

## 目录

前言.....	6
Spring 入门和 IOC 介绍.....	7
1. Spring 介绍.....	7
1.1 侵入式概念.....	8
1.2 松耦合概念.....	8
1.3 切面编程.....	9
2. 引出 Spring.....	11
2.1 IOC 控制反转.....	12
1. 不用自己组装，拿来就用。 .....	13
2. 享受单例的好处，效率高，不浪费空间。 .....	14
3. 便于单元测试，方便切换 mock 组件。 .....	14
4. 便于进行 AOP 操作，对于使用者是透明的。 .....	14
5. 统一配置，便于修改。 .....	14
3. Spring 模块.....	14
4. Core 模块快速入门.....	17
4.1 搭建配置环境.....	17
4.2 得到 Spring 容器对象【IOC 容器】.....	18
4.2.1 通过 Resource 获取 BeanFactory.....	18
4.2.2 类路径下 XML 获取 ApplicationContext.....	19
4.3 XML 配置方式.....	20
4.3.1 带参数的构造函数创建对象.....	22
4.3.2 工厂静态方法创建对象.....	23
4.3.3 工厂非静态方法创建对象.....	24
4.3.4 c 名称空间.....	25
4.3.5 装载集合.....	25
4.4 注解方式.....	26
4.5 通过 JavaConfig 方式.....	31
4.6 三种方式混合使用？ .....	33
5. bean 对象创建细节.....	34
5.1 scope 属性.....	34
5.2 lazy-init 属性.....	36
5.3 init-method 和 destroy-method.....	36
5.4 Bean 创建细节总结.....	37

对象依赖.....	39
1. 回顾以前对象依赖.....	39
1.1 直接 new 对象.....	39
1.2 写 DaoFactory , 用字符串来维护依赖关系.....	39
1.3 DaoFactory 读取配置文件.....	40
2. Spring 依赖注入.....	41
2.1 搭建测试环境.....	42
2.2 构造函数给属性赋值.....	43
2.3 通过 set 方法给属性注入值.....	46
2.4 内部 Bean.....	47
2.5 p 名称空间注入属性值.....	48
2.6 自动装配.....	49
2.6.1 XML 配置根据名字.....	49
2.6.2 XML 配置根据类型.....	50
2.7 使用注解来实现自动装配.....	51
AOP 入门.....	54
1. cglib 代理.....	54
1.1 编写 cglib 代理.....	54
2. 手动实现 AOP 编程.....	57
2.1 案例分析 : .....	58
2.2 工厂静态方法 : .....	62
2.3 工厂非静态方法.....	64
3. AOP 的概述.....	66
4. 使用 Spring AOP 开发步骤.....	69
1) 先引入 aop 相关 jar 文件 ( aspectj aop 优秀组件 ) .....	69
2) bean.xml 中引入 aop 名称空间.....	70
4.1 引入名称空间.....	70
4.2 注解方式实现 AOP 编程.....	71
4.2.1 在配置文件中开启 AOP 注解方式.....	71
4.2.2 代码 : .....	71
4.3 目标对象没有接口.....	74
4.4 AOP 注解 API.....	75
4.5 表达式优化.....	77
4.6 XML 方式实现 AOP 编程.....	78

5. 切入点表达式.....	82
5.1 查官方文档.....	82
5.2 语法解析.....	83
5.3 测试代码.....	84
JDBCTemplate 和 Spring 事务.....	86
1. 回顾对模版代码优化过程.....	86
2. 使用 Spring 的 JDBC.....	89
2.1 JdbcTemplate 查询.....	94
3. 事务控制概述.....	99
3.1 编程式事务控制.....	99
3.2 声明式事务控制.....	99
3.3 声明式事务控制教程.....	100
3.3.1 搭建配置环境.....	100
3.3.2 XML 方式实现声明式事务控制.....	104
3.3.3 使用注解的方法实现事务控制.....	106
第一步和 XML 的是一样的，必须配置事务管理器类：.....	106
第二步：开启以注解的方式来实现事务控制.....	107
4. 事务属性.....	108
4.1 事务传播行为:.....	108
4.2 当事务传播行为是 Propagation.REQUIRED.....	109
4.3 当事务传播行为是 Propagation.REQUIRED_NEW.....	110
Spring 事务原理.....	112
一、阅读本文需要的基础知识.....	112
二、两个不靠谱直觉的例子.....	113
2.1 第一个例子.....	113
2.2 第二个例子.....	115
第二个例子来源于知乎@柳树文章，文末会给出相应的 URL.....	115
2.2.1 再延伸一下.....	117
三、Spring 事务传播机制.....	119
四、多线程问题.....	121
五、啥是 BPP ? .....	123
5.1 为什么特意讲 BPP ? .....	125
六、认识 Spring 事务几个重要的接口.....	125
Spring 事务的一个线程安全问题.....	128

一、我的思考.....	130
二、图解出现的原因.....	132
三、解决问题.....	135
最后.....	137
IOC 再回顾和面试题.....	138
一、Spring IOC 全面认知.....	139
1.1IOC 和 DI 概述.....	139
1. 不用自己组装，拿来就用。 .....	140
2. 享受单例的好处，效率高，不浪费空间。 .....	140
3. 便于单元测试，方便切换 mock 组件。 .....	140
4. 便于进行 AOP 操作，对于使用者是透明的。 .....	140
5. 统一配置，便于修改。 .....	140
1.2IOC 容器的原理.....	140
1. 根据 Bean 配置信息在容器内部创建 Bean 定义注册表.....	141
2. 根据注册表加载、实例化 bean、建立 Bean 与 Bean 之间的依赖关系.....	141
3. 将这些准备就绪的 Bean 放到 Map 缓存池中，等待应用程序调用.....	141
1.3IOC 容器装配 Bean.....	147
1.3.1 装配 Bean 方式.....	147
1.3.2 依赖注入方式.....	147
1.3.3 对象之间关系.....	147
1.3.4Bean 的作用域.....	148
1.3.6 处理自动装配的歧义性.....	150
1.3.7 引用属性文件以及 Bean 属性.....	150
1.3.8 组合配置文件.....	152
1.3.9 装配 Bean 总结.....	155
二、Spring IOC 相关面试题.....	159
一、.....	164
二、.....	164
AOP 再回顾.....	167
一、Spring AOP 全面认知.....	167
1.1AOP 概述.....	168
1.2Spring AOP 原理.....	170
1.3AOP 的实现者.....	171
1.4AOP 的术语.....	172

1.5Spring 对 AOP 的支持.....	173
二、基于代理的经典 SpringAOP.....	174
首先，我们来看一下增强接口的继承关系图：.....	174
三、拥抱基于注解和命名空间的 AOP 编程.....	179
3.1 使用引介/引入功能实现为 Bean 引入新方法.....	182
3.2 在 XML 中声明切面.....	186
四、总结.....	189

## 前言

这个文档的内容纯手打，如果想要看更多的干货文章，关注我的公众号：  
Java3y。有更多的原创技术文章和干货！

目前疯狂处于疯狂更新 PDF 中，只要是 Java 后端的知识，都会有！欢迎来我公众号催更！微信搜索：Java3y

如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！公众号有我的联系方式



- Java 精美脑图
- Java 学习路线

- 开发常用工具
- 精美原创电子书

在公众号下回复「888」即可获取！！

学习不能盲目，跟着我，会让你事半功倍

文档允许随意传播，但不能修改任何内容。

电子书的整理也是挺不容易，如果你觉得有帮助，想要打赏作者，那么可以通过这个收款码打赏我，**金额不重要，心意最重要**。主要是我可以通过这个打赏情况来预计大家对这本电子书的评价，嘻嘻

## Spring 入门和 IOC 介绍

### 1. Spring 介绍

Spring 诞生：

- 创建 Spring 的目的就是用来替代更加重量级的企业级 Java 技术
- 简化 Java 的开发
  - 基于 POJO 轻量级和最小侵入式开发
  - 通过依赖注入和面向接口实现松耦合
  - 基于切面和惯例进行声明式编程
  - 通过切面和模板减少样板式代码

## 1.1 侵入式概念

侵入式：对于 EJB、Struts2 等一些传统的框架，通常是要实现特定的接口，继承特定的类才能增强功能

- 改变了 java 类的结构

非侵入式：对于 Hibernate、Spring 等框架，对现有的类结构没有影响，就能够增强 JavaBean 的功能

## 1.2 松耦合概念

前面我们在写程序的时候，都是面向接口编程，通过 DaoFactory 等方法来实现松耦合

```
private CategoryDao categoryDao = DaoFactory.getInstance().createDao("zhongfuchen  
g.dao.impl.CategoryDAOImpl", CategoryDao.class);
```

```
private BookDao bookDao = DaoFactory.getInstance().createDao("zhongfucheng.dao.imp  
l.BookDAOImpl", BookDAO.class);
```

```
private UserDao userDao = DaoFactory.getInstance().createDao("zhongfucheng.dao.imp  
l.UserDAOImpl", UserDao.class);
```

```
private OrderDao orderDao = DaoFactory.getInstance().createDao("zhongfucheng.dao.i  
mp1.OrderDAOImpl", OrderDAO.class);
```

代码如下：

```
public class BussinessServiceImpl implements BussinessServiceDao {

    private CategoryDao categoryDao = DaoFactory.getInstance().createDao("zhongfucheng.dao.impl.CategoryDAOImpl", CategoryDao.class);

    private BookDao bookDao = DaoFactory.getInstance().createDao("zhongfucheng.dao.impl.BookDAOImpl", BookDAO.class);

    private UserDao userDao = DaoFactory.getInstance().createDao("zhongfucheng.dao.impl.UserDAOImpl", UserDao.class);

    private OrderDao orderDao = DaoFactory.getInstance().createDao("zhongfucheng.dao.impl.OrderDAOImpl", OrderDAO.class);

    @Override
    @permission("添加分类")
    /*添加分类*/
    public void addCategory(Category category) { categoryDao.addCategory(category); }

    /*查找分类*/
    @Override
    public void findCategory(String id) { categoryDao.findCategory(id); }

    @Override
    @permission("查找分类")
    /*查看分类*/
    public List<Category> getAllCategory() { return categoryDao.getAllCategory(); }

    /*添加图书*/
}
```

[http://blog.csdn.net/hon\\_3y](http://blog.csdn.net/hon_3y)

DAO 层和 Service 层通过 DaoFactory 来实现松耦合，如果 Serivce 层直接 new DaoBook()，那么 DAO 和 Service 就紧耦合了【Service 层依赖紧紧依赖于 Dao】。

而 Spring 给我们更加合适的方法来实现松耦合，并且更加灵活、功能更加强大！---|IOC 控制反转（这个后面会说）

### 1.3 切面编程

切面编程也就是 AOP 编程，其实我们在之前也接触过...动态代理就是一种切面编程了...

当时我们使用动态代理+注解的方式给 Service 层的方法添加权限.

```
@Override
@permission('添加分类')
/*添加分类*/
public void addCategory(Category category) {
    categoryDao.addCategory(category);
```

```
}

/*查找分类*/
@Override
public void findCategory(String id) {
    categoryDao.findCategory(id);
}

@Override
@permission("查找分类")
/*查看分类*/
public List<Category> getAllCategory() {
    return categoryDao.getAllCategory();
}

/*添加图书*/
@Override
public void addBook(Book book) {
    bookDao.addBook(book);
}
```

Controller 调用 Service 的时候，Service 返回的是一个代理对象，代理对象得到 Controller 想要调用的方法，通过反射来看看该方法上有没有注解

如果有注解的话，那么就判断该用户是否有权限来调用此方法，如果没有权限，就抛出异常给 Controller，Controller 接收到异常，就可以提示用户没有权限了。

AOP 编程可以简单理解成：在执行某些代码前，执行另外的代码（Struts2 的拦截器也是面向切面编程【在执行 Action 业务方法之前执行拦截器】）

Spring 也为我们提供更好地方式来实现面向切面编程！

## 2. 引出 Spring

我们试着回顾一下没学 Spring 的时候，是怎么开发 Web 项目的

- 1. 实体类---|class User{ }
- 2. daoclass--| UserDao{ .. 访问 db}
- 3. service---|class UserService{ UserDao userDao = new UserDao();}
- 4. actionclass UserAction{UserService userService = new UserService();}

用户访问：Tomcat-|servlet-|service-|dao

我们来思考几个问题：

- ①：对象创建能否写死？
- ②：对象创建细节

### — 对象数量

- action 多个 【维护成员变量】
- service 一个 【不需要维护公共变量】
- dao 一个 【不需要维护公共变量】

### — 创建时间

- action 访问时候创建
  - service 启动时候创建
  - dao 启动时候创建
- ③：对象的依赖关系
- action 依赖 service

## — service 依赖 dao

对于第一个问题和第三个问题，我们可以通过 DaoFactory 解决掉(虽然不是比较好  
的解决方法)

对于第二个问题，我们要控制对象的数量和创建时间就有点麻烦了....

而 Spring 框架通过 IOC 就很好地可以解决上面的问题....

## 2.1 IOC 控制反转

Spring 的核心思想之一：Inversion of Control , 控制反转 IOC

那么控制反转是什么意思呢？？？对象的创建交给外部容器完成，这个就做控制  
反转。

- Spring 使用控制反转来实现对象不用在程序中写死
- 控制反转解决对象处理问题【把对象交给别人创建】

那么对象的对象之间的依赖关系 Spring 是怎么做的呢？？依赖注入：dependency  
injection.

- Spring 使用依赖注入来实现对象之间的依赖关系
- 在创建完对象之后，对象的关系处理就是依赖注入

上面已经说了，控制反转是通过外部容器完成的，而 Spring 又为我们提供了这么  
一个容器，我们一般将这个容器叫做：IOC 容器。

无论是创建对象、处理对象之间的依赖关系、对象创建的时间还是对象的数量，  
我们都是在 Spring 为我们提供的 IOC 容器上配置对象的信息就好了。

那么使用 IOC 控制反转这一思想有什么作用呢？？？我们来看看一些优秀的回答...

来自知乎：<https://www.zhihu.com/question/23277575/answer/24259844>

我摘取一下核心的部分：

ioc 的思想最核心的地方在于，资源不由使用资源的双方管理，而由不使用资源的第三方管理，这可以带来很多好处。第一，资源集中管理，实现资源的可配置和易管理。第二，降低了使用资源双方的依赖程度，也就是我们说的耦合度。

也就是说，甲方要达成某种目的不需要直接依赖乙方，它只需要达到的目的告诉第三方机构就可以了，比如甲方需要一双袜子，而乙方它卖一双袜子，它要把袜子卖出去，并不需要自己去直接找到一个卖家来完成袜子的卖出。它也需要找第三方，告诉别人我要卖一双袜子。这下好了，甲乙双方进行交易活动，都不需要自己直接去找卖家，相当于程序内部开放接口，卖家由第三方作为参数传入。甲乙互相不依赖，而且只有在进行交易活动的时候，甲才和乙产生联系。反之亦然。这样做什么好处么呢，甲乙可以在对方不真实存在的情况下独立存在，而且保证不交易时候无联系，想交易的时候可以很容易的产生联系。甲乙交易活动不需要双方见面，避免了双方的互不信任造成交易失败的问题。因为交易由第三方来负责联系，而且甲乙都认为第三方可靠。那么交易就能很可靠很灵活的产生和进行了。这就是 ioc 的核心思想。生活中这种例子比比皆是，支付宝在整个淘宝体系里就是庞大的 ioc 容器，交易双方之外的第三方，提供可靠性可依赖可灵活变更交易方的资源管理中心。另外人事代理也是，雇佣机构和个人之外的第三方。

=====update=====  
=====

在以上的描述中，诞生了两个专业词汇，依赖注入和控制反转所谓的依赖注入，则是，甲方开放接口，在它需要的时候，能够讲乙方传递进来(注入)所谓的控制反转，甲乙双方不相互依赖，交易活动的进行不依赖于甲乙任何一方，整个活动的进行由第三方负责管理。

参考优秀的博文②：[这里写链接内容](#)

知乎@Intopass 的回答：

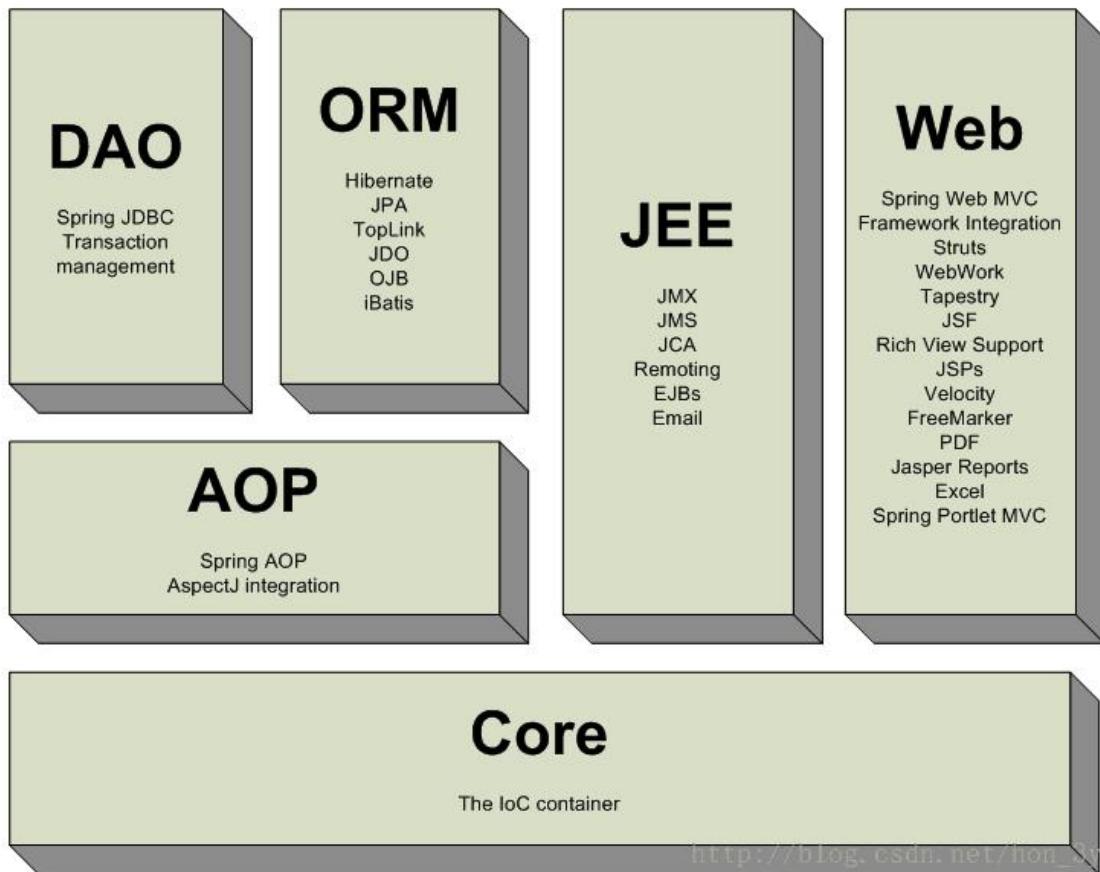
1. 不用自己组装，拿来就用。

2. 享受单例的好处，效率高，不浪费空间。
3. 便于单元测试，方便切换 mock 组件。
4. 便于进行 AOP 操作，对于使用者是透明的。
5. 统一配置，便于修改。

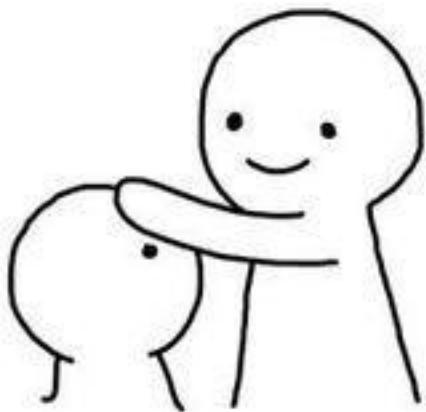
### 3. Spring 模块

Spring 可以分为 6 大模块：

- Spring Core spring 的核心功能：IOC 容器，解决对象创建及依赖关系
- Spring Web Spring 对 web 模块的支持。
  - 可以与 struts 整合，让 struts 的 action 创建交给 spring
  - spring mvc 模式
- Spring DAO Spring 对 jdbc 操作的支持 【JdbcTemplate 模板工具类】
- Spring ORM spring 对 orm 的支持：
  - 既可以与 hibernate 整合，【session】
  - 也可以使用 spring 的对 hibernate 操作的封装
- Spring AOP 切面编程
- SpringEE spring 对 javaEE 其他模块的支持



上面文主要引出了为啥我们需要使用 Spring 框架，以及大致了解了 Spring 是分为六大模块的....下面主要讲解 Spring 的 core 模块！



加油加油



如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜 Java3y 公众号有我的联系方式。更多原创技术文章可关注我的 GitHub：

<https://github.com/ZhongFuCheng3y/3y>

## 4. Core 模块快速入门

### 4.1 搭建配置环境

更新：如果使用 maven 的同学，引入 pom 文件就好了

本博文主要是 core 模块的内容，涉及到 Spring core 的开发 jar 包有五个：

- commons-logging-1.1.3.jar 日志
- spring-beans-3.2.5.RELEASE.jar bean 节点
- spring-context-3.2.5.RELEASE.jar spring 上下文节点
- spring-core-3.2.5.RELEASE.jar spring 核心功能
- spring-expression-3.2.5.RELEASE.jar spring 表达式相关表

这次使用的是 Spring3.2 版本

编写配置文件:Spring 核心的配置文件 applicationContext.xml 或者叫 bean.xml

那这个配置文件怎么写呢？？一般地，我们都知道框架的配置文件都是有约束的...

