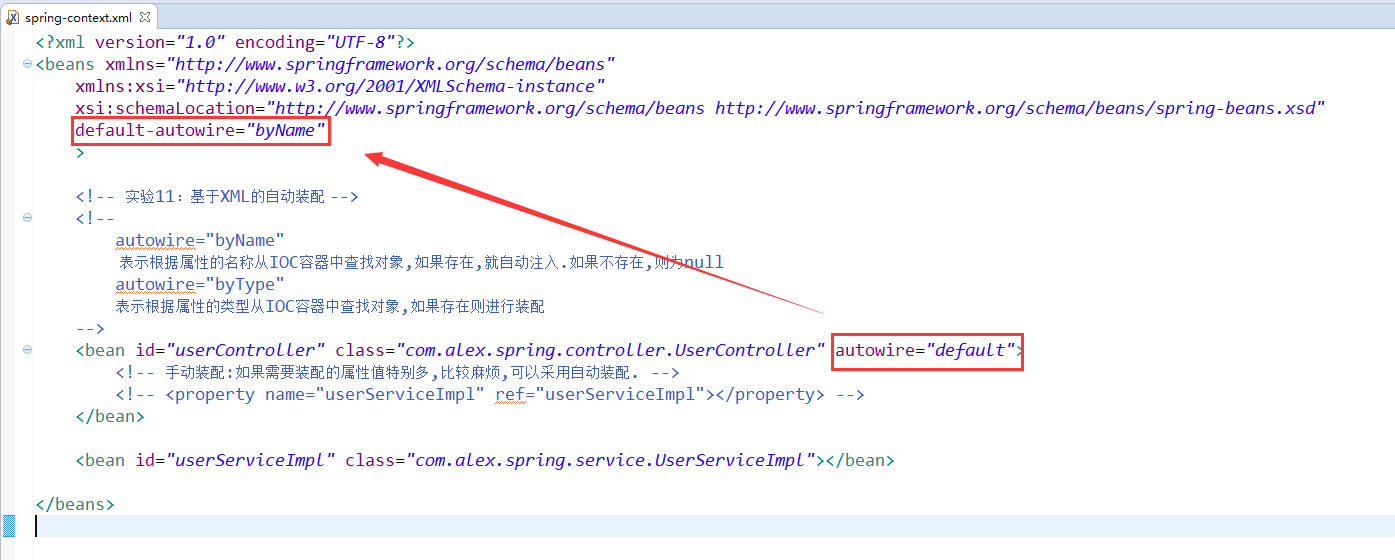
当使用autowire="byType" 的时候，出现了两个相同的类型，就会报异常

<bean id="userServiceImpl1111" class="com.alex.spring.service.UserServiceImpl"></bean>

<bean id="userServiceImpl2222" class="com.alex.spring.service.UserServiceImpl"></bean>



### 6）autowire="default"



**尖叫提示：autowire 不写，默认就是default，这个默认指的是<beans>里面的default-autowire里面的值，default-autowire不写，默认也是default，default默认就是不自动装配**

## 11、实验12（spring\_ioc10）

通过注解分别创建entity、Dao、Service、Controller★

Spring 有Spring自己的容器

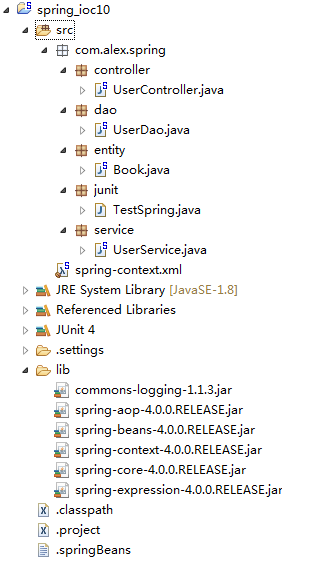
Spring MVC 有Spring MVC 自己的容器

真实项目开发的时候，Controller往往交由Spring MVC 的容器

Dao、Service交由Spring 容器

### 1）导入jar 包示意图

添加新的jar包spring-aop-4.0.0.RELEASE.jar



### 2）UserController

|  |
| --- |
| package com.alex.spring.controller;  import org.springframework.context.annotation.Scope;  import org.springframework.stereotype.Controller;  //@Controller 注解是专门用于声明表现层的类对象的，默认的id位：类名称首字母小写u serController  //自定义，不使用默认的id  //@Controller(value="userController")  @Controller  //prototype 表示多例  @Scope(value="prototype")//创建方式注解:默认是单例的@Scope(value="singleton")  public class UserController {    } |

### 3）UserDao

|  |
| --- |
| package com.alex.spring.dao;  import org.springframework.stereotype.Repository;  //@Repository(value="userDao")  @Repository  public class UserDao {  } |

### 4）Book

|  |
| --- |
| package com.alex.spring.entity;  import org.springframework.stereotype.Component;  //@Component 声明不知道是哪个层的bean对象，比如说配置类，通用性比较强  //@Component 是比较通用的注解，可以代替分层注解使用  @Component  public class Book {    private Integer id;  private String title;  private double price;  private String author;  public Book() {  }  public Book(Integer id, String title, double price, String author) {  this.id = id;  this.title = title;  this.price = price;  this.author = author;  }  public Integer getId() {  return id;  }  public void setId(Integer id) {  this.id = id;  }  public String getTitle() {  return title;  }  public void setTitle(String title) {  this.title = title;  }  public double getPrice() {  return price;  }  public void setPrice(double price) {  this.price = price;  }  public String getAuthor() {  return author;  }  public void setAuthor(String author) {  this.author = author;  }  @Override  public String toString() {  return "Book [id=" + id + ", title=" + title + ", price=" + price + ", author=" + author + "]";  }  } |

### 5）UserService

|  |
| --- |
| package com.alex.spring.service;  import org.springframework.stereotype.Service;  //@Service(value="userService")  @Service  public class UserService {  } |

### 6）spring-context.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd">  <!-- 设置扫描包 -->  <context:component-scan base-package="com.alex.spring"></context:component-scan>    </beans> |

### 7）TestSpring

|  |
| --- |
| package com.alex.spring.junit;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.alex.spring.controller.UserController;  import com.alex.spring.dao.UserDao;  import com.alex.spring.entity.Book;  import com.alex.spring.service.UserService;  public class TestSpring {  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");    //1.分层注解使用时需要增加aop包.  //Caused by: java.lang.NoClassDefFoundError: org/springframework/aop/TargetSource  //2.如果getBean()时,采用XML配置文件,获取不到对象,会返回null;采用Annotation获取不要回抛异常.  //org.springframework.beans.factory.NoSuchBeanDefinitionException: No bean named 'userServiceImpl' is defined  @Test  public void testIOC() {  UserController userController1 = (UserController) ioc.getBean("userController");  System.out.println(userController1);  UserController userController2 = (UserController) ioc.getBean("userController");  System.out.println(userController2);  UserDao userDao = (UserDao) ioc.getBean("userDao");  System.out.println(userDao);  UserService userService = (UserService) ioc.getBean("userService");  System.out.println(userService);  Book book = (Book) ioc.getBean("book");  System.out.println(book);  }  } |

## 12、实验13（spring\_ioc11）

使用context:include-filter指定扫描包时要包含的类★

### 1）spring-context.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd">    <!-- 设置扫描包 -->  <!-- 包含过滤 -->  <!-- use-default-filters="false" 默认是true, 设置为false，表示对指定包全都不扫描了-->  <context:component-scan base-package="com.alex.spring" use-default-filters="false">  <context:include-filter type="annotation" expression="org.springframework.stereotype.Service"/>  <context:include-filter type="assignable" expression="com.alex.spring.dao.UserDao"/>  </context:component-scan>  </beans> |

## 13、实验14（spring\_ioc12）

使用context:exclude-filter指定扫描包时不包含的类★

### 1）spring-context.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd">  <!-- 设置扫描包 -->  <!-- 排除过滤 -->  <context:component-scan base-package="com.alex.spring">  <!-- 只要是 @Service 的就排除 -->  <context:exclude-filter type="annotation" expression="org.springframework.stereotype.Service" />  <!-- 排除具体某一个类 -->  <context:exclude-filter type="assignable" expression="com.alex.spring.dao.UserDao" />  </context:component-scan>  </beans> |

## 14、实验15（spring\_ioc13）

使用@Autowired注解实现根据类型实现自动装配★

### 1）UserController

|  |
| --- |
| package com.alex.spring.controller;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Controller;  import com.alex.spring.service.UserServiceImpl1;  @Controller  public class UserController {  /\*\*  \* @Autowired 自动装配省略set方法  \*/  @Autowired  private UserServiceImpl1 userServiceImpl1;  } |

### 2）UserServiceImpl1

|  |
| --- |
| package com.alex.spring.service;  import org.springframework.stereotype.Service;  @Service  public class UserServiceImpl1{  } |

### 3）TestSpring

Debug运行查看是否自动装配

|  |
| --- |
| package com.alex.spring.junit;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.alex.spring.controller.UserController;  public class TestSpring {  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");    //1.分层注解使用时需要增加aop包.  //Caused by: java.lang.NoClassDefFoundError: org/springframework/aop/TargetSource  //2.如果getBean()时,采用XML配置文件,获取不到对象,会返回null;采用Annotation获取不要回抛异常.  //org.springframework.beans.factory.NoSuchBeanDefinitionException: No bean named 'userServiceImpl' is defined  @Test  public void testIOC() {  //debug运行，查看UserService是否进来了  UserController userController = (UserController) ioc.getBean("userController");  System.out.println(userController);  }  } |

### 4）spring-context.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd">    <!-- 设置扫描包 -->  <context:component-scan base-package="com.alex.spring"></context:component-scan>  </beans> |

## 15、实验16（spring\_ioc14）

如果资源类型的bean不止一个，默认根据@Autowired注解标记的成员变量名作为id查找bean，进行装配★

### 1）UserController

|  |
| --- |
| package com.alex.spring.controller;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Controller;  import com.alex.spring.service.UserService;  @Controller  public class UserController {    /\*\*  @Autowired 自动装配注解  演示各种异常情况  原理:  首先采用byType进行自动装配.  ①如果只查找到一个,自动装配  ②如果一个都没查找到,会报异常.  ③如果查找到多个:报异常.  如果查找到 多个会抛异常,框架会通过byName再进行查找.  ①如果查找到的多个bean的名字,与属性名称都不一样,会报错.  如果不希望报错,可以通过@Qualifier(value="id")指定id来进行自动装配.  如果指定的名称不存在,依然报错.(比如说非诚勿扰，有两个女生都喜欢你，你非选心动女生，心动女生又不喜欢你)  如果不希望报错,可以设置required=false,就不进行装配.  ②如果查找到的多个bean的名称之一,与属性名称一样,就将其进行装配  如果希望将查找到的多个对象都进行装配,可以采用数组,或集合(一定要指定泛型)来装配多个对象.    \*/  @Autowired  private UserService userService;    //先通过类型装配，再通过名称装配  //@Autowired  //private UserService userServiceImpl1;  } |

### 2）UserService

|  |
| --- |
| package com.alex.spring.service;  public interface UserService {  } |

### 3）UserServiceImpl1

|  |
| --- |
| package com.alex.spring.service;  import org.springframework.stereotype.Service;  @Service  public class UserServiceImpl1 implements UserService{  } |

### 4）UserServiceImpl2

|  |
| --- |
| package com.alex.spring.service;  import org.springframework.stereotype.Service;  @Service  public class UserServiceImpl2 implements UserService{  } |

### 5）TestSpring

|  |
| --- |
| package com.alex.spring.junit;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.alex.spring.controller.UserController;  public class TestSpring {  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");    //1.分层注解使用时需要增加aop包.  //Caused by: java.lang.NoClassDefFoundError: org/springframework/aop/TargetSource  //2.如果getBean()时,采用XML配置文件,获取不到对象,会返回null;采用Annotation获取不要回抛异常.  //org.springframework.beans.factory.NoSuchBeanDefinitionException: No bean named 'userServiceImpl' is defined  @Test  public void testIOC() {  //debug运行，查看UserService是否进来了  UserController userController = (UserController) ioc.getBean("userController");  System.out.println(userController);  }  } |

### 6）spring-context.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd">    <!-- 设置扫描包 -->  <context:component-scan base-package="com.alex.spring"></context:component-scan>  </beans> |

## 16、实验17（spring\_ioc15）

如果根据成员变量名作为id还是找不到bean，可以使用@Qualifier注解明确指定目标bean的id★

### 1）UserController

|  |
| --- |
| package com.alex.spring.controller;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.beans.factory.annotation.Qualifier;  import org.springframework.stereotype.Controller;  import com.alex.spring.service.UserService;  @Controller  public class UserController {  /\*\*  @Autowired 自动装配注解  演示各种异常情况  原理:  首先采用byType进行自动装配.  ①如果只查找到一个,自动装配  ②如果一个都没查找到,会报异常.  ③如果查找到多个:报异常.  如果查找到 多个会抛异常,框架会通过byName再进行查找.  ①如果查找到的多个bean的名字,与属性名称都不一样,会报错.  如果不希望报错,可以通过@Qualifier(value="id")指定id来进行自动装配.  如果指定的名称不存在,依然报错.(比如说非诚勿扰，有两个女生都喜欢你，你非选心动女生，心动女生又不喜欢你)  如果不希望报错,可以设置required=false,就不进行装配.  ②如果查找到的多个bean的名称之一,与属性名称一样,就将其进行装配  如果希望将查找到的多个对象都进行装配,可以采用数组,或集合(一定要指定泛型)来装配多个对象.    \*/  // 指定装配  @Autowired  @Qualifier(value = "userServiceImpl1")  private UserService userService;  } |

尖叫提示：其它内容同实验16是一样的

## 17、实验18（spring\_ioc16）

在方法上使用注解

在方法的形参位置使用@Qualifier注解

### 1）UserController

|  |
| --- |
| package com.alex.spring.controller;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.beans.factory.annotation.Qualifier;  import org.springframework.stereotype.Controller;  import com.alex.spring.service.UserService;  @Controller  public class UserController {  private UserService userService;  // 注解可以防止在成员变量上，也可以放置在方法上  @Autowired  @Qualifier(value = "UserServiceImpl1")  public void setUserService(UserService userService) {  this.userService = userService;  }  // 注解可以防止在成员变量上，也可以放置在方法上,还可以放置在参数上  // @Autowired  // public void setUserService(@Qualifier(value="UserServiceImpl1") UserService userService) {  // this.userService = userService;  // }  } |

尖叫提示：其它内容同实验16是一样的

## 18、实验19（spring\_ioc17）

@Autowired注解的required属性指定某个属性允许不被设置★

### 1）UserController

|  |
| --- |
| package com.alex.spring.controller;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.beans.factory.annotation.Qualifier;  import org.springframework.stereotype.Controller;  import com.alex.spring.service.UserService;  import com.alex.spring.service.UserServiceImpl1;    @Controller  public class UserController {  /\*\*  @Autowired 自动装配注解  演示各种异常情况  原理:  首先采用byType进行自动装配.  ①如果只查找到一个,自动装配  ②如果一个都没查找到,会报异常.  ③如果查找到多个:报异常.  如果查找到 多个会抛异常,框架会通过byName再进行查找.  ①如果查找到的多个bean的名字,与属性名称都不一样,会报错.  如果不希望报错,可以通过@Qualifier(value="id")指定id来进行自动装配.  如果指定的名称不存在,依然报错.(比如说非诚勿扰，有两个女生都喜欢你，你非选心动女生，心动女生又不喜欢你)  如果不希望报错,可以设置required=false,就不进行装配.  ②如果查找到的多个bean的名称之一,与属性名称一样,就将其进行装配  如果希望将查找到的多个对象都进行装配,可以采用数组,或集合(一定要指定泛型)来装配多个对象.  \*/  //能装配就装配，不能装配就算了，别报错  @Autowired(required=false)  private UserService userService;  } |

尖叫提示：其它内容同实验16是一样的

## 19、实验20（spring\_ioc18）

如果希望将查找到的多个对象都进行装配,可以采用数组,或集合(一定要指定泛型)来装配多个对象

### 1）UserController

|  |
| --- |
| package com.alex.spring.controller;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Controller;  import com.alex.spring.service.UserService;    @Controller  public class UserController {  /\*\*  @Autowired 自动装配注解  演示各种异常情况  原理:  首先采用byType进行自动装配.  ①如果只查找到一个,自动装配  ②如果一个都没查找到,会报异常.  ③如果查找到多个:报异常.  如果查找到 多个会抛异常,框架会通过byName再进行查找.  ①如果查找到的多个bean的名字,与属性名称都不一样,会报错.  如果不希望报错,可以通过@Qualifier(value="id")指定id来进行自动装配.  如果指定的名称不存在,依然报错.(比如说非诚勿扰，有两个女生都喜欢你，你非选心动女生，心动女生又不喜欢你)  如果不希望报错,可以设置required=false,就不进行装配.  ②如果查找到的多个bean的名称之一,与属性名称一样,就将其进行装配  如果希望将查找到的多个对象都进行装配,可以采用数组,或集合(一定要指定泛型)来装配多个对象.  \*/  //数组装配多个  @Autowired  private UserService[] userService;    //集合装配多个  //@Autowired  //private List<UserService> userService;    } |

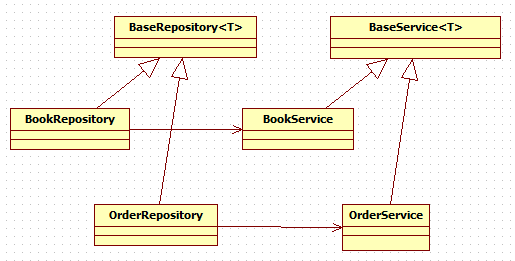
尖叫提示：其它内容同实验16是一样的

## 20、实验21（spring\_ioc19）

泛型依赖注入

### 1）简介

Spring 4.x中可以为子类注入子类对应的泛型类型的成员变量的引用。



### 2）案例实操

#### 1）BaseService

|  |
| --- |
| package com.alex.spring.bean;  import org.springframework.beans.factory.annotation.Autowired;  public class BaseService<T> {    @Autowired  private BaseRepository<T> repository;    public void add() {  repository.save();  }  } |

#### 2）BaseRepository

|  |
| --- |
| package com.alex.spring.bean;  public class BaseRepository<T> {    public void save() {  System.out.println("Saved by BaseRepository");  }  } |

#### 3）UserRepository

|  |
| --- |
| package com.alex.spring.dao;  import org.springframework.stereotype.Repository;  import com.alex.spring.bean.BaseRepository;  import com.alex.spring.entity.User;  @Repository  public class UserRepository extends BaseRepository<User>{    public void save() {  System.out.println("Saved by UserRepository");  }  } |

#### 4）User

|  |
| --- |
| package com.alex.spring.entity;  import org.springframework.stereotype.Component;  @Component  public class User {  } |

#### 5）UserService

|  |
| --- |
| package com.alex.spring.service;  import org.springframework.stereotype.Service;  import com.alex.spring.bean.BaseService;  import com.alex.spring.entity.User;  @Service  public class UserService extends BaseService<User>{  } |

#### 6）TestSpring

|  |
| --- |
| package com.alex.spring.junit;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.alex.spring.service.UserService;  public class TestSpring {  @Test  public void testIOC() {  @SuppressWarnings("resource")  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");  UserService us = (UserService) ioc.getBean("userService");  us.add();  }  } |

# 六、Spring EL

## 1、简介

Spring Expression Language，Spring表达式语言，简称SpEL。支持运行时查询并可以操作对象图。和JSP页面上的EL表达式、Struts2中用到的OGNL表达式一样，SpEL根据JavaBean风格的getXxx()、setXxx()方法定义的属性访问对象图，完全符合我们熟悉的操作习惯。

## 2、基本语法

SpEL使用#{…}作为定界符，所有在大框号中的字符都将被认为是SpEL表达式。

## 3、使用字面量

1）整数

<property name="count" value="#{5}"/>

2）小数

<property name="frequency" value="#{89.7}"/>

3）科学计数法

<property name="capacity" value="#{1e4}"/>

4）String

类型的字面量可以使用单引号或者双引号作为字符串的定界符号

<property name="name" value="#{'Chuck'}"/>

<property name="name" value='#{"Chuck"}'/>

5）Boolean

<property name="enabled" value="#{false}"/>

## 4、引用其他bean

|  |
| --- |
| <bean id="emp04" class="com.alex.parent.bean.Employee">  <property name="empId" value="1003"/>  <property name="empName" value="Kate"/>  <property name="age" value="21"/>  <property name="detp" value="#{dept}"/>  </bean> |

## 5、引用其他bean的属性值作为自己某个属性的值

|  |
| --- |
| <bean id="emp05" class="com.alex.parent.bean.Employee">  <property name="empId" value="1003"/>  <property name="empName" value="Kate"/>  <property name="age" value="21"/>  <property name="deptName" value="#{dept.deptName}"/>  </bean> |

## 6、调用非静态方法

|  |
| --- |
| <!-- 创建一个对象，在SpEL表达式中调用这个对象的方法 -->  <bean id="salaryGenerator" class="com.alex.spel.bean.SalaryGenerator"/>  <bean id="employee" class="com.alex.spel.bean.Employee">  <!-- 通过对象方法的返回值为属性赋值 -->  <property name="salayOfYear" value="#{salaryGenerator.getSalaryOfYear(5000)}"/>  </bean> |

## 7、调用静态方法

|  |
| --- |
| <bean id="employee" class="com.alex.spel.bean.Employee">  <!-- 在SpEL表达式中调用类的静态方法 -->  <property name="circle" value="#{T(java.lang.Math).PI\*20}"/>  </bean> |

## 8、运算符

1）算术运算符：+、-、\*、/、%、^

2）字符串连接：+

3）比较运算符：<、>、==、<=、>=、lt、gt、eq、le、ge

4）逻辑运算符：and, or, not, |

5）三目运算符：判断条件?判断结果为true时的取值:判断结果为false时的取值

6）正则表达式：matches

# 七、整合多个配置文件

1、Spring允许通过<import>将多个配置文件引入到一个文件中，进行配置文件的集成。这样在启动Spring容器时，仅需要指定这个合并好的配置文件就可以。

2、import元素的resource属性支持Spring的标准的路径资源



# 八、Spring AOP 初始化认识

<<Java 与 模式.pdf>>

## 1、代理设计模式

GoF95 23种

代理设计模式:

它是用于对目标对象进行访问控制,以及对目标对象进行功能扩展.

A(客户端) -> B(代理(权限控制,日志,异常处理,事务处理等)) -> E,F,G(目标)

## 2、代理方式

### 1）静态代理

一个代理类只能代理一种接口.

### 2）动态代理

一个代理类可以代理多个接口.

（1）JDK动态代理

JDK中提供的代理方式.它是基于接口的代理方式.

java.lang.reflect.Proxy

public static Object newProxyInstance(ClassLoader loader,

Class<?>[] interfaces,

InvocationHandler h)

在使用JDK动态代理时,必须实现InvocationHandler 接口,用于处理代理功能的代码实现.

A(客户端) -> B(代理(权限控制,日志,异常处理,事务处理等)) -> C(InvocationHandler ) -> E,F,G(目标)

（2）Cglib动态代理:

Cglib是开源的代理方式,它是基于类的代理方式.代理类是目标类的子类.

## 3、JDK动态代理案例（spring\_aop\_01\_jdkproxy）

### 1）ProxyProvider

|  |
| --- |
| package com.alex.proxy.jdk;  import java.lang.reflect.InvocationHandler;  import java.lang.reflect.Method;  import java.lang.reflect.Proxy;  import java.util.Arrays;  import java.util.List;  public class ProxyProvider {    private Object targetobject;  public ProxyProvider(Object targetobject) {  this.targetobject = targetobject;  }    public Object getProxy() {    //1.目标对象的类加载器对象  ClassLoader loader = targetobject.getClass().getClassLoader();    //2.目标对象所实现的接口的数组  Class<?>[] interfaces = targetobject.getClass().getInterfaces();    //3.InvocationHandler对象  InvocationHandler h = new InvocationHandler() {    /\*\*  \* proxy参数：代理对象  \* method参数：要执行代理的目标方法  \* args参数：执行目标方法时需要的参数列表  \*/  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {    //①获取目标方法的方法名  String methodName = method.getName();    //②为了能够直接打印出参数列表，将Object[]转换为List  List<Object> argList = Arrays.asList(args);    Object result = null;  try {    //③在方法开始执行之前，打印“方法开始”日志  System.out.println("[Proxy日志]["+methodName+"方法开始][参数值："+argList+"]");    //④执行被代理对象的目标方法  result = method.invoke(targetobject, args);    //⑤在方法正常结束之后，打印“方法正常结束”日志  System.out.println("[Proxy日志]["+methodName+"方法正常结束][返回值："+result+"]");    } catch (Exception e) {  e.printStackTrace();    //⑥在方法抛出异常后，打印“方法抛出异常”日志  System.out.println("[Proxy日志]["+methodName+"方法抛出异常][异常类型："+e.getClass().getName()+"]");    } finally {  //⑦在方法无论是正常结束还是抛出异常结束，都打印“方法最终结束”日志  System.out.println("[Proxy日志]["+methodName+"方法最终结束]");  }    //⑧被代理对象执行目标方法后的返回值返回  return result;  }  };    //4.创建代理对象  Object proxy = Proxy.newProxyInstance(loader, interfaces, h);    return proxy;  }  } |

### 2）MathCalculator

|  |
| --- |
| package com.alex.proxy.service;  public interface MathCalculator {    int add(int i,int j);  int sub(int i,int j);  int mul(int i, int j);  int div(int i,int j);  } |

### 3）EazyImpl

|  |
| --- |
| package com.alex.proxy.serviceimpl;  import com.alex.proxy.service.MathCalculator;  public class EazyImpl implements MathCalculator{  @Override  public int add(int i, int j) {    int result = i + j;    System.out.println("方法内部打印：result="+result);    return result;  }  @Override  public int sub(int i, int j) {    int result = i - j;    System.out.println("方法内部打印：result="+result);    return result;  }  @Override  public int mul(int i, int j) {    int result = i \* j;    System.out.println("方法内部打印：result="+result);    return result;  }  @Override  public int div(int i, int j) {    int result = i / j;    System.out.println("方法内部打印：result="+result);    return result;  }  } |

### 4）MathTest

|  |
| --- |
| package com.alex.proxy.test;  import com.alex.proxy.jdk.ProxyProvider;  import com.alex.proxy.service.MathCalculator;  import com.alex.proxy.serviceimpl.EazyImpl;  public class MathTest {    public static void main(String[] args) {    //正常调用执行  /\*MathCalculator calculator = new EazyImpl();  calculator.add(10, 5);  calculator.sub(10, 5);  calculator.mul(10, 5);  calculator.div(10, 5);\*/    //采用JDK动态代理来实现日志功能的扩展  MathCalculator calculator = new EazyImpl();  //A(客户端) -> B(代理(权限控制,日志,异常处理,事务处理等)) -> C(InvocationHandler ) -> E,F,G(目标)  //calculator = (MathCalculator) new ProxyProvider(calculator).getProxy();  System.out.println(calculator.getClass());  calculator.add(10, 5);  calculator.sub(10, 5);  calculator.mul(10, 5);  calculator.div(10, 5);    }  } |

## 4、Spring AOP 概述

1）AOP(Aspect-Oriented Programming，面向切面编程)：是一种新的方法论，是对传统 OOP(Object-Oriented Programming，面向对象编程)的补充。

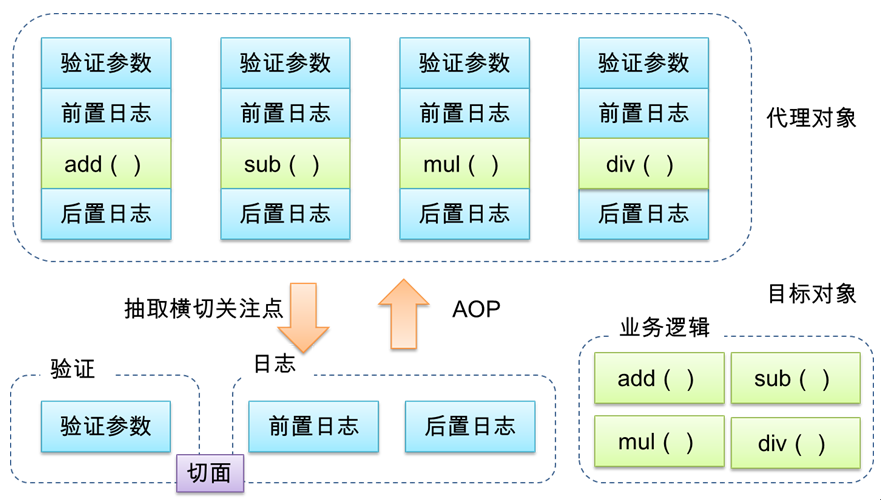
2）AOP编程操作的主要对象是切面(aspect)，而切面模块化横切关注点。

3）在应用AOP编程时，仍然需要定义公共功能，但可以明确的定义这个功能应用在哪里，以什么方式应用，并且不必修改受影响的类。这样一来横切关注点就被模块化到特殊的类里——这样的类我们通常称之为“切面”。

4）AOP的好处

每个事物逻辑位于一个位置，代码不分散，便于维护和升级

业务模块更简洁，只包含核心业务代码



## 5、Spring AOP 术语

### 1）切面(Aspect)

（1）切面是一个抽象的概念,也是一种编程思想.

（2）具体的指:

对目标程序中非业务代码进行提取出来生成一个单独程序类或组件-切面类

封装横切关注点信息的类，每个关注点体现为一个通知方法

### 2）横切关注点

当解决非业务代码问题时,将目标程序中的所有非业务代码都查找,这个非业务代码我们成为关注点.

从每个方法中抽取出来的同一类非核心业务

### 3）通知(通知方法)

切面必须要完成的各个具体工作

（1）表示将非业务代码提取到切面类中,存放到不同方法中来定义.然后在执行目标程序时,再执行这些方法

（2）有5种通知

@Before 前置通知

在执行目标方法前执行

@AfterRunning 方法返回通知

在执行目标方法返回时执行

@AfterThrowing 异常通知

在目标方法抛异常时执行

@After 后置通知

在目标方法执行完成后,不管是否存在异常,都必须执行

@Around 环绕通知

相当于前四种通知的组合使用.

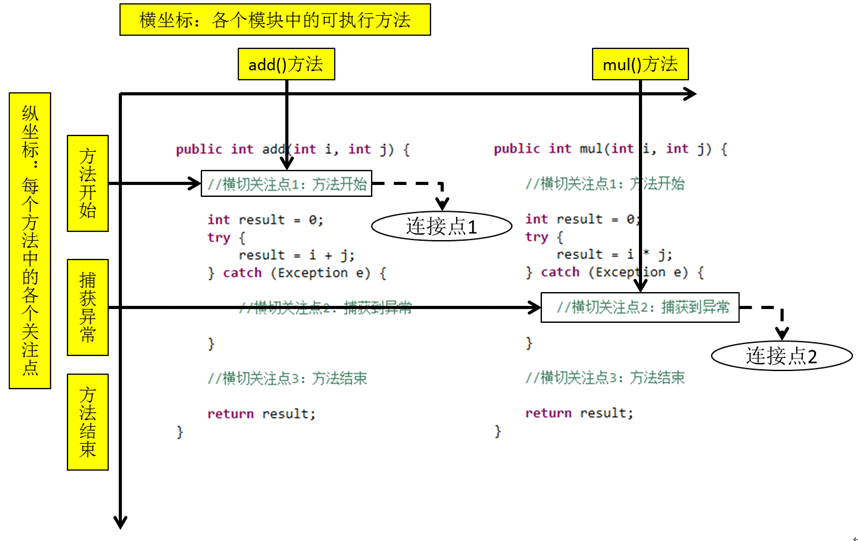
### 4）连接点

（1）一般指的就是目标对象中的方法.

（2）需要执行切面类通知方法的那个位置.

横切关注点在程序代码中的具体体现，对应程序执行的某个特定位置。例如：类某个方法调用前、调用后、方法捕获到异常后等。

在应用程序中可以使用横纵两个坐标来定位一个具体的连接点



### 5）切入点

（1）表示通过切入点表达式匹配到的连接点-切入点

定位连接点的方式。每个类的方法中都包含多个连接点，所以连接点是类中客观存在的事物。如果把连接点看作数据库中的记录，那么切入点就是查询条件——AOP可以通过切入点定位到特定的连接点。切点通过org.springframework.aop.Pointcut 接口进行描述，它使用类和方法作为连接点的查询条件

### 6）目标对象

（1）完成业务逻辑代码的对象.

被通知的对象

### 7）代理对象

（1）完成对目标对象进行功能扩展的对象.