我们可以在 spring-framework-3.2.5.RELEASE\docs\spring-framework-reference\htmlsingle\index.html 找到 XML 配置文件的约束

```
<beans xmlns='http://www.springframework.org/schema/beans'  
       xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'  
       xmlns:p='http://www.springframework.org/schema/p'  
       xmlns:context='http://www.springframework.org/schema/context'  
       xsi:schemaLocation='
```

```
http://www.springframework.org/schema/beans  
http://www.springframework.org/schema/beans/spring-beans.xsd  
http://www.springframework.org/schema/context  
http://www.springframework.org/schema/context/spring-context.xsd' |
```

```
</beans|
```

我是使用 IntelliJ Idea 集成开发工具的，可以选择自带的 Spring 配置文件，它长的是这样：

```
<?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.spring  
framework.org/schema/beans/spring-beans.xsd' |
```

```
</beans|
```

前面在介绍 Spring 模块的时候已经说了，Core 模块是：IOC 容器，解决对象创建和之间的依赖关系。

因此 Core 模块主要是学习如何得到 IOC 容器，通过 IOC 容器来创建对象、解决对象之间的依赖关系、IOC 细节。

## 4.2 得到 Spring 容器对象【IOC 容器】

Spring 容器不单单只有一个，可以归为两种类型

- Bean 工厂，BeanFactory 【功能简单】
- 应用上下文，ApplicationContext 【功能强大，一般我们使用这个】

### 4.2.1 通过 Resource 获取 BeanFactory

步骤

- 加载 Spring 配置文件

- 通过 XmlBeanFactory+配置文件来创建 IOC 容器

```
//加载 Spring 的资源文件  
Resource resource = new ClassPathResource('applicationContext.xml');  
  
//创建 IOC 容器对象【IOC 容器=工厂类+applicationContext.xml】  
BeanFactory beanFactory = new XmlBeanFactory(resource);
```

---

#### 4.2.2 类路径下 XML 获取 ApplicationContext

直接通过 ClassPathXmlApplicationContext 对象来获取

```
// 得到 IOC 容器对象  
ApplicationContext ac = new ClassPathXmlApplicationContext('applicationCont  
ext.xml');  
  
System.out.println(ac);
```

在 Spring 中总体来看可以通过四种方式来配置对象：

- 使用 XML 文件配置
- 使用注解来配置
- 使用 JavaConfig 来配置
- groovy 脚本 DSL

## 4.3 XML 配置方式

在上面我们已经可以得到 IOC 容器对象了。接下来就是在 `applicationContext.xml` 文件中配置信息【让 IOC 容器根据 `applicationContext.xml` 文件来创建对象】

首先我们先有个 JavaBean 的类

```
/*
 * Created by ozc on 2017/5/10.
 */
public class User {

    private String id;
    private String username;

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
}
```

以前我们是通过 `new User` 的方法创建对象的....

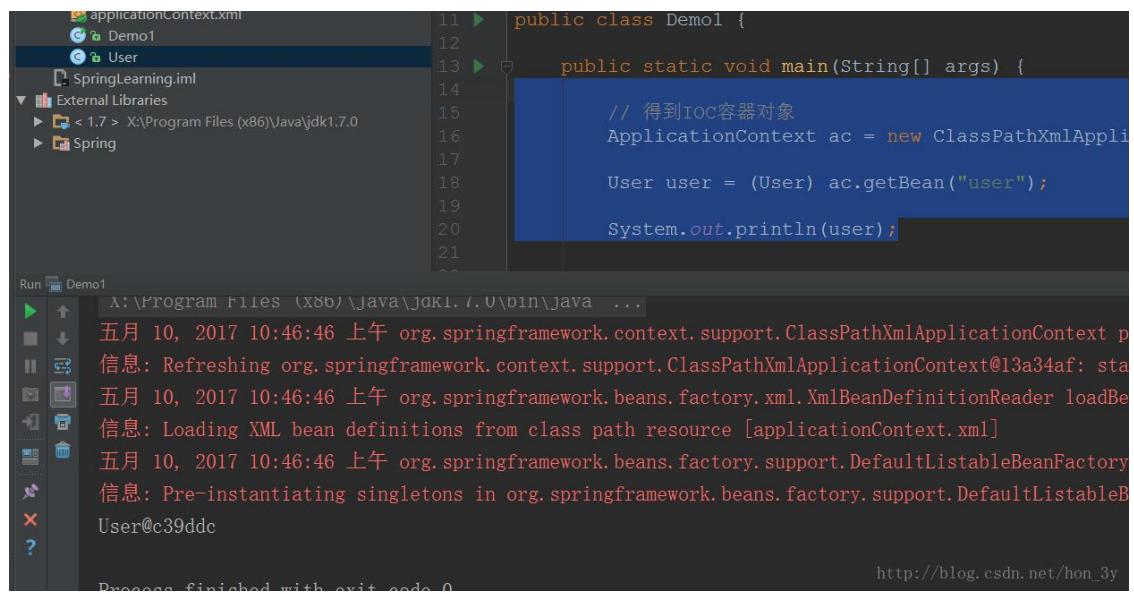
```
User user = new User();
```

现在我们有了 IOC 容器，可以让 IOC 容器帮我们创建对象了。在 applicationContext.xml 文件中配置对应的信息就行了

```
<!--  
    使用 bean 节点来创建对象  
    id 属性标识着对象  
    name 属性代表着要创建对象的类全名  
-->  
<bean id='user' class='User'/>
```

通过 IOC 容器对象获取对象：在外界通过 IOC 容器对象得到 User 对象

```
// 得到 IOC 容器对象  
ApplicationContext ac = new ClassPathXmlApplicationContext('applicationContext.x  
ml');  
  
User user = (User) ac.getBean('user');  
  
System.out.println(user);
```



The screenshot shows an IDE interface with a code editor and a terminal window.

**Code Editor:**

```
public class Demo1 {  
    public static void main(String[] args) {  
        // 得到IOC容器对象  
        ApplicationContext ac = new ClassPathXmlApplication  
        User user = (User) ac.getBean("user");  
        System.out.println(user);  
    }  
}
```

**Terminal Output:**

```
A:\Program Files (x86)\Java\JDK1.7.0\bin\java ...  
五月 10, 2017 10:46:46 上午 org.springframework.context.support.ClassPathXmlApplicationConte  
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@13a34af: sta  
五月 10, 2017 10:46:46 上午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBe  
信息: Loading XML bean definitions from class path resource [applicationContext.xml]  
五月 10, 2017 10:46:46 上午 org.springframework.beans.factory.support.DefaultListableBeanFactory  
信息: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableB  
User@c39ddc  
Process finished with exit code 0
```

http://blog.csdn.net/hon\_3y

上面我们使用的是 IOC 通过无参构造函数来创建对象，我们来回顾一下一般有几种创建对象的方式：

- 无参构造函数创建对象
- 带参数的构造函数创建对象
- 工厂创建对象
  - 静态方法创建对象
  - 非静态方法创建对象

使用无参的构造函数创建对象我们已经会了，接下来我们看看使用剩下的 IOC 容器是怎么创建对象的。

#### 4.3.1 带参数的构造函数创建对象

首先，JavaBean 就要提供带参数的构造函数：

```
public User(String id, String username) {  
    this.id = id;  
    this.username = username;  
}
```

接下来，关键是怎么配置 applicationContext.xml 文件了。

```
<bean id='user' class='User'>  
  
    <!--通过 constructor 这个节点来指定构造函数的参数类型、名称、第几个-->  
    <constructor-arg index='0' name='id' type='java.lang.String' value='1' />  
    <constructor-arg index='1' name='username' type='java.lang.String' value='zho  
ngfucheng' />  
</bean>
```

```

14
15     // 得到IOC容器对象
16     ApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");
17
18     User user = (User) ac.getBean("user");
19
20     System.out.println(user);
21
22 }
23

```

Run Demo1  
 X:\Program Files (x86)\Java\jdk1.7.0\bin\java ...  
 五月 10, 2017 11:28:03 上午 org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh  
 信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@135f44e: startup date [Wed May 10 11:28:03 CST 2017]; root of context hierarchy  
 五月 10, 2017 11:28:03 上午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions  
 信息: Loading XML bean definitions from class path resource [applicationContext.xml]  
 五月 10, 2017 11:28:03 上午 org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons  
 信息: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@1993f8e: defining beans [user]; root of factory hierarchy  
 User{id='1', username='zhongfucheng'}  
 Process finished with exit code 0

在 constructor 上如果构造函数的值是一个对象，而不是一个普通类型的值，我们就需要用到 ref 属性了，而不是 value 属性

比如说：我在 User 对象上维护了 Person 对象的值，想要在构造函数中初始化它。因此，就需要用到 ref 属性了

```
<bean id='person' class='Person' |</bean|
```

```
<bean id='user' class='User' |
```

```

<!--通过 constructor 这个节点来指定构造函数的参数类型、名称、第几个--|
<constructor-arg index='0' name='id' type='java.lang.String' value='1' |<const
ructor-arg|
<constructor-arg index='1' name='username' type='java.lang.String' ref='perso
n' |</constructor-arg|
</bean|
```

### 4.3.2 工厂静态方法创建对象

首先，使用一个工厂的静态方法返回一个对象

```
public class Factory {

    public static User getBean() {

        return new User();
    }
}
```

```
}
```

## 配置文件中使用工厂的静态方法返回对象

```
<!--工厂静态方法创建对象，直接使用 class 指向静态类，指定静态方法就行了-->
<bean id='user' class='Factory' factory-method='getBean' />

</bean>
```

The screenshot shows an IDE interface with a code editor and a terminal window. The code editor contains a Java file named 'Demo1.java' with the following content:

```
14 // 得到IOC容器对象
15 ApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");
16
17 User user = (User) ac.getBean("user");
18
19 System.out.println(user);
20
```

The terminal window shows the execution of the code and its output:

```
A:\Program Files (x86)\Java\jdk1.7.0_65\bin\java ...
五月 10, 2017 11:36:39 上午 org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@651ef4e: startup date [Wed May 10 11:36:39 CST 2017]; root of context hierarchy
五月 10, 2017 11:36:40 上午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
信息: Loading XML bean definitions from class path resource [applicationContext.xml]
五月 10, 2017 11:36:40 上午 org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons
信息: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@19f06f7: defining beans [user]; root of factory hierarchy
我是User, 我被创造了
User {id=null, username=null}
```

http://blog.csdn.net/hon\_3y

### 4.3.3 工厂非静态方法创建对象

首先，也是通过工厂的非静态方法来得到一个对象

```
public class Factory {
    public User getBean() {
        return new User();
    }
}
```

## 配置文件中使用工厂的非静态方法返回对象

```

<!-- 首先创建工厂对象-->
<bean id='factory' class='Factory'/>

<!--指定工厂对象和工厂方法-->
<bean id='user' class='User' factory-bean='factory' factory-method='getBean'/>

```

```

14 // 得到IOC容器对象
15 ApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");
16
17 User user = (User) ac.getBean("user");
18
19 System.out.println(user);
20

```

Run Demo1  
五月 10, 2017 11:36:39 上午 org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh  
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@51ef4e: startup date [Wed May 10 11:36:39 CST 2017]; root of context hierarchy  
五月 10, 2017 11:36:40 上午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions  
信息: Loading XML bean definitions from class path resource [applicationContext.xml]  
五月 10, 2017 11:36:40 上午 org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons  
信息: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@19f06f7: defining beans [user]; root of factory hierarchy  
我是User, 我被创造了  
User{id=null, username=null}

#### 4.3.4 c 名称空间

我们在使用 XML 配置创建 Bean 的时候，如果该 Bean 有构造器，那么我们使用 `<constructor-arg>` 这个节点来对构造器的参数进行赋值...

`<constructor-arg>` 未免有点太长了，为了简化配置，Spring 来提供了 c 名称空间...

要想 c 名称空间是需要导入 `xmlns:c='http://www.springframework.org/schema/c'` 的

```

<bean id='userService' class='bb.UserService' c:userDao-ref=''/>

</bean>

```

c 名称空间有个缺点：不能装配集合，当我们要装配集合的时候还是需要 `<constructor-arg>` 这个节点

#### 4.3.5 装载集合

如果对象上的属性或者构造函数拥有集合的时候，而我们又需要为集合赋值，那么怎么办？

在构造函数上，普通类型

```
<bean id='userService' class='bb.UserService' |  
  <constructor-arg |  
    <list|  
      //普通类型  
      <value|</value|  
    </list|  
  </constructor-arg|  
</bean|
```

在属性上,引用类型

```
<property name='userDao'|  
  
  <list|  
    <ref|</ref|  
  </list|  
</property|
```

#### 4.4 注解方式

自从 jdk5 有了注解这个新特性，我们可以看到 Struts2 框架、Hibernate 框架都支持使用注解来配置信息...

通过注解来配置信息就是为了简化 IOC 容器的配置，注解可以把对象添加到 IOC 容器中、处理对象依赖关系，我们来看看怎么用吧：

使用注解步骤：

- 1) 先引入 context 名称空间
  - xmlns:context='http://www.springframework.org/schema/context'
- 2) 开启注解扫描器
  - <context:component-scan base-package=' '|</context:component-scan|
  - 第二种方法：也可以通过自定义扫描类以@CompoentScan 修饰来扫描 IOC 容器的 bean 对象。如下代码：

```
//表明该类是配置类
@Configuration

//启动扫描器，扫描 bb 包下的
//也可以指定多个基础包
//也可以指定类型

@ComponentScan("bb")
public class AnnotationScan {

}
```

在使用@ComponentScan()这个注解的时候，在测试类上需要  
@ContextConfiguration 这个注解来加载配置类...

- @ContextConfiguration 这个注解又在 Spring 的 test 包下..

创建对象以及处理对象依赖关系，相关的注解：

- @ComponentScan 扫描器
- @Configuration 表明该类是配置类
- @Component 指定把一个对象加入 IOC 容器---|@Name 也可以实现相同的效果  
【一般少用】
- @Repository 作用同@Component； 在持久层使用
- @Service 作用同@Component； 在业务逻辑层使用
- @Controller 作用同@Component； 在控制层使用
- @Resource 依赖关系
  - 如果@Resource 不指定值，那么就根据类型来找，相同的类型在 IOC 容器中不能有两个
  - 如果@Resource 指定了值，那么就根据名字来找

测试代码:UserDao

```
package aa;

import org.springframework.stereotype.Repository;

/**
 * Created by ozc on 2017/5/10.
 */

//把对象添加到容器中,首字母会小写
@Repository
public class UserDao {

    public void save() {
        System.out.println("DB:保存用户");
    }
}

userService

package aa;

import org.springframework.stereotype.Service;

import javax.annotation.Resource;

//把 UserService 对象添加到 IOC 容器中,首字母会小写
@Service
public class UserService {

    //如果@Resource 不指定值,那么就根据类型来找---UserDao....当然了,IOC 容器不能
    //有两个 UserDao 类型的对象
    //@Resource
}
```

```
//如果指定了值，那么 Spring 就在 IOC 容器找有没有 id 为 userDao 的对象。  
@Resource(name = 'userDao')  
private UserDao userDao;  
  
public void save() {  
    userDao.save();  
}  
}  
  
userAction  
  
package aa;  
  
import org.springframework.stereotype.Controller;  
  
import javax.annotation.Resource;  
  
/**  
 * Created by ozc on 2017/5/10.  
 */  
  
//把对象添加到 IOC 容器中,首字母会小写  
@Controller  
public class UserAction {  
  
    @Resource(name = 'userService')  
    private UserService userService;  
  
    public String execute() {  
        userService.save();  
        return null;  
    }  
}  
  
测试
```

```

package aa;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * Created by ozc on 2017/5/10.
 */

public class App {

    public static void main(String[] args) {

        // 创建容器对象
        ApplicationContext ac = new ClassPathXmlApplicationContext("aa/applicationC
ontext.xml");

        UserAction userAction = (UserAction) ac.getBean("userAction");

        userAction.execute();
    }
}

```

The screenshot shows an IDE interface with the following details:

- Project Structure:** On the left, there's a tree view of the project structure. It includes packages like `aa` containing `User`, `UserAction`, `UserDao`, and `UserService` classes, and a `SpringLearning` module containing an `External Libraries` entry.
- Code Editor:** The main editor area displays the Java code provided above, with line numbers 11 through 21 visible.
- Run Output:** Below the editor, the terminal window shows the execution of the application. It starts with the command `java -jar App.jar` and then displays log messages from Spring Framework:
  - 信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@fb6c5f: startup date [Wed May 10 14:49:27 CST 2017]; root of context hierarchy
  - 信息: Loading XML bean definitions from class path resource [aa/applicationContext.xml]
  - 信息: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@1f9805f: defining beans [userAction, userService]; parent: org.springframework.context.support.ClassPathXmlApplicationContext@fb6c5f
  - DB:保存用户
- Bottom Status:** The status bar at the bottom right shows the URL [http://blog.csdn.net/hon\\_3y](http://blog.csdn.net/hon_3y).

## 4.5 通过 JavaConfig 方式

怎么通过 java 代码来配置 Bean 呢？？

- 编写一个 java 类，使用@Configuration 修饰该类
- 被@Configuration 修饰的类就是配置类

编写配置类：

```
@org.springframework.context.annotation.Configuration  
public class Configuration {  
  
}
```

使用配置类创建 bean：

- 使用@Bean 来修饰方法，该方法返回一个对象。
- 不管方法体内的对象是怎么创建的，Spring 可以获取得到对象就行了。
- Spring 内部会将该对象加入到 Spring 容器中
- 容器中 bean 的 ID 默认为方法名

```
@org.springframework.context.annotation.Configuration  
public class Configuration {  
  
    @Bean  
    public UserDao userDao() {  
  
        UserDao userDao = new UserDao();  
        System.out.println('我是在 configuration 中的'+userDao);  
        return userDao;  
    }  
  
}
```

- 测试代码：要使用@ContextConfiguration 加载配置类的信息【引入 test 包】

```
package bb;

import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.test.context.ContextConfiguration;

/**
 * Created by ozc on 2017/5/11.
 */
//加载配置类的信息
@ContextConfiguration(classes = Configuration.class)
public class Test2 {

    @Test
    public void test33() {

        ApplicationContext ac =
            new ClassPathXmlApplicationContext("bb/bean.xml");

        UserDao userDao = (UserDao) ac.getBean("userDao");

        System.out.println(userDao);
    }
}
```

结果如下：

A screenshot of an IDE showing a test run output. The title bar says "AOP&test". The status bar shows "15" and "16". The code editor has a test class with a single test method. The output console shows Spring framework initialization logs and a message indicating a singleton was pre-instantiated. The build status at the bottom right says "1 test".

```
五月 11, 2017 6:10:21 下午 org.springframework.context.support.  
信息: Refreshing org.springframework.context.support.ClassPath  
五月 11, 2017 6:10:21 下午 org.springframework.beans.factory.  
信息: Loading XML bean definitions from class path resource  
五月 11, 2017 6:10:21 下午 org.springframework.beans.factory.  
信息: Pre-instantiating singletons in org.springframework.be  
我是在configuration中的bb UserDao@1983bbc  
bb UserDao@1983bbc  
  
Process finished with exit code 0
```

## 4.6 三种方式混合使用？

注解和 XML 配置是可以混合使用的，JavaConfig 和 XML 也是可以混合使用的...