向目标对象应用通知之后创建的代理对象

## 6、Spring AOP AspectJ

### 1）AspectJ简介

AspectJ：Java社区里最完整最流行的AOP框架。

在Spring2.0以上版本中，可以使用基于AspectJ注解或基于XML配置的AOP。

### 2）在Spring中启用AspectJ注解支持

#### （1）导入JAR包

com.springsource.net.sf.cglib-2.2.0.jar

com.springsource.org.aopalliance-1.0.0.jar

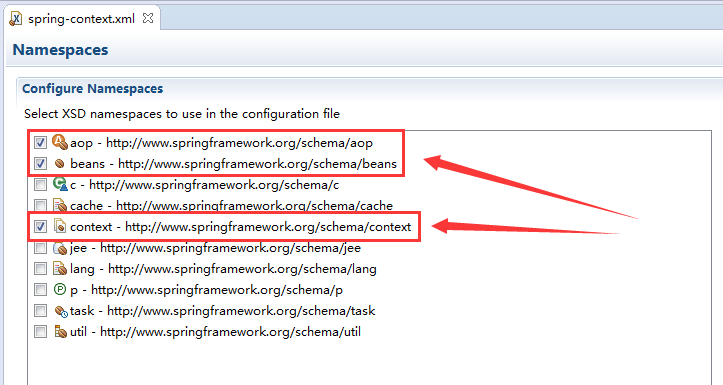
com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar

spring-aop-4.0.0.RELEASE.jar

spring-aspects-4.0.0.RELEASE.jar

尖叫提示：AOP建立在IOC基础上，所以除了拷贝AOP的jar包外，还要拷贝IOC的jar包，具体的IOC jar 包看HelloWorld 案例

#### （2）引入aop名称空间



3）配置

<aop:aspectj-autoproxy>

当Spring IOC容器侦测到bean配置文件中的<aop:aspectj-autoproxy>元素时，会自动为与AspectJ切面匹配的bean创建代理

### 3）用AspectJ注解声明切面

1）要在Spring中声明AspectJ切面，只需要在IOC容器中将切面声明为bean实例。

2）当在Spring IOC容器中初始化AspectJ切面之后，Spring IOC容器就会为那些与 AspectJ切面相匹配的bean创建代理。

3）在AspectJ注解中，切面只是一个带有@Aspect注解的Java类，它往往要包含很多通知。

4）通知是标注有某种注解的简单的Java方法。

5）AspectJ支持5种类型的通知注解：

（1）@Before：前置通知，在方法执行之前执行

（2）@After：后置通知，在方法执行之后执行

（3）@AfterRunning：返回通知，在方法返回结果之后执行

（4）@AfterThrowing：异常通知，在方法抛出异常之后执行

（5）@Around：环绕通知，围绕着方法执行

## 7、Spring AOP HelloWorld 案例

### 1）拷贝jar包

AOP建立在IOC基础上，所以除了拷贝AOP的jar包外，还要拷贝IOC的

（1）IOC jar包

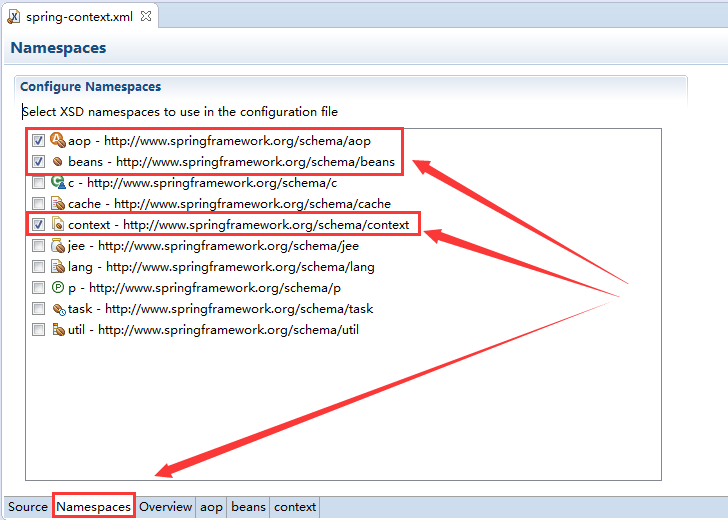
|  |
| --- |
| spring-beans-4.0.0.RELEASE.jar  spring-context-4.0.0.RELEASE.jar  spring-core-4.0.0.RELEASE.jar  spring-expression-4.0.0.RELEASE.jar  commons-logging-1.1.3.jar |

（2）AOP jar 包

|  |
| --- |
| com.springsource.net.sf.cglib-2.2.0.jar  com.springsource.org.aopalliance-1.0.0.jar  com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar  spring-aop-4.0.0.RELEASE.jar  spring-aspects-4.0.0.RELEASE.jar |

### 2）配置AOP（spring-context.xml）

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:aop="http://www.springframework.org/schema/aop"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd  http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd">  <context:component-scan base-package="com.alex.spring"></context:component-scan>    <!-- 启动AOP注解开发,自动生成代理对象. -->  <aop:aspectj-autoproxy />  </beans> |



### 3）LogAspect

定义切面类

A(客户端) -> B(代理) -> C(InvocationHandler ) -> AOP切面类(前置,后置,异常通知等(权限控制,日志,异常处理,事务处理等)) -> E,F,G(目标)

|  |
| --- |
| package com.alex.spring.aop;  import org.aspectj.lang.annotation.Aspect;  import org.aspectj.lang.annotation.Before;  import org.springframework.stereotype.Component;  /\*  \*完成日志的功能扩展  \*/  @Aspect //表示当前类是一个切面类  @Component //AOP切面类对象也需要IOC容器进行管理  public class LogAspect {    //表示当前方法为前置通知方法,所有前置通知方法会在执行目标方法前被框架所生产代理对象调用  //value属性用于指定切入点表达式:可以查询到连接点.  //@Before(value="execution(public int com.alex.spring.serviceimpl.EazyImpl.add(int, int))")  //@Before(value="execution(public int com.alex.spring.serviceimpl.EazyImpl.\*(int, int))")  //@Before(value="execution(public int com.alex.spring.serviceimpl.EazyImpl.\*(..))")  //@Before(value="execution(public int com.alex.spring.serviceimpl.EazyImpl.\*(int,..))")  //@Before(value="execution(public int com.alex.spring.serviceimpl.\*.\*(..))")  //@Before(value="execution(public int com.alex..\*.\*(..))")  //@Before(value="execution(public \* com.alex..\*.\*(..))")  //@Before(value="execution(\* com.alex..\*.\*(..))")  //@Before(value="execution(\* \*.\*(..))")  //定义切面类的通知方法  @Before(value="execution(public int com.alex.spring.serviceimpl.EazyImpl.add(int, int))")  public void beforeMethod() {  System.out.println("start method");  }  } |

### 4）MathCalculator

|  |
| --- |
| package com.alex.spring.service;  public interface MathCalculator {    int add(int i,int j);  int sub(int i,int j);  int mul(int i, int j);  int div(int i,int j);  } |

### 5）EazyImpl

|  |
| --- |
| package com.alex.spring.serviceimpl;  import org.springframework.stereotype.Component;  import com.alex.spring.service.MathCalculator;  //目标类  @Component  public class EazyImpl implements MathCalculator{  //一个方法就是一个连接点  public int add(int i, int j) {    //System.out.println("start method");    int result = i + j;    System.out.println("方法内部打印：result="+result);    //System.out.println("end method");    return result;  }  public int sub(int i, int j) {    int result = i - j;    System.out.println("方法内部打印：result="+result);    return result;  }  public int mul(int i, int j) {    int result = i \* j;    System.out.println("方法内部打印：result="+result);    return result;  }  public int div(int i, int j) {    int result = i / j;    System.out.println("方法内部打印：result="+result);    return result;  }  } |

### 6）TestAop1（JDK动态代理 spring\_aop\_02\_helloworld\_annotation）

JDK动态代理:目标类需要实现接口

|  |
| --- |
| package com.alex.spring.test;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.alex.spring.service.MathCalculator;  public class TestAop {  @Test  public void testjdkproxy() {    @SuppressWarnings("resource")  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");  //获取到的对象为目标对象的代理对象.  MathCalculator mathCalculator = ioc.getBean(MathCalculator.class);  //class com.sun.proxy.$Proxy7  //Spring框架在生产代理对象时,默认采用JDK动态代理.  //只要目标对象有接口,就会采用JDK动态代理;如果目标对象没有接口,那么会采用Cglib动态代理.  System.out.println(mathCalculator.getClass());    mathCalculator.add(10, 5);    //mathCalculator.sub(10, 5);  }  } |

### 7）TestAop2（Cglib动态代理 spring\_aop\_03\_helloworld\_annotation）

Cglib动态代理:目标类不需要实现任何接口.

|  |
| --- |
| package com.alex.spring.test;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.alex.spring.serviceimpl.EazyImpl;  public class TestAop {    @Test  public void testCGLIBproxy() {  @SuppressWarnings("resource")  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");  //Cglib动态代理:  // class com.alex.spring.serviceimpl.EazyImpl$$EnhancerByCGLIB$$7da36fc  EazyImpl eazyImpl = ioc.getBean(EazyImpl.class);  //  System.out.println(eazyImpl.getClass());    eazyImpl.add(10, 5);  }  } |

# 九、Spring AOP 细节

## 1、AOP 切入点表达式

切入表达式语法，来源于AspectJ 框架

### 1）作用

通过表达式的方式定位一个或多个具体的连接点

### 2）语法细节

（1）切入点表达式的语法格式

|  |
| --- |
| execution([权限修饰符] [返回值类型] [简单类名/全类名] [方法名]([参数列表])) |

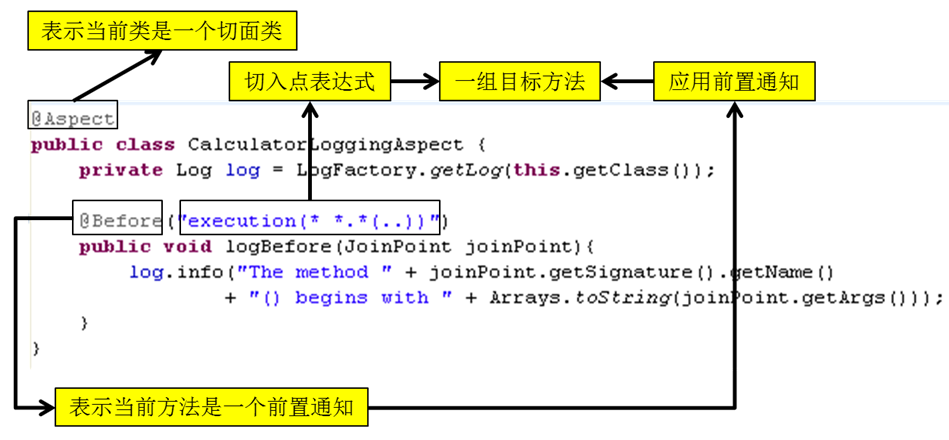
（2）举例说明

|  |  |
| --- | --- |
| 表达式 | 含义 |
| execution(\* com.alex.spring.ArithmeticCalculator.\*(..)) | ArithmeticCalculator接口中声明的所有方法第一个“\*”代表任意修饰符及任意返回值。  第二个“\*”代表任意方法。  “..”匹配任意数量、任意类型的参数。  若目标类、接口与该切面类在同一个包中可以省略包名 |
| execution(public \* ArithmeticCalculator.\*(..)) | ArithmeticCalculator接口的所有公有方法 |
| execution(public double ArithmeticCalculator.\*(..)) | ArithmeticCalculator接口中返回double类型数值的方法 |
| execution(public double ArithmeticCalculator.\*(double, ..)) | 第一个参数为double类型的方法。  “..” 匹配任意数量、任意类型的参数 |
| execution(public double ArithmeticCalculator.\*(double, double)) | 参数类型为double，double类型的方法 |

（3）在AspectJ中，切入点表达式可以通过 “&&”、“||”、“!”等操作符结合起来。

|  |  |
| --- | --- |
| 表达式 | 含义 |
| execution (\* \*.add(int,..)) || execution(\* \*.sub(int,..)) | 任意类中第一个参数为int类型的add方法或sub方法 |
| !execution (\* \*.add(int,..)) | 匹配不是——任意类中第一个参数为int类型的add方法 |

### 3）切入点表达式应用到实际的切面类中



### 4）案例实操

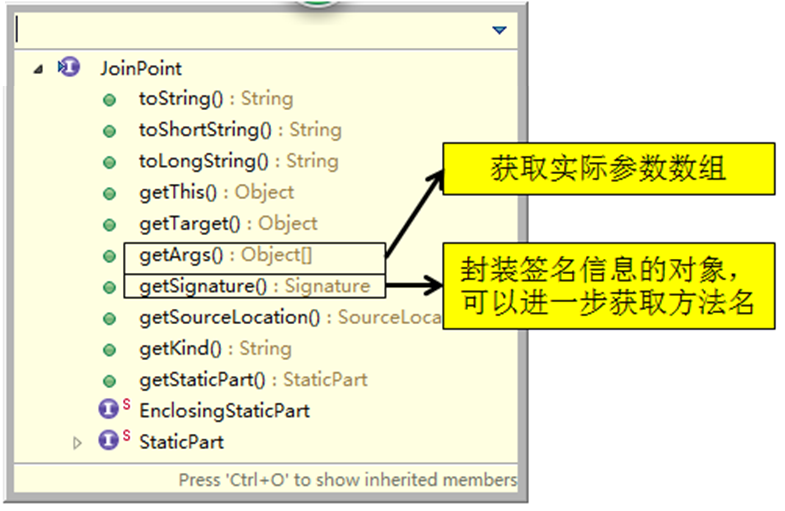
|  |
| --- |
| package com.alex.spring.aop;  import org.aspectj.lang.annotation.Aspect;  import org.aspectj.lang.annotation.Before;  import org.springframework.stereotype.Component;  /\*  \*完成日志的功能扩展  \*/  @Aspect //表示当前类是一个切面类  @Component //AOP切面类对象也需要IOC容器进行管理  public class LogAspect {    //表示当前方法为前置通知方法,所有前置通知方法会在执行目标方法前被框架所生产代理对象调用  //value属性用于指定切入点表达式:可以查询到连接点.  //@Before(value="execution(public int com.alex.spring.serviceimpl.EazyImpl.add(int, int))")  //@Before(value="execution(public int com.alex.spring.serviceimpl.EazyImpl.\*(int, int))")  //@Before(value="execution(public int com.alex.spring.serviceimpl.EazyImpl.\*(..))")  //@Before(value="execution(public int com.alex.spring.serviceimpl.EazyImpl.\*(int,..))")  //@Before(value="execution(public int com.alex.spring.serviceimpl.\*.\*(..))")  //@Before(value="execution(public int com.alex..\*.\*(..))")  //@Before(value="execution(public \* com.alex..\*.\*(..))")  //@Before(value="execution(\* com.alex..\*.\*(..))")  //@Before(value="execution(\* \*.\*(..))")  @Before(value="execution(public int com.alex.spring.serviceimpl.EazyImpl.add(int, int))")  public void beforeMethod() {  System.out.println("start method");  }  } |

## 2、AOP 获取当前连接点细节

### 1）概述

切入点表达式通常都会是从宏观上定位一组方法，和具体某个通知的注解结合起来就能够确定对应的连接点。那么就一个具体的连接点而言，我们可能会关心这个连接点的一些具体信息，例如：当前连接点所在方法的方法名、当前传入的参数值等等。这些信息都封装在JoinPoint接口的实例对象中。

### 2）JoinPoint



### 3）案例实操（spring\_aop\_04\_helloworld\_annotation）

#### （1）LogAspect

你的切面类里面，可以知道对谁做了操作

|  |
| --- |
| package com.alex.spring.aop;  import java.util.Arrays;  import org.aspectj.lang.JoinPoint;  import org.aspectj.lang.Signature;  import org.aspectj.lang.annotation.Aspect;  import org.aspectj.lang.annotation.Before;  import org.springframework.stereotype.Component;  /\*  \*完成日志的功能扩展  \*/  @Aspect // 表示当前类是一个切面类  @Component // AOP切面类对象也需要IOC容器进行管理  public class LogAspect {  // 表示当前方法为前置通知方法,所有前置通知方法会在执行目标方法前被框架所生产代理对象调用  // value属性用于指定切入点表达式:可以查询到连接点  @Before(value = "execution(public int com.alex.spring.serviceimpl.EazyImpl.add(int, int))")  public void beforeMethod(JoinPoint joinPoint) {  Signature signature = joinPoint.getSignature();  // 获取连接点的方法名称  String name = signature.getName();  // 获取连接点的连接点方法所在的类名称  String declaringTypeName = signature.getDeclaringTypeName();  // 获取连接点的方法的参数  Object[] args = joinPoint.getArgs();  System.out.println("name=" + name);  System.out.println("declaringTypeName=" + declaringTypeName);  System.out.println(declaringTypeName + "类的" + name + "方法 -start method : 参数为" + Arrays.asList(args) + "");  }  } |

#### （2）MathCalculator

|  |
| --- |
| package com.alex.spring.service;  public interface MathCalculator {  int add(int i,int j);  int sub(int i,int j);  int mul(int i, int j);  int div(int i,int j);  } |

#### （3）EazyImpl

|  |
| --- |
| package com.alex.spring.serviceimpl;  import org.springframework.stereotype.Component;  //目标类  @Component  public class EazyImpl {  //一个方法就是一个连接点  public int add(int i, int j) {    //System.out.println("start method");    int result = i + j;    System.out.println("方法内部打印：result="+result);    //System.out.println("end method");    return result;  }  public int sub(int i, int j) {    int result = i - j;    System.out.println("方法内部打印：result="+result);    return result;  }  public int mul(int i, int j) {    int result = i \* j;    System.out.println("方法内部打印：result="+result);    return result;  }  public int div(int i, int j) {    int result = i / j;    System.out.println("方法内部打印：result="+result);    return result;  }  } |

#### （4）TestAop

|  |
| --- |
| package com.alex.spring.test;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.alex.spring.serviceimpl.EazyImpl;  public class TestAop {    @Test  public void testCGLIBproxy() {    @SuppressWarnings("resource")  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");  //Cglib动态代理:  //class com.alex.spring.serviceimpl.EazyImpl$$EnhancerByCGLIB$$7da36fc  EazyImpl eazyImpl = ioc.getBean(EazyImpl.class);  System.out.println(eazyImpl.getClass());    eazyImpl.add(10, 5);  }  } |

#### （5）spring-context.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:aop="http://www.springframework.org/schema/aop"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd  http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd">  <context:component-scan base-package="com.alex.spring"></context:component-scan>    <!-- 启动AOP注解开发,自动生成代理对象. -->  <aop:aspectj-autoproxy />  </beans> |

## 3、AOP 常用通知

### 1）概述

 在具体的连接点上要执行的操作。

 一个切面可以包括一个或者多个通知。

 通知所使用的注解的值往往是切入点表达式。

### 2）有五种常用通知

（1）@Before 前置通知:

在执行目标方法前执行

（2）@AfterRunning 方法返回通知:

在执行目标方法返回时执行

（3）@AfterThrowing 异常通知:

在目标方法抛异常时执行

（4）@After 后置通知:

在目标方法执行完成后,不管是否存在异常,都必须执行.

（5）@Around 环绕通知:

相当于前四种通知的组合使用.