如果 JavaConfig 的配置类是分散的，我们一般再创建一个更高级的配置类（root），然后使用@Import 来将配置类进行组合

如果 XML 的配置文件是分散的，我们也是创建一个更高级的配置文件（root），然后使用<import>来将配置文件组合

在 JavaConfig 引用 XML

- 使用@ImportResource()

在 XML 引用 JavaConfig

- 使用<bean>节点就行了

在公司的项目中，一般我们是 XML+注解

## 5. bean 对象创建细节

既然我们现在已经初步了解 IOC 容器了，那么这些问题我们都是可以解决的。并且是十分简单【对象写死问题已经解决了，IOC 容器就是控制反转创建对象】

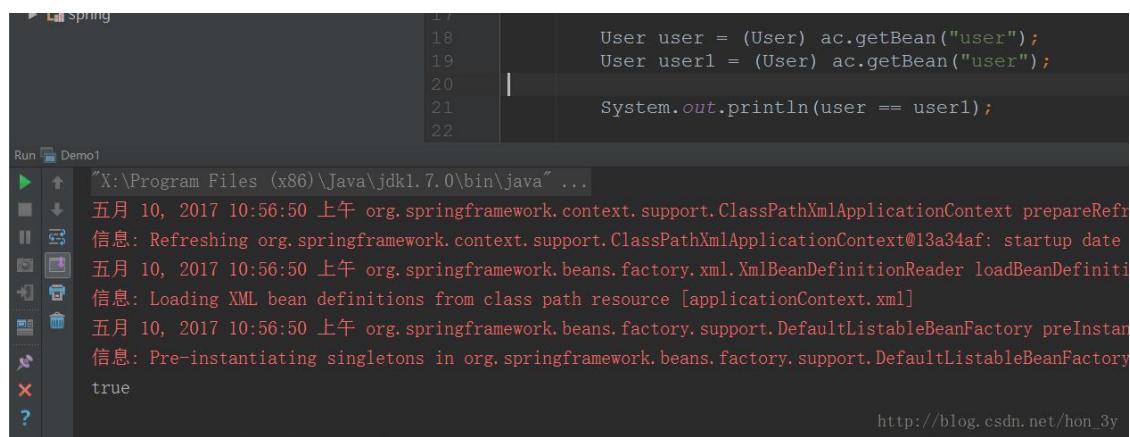
### 5.1 scope 属性

指定 scope 属性，IOC 容器就知道创建对象的时候是单例还是多例的了。

属性的值就只有两个：单例/多例



当我们使用 singleton【单例】的时候，从 IOC 容器获取的对象都是同一个：



当我们使用 prototype【多例】的时候，从 IOC 容器获取的对象都是不同的：

```

1 /          User user = (User) ac.getBean("user");
18         User user1 = (User) ac.getBean("user");
19         System.out.println(user == user1);
20
21
22
Run Demo1
"X:\Program Files (x86)\Java\jdk1.7.0\bin\java" ...
五月 10, 2017 10:57:42 上午 org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@13a34af: startup date [Wed May 10 10:5
五月 10, 2017 10:57:42 上午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
信息: Loading XML bean definitions from class path resource [applicationContext.xml]
五月 10, 2017 10:57:42 上午 org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons
信息: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@19048da: defin
false
Process finished with exit code 0

```

http://blog.csdn.net/hon\_3y

scope 属性除了控制对象是单例还是多例的，还控制着对象创建的时间！

我们在 User 的构造函数中打印出一句话，就知道 User 对象是什么时候创建了。

```

public User() {
    System.out.println('我是 User，我被创建了');
}

```

当使用 singleton 的时候，对象在 IOC 容器之前就已经创建了

```

src
  applicationContext.xml
  Demo1
  User
  SpringLearning.java
  External Libraries
    < 1.7 > X:\Program Files (x86)\Java\jdk1.7.0
    Spring
public class Demo1 {
    public static void main(String[] args) {
        // 得到IOC容器对象
        ApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");
        System.out.println("IOC容器创建了");
        User user = (User) ac.getBean("user");
        User user1 = (User) ac.getBean("user");
        System.out.println(user == user1);
    }
}

Run Demo1
"X:\Program Files (x86)\Java\jdk1.7.0\bin\java" ...
五月 10, 2017 11:06:20 上午 org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@6db33: startup date [Wed May 10 11:06:20 CST 2017]; root of context hierarchy
五月 10, 2017 11:06:20 上午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
信息: Loading XML bean definitions from class path resource [applicationContext.xml]
五月 10, 2017 11:06:20 上午 org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons
信息: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@1827a3a: defining beans [user]; root of factory hierarchy
我是 User，我被创建了
IOC容器创建了
true
Process finished with exit code 0

```

http://blog.csdn.net/hon\_3y

当使用 prototype 的时候，对象在使用的时候才创建

The screenshot shows an IDE interface with a code editor and a terminal window. The code in the editor is:

```
public static void main(String[] args) {
    // 得到IOC容器对象
    ApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");
    System.out.println("IOC容器创建了");

    User user = (User) ac.getBean("user");
    User user1 = (User) ac.getBean("user");

    System.out.println(user == user1);
}
```

The terminal window shows the execution output:

```
五月 10, 2017 11:09:07 上午 org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@20c906: startup date [Wed May 10 11:09:07 CST 2017]; root of context hierarchy
五月 10, 2017 11:09:07 上午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
信息: Loading XML bean definitions from class path resource [applicationContext.xml]
IOC容器创建了
五月 10, 2017 11:09:07 上午 org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons
我是User, 我被创建了
信息: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@19f06f7: defining beans [user]; root of factory hierarchy
我是User, 我被创建了
false
```

## 5.2 lazy-init 属性

lazy-init 属性只对 singleton【单例】的对象有效.....lazy-init 默认为 false....

有的时候，可能我们想要对象在使用的时候才创建，那么将 lazy-init 设置为 true 就行了

The screenshot shows an IDE interface with a code editor and a terminal window. The code in the editor is identical to the previous screenshot:

```
public static void main(String[] args) {
    // 得到IOC容器对象
    ApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");
    System.out.println("IOC容器创建了");

    User user = (User) ac.getBean("user");
    User user1 = (User) ac.getBean("user");

    System.out.println(user == user1);
}
```

The terminal window shows the execution output:

```
五月 10, 2017 11:12:08 上午 org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@20c906: startup date [Wed May 10 11:12:08 CST 2017]; root of context hierarchy
五月 10, 2017 11:12:09 上午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
信息: Loading XML bean definitions from class path resource [applicationContext.xml]
IOC容器创建了
信息: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@19f06f7: defining beans [user]; root of factory hierarchy
我是User, 我被创建了
true
```

## 5.3 init-method 和 destroy-method

如果我们想要对象在创建后，执行某个方法，我们指定为 init-method 属性就行了。。。

如果我们想要 IOC 容器销毁后，执行某个方法，我们指定 destroy-method 属性就行了。

```
<bean id='user' class='User' scope='singleton' lazy-init='true' init-method="" de  
stroy-method=''/>
```

## 5.4 Bean 创建细节总结

```
/**  
 * 1) 对象创建： 单例/多例  
 *      scope='singleton'，默认值，即默认是单例【service/dao/工具类】  
 *      scope='prototype'，多例；                                【Action 对象】  
 *  
 * 2) 什么时候创建？  
 *      scope='prototype' 在用到对象的时候，才创建对象。  
 *      scope='singleton' 在启动(容器初始化之前)，就已经创建了 bean，且整个  
应用只有一个。  
 * 3) 是否延迟创建  
 *      lazy-init='false' 默认为 false，不延迟创建，即在启动时候就创建对  
象  
 *      lazy-init='true' 延迟初始化，在用到对象的时候才创建对象  
 *          (只对单例有效)  
 * 4) 创建对象之后，初始化/销毁  
 *      init-method='init_user'      【对应对象的 init_user 方法，在对象创  
建之后执行】  
 *      destroy-method='destroy_user' 【在调用容器对象的 destroy 方法时候执  
行，(容器用实现类)】  
 */
```



如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜 **Java3y** 公众号有我的联系方式。更多原创技术文章可关注我的 GitHub：

<https://github.com/ZhongFuCheng3y/3y>

# 对象依赖

## 1. 回顾以前对象依赖

我们来看一下我们以前关于对象依赖，是怎么的历程

### 1.1 直接 new 对象

在最开始，我们是直接 new 对象给 service 的 userDao 属性赋值...

```
class UserService{
    UserDao userDao = new UserDao();
}
```

### 1.2 写 DaoFactory，用字符串来维护依赖关系

后来，我们发现 service 层紧紧耦合了 dao 层。我们就写了 DaoFactory，在 service 层只要通过字符串就能够创建对应的 dao 层的对象了。

DaoFactory

```
public class DaoFactory {

    private static final DaoFactory factory = new DaoFactory();
    private DaoFactory(){}

    public static DaoFactory getInstance(){
        return factory;
    }

    public <T> T createDao(String className,Class<T> clazz){
        try{
            T t = (T) Class.forName(className).newInstance();
        }
    }
}
```

```

        return t;
    }catch (Exception e) {
        throw new RuntimeException(e);
    }
}

serivce

private CategoryDao categoryDao = DaoFactory.getInstance().createDao("zhongfucheng.dao.impl.CategoryDAOImpl", CategoryDao.class);

private BookDao bookDao = DaoFactory.getInstance().createDao("zhongfucheng.dao.impl.BookDAOImpl", BookDao.class);

private UserDao userDao = DaoFactory.getInstance().createDao("zhongfucheng.dao.impl.UserDAOImpl", UserDao.class);

private OrderDao orderDao = DaoFactory.getInstance().createDao("zhongfucheng.dao.impl.OrderDAOImpl", OrderDao.class);

```

### 1.3 DaoFactory 读取配置文件

再后来，我们发现要修改 Dao 的实现类，还是得修改 service 层的源代码呀..于是我们在 DaoFactory 中读取关于 daoImpl 的配置文件，根据配置文件来创建对象，这样一来，创建的是哪个 daoImpl 对 service 层就是透明的

DaoFactory

```

public class DaoFactory {

    private UserDao userdao = null;

```

```

private DaoFactory(){
    try{
        InputStream in = DaoFactory.class.getClassLoader().getResourceAsStream("dao.properties");
        Properties prop = new Properties();
        prop.load(in);

        String daoClassName = prop.getProperty("userdao");
        userdao = (UserDao)Class.forName(daoClassName).newInstance();

    }catch (Exception e) {
        throw new RuntimeException(e);
    }
}

private static final DaoFactory instance = new DaoFactory();

public static DaoFactory getInstance(){
    return instance;
}

public UserDao createUserDao(){
    return userdao;
}

}

service

UserDao dao = DaoFactory.getInstance().createUserDao();

```

## 2. Spring 依赖注入

通过上面的历程，我们可以清晰地发现：对象之间的依赖关系，其实就是给对象上的属性赋值！因为对象上有其他对象的变量，因此存在了依赖...

Spring 提供了好几几种的方式来给属性赋值

- 1) 通过构造函数
- 2) 通过 set 方法给属性注入值
- 3) p 名称空间
- 4) 自动装配(了解)
- 5) 注解

## 2.1 搭建测试环境

UserService 中使用 userDao 变量来维护与 Dao 层之间的依赖关系，UserAction 中使用 userService 变量来维护与 Service 层之间的依赖关系。

userDao

```
public class UserDao {  
  
    public void save() {  
        System.out.println("DB:保存用户");  
    }  
}
```

userService

```
public class UserService {  
  
    private UserDao userDao;  
  
    public void save() {  
        userDao.save();  
    }  
}
```

```
userAnction

public class UserAction {

    private UserService userService;

    public String execute() {
        userService.save();
        return null;
    }
}
```

## 2.2 构造函数给属性赋值

其实我们在讲解创建带参数的构造函数的时候已经讲过了...我们还是来回顾一下呗..

我们测试 service 和 dao 的依赖关系就好了....在 service 中加入一个构造函数，参数就是 userDao

```
public UserService(UserDao userDao) {
    this.userDao = userDao;

    //看看有没有拿到 userDao
    System.out.println(userDao);
}
```

### applicationContext.xml 配置文件

```
<!--创建 userDao 对象--/
<b>bean id='userDao' class='UserDao'</b>

<!--创建 userService 对象--/
<b>bean id='userService' class='UserService'</b>
    <!--要想在 userService 层中能够引用到 userDao，就必须先创建 userDao 对象--/
    <b>constructor-arg index='0' name='userDao' type='UserDao' ref='userDao'</b></construc
```

```
tor-arg|
```

```
</bean|
```

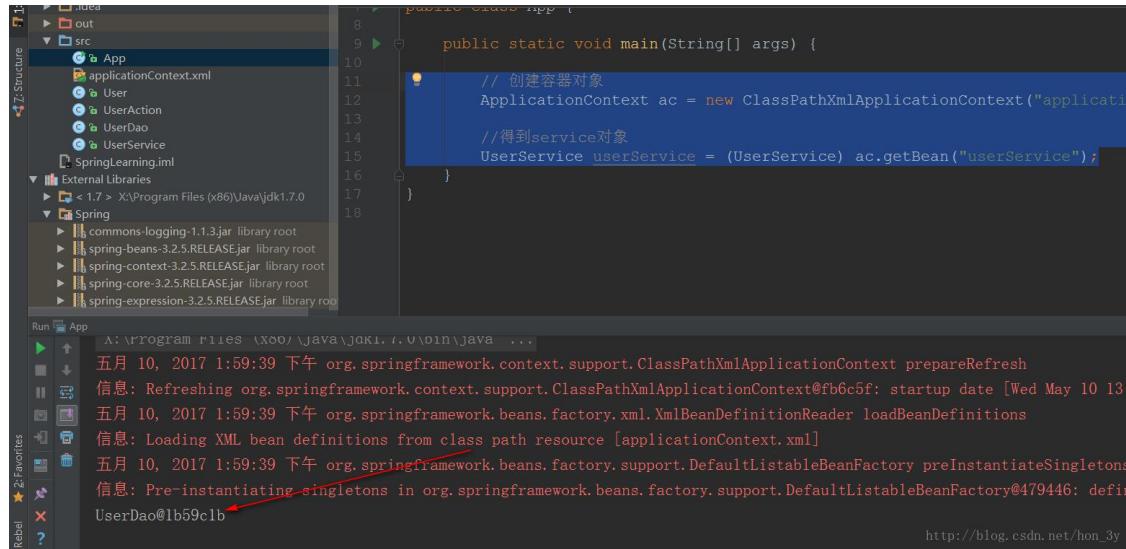
测试：可以成功获取到 userDao 对象

```
// 创建容器对象
```

```
ApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");
```

```
//得到 service 对象
```

```
UserService userService = (UserService) ac.getBean("userService");
```





如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜 **Java3y** 公众号有我的联系方式。更多原创技术文章可关注我的 GitHub：

<https://github.com/ZhongFuCheng3y/3y>

## 2.3 通过 set 方法给属性注入值

我们这里也是测试 service 和 dao 层的依赖关系就好了...在 service 层通过 set 方法来把 userDao 注入到 UserService 中

为 UserService 添加 set 方法

```
public class UserService {  
  
    private UserDao userDao;  
  
    public void setUserDao(UserDao userDao) {  
        this.userDao = userDao;  
  
        //看看有没有拿到 userDao  
        System.out.println(userDao);  
    }  
  
    public void save() {  
        userDao.save();  
    }  
}
```

applicationContext.xml 配置文件：通过 property 节点来给属性赋值

- 引用类型使用 ref 属性
- 基本类型使用 value 属性

```
<!--创建 userDao 对象--/  
<bean id='userDao' class='UserDao'/>  
  
<!--创建 userService 对象--/
```

```

<bean id='userService' class='UserService'>
    <property name='userDao' ref='userDao'/>
</bean>

```

测试：

The screenshot shows an IDE interface with the following details:

- Project Structure:** Shows a tree view with nodes: App, applicationContext.xml, User, UserAction, UserDao, UserService, SpringLearning.java, and External Libraries.
- Code Editor:** Displays Java code in a file named SpringLearning.java. The code includes imports for ApplicationContext and ClassPathXmlApplicationContext, and a main method that creates an ApplicationContext and retrieves a UserService bean.
- Run Tab:** Shows the output of the application's execution.
- Output Log:**
  - May 10, 2017 2:07:39 PM org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh
  - 信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@1de891b: startup date [Wed May 10 14:07:39 CST 2017]; root of context hierarchy
  - May 10, 2017 2:07:39 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
  - 信息: Loading XML bean definitions from class path resource [applicationContext.xml]
  - May 10, 2017 2:07:39 PM org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons
  - 信息: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@1d248b4: defining beans [userDao,userService]; root of factory hierarchy
  - UserDao@861253
- Bottom Status Bar:** Shows the URL [http://blog.csdn.net/hon\\_3y](http://blog.csdn.net/hon_3y).

## 2.4 内部 Bean

我们刚才是先创建 userDao 对象，再由 userService 对 userDao 对象进行引用...我们还有另一种思维：先创建 userService，发现 userService 需要 userDao 的属性，再创建 userDao...我们来看看这种思维方式是怎么配置的：

applicationContext.xml 配置文件：property 节点内置 bean 节点

```

<!--
1. 创建 userService，看到有 userDao 这个属性
2. 而 userDao 这个属性又是一个对象
3. 在 property 属性下又内置了一个 bean
4. 创建 userDao
-->

<bean id='userService' class='UserService'>
    <property name='userDao'>

```

```

<bean id='userDao' class='UserDao'/>
</property>
</bean>

```

## 测试

The screenshot shows an IDE interface with a code editor and a terminal window. The code editor contains a Java file named 'SpringLearning.java' with the following content:

```

10 // 创建容器对象
11 ApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");
12
13 //得到service对象
14 UserService userService = (UserService) ac.getBean("userService");
15
16 }

```

The terminal window shows the execution of the code and its output:

```

X:\Program Files (x86)\Java\jdk1.7.0\bin\java" ...
五月 10, 2017 2:14:10 下午 org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@1de891b: startup date [Wed May 10 14:14:10 CST 2017]; root of conte
五月 10, 2017 2:14:10 下午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
信息: Loading XML bean definitions from class path resource [applicationContext.xml]
五月 10, 2017 2:14:10 下午 org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons
信息: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@1e06dfc: defining beans [userService]; root
UserService@141e83f
UserDao@141e83f
Process finished with exit code 0

```

The URL [http://blog.csdn.net/hon\\_3y](http://blog.csdn.net/hon_3y) is visible at the bottom right of the terminal window.

我们发现这种思维方式和服务器访问的执行顺序是一样的，但是如果 userDao 要多次被其他 service 使用的话，就要多次配置了...

## 2.5 p 名称空间注入属性值

p 名称控件这种方式其实就是 set 方法的一种优化，优化了配置而已...p 名称空间这个内容需要在 Spring3 版本以上才能使用...我们来看看：

applicationContext.xml 配置文件：使用 p 名称空间

```

<bean id='userDao' class='UserDao'/>

<!--不用写 property 节点了，直接使用 p 名称空间-->
<bean id='userService' class='UserService' p:userDao-ref='userDao'/>

```

## 测试

```
internal Libraries
10
11     // 创建容器对象
12     ApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");
13
14     //得到service对象
15     UserService userService = (UserService) ac.getBean("userService");
16 }
17
18
19
app
[X:\Program Files (x86)\Java\jdk1.7.0\bin\java" ...
X:\Program Files (x86)\Java\jdk1.7.0\bin\java" -Didea.launcher.port=7538 -Didea.launcher.bin.path=X:\开发软件\IntelliJ IDEA 2016.2\bin" -Dfile.encoding=UTF-8 -classpath "X:\Program Files (x86)\Java\jdk1.7.0\bin" org.springframework.context.support.ClassPathXmlApplicationContext@1de891b: startup date [Wed May 10 14:18:48 CST 2017]: root of context hierarchy
五月 10, 2017 2:18:48 下午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
信息： Loading XML bean definitions from class path resource [applicationContext.xml]
五月 10, 2017 2:18:48 下午 org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons
信息： Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@1fSec00: defining beans [userDao,userService]; root of factory
UserDao@16e8e76
Process finished with exit code 0
http://blog.csdn.net/hon_3y
```

2.6 自动装配

Spring 还提供了自动装配的功能，能够非常简化我们的配置

自动装载默认是不打开的，自动装配常用的可分为两种：

- 根据名字来装配
  - 根据类型类装配

### 2.6.1 XML 配置根据名字

applicationContext.xml 配置文件：使用自动装配，根据名字

```
<bean id='userDao' class='UserDao'/>
```

< / --

### 1. 通过名字来自动装配

2.发现 userService 中有个叫 userDao 的属性

3.看看 IOC 容器中没有叫 userDao 的对象

4.如果有，就装配进去

— 1 —

```
<bean id='userService' class='UserService' autowire='byName' />
```

测试

The screenshot shows an IDE interface with a code editor and a terminal window. The code editor contains a Java file named 'SpringLearning.java' with the following content:

```
public static void main(String[] args) {
    // 创建容器对象
    ApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");
    // 得到service对象
    UserService userService = (UserService) ac.getBean("userService");
}
```

The terminal window shows the execution of the application:

```
Run App "X:\Program Files (x86)\Java\jdk1.7.0\bin\java" ...
五月 10, 2017 2:26:56 下午 org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@fb6c5f: startup date [Wed May 10 14:26:56 CST 2017]; root of context hierarchy
五月 10, 2017 2:26:56 下午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
信息: Loading XML bean definitions from class path resource [applicationContext.xml]
五月 10, 2017 2:26:56 下午 org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons
信息: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@a7d7b9: defining beans [userDao,userService]; root of factory hierarchy
UserDao@93370a
```

## 2.6.2 XML 配置根据类型

applicationContext.xml 配置文件：使用自动装配，根据类型

值得注意的是：如果使用了根据类型来自动装配，那么在 IOC 容器中只能有一个这样的类型，否则就会报错！

```
<bean id='userDao' class=' UserDao'>
```

```
<!--
```

1. 通过名字来自动装配
2. 发现 userService 中有个叫 userDao 的属性
3. 看看 IOC 容器 UserDao 类型的对象
4. 如果有，就装配进去

```
-->
```

```
<bean id='userService' class=' UserService' autowire='byType'>
```

测试：

The screenshot shows an IDE interface with a code editor and a run console.

```

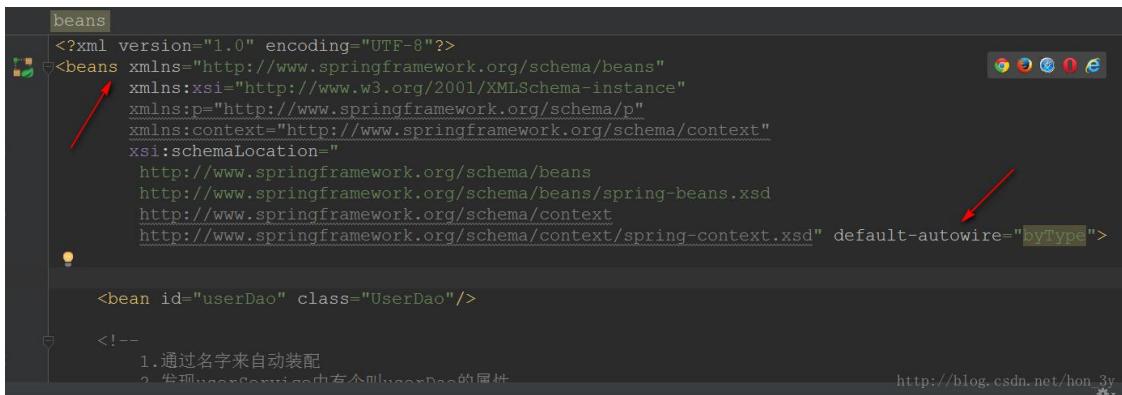
public class App {
    public static void main(String[] args) {
        // 创建容器对象
        ApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");
        // 得到service对象
        UserService userService = (UserService) ac.getBean("userService");
    }
}