### 3）案例实操（spring\_aop\_05\_helloworld\_annotation）

#### （1）LogAspect

|  |
| --- |
| package com.alex.spring.aop;  import java.util.Arrays;  import org.aspectj.lang.JoinPoint;  import org.aspectj.lang.Signature;  import org.aspectj.lang.annotation.After;  import org.aspectj.lang.annotation.AfterReturning;  import org.aspectj.lang.annotation.AfterThrowing;  import org.aspectj.lang.annotation.Aspect;  import org.aspectj.lang.annotation.Before;  import org.springframework.stereotype.Component;  /\*  \*完成日志的功能扩展  \*/  @Aspect // 表示当前类是一个切面类  @Component // AOP切面类对象也需要IOC容器进行管理  public class LogAspect {  //方法前通知  // 表示当前方法为前置通知方法,所有前置通知方法会在执行目标方法前被框架所生产代理对象调用  // value属性用于指定切入点表达式:可以查询到连接点  @Before(value = "execution(public int com.alex.spring.serviceimpl.EazyImpl.add(int, int))")  public void beforeMethod(JoinPoint joinPoint) {  Signature signature = joinPoint.getSignature();  // 获取连接点的方法名称  String name = signature.getName();  // 获取连接点的连接点方法所在的类名称  String declaringTypeName = signature.getDeclaringTypeName();  // 获取连接点的方法的参数  Object[] args = joinPoint.getArgs();  System.out.println("name=" + name);  System.out.println("declaringTypeName=" + declaringTypeName);  System.out.println(declaringTypeName + "类的" + name + "方法 -start method : 参数为" + Arrays.asList(args) + "");  }  //方法后通知  @After(value = "execution(public int com.alex.spring.serviceimpl.EazyImpl.add(int, int))")  public void afterMethod(JoinPoint joinPoint) {  Signature signature = joinPoint.getSignature();  String name = signature.getName(); // 连接点-方法名称  String declaringTypeName = signature.getDeclaringTypeName(); // 连接点方法所在的类名称  Object[] args = joinPoint.getArgs();  System.out.println(declaringTypeName + "类的" + name + "方法 -end method : 参数为" + Arrays.asList(args) + "");  }  //方法异常通知  @AfterThrowing(value = "execution(public int com.alex.spring.serviceimpl.EazyImpl.\*(int, int))", throwing = "exception")  public void afterThrowingMethod(JoinPoint joinPoint, Throwable exception) {  Signature signature = joinPoint.getSignature();  String name = signature.getName(); // 连接点-方法名称  String declaringTypeName = signature.getDeclaringTypeName(); // 连接点方法所在的类名称  @SuppressWarnings("unused")  Object[] args = joinPoint.getArgs();  System.out.println(declaringTypeName + "类的" + name + "方法 - 抛异常啦 : 异常为=" + exception.getMessage());  }  //方法返回值通知  @AfterReturning(value = "execution(public int com.alex.spring.serviceimpl.EazyImpl.\*(int, int))", returning = "result")  public void afterReturningMethod(JoinPoint joinPoint, Object result) {  Signature signature = joinPoint.getSignature();  String name = signature.getName(); // 连接点-方法名称  String declaringTypeName = signature.getDeclaringTypeName(); // 连接点方法所在的类名称  @SuppressWarnings("unused")  Object[] args = joinPoint.getArgs();  System.out.println(declaringTypeName + "类的" + name + "方法 -返回结果了 : " + result);  }  } |

#### （2）MathCalculator

|  |
| --- |
| package com.alex.spring.service;  public interface MathCalculator {  int add(int i,int j);  int sub(int i,int j);  int mul(int i, int j);  int div(int i,int j);  } |

#### （3）EazyImpl

|  |
| --- |
| package com.alex.spring.serviceimpl;  import org.springframework.stereotype.Component;  //目标类  @Component  public class EazyImpl {  //一个方法就是一个连接点  public int add(int i, int j) {  //System.out.println("start method");  int result = i + j;  System.out.println("方法内部打印：result="+result);  //System.out.println("end method");  return result;  }  public int sub(int i, int j) {  int result = i - j;  System.out.println("方法内部打印：result="+result);  return result;  }  public int mul(int i, int j) {  int result = i \* j;  System.out.println("方法内部打印：result="+result);  return result;  }  public int div(int i, int j) {  int result = i / j;  System.out.println("方法内部打印：result="+result);  return result;  }  } |

#### （4）TestAop

|  |
| --- |
| package com.alex.spring.test;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.alex.spring.serviceimpl.EazyImpl;  public class TestAop {    @Test  public void testCGLIBproxy() {    @SuppressWarnings("resource")  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");  //Cglib动态代理:  // class com.alex.spring.serviceimpl.EazyImpl$$EnhancerByCGLIB$$7da36fc  EazyImpl eazyImpl = ioc.getBean(EazyImpl.class);    System.out.println(eazyImpl.getClass());    eazyImpl.add(10, 5);  //eazyImpl.div(10, 0);  }  } |

#### （5）spring-context.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:aop="http://www.springframework.org/schema/aop"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd  http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd">  <context:component-scan base-package="com.alex.spring"></context:component-scan>    <!-- 启动AOP注解开发,自动生成代理对象. -->  <aop:aspectj-autoproxy />  </beans> |

## 4、AOP 环绕通知

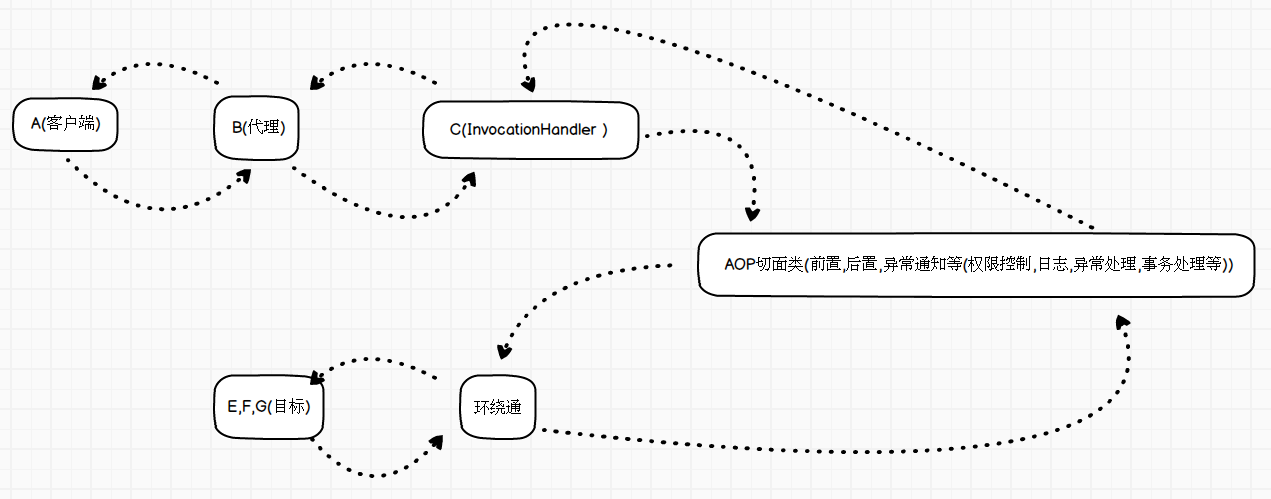
### 1）环绕通知概念

 环绕通知是所有通知类型中功能最为强大的，能够全面地控制连接点，甚至可以控制是否执行连接点。

 对于环绕通知来说，连接点的参数类型必须是ProceedingJoinPoint。它是 JoinPoint的子接口，允许控制何时执行，是否执行连接点。

 在环绕通知中需要明确调用ProceedingJoinPoint的proceed()方法来执行被代理的方法。如果忘记这样做就会导致通知被执行了，但目标方法没有被执行。

 注意：环绕通知的方法需要返回目标方法执行之后的结果，即调用 joinPoint.proceed();的返回值，否则会出现空指针异常。



### 2）环绕通知实操（spring\_aop\_06\_helloworld\_annotation）

A(客户端) -> B(代理) -> C(InvocationHandler ) -> AOP切面类(前置,后置,异常通知等(权限控制,日志,异常处理,事务处理等)) -> E,F,G(目标)

**尖叫提示：可以配置之前的四个通知，一起打debug走一下，正常情况下，前四个通知和环绕通知，不一起使用，使用其一就可以了，使用注解的方式，是先执行@after 再执行**

**@ AfterRunning 通知，使用xml配置方式，执行的顺序，是根据配置的前后顺序的**

#### （1）LogAspect

|  |
| --- |
| package com.alex.spring.aop;  import java.util.Arrays;  import org.aspectj.lang.ProceedingJoinPoint;  import org.aspectj.lang.Signature;  import org.aspectj.lang.annotation.Around;  import org.aspectj.lang.annotation.Aspect;  import org.springframework.stereotype.Component;  /\*  \*完成日志的功能扩展  \*/  @Aspect // 表示当前类是一个切面类  @Component // AOP切面类对象也需要IOC容器进行管理  public class LogAspect {  @Around(value = "execution(public int com.alex.spring.serviceimpl.EazyImpl.\*(int, int))")  public Object aroundMethod(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {  Object result = null;  // proceedingJoinPoint 是 JoinPoint 的子类  // JoinPoint Spring 框架自动调用方法  // proceedingJoinPoint 需要自己调用方法  Signature signature = proceedingJoinPoint.getSignature();  String name = signature.getName(); // 连接点-方法名称  String declaringTypeName = signature.getDeclaringTypeName(); // 连接点方法所在的类名称  Object[] args = proceedingJoinPoint.getArgs();  try {  // 前置通知  System.out.println(declaringTypeName + "类的@Around" + name + "方法 -开始了 : ");  result = proceedingJoinPoint.proceed(); // 执行目标方法.  // 后置通知  System.out.println(declaringTypeName + "类的@Around" + name + "方法 -返回结果了 : " + result);  return result;  } catch (Exception e) {  // 异常通知  System.out.println(declaringTypeName + "类的@Around" + name + "方法 - 抛异常啦 : 异常为=" + e.getMessage());  e.printStackTrace();  throw e;  } finally {  // 最后通知  System.out.println(  declaringTypeName + "类的@Around" + name + "方法 -" + Arrays.asList(args) + " 最终执行完成: 结果为" + result);  }  }  } |

#### （2）MathCalculator

|  |
| --- |
| package com.alex.spring.service;  public interface MathCalculator {    int add(int i,int j);  int sub(int i,int j);  int mul(int i, int j);  int div(int i,int j);  } |

#### （3）EazyImpl

|  |
| --- |
| package com.alex.spring.serviceimpl;  import org.springframework.stereotype.Component;  //目标类  @Component  public class EazyImpl {  //一个方法就是一个连接点  public int add(int i, int j) {    //System.out.println("start method");    int result = i + j;    System.out.println("方法内部打印：result="+result);    //System.out.println("end method");    return result;  }  public int sub(int i, int j) {    int result = i - j;    System.out.println("方法内部打印：result="+result);    return result;  }  public int mul(int i, int j) {    int result = i \* j;    System.out.println("方法内部打印：result="+result);    return result;  }  public int div(int i, int j) {    int result = i / j;    System.out.println("方法内部打印：result="+result);    return result;  }  } |

#### （4）TestAop

|  |
| --- |
| package com.alex.spring.test;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.alex.spring.serviceimpl.EazyImpl;  public class TestAop {    @Test  public void testCGLIBproxy() {    @SuppressWarnings("resource")  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");  //Cglib动态代理:class com.alex.spring.serviceimpl.EazyImpl$$EnhancerByCGLIB$$7da36fc  EazyImpl eazyImpl = ioc.getBean(EazyImpl.class);    System.out.println(eazyImpl.getClass());    eazyImpl.add(10, 5);  //eazyImpl.div(10, 0);  }  } |

#### （5）spring-context.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:aop="http://www.springframework.org/schema/aop"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd  http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd">  <context:component-scan base-package="com.alex.spring"></context:component-scan>    <!-- 启动AOP注解开发,自动生成代理对象. -->  <aop:aspectj-autoproxy />  </beans> |

## 5、AOP 重用切入点表达式

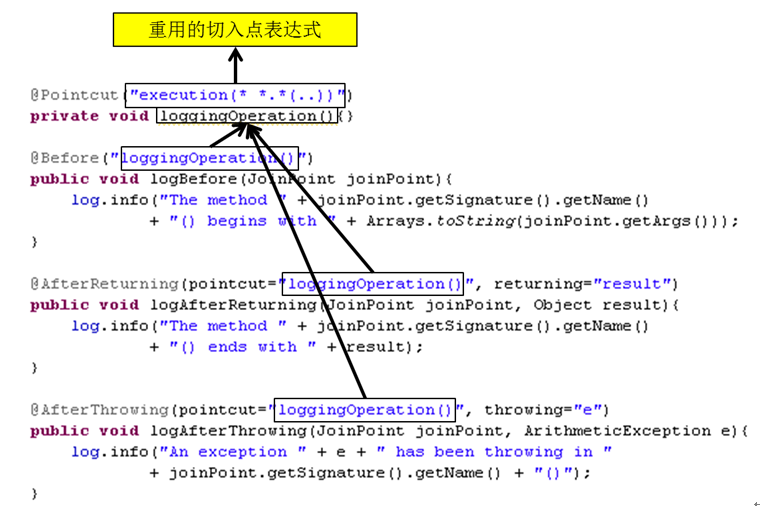
### 1）概念

 在编写AspectJ切面时，可以直接在通知注解中书写切入点表达式。但同一个切点表达式可能会在多个通知中重复出现。

 在AspectJ切面中，可以通过@Pointcut注解将一个切入点声明成简单的方法。切入点的方法体通常是空的，因为将切入点定义与应用程序逻辑混在一起是不合理的。

 切入点方法的访问控制符同时也控制着这个切入点的可见性。如果切入点要在多个切面中共用，最好将它们集中在一个公共的类中。在这种情况下，它们必须被声明为public。在引入这个切入点时，必须将类名也包括在内。如果类没有与这个切面放在同一个包中，还必须包含包名。

 其他通知可以通过方法名称引入该切入点



### 2）思想

减少冗余代码，易于维护

在通知方法上使用切入点表达式时,可以去引用一个公共的切入点表达式.

（1）在切面类中定义公共切入点表达式

@Pointcut(value="execution(public int com.alex.spring.serviceimpl.EazyImpl.\*(int, int))")

public void pointcut() {}

（2）在通知方法上应用公共切入点表达式

@Before(value="pointcut()")

@After(value="pointcut()")

@AfterThrowing(value="pointcut()",throwing="exception")

@AfterReturning(value="pointcut()",returning="result")

@Around(value="pointcut()")

**尖叫提示：value的值是@Pointcut注解的方法名**

### 3）案例实操（spring\_aop\_07\_helloworld\_annotation）

#### （1）LogAspect

|  |
| --- |
| package com.alex.spring.aop;  import java.util.Arrays;  import org.aspectj.lang.JoinPoint;  import org.aspectj.lang.ProceedingJoinPoint;  import org.aspectj.lang.Signature;  import org.aspectj.lang.annotation.After;  import org.aspectj.lang.annotation.AfterReturning;  import org.aspectj.lang.annotation.AfterThrowing;  import org.aspectj.lang.annotation.Around;  import org.aspectj.lang.annotation.Aspect;  import org.aspectj.lang.annotation.Before;  import org.aspectj.lang.annotation.Pointcut;  import org.springframework.stereotype.Component;  /\*  \*完成日志的功能扩展  \*/  @Aspect // 表示当前类是一个切面类  @Component // AOP切面类对象也需要IOC容器进行管理  public class LogAspect {    @Pointcut(value="execution(public int com.alex.spring.serviceimpl.EazyImpl.\*(int, int))")  public void pointcut() {}  //方法前通知  // 表示当前方法为前置通知方法,所有前置通知方法会在执行目标方法前被框架所生产代理对象调用  // value属性用于指定切入点表达式:可以查询到连接点  //@Before(value = "execution(public int com.alex.spring.serviceimpl.EazyImpl.add(int, int))")  @Before(value="pointcut()")  public void beforeMethod(JoinPoint joinPoint) {  Signature signature = joinPoint.getSignature();  // 获取连接点的方法名称  String name = signature.getName();  // 获取连接点的连接点方法所在的类名称  String declaringTypeName = signature.getDeclaringTypeName();  // 获取连接点的方法的参数  Object[] args = joinPoint.getArgs();  System.out.println("name=" + name);  System.out.println("declaringTypeName=" + declaringTypeName);  System.out.println(declaringTypeName + "类的" + name + "方法 -start method : 参数为" + Arrays.asList(args) + "");  }  //方法后通知  //@After(value = "execution(public int com.alex.spring.serviceimpl.EazyImpl.add(int, int))")  @After(value="pointcut()")  public void afterMethod(JoinPoint joinPoint) {  Signature signature = joinPoint.getSignature();  String name = signature.getName(); // 连接点-方法名称  String declaringTypeName = signature.getDeclaringTypeName(); // 连接点方法所在的类名称  Object[] args = joinPoint.getArgs();  System.out.println(declaringTypeName + "类的" + name + "方法 -end method : 参数为" + Arrays.asList(args) + "");  }  //方法异常通知  //@AfterThrowing(value = "execution(public int com.alex.spring.serviceimpl.EazyImpl.\*(int, int))", throwing = "exception")  @AfterThrowing(value="pointcut()",throwing="exception")  public void afterThrowingMethod(JoinPoint joinPoint, Throwable exception) {  Signature signature = joinPoint.getSignature();  String name = signature.getName(); // 连接点-方法名称  String declaringTypeName = signature.getDeclaringTypeName(); // 连接点方法所在的类名称  @SuppressWarnings("unused")  Object[] args = joinPoint.getArgs();  System.out.println(declaringTypeName + "类的" + name + "方法 - 抛异常啦 : 异常为=" + exception.getMessage());  }  //方法返回值通知  //@AfterReturning(value = "execution(public int com.alex.spring.serviceimpl.EazyImpl.\*(int, int))", returning = "result")  @AfterReturning(value="pointcut()",returning="result")  public void afterReturningMethod(JoinPoint joinPoint, Object result) {  Signature signature = joinPoint.getSignature();  String name = signature.getName(); // 连接点-方法名称  String declaringTypeName = signature.getDeclaringTypeName(); // 连接点方法所在的类名称  @SuppressWarnings("unused")  Object[] args = joinPoint.getArgs();  System.out.println(declaringTypeName + "类的" + name + "方法 -返回结果了 : " + result);  }    //@Around(value = "execution(public int com.alex.spring.serviceimpl.EazyImpl.\*(int, int))")  @Around(value="pointcut()")  public Object aroundMethod(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {  Object result = null;  // proceedingJoinPoint 是 JoinPoint 的子类  // JoinPoint Spring 框架自动调用方法  // proceedingJoinPoint 需要自己调用方法  Signature signature = proceedingJoinPoint.getSignature();  String name = signature.getName(); // 连接点-方法名称  String declaringTypeName = signature.getDeclaringTypeName(); // 连接点方法所在的类名称  Object[] args = proceedingJoinPoint.getArgs();  try {  // 前置通知  System.out.println(declaringTypeName + "类的@Around" + name + "方法 -开始了 : ");  result = proceedingJoinPoint.proceed(); // 执行目标方法.  // 后置通知  System.out.println(declaringTypeName + "类的@Around" + name + "方法 -返回结果了 : " + result);  return result;  } catch (Exception e) {  // 异常通知  System.out.println(declaringTypeName + "类的@Around" + name + "方法 - 抛异常啦 : 异常为=" + e.getMessage());  e.printStackTrace();  throw e;  } finally {  // 最后通知  System.out.println(  declaringTypeName + "类的@Around" + name + "方法 -" + Arrays.asList(args) + " 最终执行完成: 结果为" + result);  }  }  } |