```

Run App  
 X:\Program Files (x86)\Java\jdk1.7.0\bin\java" ...  
 五月 10, 2017 2:28:24 下午 org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh  
 信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@1de891b: startup date [Wed May 10 14:28:24 CST 2017]; root of context hierarchy  
 五月 10, 2017 2:28:24 下午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions  
 信息: Loading XML bean definitions from class path resource [applicationContext.xml]  
 五月 10, 2017 2:28:24 下午 org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons  
 信息: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@143c268: defining beans [userService, userDao]; root of factory hierarchy  
 UserDao@1a729c1  
 Process finished with exit code 0

[http://blog.csdn.net/hon\\_3y](http://blog.csdn.net/hon_3y)

我们这只是测试两个对象之间的依赖关系，如果我们有很多对象，我们也可以使用默认自动分配



## 2.7 使用注解来实现自动装配

@Autowired 注解来实现自动装配：

- 可以在构造器上修饰
- 也可以在 setter 方法上修饰
- 来自 java 的@Inject 和 @AutoWired 有相同的功能

如果没有匹配到 bean，又为了避免异常的出现，我们可以使用 required 属性上设置为 false。【谨慎对待】

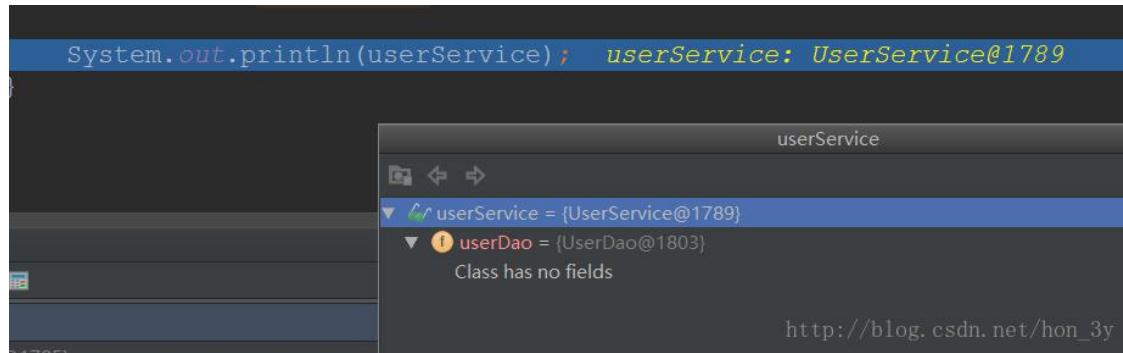
测试代码

```
@Component
public class UserService {

    private UserDao userDao;

    @Autowired
    public void setUserDao(UserDao userDao) {
        this.userDao = userDao;
    }
}
```

顺利拿到 userDao 的引用



加油~



如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜 Java3y 公众号有我的联系方式。更多原创技术文章可关注我的 GitHub：

<https://github.com/ZhongFuCheng3y/3y>

## AOP 入门

在讲解 AOP 模块之前，首先我们来讲解一下 **cglib** 代理、以及怎么手动实现 AOP 编程

### 1. cglib 代理

在讲解 cglib 之前，首先我们来回顾一下静态代理和动态代理

由于静态代理需要实现目标对象的相同接口，那么可能会导致代理类会非常非常多....不好维护----|因此出现了动态代理

动态代理也有个约束：目标对象一定是要有接口的，没有接口就不能实现动态代理.....-----|因此出现了 cglib 代理

cglib 代理也叫子类代理，从内存中构建出一个子类来扩展目标对象的功能！

- CGLIB 是一个强大的高性能的代码生成包，它可以在运行期扩展 Java 类与实现 Java 接口。它广泛的被许多 AOP 的框架使用，例如 Spring AOP 和 dynaop，为他们提供方法的 interception (拦截)。

#### 1.1 编写 cglib 代理

接下来我们就讲讲怎么写 cglib 代理：

- 需要引入 cglib — jar 文件，但是 spring 的核心包中已经包括了 cglib 功能，所以直接引入 spring-core-3.2.5.jar 即可。(如果用 maven 的同学，引入 pom 依赖就好了)
- 引入功能包后，就可以在内存中动态构建子类

- 代理的类不能为 `final`，否则报错【在内存中构建子类来做扩展，当然不能为 `final`，有 `final` 就不能继承了】
- 目标对象的方法如果为 `final/static`，那么就不会被拦截，即不会执行目标对象额外的业务方法。

```
//需要实现 MethodInterceptor 接口
public class ProxyFactory implements MethodInterceptor{

    // 维护目标对象
    private Object target;

    public ProxyFactory(Object target){
        this.target = target;
    }

    // 给目标对象创建代理对象
    public Object getProxyInstance(){
        //1. 工具类
        Enhancer en = new Enhancer();
        //2. 设置父类
        en.setSuperclass(target.getClass());
        //3. 设置回调函数
        en.setCallback(this);
        //4. 创建子类(代理对象)
        return en.create();
    }

    @Override
    public Object intercept(Object obj, Method method, Object[] args,
                           MethodProxy proxy) throws Throwable {
        System.out.println('开始事务.....');

        // 执行目标对象的方法
        return null;
    }
}
```

```
//Object returnValue = method.invoke(target, args);
proxy.invokeSuper(object, args);
System.out.println('提交事务.....');

return returnValue;
}

}
```

测试：

```
public class App {

    public static void main(String[] args) {

        UserDao userDao = new UserDao();

        UserDao factory = (UserDao) new ProxyFactory(userDao).getProxyInstance();

        factory.save();
    }
}
```

结果如下：

The screenshot shows an IDE interface with a code editor and a terminal window.

**Code Editor:**

```
5
6 public static void main(String[] args) {
7
8     UserDao userDao = new UserDao();
9
10    UserDao factory = (UserDao) new ProxyFactory(userDao)
11        .getProxyInstance();
12
13    factory.save();
14
15 }
16
17 }
```

**Terminal Window:**

```
un App
X:\Program Files (x86)\Java\jdk1.7.0\bin\java" ...
开始事务.....
DB: 保存用户
提交事务.....
Process finished with exit code 0
```

[http://blog.csdn.net/hon\\_3y](http://blog.csdn.net/hon_3y)

The screenshot shows an IDE interface with a code editor and a terminal window.

**Code Editor:**

```
5
6 public static void main(String[] args) {
7
8     UserDao userDao = new UserDao();
9     System.out.println(userDao.getClass());
10    UserDao factory = (UserDao) new ProxyFactory(userDao).getProxyInstance();
11    System.out.println(factory.getClass());
12
13 }
14
15
16
17
18 }
```

**Terminal Window:**

```
X:\Program Files (x86)\Java\jdk1.7.0\bin\java" ...
class UserDao
class UserDao$$EnhancerByCGLIB$$1c2049ea
```

A yellow callout box highlights the text "cglib代理的类型".

[http://blog.csdn.net/hon\\_3y](http://blog.csdn.net/hon_3y)

使用 cglib 就是为了弥补动态代理的不足【动态代理的目标对象一定要实现接口】

## 2. 手动实现 AOP 编程

AOP 面向切面的编程：AOP 可以实现“业务代码”与“关注点代码”分离

下面我们来看一段代码：

```
// 保存一个用户
public void add(User user) {
    Session session = null;
```

```

Transaction trans = null;
try {
    session = HibernateSessionFactoryUtils.getSession(); // 【关注点代码】
    trans = session.beginTransaction(); // 【关注点代码】

    session.save(user); // 核心业务代码

    trans.commit(); //... 【关注点代码】

} catch (Exception e) {
    e.printStackTrace();
    if(trans != null){
        trans.rollback(); //.. 【关注点代码】
    }
} finally{
    HibernateSessionFactoryUtils.closeSession(session); //...
}

【关注点代码】
}
}

```

关注点代码，就是指重复执行的代码。

业务代码与关注点代码分离，好处？

- 关注点代码写一次即可；
- 开发者只需要关注核心业务；
- 运行时期，执行核心业务代码时候动态植入关注点代码；【代理】

## 2.1 案例分析：

IUser 接口

```
public interface IUser {
```

```
    void save();  
}
```

我们一步一步来分析，首先我们的 UserDao 有一个 save()方法，每次都要开启事务和关闭事务

```
//@Component --|任何地方都能用这个  
@Repository //--|这个在 Dao 层中使用  
public class UserDao {  
  
    public void save() {  
  
        System.out.println('开始事务');  
        System.out.println('DB:保存用户');  
        System.out.println('关闭事务');  
  
    }  
  
}
```

在刚学习 java 基础的时候，我们知道：如果某些功能经常需要用到就封装成方法：

```
//@Component --|任何地方都能用这个  
@Repository //--|这个在 Dao 层中使用  
public class UserDao {  
  
    public void save() {  
  
        begin();  
        System.out.println('DB:保存用户');  
        close();  
  
    }  

```

```
public void begin() {
    System.out.println('开始事务');
}
public void close() {
    System.out.println('关闭事务');
}
}
```

现在呢，我们可能有多个 Dao，都需要有开启事务和关闭事务的功能，现在只有 UserDao 中有这两个方法，重用性还是不够高。因此我们抽取出一个类出来

```
public class AOP {

    public void begin() {
        System.out.println('开始事务');
    }
    public void close() {
        System.out.println('关闭事务');
    }
}
```

在 UserDao 维护这个变量，要用的时候，调用方法就行了。

`@Repository //--> 这个在 Dao 层中使用`

```
public class UserDao {
```

```
AOP aop;
```

```
public void save() {

    aop.begin();
    System.out.println('DB:保存用户');
    aop.close();
}

}
```

```
}
```

现在的开启事务、关闭事务还是需要我在 userDao 中手动调用。还是不够优雅。。我想要的效果：当我在调用 userDao 的 save()方法时，动态地开启事务、关闭事务。因此，我们就用到了代理。当然了，真正执行方法的都是 userDao、要干事的是 AOP，因此在代理中需要维护他们的引用。

```
public class ProxyFactory {
    //维护目标对象
    private static Object target;

    //维护关键点代码的类
    private static AOP aop;
    public static Object getProxyInstance(Object target_, AOP aop_) {
        //目标对象和关键点代码的类都是通过外界传递进来
        target = target_;
        aop = aop_;

        return Proxy.newProxyInstance(
            target.getClass().getClassLoader(),
            target.getClass().getInterfaces(),
            new InvocationHandler() {
                @Override
                public Object invoke(Object proxy, Method method, Object[] args) throws
                    Throwable {
                    aop.begin();
                    Object returnValue = method.invoke(target, args);
                    aop.close();
                    return returnValue;
                }
            }
        );
    }
}
```

```
        );
    }
}
```

## 2.2 工厂静态方法：

把 AOP 加入 IOC 容器中

```
//把该对象加入到容器中
@Component
public class AOP {

    public void begin() {
        System.out.println("开始事务");
    }

    public void close() {
        System.out.println("关闭事务");
    }
}
```

把 UserDao 放入容器中

```
@Component
public class UserDao {

    public void save() {
        System.out.println("DB:保存用户");
    }
}
```

在配置文件中开启注解扫描,使用工厂静态方法创建代理对象

```
<?xml version='1.0' encoding='UTF-8'?!
<beans xmlns='http://www.springframework.org/schema/beans'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xmlns:p='http://www.springframework.org/schema/p'
```

```
xmlns:context='http://www.springframework.org/schema/context'
xsi:schemaLocation='
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd' |
```

```
<bean id='proxy' class='aa.ProxyFactory' factory-method='getProxyInstance' |
    <constructor-arg index='0' ref='userDao'/|
    <constructor-arg index='1' ref='AOP'/|
</bean|

<context:component-scan base-package='aa'|

</beans|
```

测试，得到 UserDao 对象，调用方法

```
public class App {

    public static void main(String[ ] args) {

        ApplicationContext ac =
            new ClassPathXmlApplicationContext('aa/applicationContext.xml');

        IUser iUser = (IUser) ac.getBean('proxy');

        iUser.save();
    }
}
```

```

    }
}

13     ApplicationContext ac =
14         new ClassPathXmlApplicationContext("aa/applicationContext.xml");
15     IUser iUser = (IUser) ac.getBean("proxy");
16     System.out.println(iUser.getClass());
17     iUser.save();
18
19
20
21
22
23
24
25 }

App()
五月 11, 2017 12:35:37 下午 org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@fb6c5f: startup date [Thu May 11 12:35:37 CST 2017]; root of context hierarchy
五月 11, 2017 12:35:37 下午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
信息: Loading XML bean definitions from class path resource [aa/applicationContext.xml]
五月 11, 2017 12:35:38 下午 org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons
信息: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@a70e3a: defining beans [proxy,AOP,userDao,org.springframework.context.annotation.internalResourceEditor]
class $Proxy3
开始事务
DB:保存用户
关闭事务
http://blog.csdn.net/hon_3y

```

## 2.3 工厂非静态方法

上面使用的是工厂静态方法来创建代理类对象。我们也使用一下非静态的工厂方法创建对象。

```

package aa;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

/**
 * Created by ozc on 2017/5/11.
 */

public class ProxyFactory {

```

```
    public Object getProxyInstance(final Object target_, final AOP aop_) {
```

//目标对象和关键点代码的类都是通过外界传递进来

```

    return Proxy.newProxyInstance(
        target_.getClass().getClassLoader(),
        target_.getClass().getInterfaces(),
        new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args) throws
            Throwable {
                aop_.begin();
                Object returnValue = method.invoke(target_, args);
                aop_.close();
                return returnValue;
            }
        });
    }
}

```

配置文件:先创建工厂，再创建代理类对象

```

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

```

<!--创建工厂-->

```

<bean id='factory' class='aa.ProxyFactory'/>

<!--通过工厂创建代理--/

<bean id='IUser' class='aa.IUser' factory-bean='factory' factory-method='getProx
yInstance'>

    <constructor-arg index='0' ref='userDao'/>
    <constructor-arg index='1' ref='AOP'/>
</bean>

<context:component-scan base-package='aa'/>
</beans>

```

效果如下：

The screenshot shows an IDE interface with the following details:

- Project Explorer:** Shows files like User, ProxyFactory, UserDao, and SpringLearning.iml.
- Java Editor:** Displays the following code in the App.java file:
 

```

public class App {
    public static void main(String[] args) {
        ApplicationContext ac =
            new ClassPathXmlApplicationContext("aa/applicationContext.xml");

        IUser iUser = (IUser) ac.getBean("IUser");
        System.out.println(iUser.getClass());
        iUser.save();
    }
}

```
- Console:** Shows a log message from the application output:
 

```

五月 11, 2017 12:54:53 十一 org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons
信息: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@1ec4a0c: de

```
- Debugger:** Shows a stack trace with frames like \$Proxy3, 开始事务, DB: 保存用户, and 关闭事务.
- Bottom Status Bar:** Shows the URL [http://blog.csdn.net/hon\\_3y](http://blog.csdn.net/hon_3y).

### 3. AOP 的概述

Aop : aspect object programming 面向切面编程

- 功能：让关注点代码与业务代码分离！
- 面向切面编程就是指：对很多功能都有的重复的代码抽取，再在运行的时候往业务方法上动态植入“切面类代码”。

关注点：重复代码就叫做关注点。

```

// 保存一个用户
public void add(User user) {
    Session session = null;
    Transaction trans = null;
    try {
        session = HibernateSessionFactoryUtils.getSession(); // 【关注点代码】
        trans = session.beginTransaction(); // 【关注点代码】

        session.save(user); // 核心业务代码

        trans.commit(); //... 【关注点代码】

    } catch (Exception e) {
        e.printStackTrace();
        if(trans != null){
            trans.rollback(); //.. 【关注点代码】
        }
    } finally{
        HibernateSessionFactoryUtils.closeSession(session); //...
    }
}

```

切面：关注点形成的类，就叫切面(类)！

```

public class AOP {

    public void begin() {
        System.out.println("开始事务");
    }

    public void close() {
        System.out.println("关闭事务");
    }
}

```

```
    }  
}
```

切入点：

- 执行目标对象方法，动态植入切面代码。
- 可以通过切入点表达式，指定拦截哪些类的哪些方法；给指定的类在运行的时候植入切面类代码。

切入点表达式：

- 指定哪些类的哪些方法被拦截





如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜 **Java3y** 公众号有我的联系方式。更多原创技术文章可关注我的 GitHub：

<https://github.com/ZhongFuCheng3y/3y>

## 4. 使用 Spring AOP 开发步骤

更新：如果用 maven 的同学，引入 pom 依赖就好了

1) 先引入 aop 相关 jar 文件 (aspectj aop 优秀组件)

- spring-aop-3.2.5.RELEASE.jar 【spring3.2 源码】
- aopalliance.jar 【spring2.5 源码/lib/aopalliance】
- aspectjweaver.jar 【spring2.5 源码/lib/aspectj】或【aspectj-1.8.2\lib】
- aspectjrt.jar 【spring2.5 源码/lib/aspectj】或【aspectj-1.8.2\lib】

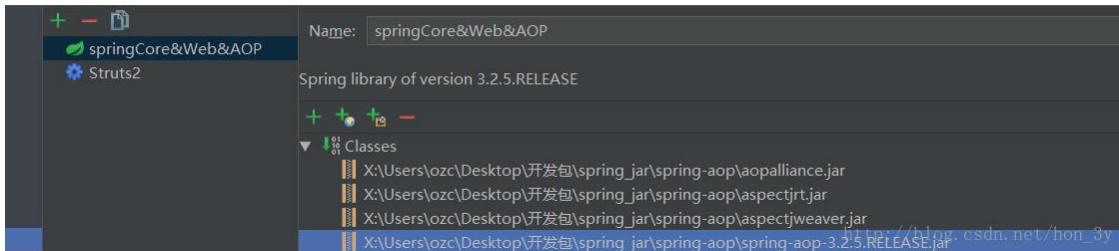
注意：用到 spring2.5 版本的 jar 文件，如果用 jdk1.7 可能会有问题。

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

## 2) bean.xml 中引入 aop 名称空间

- `xmlns:context='http://www.springframework.org/schema/context'`
- `http://www.springframework.org/schema/context`
- `http://www.springframework.org/schema/context/spring-context.xsd`

引入 4 个 jar 包：



## 4.1 引入名称空间

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

</beans>
```

## 4.2 注解方式实现 AOP 编程

我们之前手动的实现 AOP 编程是需要自己来编写代理工厂的，现在有了 Spring，就不需要我们自己写代理工厂了。Spring 内部会帮我们创建代理工厂。也就是说，不用我们自己写代理对象了。

因此，我们只要关心切面类、切入点、编写切入表达式指定拦截什么方法就可以了！

还是以上一个例子为案例，使用 Spring 的注解方式来实现 AOP 编程

### 4.2.1 在配置文件中开启 AOP 注解方式

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

```
<context:component-scan base-package='aa'/>

<!-- 开启 aop 注解方式 -->
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>
```

```
</beans>
```

### 4.2.2 代码：

切面类

```

@Component
@Aspect//指定为切面类
public class AOP {

    //里面的值为切入点表达式
    @Before("execution(* aa.*.*(..))")
    public void begin() {
        System.out.println("开始事务");
    }

    @After("execution(* aa.*.*(..))")
    public void close() {
        System.out.println("关闭事务");
    }
}

```

UserDao 实现了 IUser 接口

```

@Component
public class UserDao implements IUser {

    @Override
    public void save() {
        System.out.println("DB:保存用户");
    }

}

```

IUser 接口

```

public interface IUser {
    void save();
}

```

测试代码：

```

public class App {

    public static void main(String[] args) {

        ApplicationContext ac =
            new ClassPathXmlApplicationContext("aa/applicationContext.xml");

        //这里得到的是代理对象....
        IUser iUser = (IUser) ac.getBean("userDao");

        System.out.println(iUser.getClass());

        iUser.save();

    }
}

```

效果：

The screenshot shows an IDE interface with the following details:

- Project Structure:** The project structure on the left shows a package named "aa" containing classes "AOP", "App", "applicationContext.xml", "IUser", and "UserDao", along with a "SpringLearning.iml" file and "External Libraries".
- Code Editor:** The main editor window displays the Java code provided above.
- Run Tab:** The bottom tab bar is labeled "Run" and shows the output of the application's execution.
- Output Log:** The log output includes the following text:
  - "信息: Loading XML bean definitions from class path resource [aa/applicationContext.xml]"
  - "五月 11, 2017 2:47:32 下午 org.springframework.beans.factory.support.DefaultListableBeanFactory pre"
  - "信息: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory for beans defined in class path resource [aa/applicationContext.xml]"
  - "class \$Proxy7"
  - "开始事务"
  - "DB: 保存用户"
  - "关闭事务"
- Annotations:** A yellow callout box highlights the word "动态代理" (Dynamic Proxy) in the code editor.
- Bottom Status:** The status bar at the bottom right shows the URL "http://blog.csdn.net/hon\_3y".

---

## 4.3 目标对象没有接口

上面我们测试的是 UserDao 有 IUser 接口，内部使用的是动态代理...那么我们这次测试的是目标对象没有接口

OrderDao 没有实现接口

```
@Component  
public class OrderDao {  
  
    public void save() {  
  
        System.out.println("我已经进货了！！！");  
  
    }  
}
```

测试代码：

```
public class App {  
  
    public static void main(String[] args) {  
  
        ApplicationContext ac =  
            new ClassPathXmlApplicationContext("aa/applicationContext.xml");  
  
        OrderDao orderDao = (OrderDao) ac.getBean("orderDao");  
  
        System.out.println(orderDao.getClass());  
  
        orderDao.save();  
    }  
}
```

```
    }  
}
```

效果：

The screenshot shows an IDE interface with a code editor and a terminal window. The code editor contains Java code with annotations. A yellow callout box points to the code, containing the text "使用cglib代理". The terminal window shows Spring framework startup logs.

```
17 OrderDao orderDao = (OrderDao) ac.getBean("orderDao");  
18  
19 System.out.println(orderDao.getClass());  
20  
21 orderDao.save();  
22  
23 }  
24  
25
```

使用cglib代理

```
App (1)  
五月 11, 2017 2:54:02 下午 org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh  
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@1469658: startup date [Th  
五月 11, 2017 2:54:02 下午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions  
信息: Loading XML bean definitions from class path resource [aa/applicationContext.xml]  
五月 11, 2017 2:54:03 下午 org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiate  
信息: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@1b  
class aa.OrderDao$$EnhancerByCGLIB$$add721da  
开始事务  
我已经进货了！！！  
关闭事务
```

## 4.4 AOP 注解 API

api:

- `@Aspect` 指定一个类为切面类
- `@Pointcut('execution(* cn.itcast.eaopanno..(..))')` 指定切入点表达式
- `@Before('pointCut_()')` 前置通知：目标方法之前执行
- `@After('pointCut_()')` 后置通知：目标方法之后执行（始终执行）
- `@AfterReturning('pointCut_()')` 返回后通知：执行方法结束前执行(异常不执行)
- `@AfterThrowing('pointCut_()')` 异常通知：出现异常时候执行

- `@Around("pointCut_()")` 环绕通知：环绕目标方法执行

```
// 前置通知：在执行目标方法之前执行
@Before("pointCut_()")
public void begin(){
    System.out.println("开始事务/异常");
}

// 后置/最终通知：在执行目标方法之后执行 【无论是否出现异常最终都会执行】
@After("pointCut_()")
public void after(){
    System.out.println("提交事务/关闭");
}

// 返回后通知：在调用目标方法结束后执行 【出现异常不执行】
@AfterReturning("pointCut_()")
public void afterReturning() {
    System.out.println("afterReturning()");
}

// 异常通知：当目标方法执行异常时候执行此关注点代码
@AfterThrowing("pointCut_()")
public void afterThrowing(){
    System.out.println("afterThrowing()");
}

// 环绕通知：环绕目标方法执行
@Around("pointCut_()")
public void around(ProceedingJoinPoint pjp) throws Throwable{
    System.out.println("环绕前....");
    pjp.proceed(); // 执行目标方法
}
```

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

## 4.5 表达式优化

我们的代码是这样的：每次写 Before、After 等，都要重写一次切入点表达式，这样就不优雅了。

```
@Before("execution(* aa.*.*(..))")
public void begin() {
    System.out.println("开始事务");
}
```

```
@After("execution(* aa.*.*(..))")
public void close() {
    System.out.println("关闭事务");
}
```

于是乎，我们要使用@Pointcut 这个注解，来指定切入点表达式，在用到的地方中，直接引用就行了！

那么我们的代码就可以改造成这样了：

```
@Component
@Aspect//指定为切面类
public class AOP {

    // 指定切入点表达式，拦截哪个类的哪些方法
    @Pointcut("execution(* aa.*.*(..))")
    public void pt() {

    }

    @Before("pt()")
    public void begin() {
```

```
System.out.println('开始事务');
}

@After('pt()')
public void close() {
    System.out.println('关闭事务');
}
}
```

---

## 4.6 XML 方式实现 AOP 编程

### XML 文件配置

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

```
<!--对象实例--/
<bean id='userDao' class='aa.UserDao'/>
<bean id='orderDao' class='aa.OrderDao'/>
```

```

<!-- 切面类-->
<bean id='aop' class='aa.AOP'/>

<!--AOP 配置-->
<aop:config />

<!-- 定义切入表达式，拦截哪些方法-->
<aop:pointcut id='pointCut' expression='execution(* aa.*.*(..))' />

<!-- 指定切面类是哪个-->
<aop:aspect ref='aop' >

    <!--指定来拦截的时候执行切面类的哪些方法-->
    <aop:before method='begin' pointcut-ref='pointCut' />
    <aop:after method='close' pointcut-ref='pointCut' />

</aop:aspect>
</aop:config>

</beans>

```

测试：

```

public class App {

    @Test
    public void test1() {

        ApplicationContext ac =
            new ClassPathXmlApplicationContext("aa/applicationContext.xml");

        OrderDao orderDao = (OrderDao) ac.getBean("orderDao");
    }
}

```

```
System.out.println(orderDao.getClass());  
  
orderDao.save();  
  
}  
  
@Test  
public void test2() {  
  
    ApplicationContext ac =  
        new ClassPathXmlApplicationContext("aa/applicationContext.xml");  
  
    IUser userDao = (IUser) ac.getBean("userDao");  
  
    System.out.println(userDao.getClass());  
  
    userDao.save();  
  
}  
}
```

测试 OrderDao

SpringLearning C:\SpringLearning

```

13     public void test1() {
14
15         ApplicationContext ac =
16             new ClassPathXmlApplicationContext("aa/applicationContext.xml");
17
18         OrderDao orderDao = (OrderDao) ac.getBean("orderDao");
19
20         System.out.println(orderDao.getClass());
21
22         orderDao.save();
23
24     }
25
26     @Test
27     public void test2() {
28
29         ApplicationContext ac =
30             new ClassPathXmlApplicationContext("aa/applicationContext.xml");
31
32         IUser userDao = (IUser) ac.getBean("userDao");
33
34         System.out.println(userDao.getClass());
35
36         userDao.save();
37
38     }
39
40 }
```

App (aa) 637ms

App (aa) 637ms

```

[X:\Program Files (x86)\Java\jdk1.7.0\bin\java" ...]
五月 11, 2017 3:16:12 下午 org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@1ea817f: startup date [Thu May 11 15:16:12 CST 2017]; root of context hierarchy
五月 11, 2017 3:16:12 下午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
信息: Loading XML bean definitions from class path resource [aa/applicationContext.xml]
五月 11, 2017 3:16:12 下午 org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons
信息: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@32b87: defining beans [userDao, orderDao, app, org.springframework.context.annotation.internalResourceEditorRegistrar]
class aa.OrderDao$$EnhancerByCGLIB$$4a4fd1d5
开始事务
我已经进货了!!!
关闭事务

Process finished with exit code 0
```

http://blog.csdn.net/hon\_3y

## 测试 UserDao

SpringLearning C:\SpringLearning

```

13     public void test1() {
14
15         ApplicationContext ac =
16             new ClassPathXmlApplicationContext("aa/applicationContext.xml");
17
18         IUser userDao = (IUser) ac.getBean("userDao");
19
20         System.out.println(userDao.getClass());
21
22         userDao.save();
23
24     }
25
26     @Test
27     public void test2() {
28
29         ApplicationContext ac =
30             new ClassPathXmlApplicationContext("aa/applicationContext.xml");
31
32         IUser userDao = (IUser) ac.getBean("userDao");
33
34         System.out.println(userDao.getClass());
35
36         userDao.save();
37
38     }
39
40 }
```

App (aa) 887ms

App (aa) 887ms

```

[X:\Program Files (x86)\Java\jdk1.7.0\bin\java" ...
五月 11, 2017 3:17:27 下午 org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh
信息: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@dd794: startup date [Thu May 11 15:17:27 CST 2017]; root of context hierarchy
五月 11, 2017 3:17:27 下午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
信息: Loading XML bean definitions from class path resource [aa/applicationContext.xml]
五月 11, 2017 3:17:27 下午 org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons
信息: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@19ebfd1: defining beans [userDao, orderDao, app, org.springframework.context.annotation.internalResourceEditorRegistrar]
class SProxy
开始事务
DB:保存用户
关闭事务

Process finished with exit code 0
```

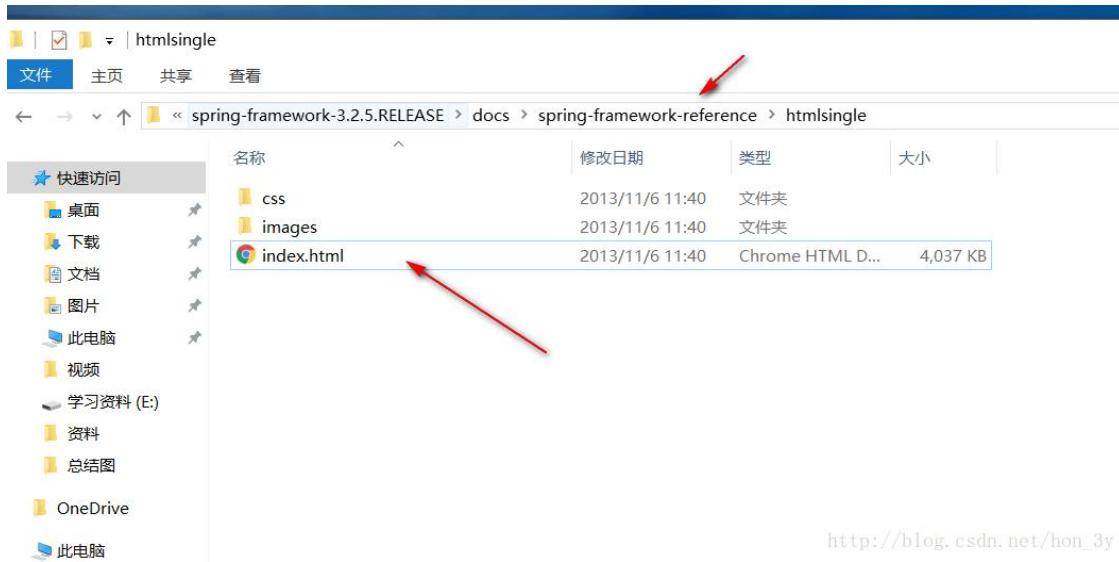
http://blog.csdn.net/hon\_3y

## 5. 切入点表达式

切入点表达式主要就是来配置拦截哪些类的哪些方法

### 5.1 查官方文档

我们去文档中找找它的语法..



在文档中搜索:execution(

The screenshot shows the search results for 'execution(' in the Spring Framework documentation. The results page has a header '第 6 页, 共 44 条' with a red arrow pointing to it. Below the header, there is a snippet of XML code demonstrating the use of the execution pointcut. A red arrow points to the word 'execution'. Further down, under 'Examples', another red arrow points to the word 'execution'. A yellow box highlights the word '语法' (syntax) next to a link. The page also contains a detailed explanation of the execution pointcut syntax.

The execution pointcut designator is the most often used. The format of an execution expression is:

```
① execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern))
```

All parts except the returning type pattern (ret-type-pattern in the snippet above), name pattern, and parameters pattern are optional. The returning type pattern determines what the return type of the method must be in order for a join point to be matched. Most frequently you will use \* as the returning type pattern, which matches any return type. A fully-qualified type name will match only when the method returns the given type. The name pattern matches the method name. You can use the \* wildcard as all or part of a name pattern. The parameters pattern is slightly more complex: () matches a method that takes no parameters, whereas .. matches any number of parameters (zero or more). The pattern (\*) matches a method taking one parameter of any type, (\*,String) matches a method taking two parameters, the first can be of any type, the second must be a String. Consult the Language Semantics section of the AspectJ Programming Guide for more information.

Some examples of common pointcut expressions are given below.

## 5.2 语法解析

那么它的语法是这样子的：

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern  
param-pattern) throws-pattern?)
```

符号讲解：

- ?号代表 0 或 1，可以不写
- “\*” 号代表任意类型，0 或多
- 方法参数为..表示为可变参数

参数讲解：

- modifiers-pattern? 【修饰的类型，可以不写】
- ret-type-pattern 【方法返回值类型，必写】
- declaring-type-pattern? 【方法声明的类型，可以不写】
- name-pattern(param-pattern) 【要匹配的名称，括号里面是方法的参数】
- throws-pattern? 【方法抛出的异常类型，可以不写】

官方也有给出一些例子给我们理解：

Some examples of common pointcut expressions are given below.

- the execution of any public method:  
`execution(public * *(..))` 以public修饰的，方法返回值任意，方法名任意
- the execution of any method with a name beginning with "set":  
`execution(* set*(..))` 任意返回值，以set开头的方法
- the execution of any method defined by the AccountService interface:  
`execution(* com.xyz.service.AccountService.*(..))` 返回值任意，以com.xyz.service.AccountService类下的所有方法
- the execution of any method defined in the service package:  
`execution(* com.xyz.service.*.*(..))` 返回值任意，以service包下的所有方法
- the execution of any method defined in the service package or a sub-package:  
`execution(* com.xyz.service..*.*(..))` 返回值任意，以service包或子包的所有方法

### 5.3 测试代码

```
<!-- 【拦截所有 public 方法】 -->  
<!--<aop:pointcut expression='execution(public * *(..))' id='pt'/|--|  
  
<!-- 【拦截所有 save 开头的方法】 -->  
<!--<aop:pointcut expression='execution(* save*(..))' id='pt'/|--|  
  
<!-- 【拦截指定类的指定方法，拦截时候一定要定位到方法】 -->  
<!--<aop:pointcut expression='execution(public * cn.itcast.g_pointcut.OrderDao.save(..))' id='pt'/|--|  
  
<!-- 【拦截指定类的所有方法】 -->  
<!--<aop:pointcut expression='execution(* cn.itcast.g_pointcut.UserDao.*(..))' id='pt'/|--|  
  
<!-- 【拦截指定包，以及其自包下所有类的所有方法】 -->  
<!--<aop:pointcut expression='execution(* cn..*.*(..))' id='pt'/|--|  
  
<!-- 【多个表达式】 -->  
<!--<aop:pointcut expression='execution(* cn.itcast.g_pointcut.UserDao.save()) || execution(* cn.itcast.g_pointcut.OrderDao.save())' id='pt'/|--|  
     <!--<aop:pointcut expression='execution(* cn.itcast.g_pointcut.UserDao.save()) or execution(* cn.itcast.g_pointcut.OrderDao.save())' id='pt'/|--|  
         <!-- 下面 2 个且关系的，没有意义 -->  
             <!--<aop:pointcut expression='execution(* cn.itcast.g_pointcut.UserDao.save()) && execution(* cn.itcast.g_pointcut.OrderDao.save())' id='pt'/|--|  
                 <!--<aop:pointcut expression='execution(* cn.itcast.g_pointcut.UserDao.save()) and execution(* cn.itcast.g_pointcut.OrderDao.save())' id='pt'/|--|  
  
<!-- 【取非值】 -->  
<!--<aop:pointcut expression='!execution(* cn.itcast.g_pointcut.OrderDao.save())' id='pt'/|--|
```

加油！



如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜 **Java3y** 公众号有我的联系方式。更多原创技术文章可关注我的 GitHub：

<https://github.com/ZhongFuCheng3y/3y>

# JDBCTemplate 和 Spring 事务

上一篇 Spring 博文主要讲解了如何使用 Spring 来实现 AOP 编程，本博文主要讲解 Spring 的 DAO 模块对 JDBC 的支持，以及 Spring 对事务的控制...

对于 JDBC 而言，我们肯定不会陌生，我们在初学的时候肯定写过非常非常多的 JDBC 模板代码！

## 1. 回顾对模版代码优化过程

我们来回忆一下我们怎么对模板代码进行优化的！

首先来看一下我们原生的 JDBC：需要手动去数据库的驱动从而拿到对应的连接..

```
try {
    String sql = 'insert into t_dept(deptName) values('test')';
    Connection con = null;
    Statement stmt = null;
    Class.forName('com.mysql.jdbc.Driver');
    // 连接对象
    con = DriverManager.getConnection('jdbc:mysql://hib_demo', 'root',
                                    'root');
    // 执行命令对象
    stmt = con.createStatement();
    // 执行
    stmt.execute(sql);

    // 关闭
    stmt.close();
    con.close();
} catch (Exception e) {
```

```
        e.printStackTrace();
    }
}
```

因为 JDBC 是面向接口编程的，因此数据库的驱动都是由数据库的厂商给做到好了，我们只要加载对应的数据库驱动，便可以获取对应的数据库连接....因此，我们写了一个工具类，专门来获取与数据库的连接(Connection)，当然啦，为了更加灵活，我们的工具类是读取配置文件的方式做的。

```
/*
 * 连接数据库的 driver , url , username , password 通过配置文件来配置，可以增加灵活性
 * 当我们需要切换数据库的时候，只需要在配置文件中改以上的信息即可
 *
 */
private static String driver = null;
private static String url = null;
private static String username = null;
private static String password = null;

static {
    try {

        //获取配置文件的读入流
        InputStream inputStream = UtilsDemo.class.getClassLoader().getResourceAsStream("db.properties");

        Properties properties = new Properties();
        properties.load(inputStream);

        //获取配置文件的信息
        driver = properties.getProperty("driver");
        url = properties.getProperty("url");
        username = properties.getProperty("username");
    }
}
```

```
password = properties.getProperty("password");

//加载驱动类
Class.forName(driver);

}

} catch (IOException e) {
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}

}

public static Connection getConnection() throws SQLException {
    return DriverManager.getConnection(url,username,password);
}
public static void release(Connection connection, Statement statement, ResultSet resultSet) {

    if (resultSet != null) {
        try {
            resultSet.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    if (statement != null) {
        try {
            statement.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

```
if (connection != null) {  
    try {  
        connection.close();  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}
```

经过上面一层的封装，我们可以在使用的地方直接使用工具类来得到与数据库的连接...那么比原来就方便很多了！但是呢，每次还是需要使用 Connection 去创建一个 Statement 对象。并且无论是什么方法，其实就是 SQL 语句和传递进来的参数不同！

于是我们可以使用 **DBUtils** 这样的组件来解决上面的问题

## 2. 使用 Spring 的 JDBC

上面已经回顾了一下以前我们的 JDBC 开发了，那么看看 Spring 对 JDBC 又是怎么优化的

首先，想要使用 Spring 的 JDBC 模块，就必须引入两个 jar 文件：

- 引入 jar 文件（如果用 maven 的同学，直接使用 pom 文件依赖导入就好了）
  - spring-jdbc-3.2.5.RELEASE.jar
  - spring-tx-3.2.5.RELEASE.jar

首先还是看一下我们原生的 JDBC 代码：获取 Connection 是可以抽取出来的，直接使用 dataSource 来得到 Connection 就行了。

```
public void save() {  
    try {  
        String sql = 'insert into t_dept(deptName) values('test');';  
        Connection con = null;
```

```

        Statement stmt = null;
        Class.forName("com.mysql.jdbc.Driver");
        // 连接对象
        con = DriverManager.getConnection("jdbc:mysql://hib_demo", "root",
        "root");
        // 执行命令对象
        stmt = con.createStatement();
        // 执行
        stmt.execute(sql);

        // 关闭
        stmt.close();
        con.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

值得注意的是，JDBC 对 C3P0 数据库连接池是有很好的支持的。因此我们直接可以使用 Spring 的依赖注入，在配置文件中配置 dataSource 就行了！

```

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"|
    <property name="driverClass" value="com.mysql.jdbc.Driver" |</propert
y|
    <property name="jdbcUrl" value="jdbc:mysql://hib_demo" |</property|
    <property name="user" value="root" |</property|
    <property name="password" value="root" |</property|
    <property name="initialPoolSize" value="3" |</property|
    <property name="maxPoolSize" value="10" |</property|
    <property name="maxStatements" value="100" |</property|
    <property name="acquireIncrement" value="2" |</property|
</bean|

```

```

// IOC 容器注入

private DataSource dataSource;

public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
}

public void save() {
    try {
        String sql = 'insert into t_dept(deptName) values('test')';
        Connection con = null;
        Statement stmt = null;
        // 连接对象
        con = dataSource.getConnection();
        // 执行命令对象
        stmt = con.createStatement();
        // 执行
        stmt.execute(sql);

        // 关闭
        stmt.close();
        con.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

Spring 来提供了 JdbcTemplate 这么一个类给我们使用！它封装了 DataSource，也就是说我们可以在 Dao 中使用 JdbcTemplate 就行了。

创建 dataSource，创建 jdbcTemplate 对象

```

<?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:c='http://www.springframework.org/schema/c'
xsi:schemaLocation='http://www.springframework.org/schema/beans http://www.spring
framework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema
/context http://www.springframework.org/schema/context/spring-context.xsd'|
```

```
<bean id='dataSource' class='com.mchange.v2.c3p0.ComboPooledDataSource'|
<property name='driverClass' value='com.mysql.jdbc.Driver' |</property|
<property name='jdbcUrl' value='jdbc:mysql:///zhongfucheng' |</property|
<property name='user' value='root' |</property|
<property name='password' value='root' |</property|
<property name='initialPoolSize' value='3' |</property|
<property name='maxPoolSize' value='10' |</property|
<property name='maxStatements' value='100' |</property|
<property name='acquireIncrement' value='2' |</property|
</bean|
```

```
<!--扫描注解--|
<context:component-scan base-package='bb' /|
```

```
<!-- 2. 创建 JdbcTemplate 对象 --|
<bean id='jdbcTemplate' class='org.springframework.jdbc.core.JdbcTemplate'|
<property name='dataSource' ref='dataSource' |</property|
</bean|
```

```
</beans|
```

```
userDao
```

```
package bb;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
```

```
import org.springframework.stereotype.Component;

/**
 * Created by ozc on 2017/5/10.
 */

@Component
public class UserDao implements IUser {

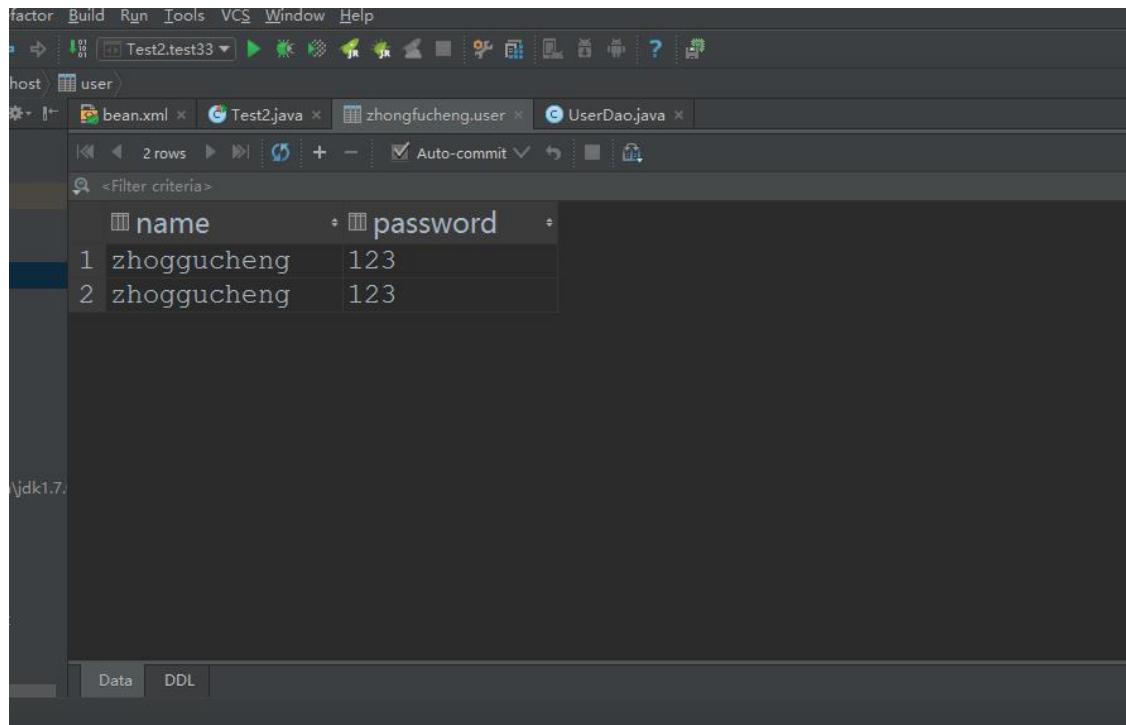
    //使用 Spring 的自动装配
    @Autowired
    private JdbcTemplate template;

    @Override
    public void save() {
        String sql = 'insert into user(name,password) values('zhoggucheng','123')';
        template.update(sql);
    }

}

测试：

@Test
public void test33() {
    ApplicationContext ac = new ClassPathXmlApplicationContext("bb/bean.xml");
    UserDao userDao = (UserDao) ac.getBean("userDao");
    userDao.save();
}
```



The screenshot shows the DataGrip IDE interface. At the top, there's a menu bar with options like 'factor', 'Build', 'Run', 'Tools', 'VCS', 'Window', and 'Help'. Below the menu is a toolbar with various icons. The main workspace shows a database connection named 'user'. In the center, there's a table view with two rows of data. The columns are labeled 'name' and 'password'. The first row has 'zhoggucheng' in the 'name' column and '123' in the 'password' column. The second row also has 'zhoggucheng' in the 'name' column and '123' in the 'password' column. Below the table, there are tabs for 'Data' and 'DDL'. At the bottom of the screen, there's a terminal window displaying command-line output.