#### （2）MathCalculator

|  |
| --- |
| package com.alex.spring.service;  public interface MathCalculator {    int add(int i,int j);  int sub(int i,int j);  int mul(int i, int j);  int div(int i,int j);  } |

#### （3）EazyImpl

|  |
| --- |
| package com.alex.spring.serviceimpl;  import org.springframework.stereotype.Component;  //目标类  @Component  public class EazyImpl {  //一个方法就是一个连接点  public int add(int i, int j) {    //System.out.println("start method");    int result = i + j;    System.out.println("方法内部打印：result="+result);    //System.out.println("end method");    return result;  }  public int sub(int i, int j) {    int result = i - j;    System.out.println("方法内部打印：result="+result);    return result;  }  public int mul(int i, int j) {    int result = i \* j;    System.out.println("方法内部打印：result="+result);    return result;  }  public int div(int i, int j) {    int result = i / j;    System.out.println("方法内部打印：result="+result);    return result;  }  } |

#### （4）TestAop

|  |
| --- |
| package com.alex.spring.test;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.alex.spring.serviceimpl.EazyImpl;  public class TestAop {    @Test  public void testCGLIBproxy() {    @SuppressWarnings("resource")  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");  //Cglib动态代理:class com.alex.spring.serviceimpl.EazyImpl$$EnhancerByCGLIB$$7da36fc  EazyImpl eazyImpl = ioc.getBean(EazyImpl.class);    System.out.println(eazyImpl.getClass());    eazyImpl.add(10, 5);  //eazyImpl.div(10, 0);  }  } |

#### （5）spring-context.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:aop="http://www.springframework.org/schema/aop"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd  http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd">  <context:component-scan base-package="com.alex.spring"></context:component-scan>    <!-- 启动AOP注解开发,自动生成代理对象. -->  <aop:aspectj-autoproxy />  </beans> |

## 6、AOP 指定页面的优先级

### 1）页面优先级概念

在同一个连接点上应用不止一个切面时，除非明确指定，否则它们的优先级是不确定的。

切面的优先级可以通过实现Ordered接口或利用@Order注解指定。

实现Ordered接口，getOrder()方法的返回值越小，优先级越高。

若使用@Order注解，序号出现在注解中

### 2）语法

@Order(value=0)

值越小优先级越高

### 3）案例实操（spring\_aop\_08\_helloworld\_annotation）

#### （1）LogAspect

|  |
| --- |
| package com.alex.spring.aop;  import java.util.Arrays;  import org.aspectj.lang.JoinPoint;  import org.aspectj.lang.ProceedingJoinPoint;  import org.aspectj.lang.Signature;  import org.aspectj.lang.annotation.After;  import org.aspectj.lang.annotation.AfterReturning;  import org.aspectj.lang.annotation.AfterThrowing;  import org.aspectj.lang.annotation.Around;  import org.aspectj.lang.annotation.Aspect;  import org.aspectj.lang.annotation.Before;  import org.aspectj.lang.annotation.Pointcut;  import org.springframework.core.annotation.Order;  import org.springframework.stereotype.Component;  /\*  \*完成日志的功能扩展  \*/  @Aspect // 表示当前类是一个切面类  @Component // AOP切面类对象也需要IOC容器进行管理  @Order(6)  public class LogAspect {    @Pointcut(value="execution(public int com.alex.spring.serviceimpl.EazyImpl.\*(int, int))")  public void pointcut() {}  //方法前通知  // 表示当前方法为前置通知方法,所有前置通知方法会在执行目标方法前被框架所生产代理对象调用  // value属性用于指定切入点表达式:可以查询到连接点  //@Before(value = "execution(public int com.alex.spring.serviceimpl.EazyImpl.add(int, int))")  @Before(value="pointcut()")  public void beforeMethod(JoinPoint joinPoint) {  Signature signature = joinPoint.getSignature();  // 获取连接点的方法名称  String name = signature.getName();  // 获取连接点的连接点方法所在的类名称  String declaringTypeName = signature.getDeclaringTypeName();  // 获取连接点的方法的参数  Object[] args = joinPoint.getArgs();  System.out.println("name=" + name);  System.out.println("declaringTypeName=" + declaringTypeName);  System.out.println(declaringTypeName + "类的" + name + "方法 -start method : 参数为" + Arrays.asList(args) + "");  }  } |

#### （2）TxAspect

|  |
| --- |
| package com.alex.spring.aop;  import org.aspectj.lang.JoinPoint;  import org.aspectj.lang.annotation.Aspect;  import org.aspectj.lang.annotation.Before;  import org.springframework.core.annotation.Order;  import org.springframework.stereotype.Component;  @Aspect  @Component  @Order(5) //数值越小，优先级越高  public class TxAspect {    @Before(value="com.alex.spring.aop.LogAspect.pointcut()")  public void beforeMethod(JoinPoint joinPoint) {  System.out.println("Tx.............");  }  } |

#### （3）MathCalculator

|  |
| --- |
| package com.alex.spring.service;  public interface MathCalculator {    int add(int i,int j);  int sub(int i,int j);  int mul(int i, int j);  int div(int i,int j);  } |

#### （4）EazyImpl

|  |
| --- |
| package com.alex.spring.serviceimpl;  import org.springframework.stereotype.Component;  //目标类  @Component  public class EazyImpl {  //一个方法就是一个连接点  public int add(int i, int j) {  //System.out.println("start method");  int result = i + j;  System.out.println("方法内部打印：result="+result);  //System.out.println("end method");  return result;  }  public int sub(int i, int j) {    int result = i - j;    System.out.println("方法内部打印：result="+result);    return result;  }  public int mul(int i, int j) {    int result = i \* j;    System.out.println("方法内部打印：result="+result);    return result;  }  public int div(int i, int j) {    int result = i / j;    System.out.println("方法内部打印：result="+result);    return result;  }  } |

#### （5）TestAop

|  |
| --- |
| package com.alex.spring.test;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.alex.spring.serviceimpl.EazyImpl;  public class TestAop {    @Test  public void testCGLIBproxy() {    @SuppressWarnings("resource")  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");  //Cglib动态代理:class com.alex.spring.serviceimpl.EazyImpl$$EnhancerByCGLIB$$7da36fc  EazyImpl eazyImpl = ioc.getBean(EazyImpl.class);    System.out.println(eazyImpl.getClass());    eazyImpl.add(10, 5);  //eazyImpl.div(10, 0);  }  } |

#### （6）spring-context.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:aop="http://www.springframework.org/schema/aop"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd  http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd">  <context:component-scan base-package="com.alex.spring"></context:component-scan>    <!-- 启动AOP注解开发,自动生成代理对象. -->  <aop:aspectj-autoproxy />  </beans> |

## 7、AOP XML

### 1）AOP XML 概述

除了使用AspectJ注解声明切面，Spring也支持在bean配置文件中声明切面。这种声明是通过aop名称空间中的XML元素完成的。

正常情况下，基于注解的声明要优先于基于XML的声明。通过AspectJ注解，切面可以与AspectJ兼容，而基于XML的配置则是Spring专有的。由于AspectJ得到越来越多的 AOP框架支持，所以以注解风格编写的切面将会有更多重用的机会。

### 2）AOP XML 配置细节

在bean配置文件中，所有的Spring AOP配置都必须定义在<aop:config>元素内部。对于每个切面而言，都要创建一个<aop:aspect>元素来为具体的切面实现引用后端bean实例。

切面bean必须有一个标识符，供<aop:aspect>元素引用。



### 3）声明切入点

 切入点使用<aop:pointcut>元素声明。

 切入点必须定义在<aop:aspect>元素下，或者直接定义在<aop:config>元素下。

 定义在<aop:aspect>元素下：只对当前切面有效

 定义在<aop:config>元素下：对所有切面都有效

 基于XML的AOP配置不允许在切入点表达式中用名称引用其他切入点。

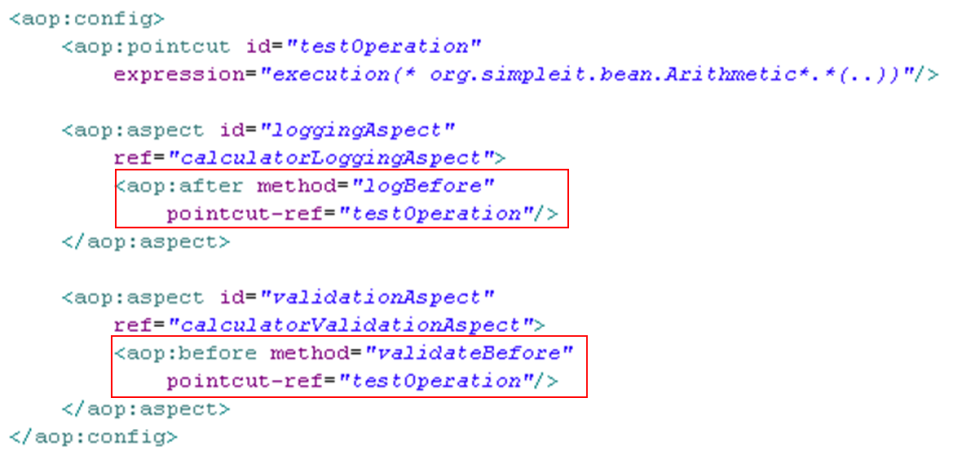


### 4）声明通知

在aop名称空间中，每种通知类型都对应一个特定的XML元素。

通知元素需要使用<pointcut-ref>来引用切入点，或用<pointcut>直接嵌入切入点表达式。

method属性指定切面类中通知方法的名称



### 5）案例实操（spring\_aop\_09\_helloworld\_xml）

#### （1）LogAspect

|  |
| --- |
| package com.alex.spring.aop;  import java.util.Arrays;  import org.aspectj.lang.JoinPoint;  import org.aspectj.lang.ProceedingJoinPoint;  import org.aspectj.lang.Signature;  /\*  \*完成日志的功能扩展.  \*/  public class LogAspect {  public void beforeMethod(JoinPoint joinPoint) {  Signature signature = joinPoint.getSignature();  String name = signature.getName(); // 连接点-方法名称  String declaringTypeName = signature.getDeclaringTypeName(); // 连接点方法所在的类名称  System.out.println("name=" + name);  System.out.println("declaringTypeName=" + declaringTypeName);  Object[] args = joinPoint.getArgs();  System.out.println(declaringTypeName + "类的" + name + "方法 -start method : 参数为" + Arrays.asList(args) + "");  }  public void afterMethod(JoinPoint joinPoint) {  Signature signature = joinPoint.getSignature();  String name = signature.getName(); // 连接点-方法名称  String declaringTypeName = signature.getDeclaringTypeName(); // 连接点方法所在的类名称  Object[] args = joinPoint.getArgs();  System.out.println(declaringTypeName + "类的" + name + "方法 -end method : 参数为" + Arrays.asList(args) + "");  }  public void afterThrowingMethod(JoinPoint joinPoint, Throwable exception) {  Signature signature = joinPoint.getSignature();  String name = signature.getName(); // 连接点-方法名称  String declaringTypeName = signature.getDeclaringTypeName(); // 连接点方法所在的类名称  Object[] args = joinPoint.getArgs();  System.out.println(declaringTypeName + "类的" + name + "方法 - 抛异常啦 : 异常为=" + exception.getMessage());  }  public void afterReturningMethod(JoinPoint joinPoint, Object result) {  Signature signature = joinPoint.getSignature();  String name = signature.getName(); // 连接点-方法名称  String declaringTypeName = signature.getDeclaringTypeName(); // 连接点方法所在的类名称  Object[] args = joinPoint.getArgs();  System.out.println(declaringTypeName + "类的" + name + "方法 -返回结果了 : " + result);  }  public Object aroundMethod(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {  Object result = null;  Signature signature = proceedingJoinPoint.getSignature();  String name = signature.getName(); // 连接点-方法名称  String declaringTypeName = signature.getDeclaringTypeName(); // 连接点方法所在的类名称  Object[] args = proceedingJoinPoint.getArgs();  try {  System.out.println(declaringTypeName + "类的@Around" + name + "方法 -开始了 : ");  result = proceedingJoinPoint.proceed(); // 执行目标方法.  System.out.println(declaringTypeName + "类的@Around" + name + "方法 -返回结果了 : " + result);  return result;  } catch (Exception e) {  System.out.println(declaringTypeName + "类的@Around" + name + "方法 - 抛异常啦 : 异常为=" + e.getMessage());  e.printStackTrace();  throw e;  } finally {  System.out.println(  declaringTypeName + "类的@Around" + name + "方法 -" + Arrays.asList(args) + " 最终执行完成: 结果为" + result);  }  }  } |

#### （2）TxAspect

|  |
| --- |
| package com.alex.spring.aop;  import org.aspectj.lang.JoinPoint;  public class TxAspect {  public void beforeMethod(JoinPoint joinPoint) {  System.out.println("Tx.............");  }  } |

#### （3）MathCalculator

|  |
| --- |
| package com.alex.spring.service;  public interface MathCalculator {    int add(int i,int j);  int sub(int i,int j);  int mul(int i, int j);  int div(int i,int j);  } |

#### （4）EazyImpl

|  |
| --- |
| package com.alex.spring.serviceimpl;  import com.alex.spring.service.MathCalculator;  //目标类  public class EazyImpl implements MathCalculator{  //一个方法就是一个连接点  public int add(int i, int j) {  //System.out.println("start method");  int result = i + j;  System.out.println("方法内部打印：result="+result);  //System.out.println("end method");  return result;  }  public int sub(int i, int j) {  int result = i - j;  System.out.println("方法内部打印：result="+result);  return result;  }  public int mul(int i, int j) {  int result = i \* j;  System.out.println("方法内部打印：result="+result);  return result;  }  public int div(int i, int j) {  int result = i / j;  System.out.println("方法内部打印：result="+result);  return result;  }  } |

#### （5）TestAOP

|  |
| --- |
| package com.alex.spring.test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.alex.spring.service.MathCalculator;  public class TestAOP {  public static void main(String[] args) {  @SuppressWarnings("resource")  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");  // 获取到的对象为目标对象的代理对象.  MathCalculator mathCalculator = ioc.getBean(MathCalculator.class);  // class com.sun.proxy.$Proxy7  // Spring框架在生产代理对象时,默认采用JDK动态代理.  // 只要目标对象有接口,就会采用JDK动态代理;如果目标对象没有接口,那么会采用Cglib动态代理.  // System.out.println(mathCalculator.getClass());  mathCalculator.add(10, 5);  }  } |

#### （6）spring-context.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:aop="http://www.springframework.org/schema/aop"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd  http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd">  <context:component-scan base-package="com.alex.spring"/>    <!-- 声明目标对象 -->  <bean id="mathCalculator" class="com.alex.spring.serviceimpl.EazyImpl"></bean>  <!-- 声明切面对象 -->  <bean id="logAspect" class="com.alex.spring.aop.LogAspect"></bean>  <bean id="txAspect" class="com.alex.spring.aop.TxAspect"></bean>    <!-- 组合目标对象和切面对象之间的关系. -->  <aop:config>  <!-- 公共的切入点表达式声明 -->  <!-- <aop:pointcut expression="execution(public int com.alex.spring.serviceimpl.EazyImpl.\*(int, int))" id="pointcut"/> -->    <aop:aspect id="txAspect" ref="txAspect" order="10">  <aop:pointcut expression="execution(public int com.alex.spring.serviceimpl.EazyImpl.\*(int, int))" id="txPointcut"/>  <aop:before method="beforeMethod" pointcut-ref="txPointcut"/>  </aop:aspect>    <aop:aspect id="logAspect" ref="logAspect" order="5">  <!-- 定义当前切面类的局部切入点表达式 -->  <aop:pointcut expression="execution(public int com.alex.spring.serviceimpl.EazyImpl.\*(int, int))" id="logPointcut"/>  <!-- 声明通知方法 -->  <aop:before method="beforeMethod" pointcut-ref="logPointcut"/>  <aop:after method="afterMethod" pointcut-ref="logPointcut"/>  <aop:after-returning method="afterReturningMethod" pointcut-ref="logPointcut" returning="result"/>  <aop:after-throwing method="afterThrowingMethod" pointcut-ref="logPointcut" throwing="exception"/>  <aop:around method="aroundMethod" pointcut-ref="logPointcut"/>  </aop:aspect>  </aop:config>  </beans> |

**尖叫提示: returning="result" 和 throwing="exception" 里面的值不能随便写，要和方法里面的参数一致**



## 8、Annotation 和 XML 比较

### 1）XML

（1）配置一目了然.