```
1 test passed - 1s 669ms
am Files (x86)\Java\jdk1.7.0\bin\java" ...
2017 10:37:22 下午 org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh
刷新 org.springframework.context.support.ClassPathXmlApplicationContext@1e8ba10: startup date [Sun May 14 22:37:22 CST 2017]; root of context hierarchy
2017 10:37:23 下午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
正在从类路径资源 [bb/bean.xml] 加载 XML bean 定义
2017 10:37:23 下午 org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons
正在为单例在 org.springframework.beans.factory.support.DefaultListableBeanFactory@137c1f1: 定义 beans [dataSource, userDao, org.springframework.jdbc.core.JdbcTemplate]
2017 10:37:23 下午 com.mchange.v2.log.MLog <clinit>
正在使用 Java 1.4+ 标准日志。
2017 10:37:23 下午 com.mchange.v2.c3p0.C3P0Registry banner
正在初始化 c3p0-0.9.2-pre1 [构建于 2010-05-27 01:00:49 -0400; debug? true; trace: 10]
2017 10:37:24 下午 com.mchange.v2.c3p0.impl.AbstractPoolBackedDataSource getPoolManager
正在初始化 c3p0 pool... com.mchange.v2.c3p0.ComboPooledDataSource [ acquireIncrement -> 2, acquireRetryAttempts -> 30, acquireRetryDelay -> 1000, aut
http://blog.csdn.net/hon_3y
```

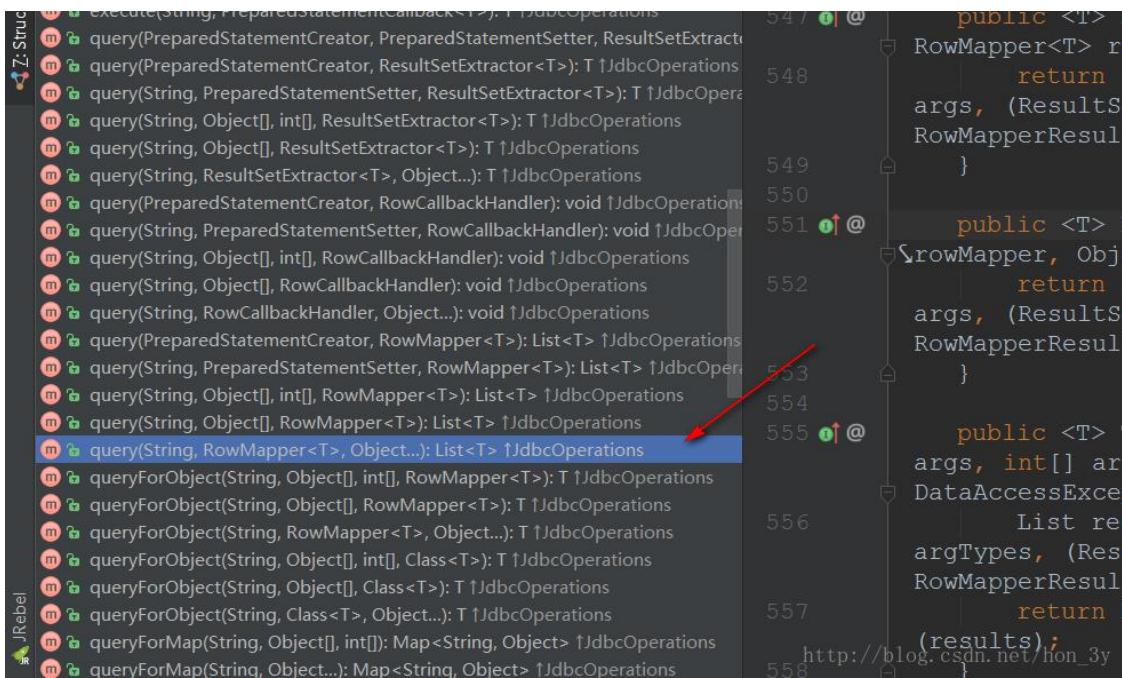
## 2.1 JdbcTemplate 查询

我们要是使用 JdbcTemplate 查询会发现有很多重载了 query()方法

```
m  ↪ query(PreparedStatementCreator psc, PreparedStatementSetter pss, ResultSet... rs)
m  ↪ query(PreparedStatementCreator psc, ResultSetExtractor<T> rse)           T
m  ↪ query(PreparedStatementCreator psc, RowCallbackHandler rch)                 void
m  ↪ query(String sql, Object[] args, int[] argTypes, ResultSetExtractor<T> rse)   T
m  ↪ query(String sql, Object[] args, int[] argTypes, RowCallbackHandler rch)    void
m  ↪ query(PreparedStatementCreator psc, RowMapper<T> rowMapper)                  List<T>
m  ↪ query(String sql, Object[] args, int[] argTypes, RowMapper<T> rowMapper)    List<T>
m  ↪ query(String sql, Object[] args, ResultSetExtractor<T> rse)                T
m  ↪ query(String sql, Object[] args, RowCallbackHandler rch)                   void
m  ↪ query(String sql, Object[] args, RowMapper<T> rowMapper)                  List<T>
m  ↪ query(String sql, PreparedStatementSetter pss, ResultSetExtractor<T> rse)   T
m  ↪ query(String sql, PreparedStatementSetter pss, RowCallbackHandler rch)      void
m  ↪ query(String sql, PreparedStatementSetter pss, RowMapper<T> rowMapper)     List<T>
m  ↪ query(String sql, ResultSetExtractor<T> rse)                            T
m  ↪ query(String sql, ResultSetExtractor<T> rse, Object... args)             T
m  ↪ query(String sql, RowCallbackHandler rch)                           void
m  ↪ query(String sql, RowCallbackHandler rch, Object... args)               void
m  ↪ query(String sql, RowMapper<T> rowMapper)                      List<T>
m  ↪ query(String sql, RowMapper<T> rowMapper, Object... args)            List<T>
m  ↪ queryForList(String sql)                                         List<Map<String, Object>>
m  ↪ queryForList(String sql, Class<T> elementType)                     List<T>
Ctrl+向下箭头 and Ctrl+向上箭头 will move caret down and up in the editor >> http://blog.csdn.net/fan_3v
```

一般地，如果我们使用 `queryForMap()`，那么只能封装一行的数据，如果封装多行的数据、那么就会报错！并且，Spring 是不知道我们想把一行数据封装成是什么样的，因此返回值是 `Map` 集合...我们得到 `Map` 集合的话还需要我们自己去转换成自己需要的类型。

我们一般使用下面这个方法：



我们可以实现 RowMapper，告诉 Spring 我们将每行记录封装成怎么样的。

```

public void query(String id) {
    String sql = 'select * from USER where password=?';

    List<User> query = template.query(sql, new RowMapper<User>() {
        //将每行记录封装成 User 对象
        @Override
        public User mapRow(ResultSet resultSet, int i) throws SQLException {
            User user = new User();
            user.setName(resultSet.getString('name'));
            user.setPassword(resultSet.getString('password'));

            return user;
        }
    }, id);
}

```

```
        System.out.println(query);
    }
```

The screenshot shows an IDE interface with a code editor and a terminal window.

**Code Editor:**

```
10 public class Test2 {
11
12     @Test
13     public void test33() {
14         ApplicationContext ac = new ClassPathXmlApplicationContext("bb/bean.xml");
15
16         UserDao userDao = (UserDao) ac.getBean("userDao");
17         userDao.query("222");
18     }
19 }
20
```

**Terminal/Terminal Output:**

```
Test2 (b 1s 96ms) 1 test passed - 1s 96ms
信息: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@19de0ec: defining beans [dataSource,userDao,com.mchange.v2.c3p0.C3P0Registry.banner];
五月 15, 2017 12:50:34 下午 com.mchange.v2.log.MLog <clinit>
信息: MLog clients using java 1.4+ standard logging.
五月 15, 2017 12:50:34 下午 com.mchange.v2.c3p0.C3P0Registry banner
信息: Initializing c3p0-0.9.2-pre1 [built 27-May-2010 01:00:49 -0400, debug? true, trace: 10]
五月 15, 2017 12:50:34 下午 com.mchange.v2.c3p0.impl.AbstractPoolBackedDataSource getPoolManager
信息: Initializing c3p0 pool... com.mchange.v2.c3p0.ComboPooledDataSource [ acquireIncrement -> 2, acquireRetryAttempts -> 30, acquireRetryDelay -> 1000
[User [name='zhong', password='222']]
```

[http://blog.csdn.net/hon\\_3y](http://blog.csdn.net/hon_3y)

---

当然了，一般我们都是将每行记录封装成一个 JavaBean 对象的，因此直接实现 RowMapper，在使用的时候创建就好了。

```
class MyResult implements RowMapper<Dept>{

    // 如何封装一行记录

    @Override
    public Dept mapRow(ResultSet rs, int index) throws SQLException {
        Dept dept = new Dept();
        dept.setDeptId(rs.getInt("deptId"));
        dept.setDeptName(rs.getString("deptName"));
        return dept;
    }
}
```



如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜 **Java3y** 公众号有我的联系方式。更多原创技术文章可关注我的 GitHub：

<https://github.com/ZhongFuCheng3y/3y>

### 3. 事务控制概述

下面主要讲解 Spring 的事务控制，如何使用 Spring 来对程序进行事务控制....

- Spring 的事务控制是属于 Spring Dao 模块的。

一般地，我们事务控制都是在 service 层做的。。为什么是在 service 层而不是在 dao 层呢？？有没有这样的疑问...

service 层是业务逻辑层，service 的方法一旦执行成功，那么说明该功能没有出错。

一个 service 方法可能要调用 dao 层的多个方法...如果在 dao 层做事务控制的话，一个 dao 方法出错了，仅仅把事务回滚到当前 dao 的功能，这样是不合适的[因为我们的业务由多个 dao 方法组成]。如果没有出错，调用完 dao 方法就 commit 了事务，这也是不合适的[导致太多的 commit 操作]。

事务控制分为两种：

- 编程式事务控制
- 声明式事务控制

#### 3.1 编程式事务控制

自己手动控制事务，就叫做编程式事务控制。

- Jdbc 代码：Conn.setAutoCommit(false); // 设置手动控制事务
- Hibernate 代码：Session.beginTransaction(); // 开启一个事务
- 特点：细粒度的事务控制：可以对指定的方法、指定的方法的某几行添加事务控制（比较灵活，但开发起来比较繁琐：每次都要开启、提交、回滚。）

#### 3.2 声明式事务控制

Spring 提供对事务的控制管理就叫做声明式事务控制

Spring 提供了对事务控制的实现。

- 如果用户想要使用 Spring 的事务控制，只需要配置就行了。

- 当不用 Spring 事务的时候，直接移除就行了。
- 特点：Spring 的事务控制是基于 Spring AOP 实现的。因此它的耦合度是非常低的。【粗粒度的事务控制：只能给整个方法应用事务，不可以对方法的某几行应用事务。】(因为 aop 拦截的是方法。)

Spring 给我们提供了事务的管理器类，事务管理器类又分为两种，因为 JDBC 的事务和 Hibernate 的事务是不一样的。

- Spring 声明式事务管理器类：
  - Jdbc 技术：DataSourceTransactionManager
  - Hibernate 技术：HibernateTransactionManager

---

### 3.3 声明式事务控制教程

我们基于 Spring 的 JDBC 来做例子吧

引入相关 jar 包(如果用 maven，那引入 pom 依赖就好了)

- AOP 相关的 jar 包【因为 Spring 的声明式事务控制是基于 AOP 的，那么就需要引入 AOP 的 jar 包。】
- 引入 tx 名称空间
- 引入 AOP 名称空间
- 引入 jdbcjar 包【jdbc.jar 包和 tx.jar 包】

---

#### 3.3.1 搭建配置环境

编写一个接口

```
public interface IUser {  
    void save();  
}
```

UserDao 实现类，使用 JdbcTemplate 对数据库进行操作！

```

@Repository
public class UserDao implements IUser {

    //使用 Spring 的自动装配
    @Autowired
    private JdbcTemplate template;

    @Override
    public void save() {
        String sql = 'insert into user(name,password) values('zhong','222)';
        template.update(sql);
    }

}

userService

@Service
public class UserService {

    @Autowired
    private UserDao userDao;
    public void save() {

        userDao.save();
    }
}

```

bean.xml 配置：配置数据库连接池、JdbcTemplate 对象、扫描注解

```

<?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:c='http://www.springframework.org/schema/c'
       xsi:schemaLocation='http://www.springframework.org/schema/beans http://www.spring
framework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema

```

```
/context http://www.springframework.org/schema/context/spring-context.xsd|
```

```
<!-- 数据连接池配置-->
<bean id='dataSource' class='com.mchange.v2.c3p0.ComboPooledDataSource'>
    <property name='driverClass' value='com.mysql.jdbc.Driver' />
    <property name='jdbcUrl' value='jdbc:mysql:///zhongfucheng' />
    <property name='user' value='root' />
    <property name='password' value='root' />
    <property name='initialPoolSize' value='3' />
    <property name='maxPoolSize' value='10' />
    <property name='maxStatements' value='100' />
    <property name='acquireIncrement' value='2' />
</bean>
```

```
<!-- 扫描注解-->
<context:component-scan base-package='bb' />
```

```
<!-- 2. 创建 JdbcTemplate 对象 -->
<bean id='jdbcTemplate' class='org.springframework.jdbc.core.JdbcTemplate'>
    <property name='dataSource' ref='dataSource' />
</bean>

</beans>
```

---

前面搭建环境的时候，是没有任何的事务控制的。也就是说，当我在 service 中调用两次 userDao.save()，即时在中途有异常抛出，还是可以在数据库插入一条记录的。

Service 代码：

```
@Service
public class UserService {

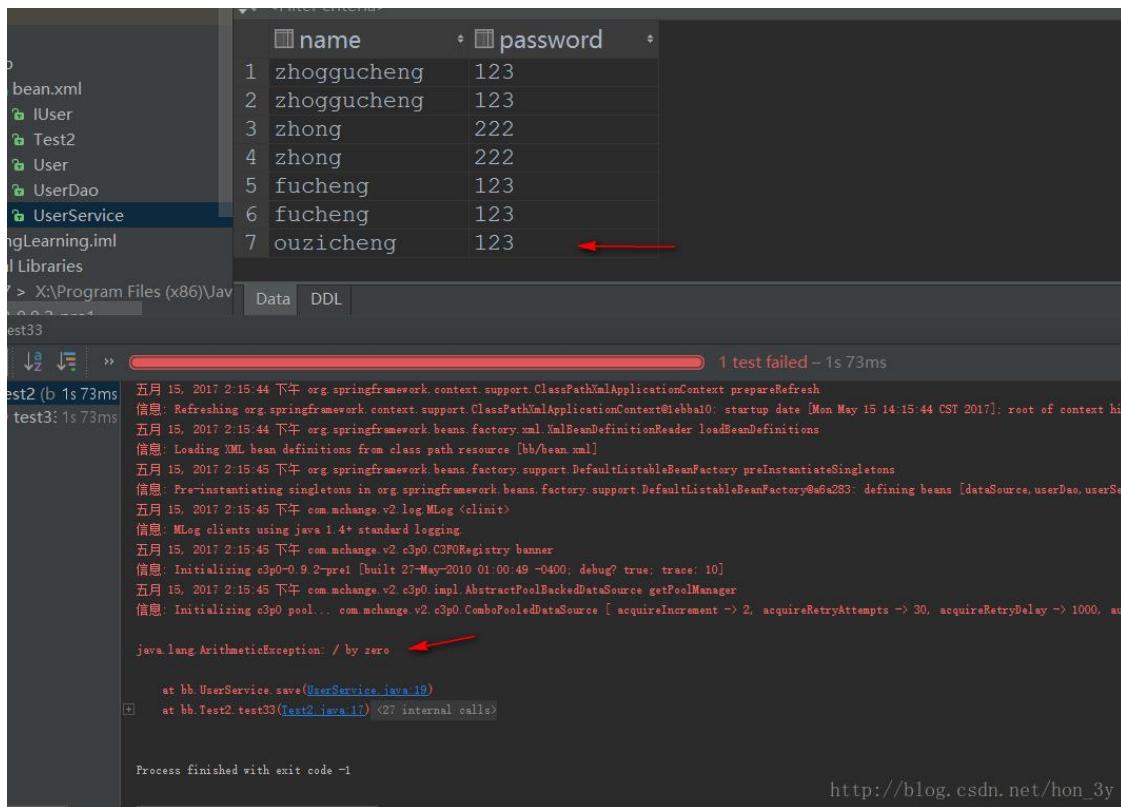
    @Autowired
    private UserDao userDao;
    public void save() {

        userDao.save();
        int i = 1 / 0;
        userDao.save();
    }
}
```

测试代码：

```
public class Test2 {

    @Test
    public void test33() {
        ApplicationContext ac = new ClassPathXmlApplicationContext("bb/bean.xml");
        UserService userService = (UserService) ac.getBean("userService");
        userService.save();
    }
}
```



### 3.3.2 XML 方式实现声明式事务控制

首先，我们要配置事务的管理器类：因为 JDBC 和 Hibernate 的事务控制是不同的。

```

<!--1.配置事务的管理器类:JDBC--/
<bean id='txManage' class='org.springframework.jdbc.datasource.DataSourceTransactionManager'>

<!--引用数据库连接池--/
<property name='dataSource' ref='dataSource'/>
</bean>

```

再而，配置事务管理器类如何管理事务

```

<!--2.配置如何管理事务-->
<tx:advice id='txAdvice' transaction-manager='txManage' |

    <!--配置事务的属性-->
    <tx:attributes|
        <!--所有的方法，并不是只读-->
        <tx:method name='*' read-only='false' />
    </tx:attributes|
</tx:advice|

```

最后，配置拦截哪些方法，

```

<!--3.配置拦截哪些方法+事务的属性-->
<aop:config|
    <aop:pointcut id='pt' expression='execution(* bb.UserService.*(..))' />
    <aop:advisor advice-ref='txAdvice' pointcut-ref='pt' |</aop:advisor|
</aop:config|