（2）通过DOM或SAX进行解析操作.

（3）XML配置发现错误问题往往是在系统运行时发现.

（4）修改配置更加方便.

### 2）Annotation

（1）配置是散落在每一个类中,查看不方便.没有源码,看不到配置.

（2）通过JVM的反射机制进行解析操作.

（3）Annotation注解配置发现问题往往在编译代码时就可以发现.

（4）修改配置不方便,一旦修改配置,需要重新编译,打包,部署.

# 十、Spring JDBCTemplate

<https://blog.csdn.net/CSDN_HELIU/article/details/83272808>

jdbc Template 类似于dbutils

Spring JDBCTemplate 我们就不作为重点来掌握了，大家听一听就可以了，因为我们真正做项目开发的时候，我们是不用这个东西的，我们底层用的是mybatis框架或者hibernate框架

Spring JDBCTemplate 太简单，是Spring 提供的，虽然也能做数据持久化，但是由于太简单，功能太单一，所以我们就不用它了，那么我们又为什么要讲Spring JDBCTemplate 呢？？因为我们还要去讲Spring 事务，而事务是基于数据的操作的，在没有讲mybatis之前就先讲Spring JDBCTemplate，好让大家有所了解，Spring JDBCTemplate不用是不用，但是学好，对后面的框架学习是有所帮助的，来吧，学习知识的时候，不要怕多，哈哈哈哈 ~~

## 1、概述

为了使JDBC更加易于使用，Spring在JDBC API上定义了一个抽象层，以此建立一个JDBC存取框架。

作为Spring JDBC框架的核心，JDBC模板的设计目的是为不同类型的JDBC操作提供模板方法，通过这种方式，可以在尽可能保留灵活性的情况下，将数据库存取的工作量降到最低。

可以将Spring的JdbcTemplate看作是一个小型的轻量级持久化层框架，和我们之前使用过的DBUtils风格非常接近。

## 2、环境准备

### 1）导入JAR包

（1）IOC容器所需要的JAR包

|  |
| --- |
| commons-logging-1.1.1.jar  spring-beans-4.0.0.RELEASE.jar  spring-context-4.0.0.RELEASE.jar  spring-core-4.0.0.RELEASE.jar  spring-expression-4.0.0.RELEASE.jar |

（2）JdbcTemplate所需要的JAR包

|  |
| --- |
| spring-jdbc-4.0.0.RELEASE.jar  spring-orm-4.0.0.RELEASE.jar  spring-tx-4.0.0.RELEASE.jar |

（3）数据库驱动和数据源

|  |
| --- |
| c3p0-0.9.1.2.jar  mysql-connector-java-5.1.7-bin.jar |

### 2）sql 语句

|  |
| --- |
| create database jdbc\_template;  USE jdbc\_template;  DROP TABLE IF EXISTS employee;  CREATE TABLE employee (  emp\_id int(11) NOT NULL AUTO\_INCREMENT,  emp\_name char(100) DEFAULT NULL,  salary double DEFAULT NULL,  PRIMARY KEY (emp\_id)  ) ENGINE=InnoDB AUTO\_INCREMENT=1 DEFAULT CHARSET=utf8;  insert into employee(emp\_id,emp\_name,salary)  values (1,'Susan',5000.23),  (2,'Julian',4234.77),  (3,'Papu',9034.51),  (4,'Babala',8054.33),  (5,'Kasier',6039.11),  (6,'Owen',7714.11); |

### 3）db.properties

|  |
| --- |
| user=root  password=root  jdbcUrl=jdbc:mysql://localhost:3306/jdbc\_template  driverClass=com.mysql.jdbc.Driver  initialPoolSize=30  minPoolSize=10  maxPoolSize=100  acquireIncrement=5  maxStatements=1000  maxStatementsPerConnection=10 |

### 4）spring-context.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd">  <!-- 设置扫描包 -->  <context:component-scan base-package="com.alex.spring"></context:component-scan>  <!-- 读取外部文件 -->  <context:property-placeholder location="classpath:/db.properties" />  <!-- C3P0 数据源 -->  <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">  <property name="user" value="${user}" />  <property name="password" value="${password}" />  <property name="jdbcUrl" value="${jdbcUrl}" />  <property name="driverClass" value="${driverClass}" />  <property name="initialPoolSize" value="${initialPoolSize}" />  <property name="minPoolSize" value="${minPoolSize}" />  <property name="maxPoolSize" value="${maxPoolSize}" />  <property name="acquireIncrement" value="${acquireIncrement}" />  <property name="maxStatements" value="${maxStatements}" />  <property name="maxStatementsPerConnection" value="${maxStatementsPerConnection}" />  </bean>  <!-- jdbcTemplate 对象 -->  <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">  <property name="dataSource" ref="dataSource" />  </bean>  </beans> |

## 3、持久化操作

### 1）增删改

JdbcTemplate.update(String, Object...)

### 2）批量增删改

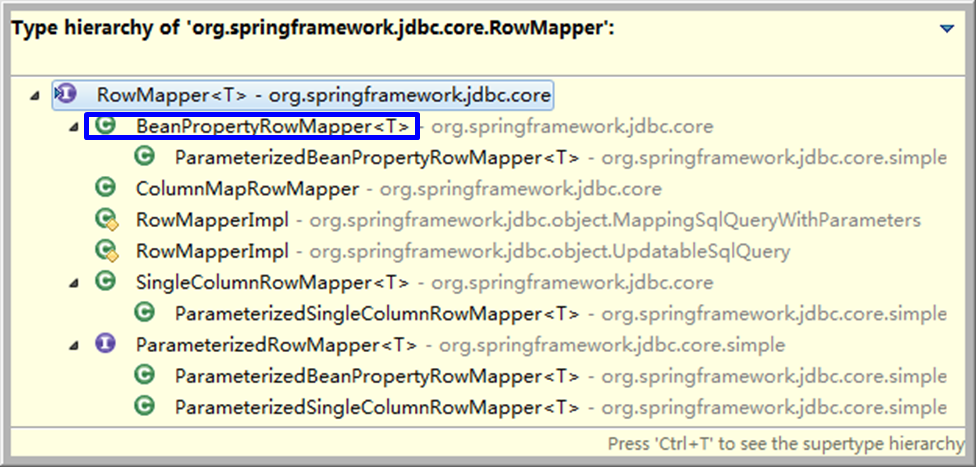
JdbcTemplate.batchUpdate(String, List<Object[]>)

Object[]封装了SQL语句每一次执行时所需要的参数

List集合封装了SQL语句多次执行时的所有参数

### 3）查询单行

JdbcTemplate.queryForObject(String, RowMapper<Department>, Object...)



### 4）查询多行

JdbcTemplate.query(String, RowMapper<Department>, Object...)

RowMapper对象依然可以使用BeanPropertyRowMapper

### 5）查询单一值

JdbcTemplate.queryForObject(String, Class, Object...)

## 4、使用具名参数的JdbcTemplate

### 1）关于具名参数

在Hibernate的HQL查询中我们体验过具名参数的使用，相对于基于位置的参数，具名参数具有更好的可维护性，在SQL语句中参数较多时可以考虑使用具名参数。

在Spring中可以通过NamedParameterJdbcTemplate类的对象使用带有具名参数的SQL语句。

### 2）通过IOC容器创建NamedParameterJdbcTemplate对象

|  |
| --- |
| <!-- 配置可以使用具名参数的JDBCTemplate类对象 -->  <bean  id="namedTemplate"  class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">  <!-- 没有无参构造器，必须传入数据源或JdbcTemplate对象 -->  <constructor-arg ref="dataSource"/>  </bean> |

### 3）具名参数在SQL语句中的格式

|  |
| --- |
| INSERT INTO depts (dept\_name) VALUES (:deptName) |

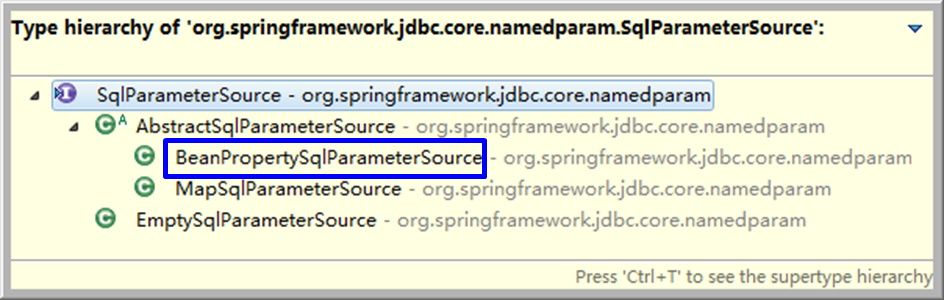
### 4）具名参数传入

（1）通过Map对象传入

|  |
| --- |
| NamedParameterJdbcTemplate.update(String sql, Map<String, ?> map) |

**尖叫提示：Map的键是参数名，值是参数值**

（2）通过SqlParameterSource对象传入



|  |
| --- |
| String sql = "INSERT INTO depts (dept\_name) VALUES (:deptName)";  Department department = new Department(null, "YYY", null);  SqlParameterSource sqlParameterSource =  new BeanPropertySqlParameterSource(department);  namedTemplate.update(sql, sqlParameterSource); |

## 5、使用JdbcTemplate实现Dao（spring\_jdbc\_07\_template）

通过IOC容器自动注入

JdbcTemplate类是线程安全的，所以可以在IOC容器中声明它的单个实例，并将这个实例注入到所有的Dao实例中

|  |
| --- |
| @Repository  public class EmployeeDao {    @Autowired  private JdbcTemplate jdbcTemplate;    public Employee get(Integer id){  //…  }  } |

### 1）code

#### （1）EmployeeDao

|  |
| --- |
| package com.alex.spring.dao;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.jdbc.core.BeanPropertyRowMapper;  import org.springframework.jdbc.core.JdbcTemplate;  import org.springframework.jdbc.core.RowMapper;  import org.springframework.stereotype.Repository;  import com.alex.spring.entity.Employee;  @Repository  public class EmployeeDao {  @Autowired  private JdbcTemplate jdbcTemplate;  public Employee get(Integer id) {  String sql = "select salary,emp\_name,emp\_id as id from employee where emp\_id=?";  RowMapper rowMapper = new BeanPropertyRowMapper(Employee.class);  Employee employee = (Employee) jdbcTemplate.queryForObject(sql, rowMapper, id);  return employee;  }  } |

#### （2）Employee

|  |
| --- |
| package com.alex.spring.entity;  import org.springframework.stereotype.Component;  @Component  public class Employee {  // 使用Integer 不赋值的时候是null,int 不赋值的时候是0  // Integer 是面向对象的，是引用类型  // int 是做计算用的，封装一个实体类是不涉及任何一个实体类的，所以这个时候尽量不要用小写的int,就用包装类型就可以了  private Integer id;  private String empName;  private Double salary;  public Integer getId() {  return id;  }  public void setId(Integer id) {  this.id = id;  }  public String getEmpName() {  return empName;  }  public void setEmpName(String empName) {  this.empName = empName;  }  public Double getSalary() {  return salary;  }  public void setSalary(Double salary) {  this.salary = salary;  }  //为了打印方便，重写toString方法  @Override  public String toString() {  return "Employee [id=" + id + ", empName=" + empName + ", salary=" + salary + "]";  }  } |

#### （3）TestSpring

|  |
| --- |
| package com.alex.spring.junit;  import java.sql.SQLException;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.alex.spring.dao.EmployeeDao;  import com.alex.spring.entity.Employee;  public class TestSpring {  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");  EmployeeDao employeeDao = (EmployeeDao) ioc.getBean("employeeDao");    //@Bean  // 查询最大salary  @Test  public void test6() throws SQLException {  Employee employee = employeeDao.get(5);  System.out.println(employee);  }  } |

### 2）注解版入口

|  |
| --- |
| @Configuration  @ComponentScan(basePackages = {"com.enl"})//添加该注解，是为了扫描到controller 、service、 dao 层  @EnableAutoConfiguration //启用自动配置  @ImportResource({"classpath:application-\*.xml"})  public class EnlightenmentApplication {    public static void main(String[] args) {  SpringApplication.run(EnlightenmentApplication.class, args);  }  } |

# 十一、Spring JDBCTemplate 实验

## 1、JDBCTemplate实验项目预览

实验1：测试数据源

实验2：将emp\_id=5的记录的salary字段更新为1300.00

实验3：批量插入

实验4：查询emp\_id=5的数据库记录，封装为一个Java对象返回

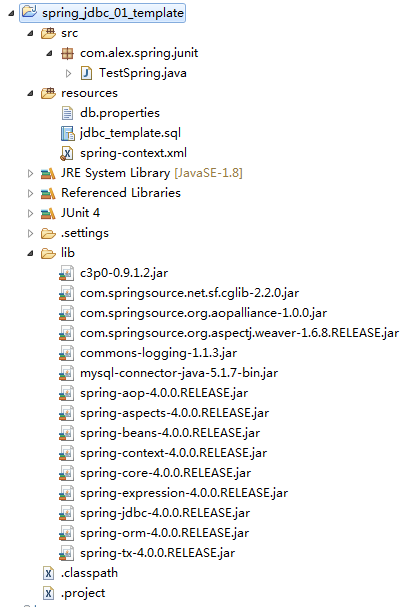
实验5：查询salary>4000的数据库记录，封装为List集合返回

实验6：查询最大salary

## 2、实验1（spring\_jdbc\_01\_template）

测试数据源

### 1）项目结构



**尖叫提示：主要看有什么jar包**

### 2）db.properties

|  |
| --- |
| user=root  password=root  jdbcUrl=jdbc:mysql://localhost:3306/jdbc\_template  driverClass=com.mysql.jdbc.Driver  initialPoolSize=30  minPoolSize=10  maxPoolSize=100  acquireIncrement=5  maxStatements=1000  maxStatementsPerConnection=10 |

### 3）jdbc\_template.sql

|  |
| --- |
| create database jdbc\_template;  USE jdbc\_template;  DROP TABLE IF EXISTS employee;  CREATE TABLE employee (  emp\_id int(11) NOT NULL AUTO\_INCREMENT,  emp\_name char(100) DEFAULT NULL,  salary double DEFAULT NULL,  PRIMARY KEY (emp\_id)  ) ENGINE=InnoDB AUTO\_INCREMENT=1 DEFAULT CHARSET=utf8;  insert into employee(emp\_id,emp\_name,salary)  values (1,'Susan',5000.23),  (2,'Julian',4234.77),  (3,'Papu',9034.51),  (4,'Babala',8054.33),  (5,'Kasier',6039.11),  (6,'Owen',7714.11); |

### 4）spring-context.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd">  <!-- 设置扫描包 -->  <context:component-scan base-package="com.alex.spring"></context:component-scan>  <!-- 读取外部文件 -->  <context:property-placeholder location="classpath:/db.properties" />  <!-- C3P0 数据源 -->  <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">  <property name="user" value="${user}" />  <property name="password" value="${password}" />  <property name="jdbcUrl" value="${jdbcUrl}" />  <property name="driverClass" value="${driverClass}" />  <property name="initialPoolSize" value="${initialPoolSize}" />  <property name="minPoolSize" value="${minPoolSize}" />  <property name="maxPoolSize" value="${maxPoolSize}" />  <property name="acquireIncrement" value="${acquireIncrement}" />  <property name="maxStatements" value="${maxStatements}" />  <property name="maxStatementsPerConnection" value="${maxStatementsPerConnection}" />  </bean>  <!-- jdbcTemplate 对象 -->  <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">  <property name="dataSource" ref="dataSource" />  </bean>  </beans> |

### 5）TestSpring

|  |
| --- |
| package com.alex.spring.junit;  import java.sql.Connection;  import java.sql.SQLException;  import javax.sql.DataSource;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import org.springframework.jdbc.core.JdbcTemplate;  public class TestSpring {  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");  JdbcTemplate jdbcTemplate = (JdbcTemplate) ioc.getBean("jdbcTemplate");    //实验1：测试数据源  @Test  public void test1() throws SQLException {  DataSource dataSource = (DataSource) ioc.getBean("dataSource");  Connection connection = dataSource.getConnection();  System.out.println(connection);    }  } |

## 3、实验2（spring\_jdbc\_02\_template）

增、删、改

### 1）TestSpring

将emp\_id=5的记录的salary字段更新为1300.00

|  |
| --- |
| package com.alex.spring.junit;  import java.sql.SQLException;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import org.springframework.jdbc.core.JdbcTemplate;  public class TestSpring {  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");  JdbcTemplate jdbcTemplate = (JdbcTemplate) ioc.getBean("jdbcTemplate");    //实验2：将emp\_id=5的记录的salary字段更新为1300.00  @Test  public void test2() throws SQLException {  String sql = "update employee set salary=? where emp\_id=?";  Object[] object = {1300.0,5};  int update = jdbcTemplate.update(sql, object);  System.out.println(update > 0 ? "成功":"失败");  }  } |

**尖叫提示：其它同实验1**

## 4、实验3（spring\_jdbc\_03\_template）

批量增、删、改

批量插入

### 1）TestSpring

|  |
| --- |
| package com.alex.spring.junit;  import java.sql.SQLException;  import java.util.ArrayList;  import java.util.List;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import org.springframework.jdbc.core.JdbcTemplate;  public class TestSpring {  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");  JdbcTemplate jdbcTemplate = (JdbcTemplate) ioc.getBean("jdbcTemplate");    //实验3：批量插入  @Test  public void test3() throws SQLException {  String sql = "insert into employee(emp\_name,salary) values(?,?)";  List<Object[]> batchAtgs = new ArrayList<Object[]>();  batchAtgs.add(new Object[]{"aaa",12345.8});  batchAtgs.add(new Object[]{"bbb",3456.8});  batchAtgs.add(new Object[]{"ccc",7856.8});  batchAtgs.add(new Object[]{"ddd",98456.8});  int[] batchUpdate = jdbcTemplate.batchUpdate(sql,batchAtgs);  for(int i : batchUpdate){  System.out.println(i > 0 ? "成功":"失败");  }  }  } |

**尖叫提示：其它同实验1**

## 5、实验4（spring\_jdbc\_04\_template）

查询emp\_id=5的数据库记录，封装为一个Java对象返回

### 1）Employee

|  |
| --- |
| package com.alex.spring.entity;  public class Employee {  // 使用Integer 不赋值的时候是null,int 不赋值的时候是0  // Integer 是面向对象的，是引用类型  // int 是做计算用的，封装一个实体类是不涉及任何一个实体类的，所以这个时候尽量不要用小写的int,就用包装类型就可以了  private Integer id;  private String empName;  private Double salary;  public Integer getId() {  return id;  }  public void setId(Integer id) {  this.id = id;  }  public String getEmpName() {  return empName;  }  public void setEmpName(String empName) {  this.empName = empName;  }  public Double getSalary() {  return salary;  }  public void setSalary(Double salary) {  this.salary = salary;  }  //为了打印方便，重写toString方法  @Override  public String toString() {  return "Employee [id=" + id + ", empName=" + empName + ", salary=" + salary + "]";  }  } |

### 2）TestSpring

|  |
| --- |
| package com.alex.spring.junit;  import java.sql.SQLException;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import org.springframework.jdbc.core.BeanPropertyRowMapper;  import org.springframework.jdbc.core.JdbcTemplate;  import org.springframework.jdbc.core.RowMapper;  import com.alex.spring.entity.Employee;  public class TestSpring {  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");  JdbcTemplate jdbcTemplate = (JdbcTemplate) ioc.getBean("jdbcTemplate");    //实验4：查询emp\_id=5的数据库记录，封装为一个Java对象返回  @SuppressWarnings("unchecked")  @Test  public void test4() throws SQLException {  //String sql = "select \* from employee where emp\_id=?";  //sql 语句查询结果映射到对象属性中，  // salary -> salary 是什么就是什么  // emp\_name -> empName 下划线去掉，下划线后边的首字母变大写拼接在一起  // emp\_id as id -> id 实在找不到，可以通过as 起别名的方式来映射上  // 这个原理是非常重要的，因为后边 mybatis 也是这么做的  //打乱查询顺序，看看能不能封装进来  //String sql = "select salary,emp\_name,emp\_id from employee where emp\_id=?";  String sql = "select salary,emp\_name,emp\_id as id from employee where emp\_id=?";  //Ctrl + T  @SuppressWarnings("rawtypes")  RowMapper rowMapper = new BeanPropertyRowMapper(Employee.class);  Employee employee = (Employee) jdbcTemplate.queryForObject(sql, rowMapper,5);  System.out.println(employee);  }  } |

**尖叫提示：其它同实验1**

## 6、实验5（spring\_jdbc\_05\_template）

查询salary>4000的数据库记录，封装为List集合返回

### 1）TestSpring

|  |
| --- |
| package com.alex.spring.junit;  import java.sql.SQLException;  import java.util.List;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import org.springframework.jdbc.core.BeanPropertyRowMapper;  import org.springframework.jdbc.core.JdbcTemplate;  import org.springframework.jdbc.core.RowMapper;  import com.alex.spring.entity.Employee;  public class TestSpring {  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");  JdbcTemplate jdbcTemplate = (JdbcTemplate) ioc.getBean("jdbcTemplate");    //实验5：查询salary>4000的数据库记录，封装为List集合返回  @Test  public void test5() throws SQLException {  String sql = "select salary,emp\_name,emp\_id as id from employee where salary>?";  RowMapper<Employee> rowMapper = new BeanPropertyRowMapper<Employee>(Employee.class);  List<Employee> emplist = jdbcTemplate.query(sql, rowMapper,4000);  for(Employee employee:emplist){  System.out.println(employee);  }  }  } |

**尖叫提示：其它同实验4**

## 7、实验6（spring\_jdbc\_06\_template）

查询最大salary

### 1）TestSpring

|  |
| --- |
| package com.alex.spring.junit;  import java.sql.SQLException;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import org.springframework.jdbc.core.JdbcTemplate;  public class TestSpring {  ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");  JdbcTemplate jdbcTemplate = (JdbcTemplate) ioc.getBean("jdbcTemplate");  // 查询最大salary  @Test  public void test6() throws SQLException {  String sql = "select max(salary) from employee";  Double maxSalary = jdbcTemplate.queryForObject(sql, Double.class);  System.out.println(maxSalary);  }  } |

**尖叫提示：其它同实验4**

# 十二、Spring 事务

## 1、事务的概述

事务是保证一系列操作的完整性和一致性.一般是作为一个整体来进行操作.最终目的还是保证数据的安全.

1）在JavaEE企业级开发的应用领域，为了保证数据的完整性和一致性，必须引入数据库事务的概念，所以事务管理是企业级应用程序开发中必不可少的技术。

2）事务就是一组由于逻辑上紧密关联而合并成一个整体(工作单元)的多个数据库操作，这些操作要么都执行，要么都不执行。

## 2、事务的特性（ACID）

（1）原子性(atomicity)

“原子”的本意是“不可再分”，事务的原子性表现为一个事务中涉及到的多个操作在逻辑上缺一不可。事务的原子性要求事务中的所有操作要么都执行，要么都不执行。

（2）一致性(consistency)

“一致”指的是数据的一致，具体是指：所有数据都处于满足业务规则的一致性状态。一致性原则要求：一个事务中不管涉及到多少个操作，都必须保证事务执行之前数据是正确的，事务执行之后数据仍然是正确的。如果一个事务在执行的过程中，其中某一个或某几个操作失败了，则必须将其他所有操作撤销，将数据恢复到事务执行之前的状态，这就是回滚。

（3）隔离性(isolation)

在应用程序实际运行过程中，事务往往是并发执行的，所以很有可能有许多事务同时处理相同的数据，因此每个事务都应该与其他事务隔离开来，防止数据损坏。隔离性原则要求多个事务在并发执行过程中不会互相干扰。

（4）持久性(durability)

持久性原则要求事务执行完成后，对数据的修改永久的保存下来，不会因各种系统错误或其他意外情况而受到影响。通常情况下，事务对数据的修改应该被写入到持久化存储器中。

## 3、事务的隔离性

### 1）事务执行过程中可能出现哪些问题?

（1）数据丢失.

例如:张三给李四汇款,张三的钱减少了,但是李四的钱没增加.

（2）脏读数据

例如:

一个事务T1读取数据时,另一个事务T2对同一个数据进行修改,导致T1事务两次读取的数据不一致.

因为T2没有提交事务,T1就把T2修改的数据看到了,这就是脏数据,因为T2有可能回滚事务.

（3）不可重复读.

例如:

一个事务T1读取数据时,另一个事务T2对同一个数据进行修改,并且提交事务.

导致T1读取的两次结果是不一致的.

**尖叫提示：脏读和不可重复读的区别是脏读是可以回滚的，不可重复读，读到的数据是已经提交事务的**

（4）幻读:

一个事务T1正在做统计操作,另外的事务T2对数据的数据进行修改.

当T3事务又来进行同样的统计操作,导致T1和T3两个事务统计的结果不一致.

### 2）事务隔离级别

（1）1未提交读

int TRANSACTION\_READ\_UNCOMMITTED = 1;

解决:数据丢失

（2）2已提交读

int TRANSACTION\_READ\_COMMITTED = 2;

解决:数据丢失,脏读数据

（3）4可重复读

int TRANSACTION\_REPEATABLE\_READ = 4;

解决:数据丢失,脏读数据,不可重复读

（4）8序列化/不可并发

int TRANSACTION\_SERIALIZABLE = 8;

解决:数据丢失,脏读数据,不可重复读,幻读

（5）说明:

事务隔离级别越高,数据操作越安全

MySQL: 默认事务级别4

Oracle : 默认事务级别2

如果修改事务隔离级别:

void setTransactionIsolation(int level) throws SQLException;

获取事务隔离级别

int getTransactionIsolation() throws SQLException;

设置只读事务

void setReadOnly(boolean readOnly) throws SQLException;

只读事务的好处，数据库底层对只读查询进行优化处理,提高查询效率.

事务控制:JDBC

|  |
| --- |
| Connection con = …. ;  try{  //void setAutoCommit(boolean autoCommit) throws SQLException;  //JDBC中事务默认是自动提交的.如果希望手动提交事务,需要设置为false  con.setAutoCommit(false); //开启事务  con.setTransactionIsolation(Connection.TRANSACTION\_READ\_COMMITTED); //修改事务默认隔离级别:  //进行一系列SQL操作  con.commit(); //提交事务  }catch(Exception e){  con.rollback() ;//回滚事务  }finally{  con.close();  } |

## 4、事务管理方式

1）手动编程式事务:

2）自动声明式事务:

（1）表示框架来管理事务,只需要做简单配置,框架就会自动增加事务操作.

（2）事务就是一个AOP切面.

## 5、声明式事务（搭建环境）

### 1）事务边界

（1）什么时候开启事务,什么时候结束事务.

（2）在项目中,事务一般都是加在业务层方法上的.也就是业务层方法调用前开启事务,业务层方法调用完成提交事务,如果有异常就回滚事务.

### 2）Spring的声明式事务

基于注解的开发方式

（1）启用基于注解的声明式事务

|  |
| --- |
| <!-- 事务切面 -->  <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  <property name="dataSource" ref="dataSource"></property>  </bean>  <!-- 开启基于注解的声明式事务  transaction-manager="transactionManager" 默认值,可以省略.  -->  <tx:annotation-driven transaction-manager="transactionManager"/> |

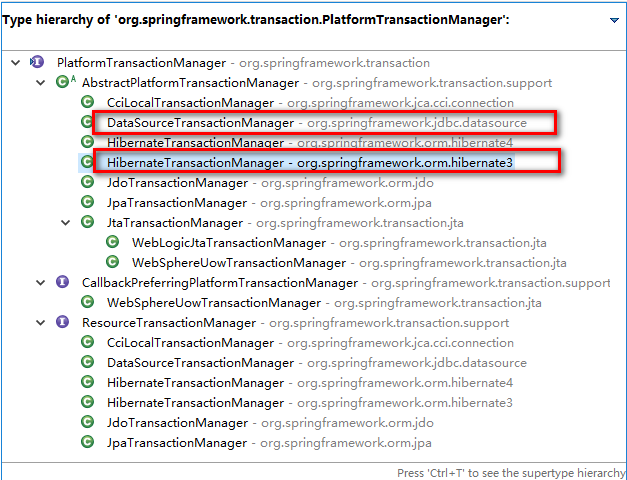
（2）在业务方法上增加@Transactional注解进行修饰.

### 3）声明式事务原理

Spring框架集成持久化层技术,管理事务,定义PlatformTransactionManager接口:

DataSourceTransactionManager : 可以用于集成JdbcTemplate,以及MyBatis框架.

HibernateTransactionManager:可以用于集成Hibernate框架.

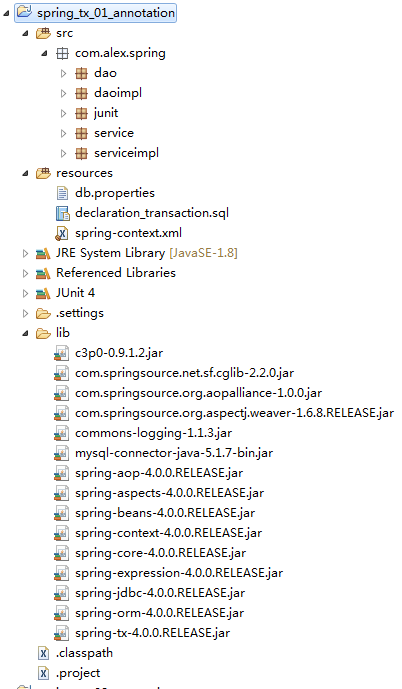


### 4）案例实操（没有使用事务spring\_tx\_01\_annotation）

在没有使用事务的情况下，演示sql异常，比如说up@erdate和没有异常

#### （1）项目结构

主要看jar包



#### （2）db.properties

|  |
| --- |
| user=root  password=root  jdbcUrl=jdbc:mysql://localhost:3306/tx  driverClass=com.mysql.jdbc.Driver  initialPoolSize=30  minPoolSize=10  maxPoolSize=100  acquireIncrement=5  maxStatements=1000  maxStatementsPerConnection=10 |

#### （3）declaration\_transaction.sql

|  |
| --- |
| /\*  SQLyog Ultimate v9.20  MySQL - 5.1.37-community : Database - tx  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*/    /\*!40101 SET NAMES utf8 \*/;  /\*!40101 SET SQL\_MODE=''\*/;  /\*!40014 SET @OLD\_UNIQUE\_CHECKS=@@UNIQUE\_CHECKS, UNIQUE\_CHECKS=0 \*/;  /\*!40014 SET @OLD\_FOREIGN\_KEY\_CHECKS=@@FOREIGN\_KEY\_CHECKS, FOREIGN\_KEY\_CHECKS=0 \*/;  /\*!40101 SET @OLD\_SQL\_MODE=@@SQL\_MODE, SQL\_MODE='NO\_AUTO\_VALUE\_ON\_ZERO' \*/;  /\*!40111 SET @OLD\_SQL\_NOTES=@@SQL\_NOTES, SQL\_NOTES=0 \*/;  CREATE DATABASE /\*!32312 IF NOT EXISTS\*/`tx` /\*!40100 DEFAULT CHARACTER SET gb2312 \*/;  USE `tx`;  /\*Table structure for table `account` \*/  DROP TABLE IF EXISTS `account`;  CREATE TABLE `account` (  `username` varchar(50) NOT NULL,  `balance` int(11) DEFAULT NULL,  PRIMARY KEY (`username`)  ) ENGINE=InnoDB DEFAULT CHARSET=gb2312;  /\*Data for the table `account` \*/  insert into `account`(`username`,`balance`) values ('Jerry',800),('Tom',100000);  /\*Table structure for table `book` \*/  DROP TABLE IF EXISTS `book`;  CREATE TABLE `book` (  `isbn` varchar(50) NOT NULL,  `book\_name` varchar(100) DEFAULT NULL,  `price` int(11) DEFAULT NULL,  PRIMARY KEY (`isbn`)  ) ENGINE=InnoDB DEFAULT CHARSET=gb2312;  /\*Data for the table `book` \*/  insert into `book`(`isbn`,`book\_name`,`price`) values ('ISBN-001','book01',100),('ISBN-002','book02',200),('ISBN-003','book03',300),('ISBN-004','book04',400),('ISBN-005','book05',500);  /\*Table structure for table `book\_stock` \*/  DROP TABLE IF EXISTS `book\_stock`;  CREATE TABLE `book\_stock` (  `isbn` varchar(50) NOT NULL,  `stock` int(11) DEFAULT NULL,  PRIMARY KEY (`isbn`)  ) ENGINE=InnoDB DEFAULT CHARSET=gb2312;  /\*Data for the table `book\_stock` \*/  insert into `book\_stock`(`isbn`,`stock`) values ('ISBN-001',1000),('ISBN-002',2000),('ISBN-003',3000),('ISBN-004',4000),('ISBN-005',5000);  /\*!40101 SET SQL\_MODE=@OLD\_SQL\_MODE \*/;  /\*!40014 SET FOREIGN\_KEY\_CHECKS=@OLD\_FOREIGN\_KEY\_CHECKS \*/;  /\*!40014 SET UNIQUE\_CHECKS=@OLD\_UNIQUE\_CHECKS \*/;  /\*!40111 SET SQL\_NOTES=@OLD\_SQL\_NOTES \*/; |

#### （4）spring-context.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"  xmlns:aop="http://www.springframework.org/schema/aop"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd  http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd">  <!-- 设置扫描包 -->  <context:component-scan base-package="com.alex.spring"></context:component-scan>  <!-- 启动AOP注解开发,自动生成代理对象. -->  <aop:aspectj-autoproxy />  <!-- 读取外部文件 -->  <context:property-placeholder location="classpath:/db.properties" />  <!-- C3P0 数据源 -->  <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">  <property name="user" value="${user}" />  <property name="password" value="${password}" />  <property name="jdbcUrl" value="${jdbcUrl}" />  <property name="driverClass" value="${driverClass}" />  <property name="initialPoolSize" value="${initialPoolSize}" />  <property name="minPoolSize" value="${minPoolSize}" />  <property name="maxPoolSize" value="${maxPoolSize}" />  <property name="acquireIncrement" value="${acquireIncrement}" />  <property name="maxStatements" value="${maxStatements}" />  <property name="maxStatementsPerConnection" value="${maxStatementsPerConnection}" />  </bean>  <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">  <property name="dataSource" ref="dataSource" />  </bean>  </beans> |