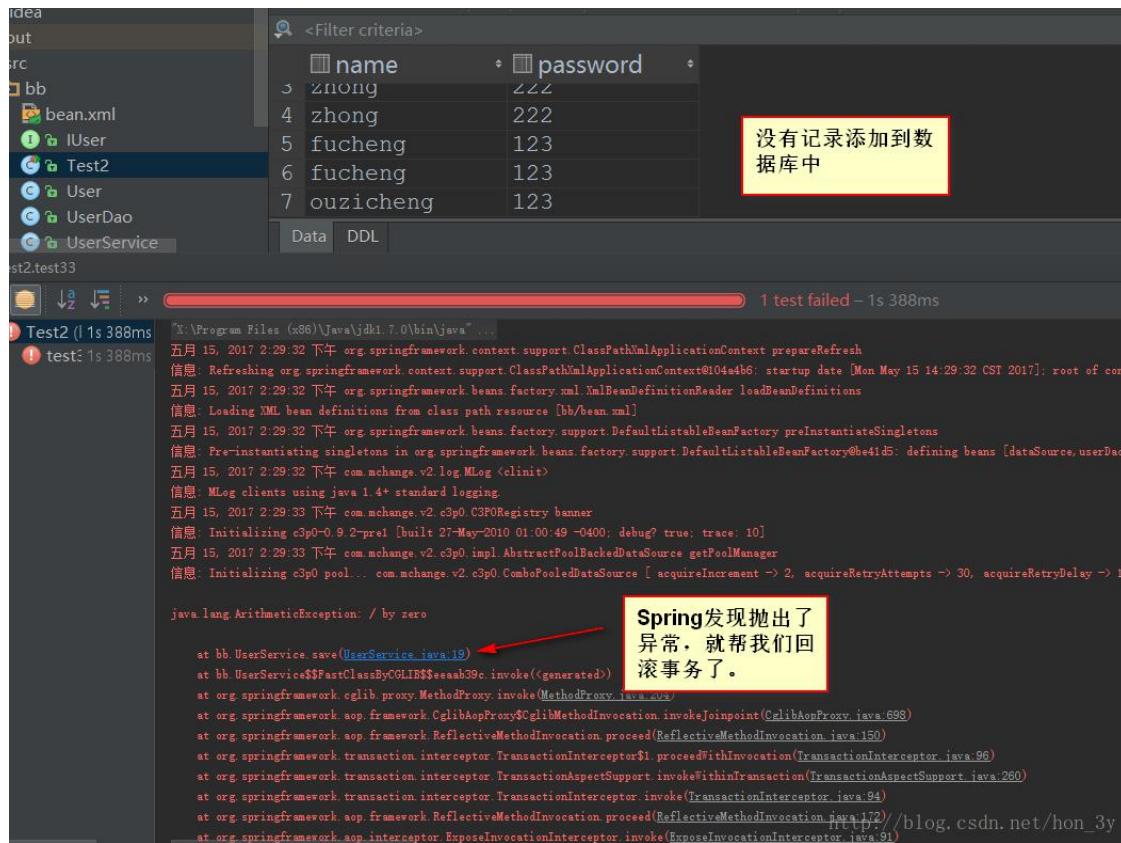
```

配置完成之后，service 中的方法都应该被 Spring 的声明式事务控制了。因此我们再次测试一下：

```

@Test
public void test33() {
    ApplicationContext ac = new ClassPathXmlApplicationContext("bb/bean.xml");
    UserService userService = (UserService) ac.getBean("userService");
    userService.save();
}

```



### 3.3.3 使用注解的方法实现事务控制

当然了，有的人可能觉得到 XML 文件上配置太多东西了。Spring 也提供了使用注解的方式来实现对事务控制

第一步和 XML 的是一样的，必须配置事务管理器类：

```

<!--1.配置事务的管理器类:JDBC-->

<bean id='txManage' class='org.springframework.jdbc.datasource.DataSourceTransac
tionManager'>

    <!--引用数据库连接池-->

    <property name='dataSource' ref='dataSource' />

</bean>

```

## 第二步：开启以注解的方式来实现事务控制

```
<!--开启以注解的方式来实现事务控制-->  
<tx:annotation-driven transaction-manager="txManage"/>
```

最后，想要控制哪个方法事务，在其前面添加`@Transactional`这个注解就行了！如果想要控制整个类的事务，那么在类上面添加就行了。

```
@Transactional  
public void save() {  
  
    userDao.save();  
  
    int i = 1 / 0;  
    userDao.save();  
}
```



The screenshot shows a code editor with Java code. A red arrow points to the `@Transactional` annotation on line 10. The code is as follows:

```
* Created by ozc on 2017/5/11.  
*/  
@Transactional  
@Service  
public class UserService {  
  
    @Autowired  
    private UserDao userDao;  
  
    public void save() {  
  
        userDao.save();  
  
        int i = 1 / 0;  
        userDao.save();  
    }  
}
```

http://blog.csdn.net/hon\_3y

## 4.事务属性

其实我们在 XML 配置管理器类如何管理事务，就是在指定事务的属性！我们来看一下事务的属性有什么：

```
public @interface Transactional {  
    String value() default "";  
  
    Propagation propagation() default Propagation.REQUIRED;  
  
    Isolation isolation() default Isolation.DEFAULT;  
  
    int timeout() default -1; // 超时  
  
    boolean readOnly() default false; // 只读  
  
    Class<? extends Throwable>[] rollbackFor() default {};  
  
    String[] rollbackForClassName() default {};  
  
    Class<? extends Throwable>[] noRollbackFor() default {};  
}
```

http://blog.csdn.net/hon\_3y

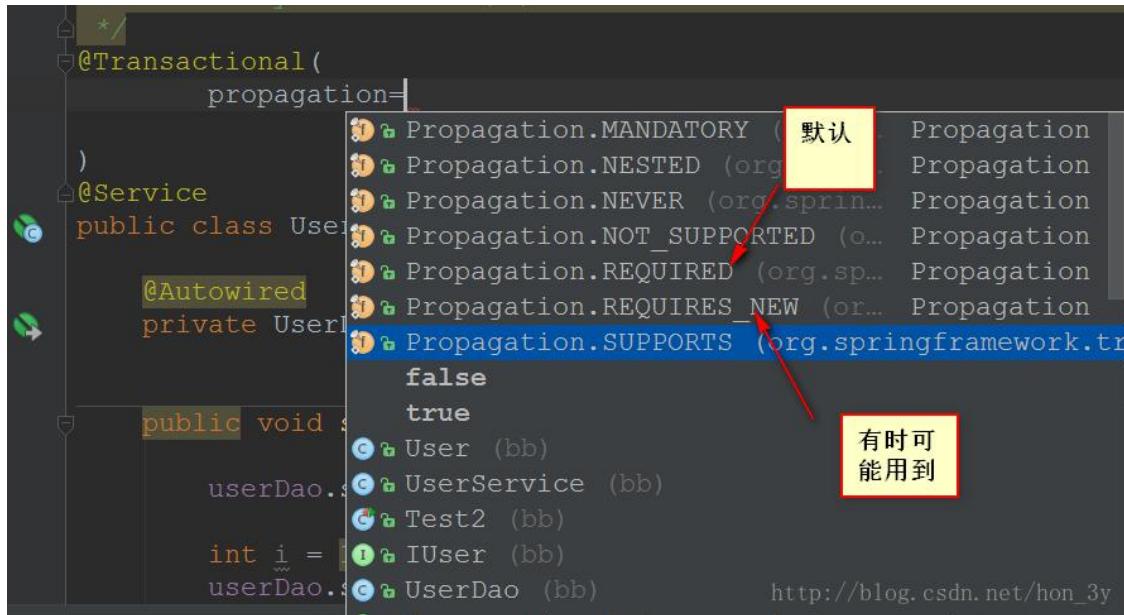
The code block shows annotations for the Transactional interface. A red box labeled '事务的传播行为' (Propagation Behavior) points to the 'propagation()' method. A yellow box labeled '超时' (Timeout) points to the 'timeout()' method. A yellow box labeled '只读' (Read-only) points to the 'readOnly()' method. A yellow box labeled '数据库的隔离方式' (Isolation Level) points to the 'isolation()' method. A red box labeled '遇到什么异常不回滚' (What exception leads to no rollback) points to the 'noRollbackFor()' method.

### 4.1 事务传播行为：

看了上面的事务属性，没有接触过的属性其实就这么一个：propagation = Propagation.REQUIRED 事务的传播行为。

事务传播行为的属性有以下这么多个，常用的就只有两个：

- Propagation.REQUIRED 【如果当前方法已经有事务了，加入当前方法事务】
- Propagation.REQUIRED\_NEW 【如果当前方法有事务了，当前方法事务会挂起。  
始终开启一个新的事务，直到新的事务执行完、当前方法的事务才开始】



## 4.2 当事务传播行为是 Propagation.REQUIRED

现在有一个日志类，它的事务传播行为是 Propagation.REQUIRED

```

Class Log{
    Propagation.REQUIRED
    insertLog();
}

```

现在，我要在保存之前记录日志

```

Propagation.REQUIRED
Void saveDept(){
    insertLog();
    saveDept();
}

```

`saveDept()`本身就存在着一个事务，当调用 `insertLog()`的时候，`insertLog()`的事务会加入到 `saveDept()`事务中

也就是说，`saveDept()`方法内始终是一个事务，如果在途中出现了异常，那么 `insertLog()`的数据是会被回滚的【因为在同一事务内】

```
Void saveDept(){
    insertLog(); // 加入当前事务
    .. 异常，会回滚
    saveDept();
}
```

### 4.3 当事务传播行为是 Propagation.REQUIRED\_NEW

现在有一个日志类，它的事务传播行为是 Propagation.REQUIRED\_NEW

```
Class Log{
    Propagation.REQUIRED
    insertLog();
}
```

现在，我要在保存之前记录日志

```
Propagation.REQUIRED
Void saveDept(){
    insertLog();
    saveDept();
}
```

当执行到 saveDept()中的 insertLog()方法时，insertLog()方法发现 saveDept()已经存在事务了，insertLog()会独自新开一个事务，直到事务关闭之后，再执行下面的方法

如果在中途抛出了异常，insertLog()是不会回滚的，因为它的事务是自己的，已经提交了

```
Void saveDept(){
    insertLog(); // 始终开启事务
    .. 异常，日志不会回滚
    saveDept();
}
```



如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜 Java3y 公众号有我的联系方式。更多原创技术文章可关注我的 GitHub：

<https://github.com/ZhongFuCheng3y/3y>

## Spring 事务原理

Spring 事务管理我相信大家都用得很多，但可能仅仅局限于一个@Transactional 注解或者在 XML 中配置事务相关的东西。不管怎么说，日常可能足够我们去用了。但作为程序员，无论是为了面试还是说更好把控自己写的代码，还是应该得多多了解一下 Spring 事务的一些细节。

这里我抛出几个问题，看大家能不能瞬间答得上：

- 如果嵌套调用含有事务的方法，在 Spring 事务管理中，这属于哪个知识点？
- 我们使用的框架可能是 Hibernate/JPA 或者是 Mybatis，都知道的底层是需要一个 session/connection 对象来帮我们执行操作的。要保证事务的完整性，我们需要多组数据库操作要使用同一个 session/connection 对象，而我们又知道 Spring IOC 所管理的对象默认都是单例的，这为啥我们在使用的时候不会引发线程安全问题呢？内部 Spring 到底干了什么？
- 人家所说的 BPP 又是啥东西？
- Spring 事务管理重要接口有哪几个？

### 一、阅读本文需要的基础知识

阅读这篇文章的同学我默认大家都对 Spring 事务相关知识有一定的了解了。(ps:如果不了解点解具体的文章去阅读再回到这里来哦)

我们都知道，Spring 事务是 Spring AOP 的最佳实践之一，所以说 [AOP 入门基础知识\(简单配置，使用\)](#) 是需要先知道的。如果想更加全面了解 AOP 可以看这篇文章：[AOP 重要知识点\(术语介绍、全面使用\)](#)。说到 AOP 就不能不说 [AOP 底层原理：动态](#)

代理设计模式。到这里，对 AOP 已经有一个基础的认识了。于是我们就可以使用 XML/注解方式来配置 Spring 事务管理。

在 IOC 学习中，可以知道的是 Spring 中 Bean 的生命周期(引出 BPP 对象)并且 IOC 所管理的对象默认都是单例的：单例设计模式，单例对象如果有‘状态’(有成员变量)，那么多线程访问这个单例对象，可能就造成线程不安全。那么何为线程安全？，解决线程安全有很多方式，但其中有一种：让每一个线程都拥有自己的一个变量：[ThreadLocal](#)

如果对我以上说的知识点不太了解的话，建议点击蓝字进去学习一番。

## 二、两个不靠谱直觉的例子

### 2.1 第一个例子

之前朋友问了我一个例子：

在 Service 层抛出 Exception，在 Controller 层捕获，那如果在 Service 中有异常，那会事务回滚吗？

// Service 方法

```
@Transactional  
public Employee addEmployee() throws Exception {  
  
    Employee employee = new Employee('3y', 23);  
    employeeRepository.save(employee);  
    // 假设这里出了 Exception  
    int i = 1 / 0;  
  
    return employee;  
}
```

// Controller 调用

```

@RequestMapping("/add")
public Employee addEmployee() {
    Employee employee = null;
    try {
        employee = employeeService.addEmployee();
    } catch (Exception e) {
        e.printStackTrace();
    }
    return employee;
}

```

我第一反应：不会回滚吧。

- 我当时是这样想的：因为 Service 层已经抛出了异常，由 Controller 捕获。那是否回滚应该由 Controller 的 catch 代码块中逻辑来决定，如果 catch 代码块没有回滚，那应该是不会回滚。

但朋友经过测试说，可以回滚阿。(pappapa 打脸)

```

@Transactional
public Employee addEmployee() throws Exception {
    Employee employee = new Employee(id: 3, name: "3y", age: 23);
    employeeRepository.save(employee);
    int i = 1 / 0;
    return employee;
}

EmployeeService > addEmployee()
发生了异常，并没有记录
写入到数据库中

JavaApplication
Console Endpoints
2019-01-28 19:27:02.404  INFO 6204 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet
aaa
Hibernate: select employee0_.id as id1_0_0_, employee0_.age as age2_0_0_, employee0_
Hibernate: select next_val as id_val from hibernate_sequence for update
Hibernate: update hibernate_sequence set next_val=? where next_val=?
java.lang.ArithmetricException: / by zero
    at com.springtest.service.EmployeeService.addEmployee(EmployeeService.java:23)
    at com.springtest.service.EmployeeService$$FastClassBySpringCGLIB$$5f58bd8c.invoke
    at org.springframework.cglib.proxy.MethodProxy.invoke(MethodProxy.java:204)
    at org.springframework.aop.framework.CglibAopProxy$CglibMethodInvocation.invoke
    at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(Reflect
    at org.springframework.transaction.interceptor.TransactionAspectSupport.invokeW
    at org.springframework.transaction.interceptor.TransactionInterceptor.invoke(Tr
    at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(Reflect
    at org.springframework.aop.framework.CglibAopProxy$DynamicAdvisedInterceptor.in
    at com.springtest.service.EmployeeService$$EnhancerBySpringCGLIB$$81d3af95.addE

```

看了一下文档，原来文档有说明：

By default checked exceptions do not result in the transactional interceptor marking the transaction for rollback and instances of RuntimeException and its subclasses do

结论：如果是编译时异常不会自动回滚，如果是运行时异常，那会自动回滚！

## 2.2 第二个例子

第二个例子来源于知乎@柳树文章，文末会给出相应的 URL

我们都知道，带有@Transactional注解所包围的方法就能被Spring事务管理起来，那如果我在当前类下使用一个没有事务的方法去调用一个有事务的方法，那我们这次调用会怎么样？是否会有事务呢？

用代码来描述一下：

```
// 没有事务的方法去调用有事务的方法
public Employee addEmployee2Controller() throws Exception {
    return this.addEmployee();
}

@Transactional
public Employee addEmployee() throws Exception {
    employeeRepository.deleteAll();
    Employee employee = new Employee('3y', 23);

    // 模拟异常
    int i = 1 / 0;

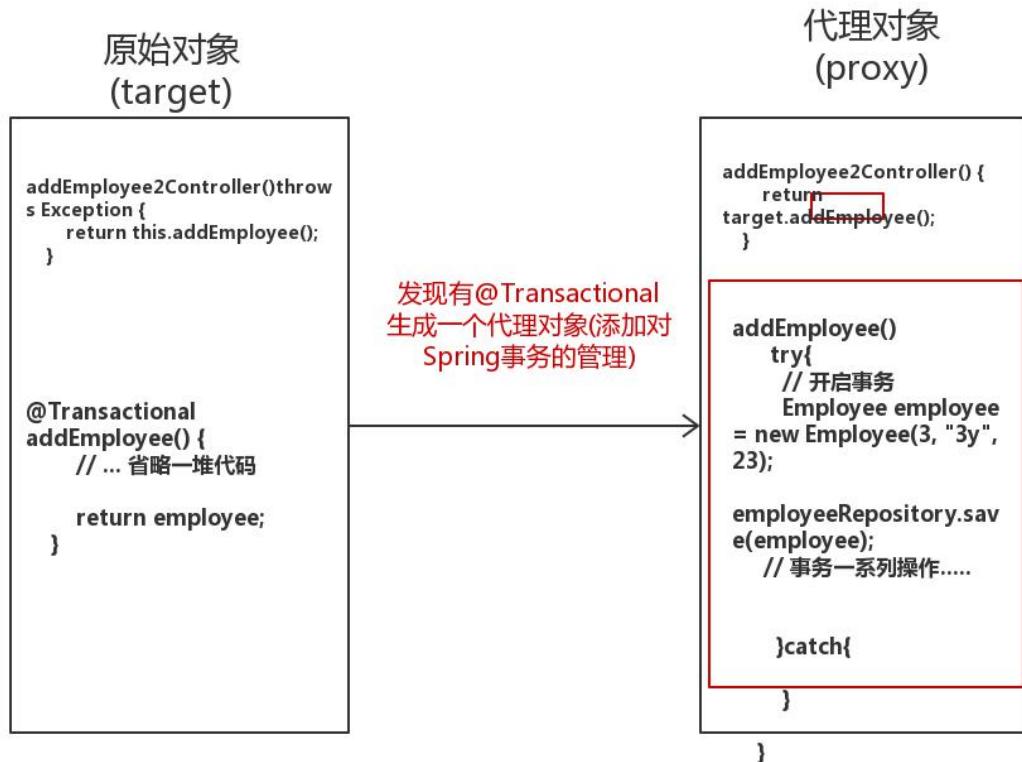
    return employee;
}
```

我第一直觉是：这跟Spring事务的传播机制有关吧。

其实这跟Spring事务的传播机制没有关系，下面我讲述一下：

- Spring 事务管理用的是 AOP，AOP 底层用的是动态代理。所以如果我们在类或者方法上标注注解@Transactional，那么会生成一个代理对象。

接下来我用图来说明一下：



显然地，我们拿到的是代理(Proxy)对象，调用 addEmployee2Controller()方法，而 addEmployee2Controller()方法的逻辑是 target.addEmployee()，调用回原始对象 (target)的 addEmployee()。所以这次的调用压根就没有事务存在，更谈不上说 Spring 事务传播机制了。

原有的数据：

	<b>id</b>	<b>age</b>	<b>name</b>
1	7	23	3y
2	6	23	3y
3	5	23	3y

测试结果：压根就没有事务的存在

The screenshot shows a Java application interface with a table and a console log window.

**Table Data:**

	<b>id</b>	<b>age</b>	<b>name</b>
1	7	23	3y
2	6	23	3y
3	5	23	3y

**Console Log:**

```

Run JavaApplication
Console Endpoints
2019-01-28 20:23:15.595  INFO 9756 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet      :
2019-01-28 20:23:15.849  INFO 9756 --- [nio-8080-exec-21] o.h.h.i.QueryTranslatorFactoryInitiator  :
Hibernate: select employee0_.id as id1_0_, employee0_.age as age2_0_, employee0_.name as name3_0_  fr
Hibernate: delete from employee where id=? 
Hibernate: delete from employee where id=? 
Hibernate: delete from employee where id=? 
java.lang.ArithmeticException: / by zero
        at com.springtest.service.EmployeeService.addEmployee(EmployeeService.java:32)
        at com.springtest.service.EmployeeService.addEmployee2Controller(EmployeeService.java:20)
        at com.springtest.service.EmployeeService$$FastClassBySpringCGLIB$$5f58bd8c.invoke(<generated>)
        at org.springframework.cglib.proxy.MethodProxy.invoke(MethodProxy.java:204)

```

A red arrow points from a text box containing the message "抛出了运行时异常，同样地我们的数据也没有了-----说明这次调用根本没有事务的管理！" to the log entry "java.lang.ArithmeticException: / by zero".

## 2.2.1 再延伸一下

从上面的测试我们可以发现：如果是在本类中没有事务的方法来调用标注注解 @Transactional 方法，最后的结论是没有事务的。那如果我将这个标注注解的方法移到别的 Service 对象上，有没有事务？

```

@Service
public class TestService {

```

```
    @Autowired
```

```
private EmployeeRepository employeeRepository;

@Transactional
public Employee addEmployee() throws Exception {

    employeeRepository.deleteAll();

    Employee employee = new Employee("3y", 23);

    // 模拟异常
    int i = 1 / 0;

    return employee;
}

@Service
public class EmployeeService {

    @Autowired
    private TestService testService;
    // 没有事务的方法去调用别的类有事务的方法

    public Employee addEmployee2Controller() throws Exception {
        return testService.addEmployee();
    }
}
```

测试结果：

	<b>id</b>	<b>age</b>	<b>name</b>
1	1	3	35

抛出了运行时异常，但我们的数据还是存在的