#### （5）BookShopDao

|  |
| --- |
| package com.alex.spring.dao;  public interface BookShopDao {  // [1]根据isbn的值查询书的价格  int findBookPriceByIsbn(String isbn);    // [2]根据isbn的值减少书的库存，假设每次都只买1本书  int updateBookStock(String isbn);    // [3]根据用户名减少用户账户中的余额，减少的额度就是书的价格  int updateAccount(String username,int price);  } |

#### （6）BookShopDaoImpl

|  |
| --- |
| package com.alex.spring.daoimpl;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.jdbc.core.JdbcTemplate;  import org.springframework.stereotype.Repository;  import com.alex.spring.dao.BookShopDao;  @Repository  public class BookShopDaoImpl implements BookShopDao {  @Autowired  private JdbcTemplate jdbcTemplate ;    @Override  public int findBookPriceByIsbn(String isbn) {  String sql = "select price from book where isbn=?";  return jdbcTemplate.queryForObject(sql, Integer.class, isbn);  }  @Override  public int updateBookStock(String isbn) {  String sql = "update book\_stock set stock=stock-1 where isbn=?";  return jdbcTemplate.update(sql, isbn);  }  @Override  public int updateAccount(String username, int price) {  String sql = "update account set balance=balance-? where username=?";  return jdbcTemplate.update(sql, price,username);  }  } |

#### （7）BookShopService

|  |
| --- |
| package com.alex.spring.service;  import java.io.FileNotFoundException;  public interface BookShopService {  public void purchase(String isbn,String username) throws FileNotFoundException;  } |

#### （8）BookShopServiceImpl

|  |
| --- |
| package com.alex.spring.serviceimpl;  import java.io.FileNotFoundException;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Service;  import com.alex.spring.dao.BookShopDao;  import com.alex.spring.service.BookShopService;  @Service  public class BookShopServiceImpl implements BookShopService {  @Autowired  private BookShopDao bookShopDao ;  @Override  public void purchase(String isbn, String username) throws FileNotFoundException {    int price = bookShopDao.findBookPriceByIsbn(isbn);  System.out.println(isbn + "这本书，它的价格是： price:" + price);    bookShopDao.updateBookStock(isbn);  bookShopDao.updateAccount(username, price);  }  } |

#### （9）TestSpringTx

|  |
| --- |
| package com.alex.spring.junit;  import java.io.FileNotFoundException;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.alex.spring.service.BookShopService;  public class TestSpringTx {    ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");  BookShopService bookShopService = ioc.getBean(BookShopService.class);    @Test  public void test() throws FileNotFoundException {  System.out.println("bookShopService="+bookShopService.getClass());  bookShopService.purchase("ISBN-001", "Tom");  }  } |

### 5）案例实操（使用了事务 spring\_tx\_02\_annotation）

在使用了事务的情况下，演示sql异常，比如说up@erdate，和没有异常

#### （1）spring-context.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:aop="http://www.springframework.org/schema/aop"  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-4.0.xsd  http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd  http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.0.xsd">  <!-- 设置扫描包 -->  <context:component-scan base-package="com.alex.spring"></context:component-scan>  <!-- 启动AOP注解开发,自动生成代理对象. -->  <aop:aspectj-autoproxy />  <!-- 读取外部文件 -->  <context:property-placeholder location="classpath:/db.properties" />  <!-- C3P0 数据源 -->  <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">  <property name="user" value="${user}" />  <property name="password" value="${password}" />  <property name="jdbcUrl" value="${jdbcUrl}" />  <property name="driverClass" value="${driverClass}" />  <property name="initialPoolSize" value="${initialPoolSize}" />  <property name="minPoolSize" value="${minPoolSize}" />  <property name="maxPoolSize" value="${maxPoolSize}" />  <property name="acquireIncrement" value="${acquireIncrement}" />  <property name="maxStatements" value="${maxStatements}" />  <property name="maxStatementsPerConnection" value="${maxStatementsPerConnection}" />  </bean>  <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">  <property name="dataSource" ref="dataSource" />  </bean>    <!-- 事务切面 -->  <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  <property name="dataSource" ref="dataSource"></property>  </bean>    <!-- 开启基于注解的声明式事务  transaction-manager="transactionManager" 默认值,可以省略.  当事务切面的id是transactionManager的时候，transaction-manager="transactionManager"是可以省略的  -->  <tx:annotation-driven transaction-manager="transactionManager"/>  </beans> |

#### （2）BookShopServiceImpl

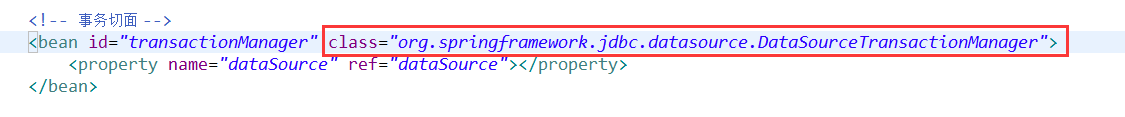
|  |
| --- |
| package com.alex.spring.serviceimpl;  import java.io.FileNotFoundException;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Service;  import org.springframework.transaction.annotation.Transactional;  import com.alex.spring.dao.BookShopDao;  import com.alex.spring.service.BookShopService;  @Service  **//@Transactional // 事务注解是可以放到类上面的，这样类中的所有方法都是带事务的**  public class BookShopServiceImpl implements BookShopService {  @Autowired  private BookShopDao bookShopDao ;  **@Transactional**  @Override  public void purchase(String isbn, String username) throws FileNotFoundException {    int price = bookShopDao.findBookPriceByIsbn(isbn);  System.out.println(isbn + "这本书，它的价格是： price:" + price);    bookShopDao.updateBookStock(isbn);  bookShopDao.updateAccount(username, price);  }  } |

**尖叫提示：其他内容同spring\_tx\_01\_annotation项目**

### 6）案例实操（事务原理 spring\_tx\_03\_annotation）

#### （1）TestSpringTx

|  |
| --- |
| package com.alex.spring.junit;  import java.io.FileNotFoundException;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import com.alex.spring.service.BookShopService;  import com.alex.spring.serviceimpl.BookShopServiceImpl;  public class TestSpringTx {    ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-context.xml");  BookShopService bookShopService = ioc.getBean(BookShopService.class); // √ 正确  //有接口的时候，使用的是jdbc的代理，没有接口的时候，使用的是Cglib 动态代理  //BookShopService bookShopService1 = (BookShopService) ioc.getBean("bookShopServiceImpl"); //√ 正确  //BookShopService bookShopService2 = ioc.getBean(BookShopServiceImpl.class); //× 不正确    @Test  public void test() throws FileNotFoundException {  System.out.println("bookShopService="+bookShopService.getClass());  bookShopService.purchase("ISBN-001", "Tom");  }  } |



**尖叫提示：其他内容同spring\_tx\_02\_annotation项目**

## 6、声明式事务相关概念

### 1）事务传播行为（spring\_tx\_04\_annotation）

方法嵌套调用时,事务的开启的方式.

（1）propagation=Propagation.REQUIRED 默认传播行为

表示a方法调用b方法时:

a方法有事务,那么b方法使用a方法开启的事务.

a方法没有事务,那么b方法要为自己开启一个新事务

（2）propagation=Propagation.REQUIRES\_NEW

表示a方法调用b方法时:

a方法不管有没有事务,b方法总为自己开启一个新事务.

案例实操

BookShopServiceImpl

|  |
| --- |
| package com.alex.spring.serviceimpl;  import java.io.FileNotFoundException;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Service;  import org.springframework.transaction.annotation.Propagation;  import org.springframework.transaction.annotation.Transactional;  import com.alex.spring.dao.BookShopDao;  import com.alex.spring.service.BookShopService;  @Service  public class BookShopServiceImpl implements BookShopService {  @Autowired  private BookShopDao bookShopDao;  @Override  @Transactional(propagation = Propagation.REQUIRED)  public void purchase(String isbn, String username) throws FileNotFoundException {  int price = bookShopDao.findBookPriceByIsbn(isbn);  System.out.println(isbn + "这本书，它的价格是： price:" + price);  bookShopDao.updateBookStock(isbn);    try {  a(username, price);  } catch (Exception e) {  e.printStackTrace();  //throw e;  }  }  @Transactional(propagation = Propagation.REQUIRES\_NEW)  public void a(String username, int price) {  bookShopDao.updateAccount(username, price);  }  } |

**尖叫提示：其他内容同spring\_tx\_02\_annotation项目**

### 2）事务隔离级别（spring\_tx\_05\_annotation）

MySQL : 4 默认值 //debug调用， 修改数据库

isolation=Isolation.READ\_UNCOMMITTED

select \* from book where isbn='ISBN-001' for update; //行级锁

表示对查询行进行锁定.类似多线程的同步锁.

案例实操

BookShopServiceImpl

|  |
| --- |
| package com.alex.spring.serviceimpl;  import java.io.FileNotFoundException;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Service;  import org.springframework.transaction.annotation.Isolation;  import org.springframework.transaction.annotation.Propagation;  import org.springframework.transaction.annotation.Transactional;  import com.alex.spring.dao.BookShopDao;  import com.alex.spring.service.BookShopService;  @Service  public class BookShopServiceImpl implements BookShopService {  @Autowired  private BookShopDao bookShopDao;  @Override  @Transactional(propagation = Propagation.REQUIRED, isolation = Isolation.DEFAULT)  public void purchase(String isbn, String username) throws FileNotFoundException {  // debug调用，修改数据库  // isolation=Isolation.DEFAULT  // isolation=Isolation.READ\_COMMITTED  int price = bookShopDao.findBookPriceByIsbn(isbn);  price = bookShopDao.findBookPriceByIsbn(isbn);  System.out.println(isbn + "这本书，它的价格是： price:" + price);  bookShopDao.updateBookStock(isbn);  bookShopDao.updateAccount(username, price);  }  } |

**尖叫提示：其他内容同spring\_tx\_02\_annotation项目**

isolation=Isolation.READ\_UNCOMMITTED

|  |
| --- |
| window + R -> cmd  mysql> mysql -uroot -proot  mysql> use tx;  mysql> set autocommit=false  mysql> update book set price = 500 where isbn=’ISBN-001’;  mysql> rollback;  mysql> select \* from book where isbn=’ISBN-001’; |

### 3）事务回滚策略（spring\_tx\_06\_annotation）

Spring框架事务回滚策略:

默认对RuntimeException,Error异常类型进行事务回滚;对Exception编译期异常默认不进行回滚的.

如果需要修改默认事务回滚策略:

rollbackFor= {Exception.class} 表示设置遇到哪种异常需要事务回滚

noRollbackFor= {FileNotFoundException.class} 表示设置遇到哪种异常不需要回滚.

案例实操

BookShopServiceImpl

|  |
| --- |
| package com.alex.spring.serviceimpl;  import java.io.FileNotFoundException;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Service;  import org.springframework.transaction.annotation.Transactional;  import com.alex.spring.dao.BookShopDao;  import com.alex.spring.service.BookShopService;  @Service  public class BookShopServiceImpl implements BookShopService {  @Autowired  private BookShopDao bookShopDao;  @Override  @Transactional(rollbackFor = { Exception.class },  noRollbackFor = { FileNotFoundException.class })  public void purchase(String isbn, String username) throws FileNotFoundException {  int price = bookShopDao.findBookPriceByIsbn(isbn);  System.out.println(isbn + "这本书，它的价格是： price:" + price);  bookShopDao.updateBookStock(isbn);  // Spring框架事务回滚策略:  // 默认对RuntimeException,Error异常类型进行事务回滚;对Exception编译期异常默认不进行回滚的.  // java.lang.ArithmeticException: / by zero  // int i = 1/0 ;  // java.io.FileNotFoundException: D:\xx\yy\zz\abc.txt (系统找不到指定的路径。)  // File file = new File("D:/xx/yy/zz/abc.txt");  // FileInputStream fis = new FileInputStream(file);  bookShopDao.updateAccount(username, price);  }  } |

**尖叫提示：其他内容同spring\_tx\_02\_annotation项目**

### 4）只读属性（spring\_tx\_07\_annotation）

readOnly=true 表示连接是只读的.对事务数据只能查看,不能修改.否则,会报异常.

作用:如果以只读的方式来操作数据,数据库底层会进行事务的优化,提高数据访问效率.

Caused by: java.sql.SQLException: Connection is read-only. Queries leading to data modification are not allowed

案例实操

BookShopServiceImpl

|  |
| --- |
| package com.alex.spring.serviceimpl;  import java.io.FileNotFoundException;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Service;  import org.springframework.transaction.annotation.Transactional;  import com.alex.spring.dao.BookShopDao;  import com.alex.spring.service.BookShopService;  @Service  public class BookShopServiceImpl implements BookShopService {  @Autowired  private BookShopDao bookShopDao ;    @Override  @Transactional(readOnly=true)  public void purchase(String isbn, String username) throws FileNotFoundException {    int price = bookShopDao.findBookPriceByIsbn(isbn);  System.out.println(isbn + "这本书，它的价格是： price:" + price);    bookShopDao.updateBookStock(isbn);  bookShopDao.updateAccount(username, price);  }  } |

**尖叫提示：其他内容同spring\_tx\_02\_annotation项目**

### 5）超时属性（spring\_tx\_08\_annotation）

（1）作用

如果事务进行过中时间比较长,会过长占用系统资源.

可以设置超时数据,在规定时间内完成不了事务,就立即释放资源,提高提高系统可用性.

（2）timeout=3

org.springframework.transaction.TransactionTimedOutException: Transaction timed out: deadline was Fri Oct 20 16:47:05 CST 2017

案例实操

BookShopServiceImpl

|  |
| --- |
| package com.alex.spring.serviceimpl;  import java.io.FileNotFoundException;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Service;  import org.springframework.transaction.annotation.Transactional;  import com.alex.spring.dao.BookShopDao;  import com.alex.spring.service.BookShopService;  @Service  public class BookShopServiceImpl implements BookShopService {  @Autowired  private BookShopDao bookShopDao ;    @Override  @Transactional(timeout=3)//单位是秒  public void purchase(String isbn, String username) throws FileNotFoundException {    //org.springframework.transaction.TransactionTimedOutException: Transaction timed out: deadline was Fri Oct 20 16:47:05 CST 2017  //线程休眠放置事务开始位置,会记录在超时时间访问内,放置语句最后,不计算在内.  try {  Thread.sleep(5000);  } catch (InterruptedException e) {  e.printStackTrace();  }    int price = bookShopDao.findBookPriceByIsbn(isbn);  System.out.println(isbn + "这本书，它的价格是： price:" + price);    bookShopDao.updateBookStock(isbn);  bookShopDao.updateAccount(username, price);  }  } |

**尖叫提示：其他内容同spring\_tx\_02\_annotation项目**

## 7、基于XML的声明式事务（spring\_tx\_09\_annotation）

### 1）BookShopServiceImpl

去掉注释方式的事务

|  |
| --- |
| package com.alex.spring.serviceimpl;  import java.io.FileNotFoundException;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Service;  import com.alex.spring.dao.BookShopDao;  import com.alex.spring.service.BookShopService;  @Service  public class BookShopServiceImpl implements BookShopService {  @Autowired  private BookShopDao bookShopDao ;  @Override  public void purchase(String isbn, String username) throws FileNotFoundException {    int price = bookShopDao.findBookPriceByIsbn(isbn);  System.out.println(isbn + "这本书，它的价格是： price:" + price);    bookShopDao.updateBookStock(isbn);  bookShopDao.updateAccount(username, price);  }  } |

### 2）spring-context.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"  xmlns:aop="http://www.springframework.org/schema/aop" 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-4.0.xsd  http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd  http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.0.xsd">  <!-- 设置扫描包 -->  <context:component-scan base-package="com.alex.spring"></context:component-scan>  <!-- 启动AOP注解开发,自动生成代理对象. -->  <aop:aspectj-autoproxy />  <!-- 读取外部文件 -->  <context:property-placeholder location="classpath:/db.properties" />  <!-- C3P0 数据源 -->  <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">  <property name="user" value="${user}" />  <property name="password" value="${password}" />  <property name="jdbcUrl" value="${jdbcUrl}" />  <property name="driverClass" value="${driverClass}" />  <property name="initialPoolSize" value="${initialPoolSize}" />  <property name="minPoolSize" value="${minPoolSize}" />  <property name="maxPoolSize" value="${maxPoolSize}" />  <property name="acquireIncrement" value="${acquireIncrement}" />  <property name="maxStatements" value="${maxStatements}" />  <property name="maxStatementsPerConnection" value="${maxStatementsPerConnection}" />  </bean>  <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">  <property name="dataSource" ref="dataSource" />  </bean>  <!-- 事务切面 -->  <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  <property name="dataSource" ref="dataSource"></property>  </bean>  <aop:config>  <aop:pointcut expression="execution(\* com.alex.spring..\*Service.\*(..))" id="txPointcut" />  <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut" />  </aop:config>  <!-- 配置事务通知 -->  <tx:advice id="txAdvice" transaction-manager="transactionManager">  <tx:attributes>  <tx:method name="add\*" propagation="REQUIRED" isolation="DEFAULT" rollback-for="java.lang.Exception" />  <tx:method name="delete\*" propagation="REQUIRED" isolation="DEFAULT" rollback-for="java.lang.Exception" />  <tx:method name="update\*" propagation="REQUIRED" isolation="DEFAULT" rollback-for="java.lang.Exception" />  <tx:method name="query\*" read-only="true" />  </tx:attributes>  </tx:advice>  </beans> |