```

Run JavaByApplication
Console Endpoints
2019-01-28 20:52:02.994 INFO 4440 --- [nio-8080-exec-2] o.a.c.c.C.[Tomcat].[localhost]...
2019-01-28 20:52:02.994 INFO 4440 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet...
2019-01-28 20:52:03.081 INFO 4440 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet...
2019-01-28 20:52:03.273 INFO 4440 --- [nio-8080-exec-2] o.h.h.i.QueryTranslatorFactory...
Hibernate: select employee0_.id as id1_0_, employee0_.age as age2_0_, employee0_.name as...
java.lang.ArithmetricException: / by zero
    at com.springtest.service.TestService.addEmployee(TestService.java:25)
    at com.springtest.service.TestService$$FastClassBySpringCGLIB$$8d3b31a8.invoke(<gen...
    at org.springframework.cglib.proxy.MethodProxy.invoke(MethodProxy.java:204)
    at org.springframework.aop.framework.CglibAopProxy$CglibMethodInvocation.invokeJoinp...

```

因为我们用的是代理对象(Proxy)去调用 addEmployee()方法，那就当然有事务了。

看完这两个例子，有没有觉得 3y 的直觉是真的水！

### 三、Spring 事务传播机制

如果嵌套调用含有事务的方法，在 Spring 事务管理中，这属于哪个知识点？

在当前含有事务方法内部调用其他的方法(无论该方法是否含有事务)，这就属于 Spring 事务传播机制的知识点范畴了。

Spring 事务基于 Spring AOP，Spring AOP 底层用的动态代理，动态代理有两种方式：

- 基于接口代理(JDK 代理)
  - 基于接口代理，凡是类的方法非 **public** 修饰，或者用了 **static** 关键字修饰，那这些方法都不能被 Spring AOP 增强
- 基于 CGLib 代理(子类代理)

- 基于子类代理，凡是类的方法使用了 `private`、`static`、`final` 修饰，那这些方法都不能被 Spring AOP 增强

至于为啥以上的情况不能增强，用你们的脑瓜子想一下就知道了。

值得说明的是：那些不能被 Spring AOP 增强的方法并不是不能在事务环境下工作了。只要它们被外层的事务方法调用了，由于 Spring 事务管理的传播级别，内部方法也可以工作在外部方法所启动的事务上下文中。

至于 Spring 事务传播机制的几个级别，我在这里就不贴出来了。这里只是再次解释“啥情况才是属于 Spring 事务传播机制的范畴”。





如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜 **Java3y** 公众号有我的联系方式。更多原创技术文章可关注我的 GitHub：

<https://github.com/ZhongFuCheng3y/3y>

## 四、多线程问题

我们使用的框架可能是 Hibernate/JPA 或者是 Mybatis，都知道的底层是需要一个 session/connection 对象来帮我们执行操作的。要保证事务的完整性，我们需要**多组数据库操作要使用同一个 session/connection 对象**，而我们又知道 Spring IOC 所管理的对象默认都是**单例的**，这为啥我们在使用的时候不会引发线程安全问题呢？内部 Spring 到底干了什么？

回想一下当年我们学 Mybatis 的时候，是怎么编写 **Session 工具类**？

```

public class MybatisUtil {
    private static ThreadLocal<SqlSession> threadLocal = new ThreadLocal<SqlSession>();
    private static SqlSessionFactory sqlSessionFactory;
    /**
     * 加载位于src/mybatis.xml配置文件
     */
    static{
        try {
            Reader reader = Resources.getResourceAsReader("mybatis.xml");
            sqlSessionFactory = new SqlSessionFactoryBuilder().build(reader);
        } catch (IOException e) {
            e.printStackTrace();
            throw new RuntimeException(e);
        }
    }
    /**
     * 禁止外界通过new方法创建
     */
    private MybatisUtil(){}
    /**
     * 获取SqlSession
     */
    public static SqlSession getSqlSession(){
        //从当前线程中获取sqlsession对象
        SqlSession sqlSession = threadLocal.get();
        //如果SqlSession对象为空
        if(sqlSession == null){
            //在sqlSessionFactory非空的情况下，获取sqlSession对象
            sqlSession = sqlSessionFactory.openSession();
            //将SqlSession对象与当前线程绑定在一起
            threadLocal.set(sqlSession);
        }
        //返回sqlsession对象
        return sqlSession;
    }
}

```

没错，用的就是 ThreadLocal，同样地，Spring 也是用的 ThreadLocal。

以下内容来源《精通 Spring4.x》

我们知道在一般情况下，只有无状态的 Bean 才可以在多线程环境下共享，在 Spring 中，绝大部分 Bean 都可以声明为 singleton 作用域。就是因为 Spring 对一些 Bean（如 RequestContextHolder、**TransactionSynchronizationManager**、LocaleContextHolder 等）中非线程安全状态的“状态性对象”采用 ThreadLocal 封装，让它们也成为线程安全的“状态性对象”，因此，有状态的 Bean 就能够以 singleton 的方式在多线程中工作。

我们可以试着点一下进去 TransactionSynchronizationManager 中看一下：

```
public abstract class TransactionSynchronizationManager {  
    private static final Log logger = LogFactory.getLog(TransactionSynchronizationManager.class);  
  
    private static final ThreadLocal<Map<Object, Object>> resources =  
        new NamedThreadLocal<>("Transactional resources");  
  
    private static final ThreadLocal<Set<TransactionSynchronization>> synchronizations =  
        new NamedThreadLocal<>("Transaction synchronizations");  
  
    private static final ThreadLocal<String> currentTransactionName =  
        new NamedThreadLocal<>("Current transaction name");  
  
    private static final ThreadLocal<Boolean> currentTransactionReadOnly =  
        new NamedThreadLocal<>("Current transaction read-only status");  
  
    private static final ThreadLocal<Integer> currentTransactionIsolationLevel =  
        new NamedThreadLocal<>("Current transaction isolation level");  
  
    private static final ThreadLocal<Boolean> actualTransactionActive =  
        new NamedThreadLocal<>("Actual transaction active");  
}
```

## 五、啥是 BPP？

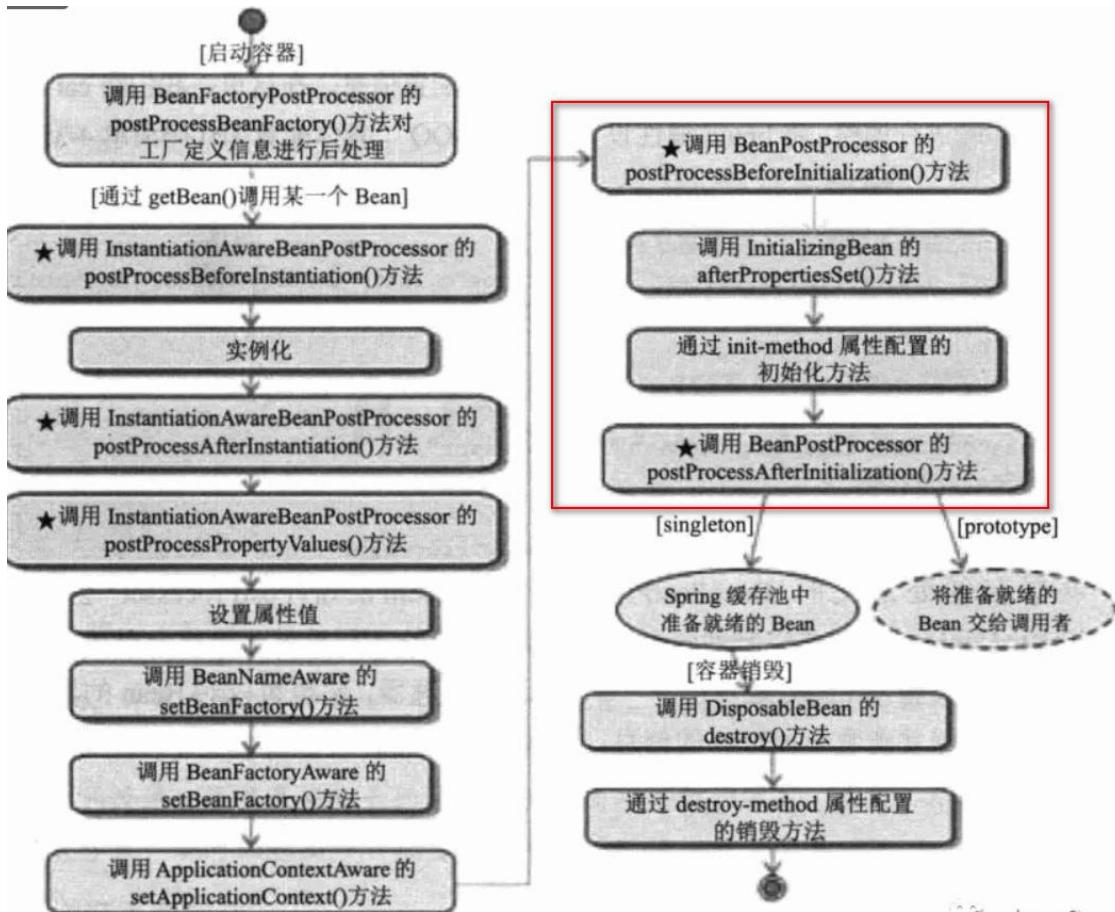
BBP 的全称叫做：BeanPostProcessor，一般我们俗称对象后处理器

- 简单来说，通过 BeanPostProcessor 可以对我们的对象进行“加工处理”。

Spring 管理 Bean(或者说 Bean 的生命周期)也是一个常考的知识点，我在秋招也重新整理了一下步骤，因为比较重要，所以还是在这里贴一下吧：

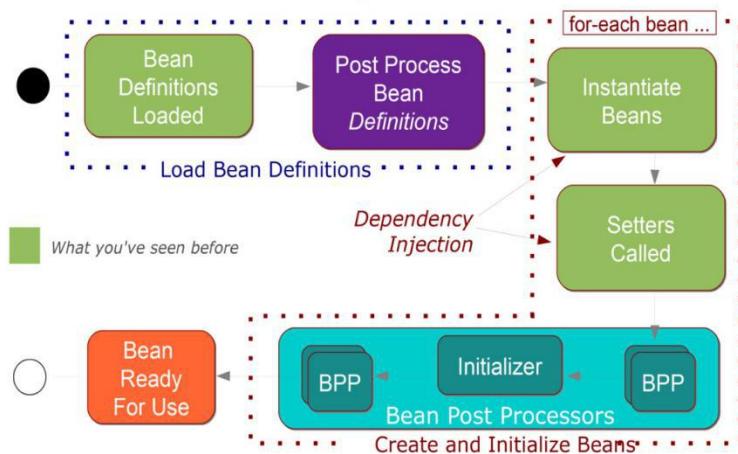
1. ResouceLoader 加载配置信息
2. BeanDefintionReader 解析配置信息，生成一个一个的 BeanDefintion
3. BeanDefintion 由 BeanDefintionRegistry 管理起来
4. BeanFactoryPostProcessor 对配置信息进行加工(也就是处理配置的信息，一般通过 PropertyPlaceholderConfigurer 来实现)
5. 实例化 Bean
6. 如果该 Bean 配置/实现了 InstantiationAwareBean，则调用对应的方法
7. 使用 BeanWarpper 来完成对象之间的属性配置(依赖)
8. 如果该 Bean 配置/实现了 Aware 接口，则调用对应的方法
9. 如果该 Bean 配置了 BeanPostProcessor 的 before 方法，则调用

10. 如果该 Bean 配置了 init-method 或者实现 InstantiationBean，则调用对应的方法
11. 如果该 Bean 配置了 BeanPostProcessor 的 after 方法，则调用
12. 将对象放入到 HashMap 中
13. 最后如果配置了 destroy 或者 DisposableBean 的方法，则执行销毁操作



其中也有关于 BPP 图片：

## Bean Initialization Steps



### 5.1 为什么特意讲 BPP ?

Spring AOP 编程底层通过的是动态代理技术，在调用的时候肯定用的是代理对象。那么 Spring 是怎么做的呢？

我只需要写一个 BPP，在 postProcessBeforeInitialization 或者 postProcessAfterInitialization 方法中，对对象进行判断，看他需不需要织入切面逻辑，如果需要，那我就根据这个对象，生成一个代理对象，然后返回这个代理对象，那么最终注入容器的，自然就是代理对象了。

Spring 提供了 BeanPostProcessor，就是让我们可以对有需要的对象进行“加工处理”啊！

## 六、认识 Spring 事务几个重要的接口

Spring 事务可以分为两种：

- 编程式事务(通过代码的方式来实现事务)
- 声明式事务(通过配置的方式来实现事务)

编程式事务在 Spring 实现相对简单一些，而声明式事务因为封装了大量的东西(一般我们使用简单，里头都非常复杂)，所以声明式事务实现要难得多。

在编程式事务中有以下几个重要的接口：

- TransactionDefinition：定义了 Spring 兼容的事务属性(比如事务隔离级别、事务传播、事务超时、是否只读状态)
- TransactionStatus：代表了事务的具体运行状态(获取事务运行状态的信息，也可以通过该接口间接回滚事务等操作)
- PlatformTransactionManager：事务管理器接口(定义了一组行为，具体实现交由不同的持久化框架来完成---类比 JDBC)

The screenshot shows an IDE interface with several tabs at the top: Structure, Project, AnnotationAwareAspectJAutoProxyCreator.java, TransactionDefinition.java, and PlatformTransactionManager.java. The PlatformTransactionManager.java tab is active, displaying the interface definition:

```

38
39
40 * 定义一组行为
41 * @author Juergen Hoeller
42 * @since 16.05.2003
43 * @see org.springframework.transaction.support.
44 * @see org.springframework.transaction.intercep
45 * @see org.springframework.transaction.intercep
46 */
public interface PlatformTransactionManager {

```

A tooltip "常见的事务管理器类实现 (相信你在XML中也配置过)" is shown over the list of implementations below:

- AbstractPlatformTransactionManager (org.springframework.transaction.support)
- CallbackPreferringPlatformTransactionManager (org.springframework.transaction.sup)
- CciLocalTransactionManager (org.springframework.jca.cci.connection)
- ChainedTransactionManager (org.springfr)
- DataSourceTransactionManager (org.spring (常见的事务管理器类实现 (相信你在XML中也配置过) ) Maven: org.springfr
- HibernateTransactionManager (org.springframewo (常见的事务管理器类实现 (相信你在XML中也配置过) ) Maven: org.springframewo
- JpaTransactionManager (org.springframewo.jpa)
- JtaTransactionManager (org.springframewo.transaction.jta)
- ResourceTransactionManager (org.springframewo.transaction.support)
- WebLogicJtaTransactionManager (org.springframewo.transaction.jta)
- WebSphereUowTransactionManager (org.springframewo.transaction.jta)

在声明式事务中，除了 TransactionStatus 和 PlatformTransactionManager 接口，还有几个重要的接口：

- TransactionProxyFactoryBean：生成代理对象
- TransactionInterceptor：实现对象的拦截
- TransactionAttribute：事务配置的数据



如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜 **Java3y** 公众号有我的联系方式。更多原创技术文章可关注我的 GitHub：

<https://github.com/ZhongFuCheng3y/3y>

# Spring 事务的一个线程安全问题

只有光头才能变强。

文本已收录至我的 GitHub 仓库，欢迎 Star：

<https://github.com/ZhongFuCheng3y/3y>

大年初二，朋友问了我一个技术的问题(朋友实在是好学，佩服！)

该问题来源知乎(synchronized 锁问题)：

- <https://www.zhihu.com/question/277812143>

开启 10000 个线程，每个线程给员工表的 money 字段【初始值是 0】加 1，没有使用悲观锁和乐观锁，但是在业务层方法上加了 synchronized 关键字，问题是代码执行完毕后数据库中的 money 字段不是 10000，而是小于 10000 问题出在哪里？

Service 层代码：

```
18  
19- @Transactional  
20  public synchronized void increaseMoneyWithPessimisticLock(Integer id) {  
21      Employee employee = synDao.findByIdWithPessimisticLock(id);  
22      logger.debug(employee.getMoney() + "-----");  
23      final Integer oldMoney = employee.getMoney();  
24      logger.debug("oldMoney: {}", oldMoney);  
25      employee.setMoney(oldMoney + 1);  
26      synDao.updateEmployee(employee);  
27  }  
28 }  
29
```

SQL 代码(没有加悲观/乐观锁)：

```
<mapper namespace="com.zisi.activitiweb.syn.mapper.EmployeeMapper">
    <select id="findByIdWithPessimisticLock" resultType="com.zisi.activitiweb.syn.entity.Employee">
        SELECT * FROM employee WHERE id = #{id} <!-- FOR UPDATE -->
    </select>

    <update id="updateEmployee">
        update
            employee
        set
            money = #{money},
            version = #{version}
        where id = #{id}
    </update>
```

用 1000 个线程跑代码：

```
<mapper namespace="com.zisi.activitiweb.syn.mapper.EmployeeMapper">
    <select id="findByIdWithPessimisticLock" resultType="com.zisi.activitiweb.syn.entity.Employee">
        SELECT * FROM employee WHERE id = #{id} <!-- FOR UPDATE -->
    </select>

    <update id="updateEmployee">
        update
            employee
        set
            money = #{money},
            version = #{version}
        where id = #{id}
    </update>
```

简单来说：多线程跑一个使用 **synchronized** 关键字修饰的方法，方法内操作的是数据库，按正常逻辑应该最终的值是 1000，但经过多次测试，结果是低于 1000。这是为什么呢？

## 一、我的思考

既然测试出来的结果是低于 1000，那说明这段代码不是线程安全的。不是线程安全的，那问题出现在哪呢？众所周知，`synchronized` 方法能够保证所修饰的代码块、方法保证有序性、原子性、可见性。

讲道理，以上的代码跑起来，问题中 Service 层的 `increaseMoney()` 是有序的、原子的、可见的，所以断定跟 `synchronized` 应该没关系。

(参考我之前写过的 `synchronize` 锁笔记：[Java 锁机制了解一下](#))

既然 Java 层面上找不到原因，那分析一下数据库层面的吧(因为方法内操作的是数据库)。在 `increaseMoney()` 方法前加了 `@Transactional` 注解，说明这个方法是带有事务的。事务能保证同组的 SQL 要么同时成功，要么同时失败。讲道理，如果没有报错的话，应该每个线程都对 `money` 值进行+1。从理论上来说，结果应该是 1000 的才对。

(参考我之前写过的 Spring 事务：[一文带你看懂 Spring 事务！](#))

根据上面的分析，我怀疑是提问者没测试好(hhhh，逃)，于是我也跑去测试了一下，发现是以提问者的方式来使用是真的有问题。

首先贴一下我的测试代码：

```
@RestController
public class EmployeeController {

    @Autowired
    private EmployeeService employeeService;

    @RequestMapping('/add')
    public void addEmployee() {
        for (int i = 0; i < 1000; i++) {
            new Thread(() -> employeeService.addEmployee()).start();
        }
    }
}
```

```
}

@Service
public class EmployeeService {

    @Autowired
    private EmployeeRepository employeeRepository;

    @Transactional
    public synchronized void addEmployee() {

        // 查出 ID 为 8 的记录，然后每次将年龄增加一
        Employee employee = employeeRepository.getOne(8);
        System.out.println(employee);
        Integer age = employee.getAge();
        employee.setAge(age + 1);

        employeeRepository.save(employee);
    }
}
```

简单地打印了每次拿到的 employee 值，并且拿到了 SQL 执行的顺序，如下(贴出小部分)：

```

2019-02-08 20:40:58.944 INFO 11976 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring FrameworkServlet 'dispatcherServlet'
2019-02-08 20:40:58.944 INFO 11976 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : FrameworkServlet 'dispatcherServlet' initialized in '0.001' ms
Hibernate: select employee0_.id as id1_0_0_, employee0_.age as age2_0_0_, employee0_.name as name3_0_0_ from employee employee0_
Employee{id=8, name='3y', age=1279}
Hibernate: select employee0_.id as id1_0_0_, employee0_.age as age2_0_0_, employee0_.name as name3_0_0_ from employee employee0_
Employee{id=8, name='3y', age=1279}
Hibernate: select employee0_.id as id1_0_0_, employee0_.age as age2_0_0_, employee0_.name as name3_0_0_ from employee employee0_
Employee{id=8, name='3y', age=1279}
Hibernate: select employee0_.id as id1_0_0_, employee0_.age as age2_0_0_, employee0_.name as name3_0_0_ from employee employee0_
Employee{id=8, name='3y', age=1279}
Hibernate: select employee0_.id as id1_0_0_, employee0_.age as age2_0_0_, employee0_.name as name3_0_0_ from employee employee0_
Employee{id=8, name='3y', age=1279}
Hibernate: update employee set age=?, name=? where id=?
Hibernate: select employee0_.id as id1_0_0_, employee0_.age as age2_0_0_, employee0_.name as name3_0_0_ from employee employee0_
Employee{id=8, name='3y', age=1280}
Hibernate: update employee set age=?, name=? where id=?

```

从打印的情况我们可以得出：多线程情况下并没有串行执行 addEmployee()方法。这就导致对同一个值做重复的修改，所以最终的数值比 1000 要少。

## 二、图解出现的原因

发现并不是同步执行的，于是我就怀疑 synchronized 关键字和 Spring 肯定有点冲突。于是根据这两个关键字搜了一下，找到了问题所在。

我们知道 Spring 事务的底层是 Spring AOP，而 Spring AOP 的底层是动态代理技术。跟大家一起回顾一下动态代理：

```

public static void main(String[] args) {

    // 目标对象
    Object target;

    Proxy.newProxyInstance(ClassLoader.getSystemClassLoader(), Main.class, new InvocationHandler() {
        @Override
        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
            // 但凡带有@Transactional 注解的方法都会被拦截
            // 1... 开启事务
        }
    });
}

```

```
method.invoke(target);
```

// 2... 提交事务

```
return null;
```

```
}
```

```
});
```

```
}
```

(详细请参考我之前写过的动态代理：[给女朋友讲解什么是代理模式](#))

实际上 Spring 做的处理跟以上的思路是一样的，我们可以看一下 TransactionAspectSupport 类中 invokeWithinTransaction()：

```
// If the transaction attribute is null, the method is non-transactional.
TransactionAttributeSource tas = getTransactionAttributeSource();
final TransactionAttribute txAttr = (tas != null ? tas.getTransactionAttribute(method, targetClass) : null);
final PlatformTransactionManager tm = determineTransactionManager(txAttr);
final String joinpointIdentification = methodIdentification(method, targetClass, txAttr);

if (txAttr == null || !(tm instanceof CallbackPreferringPlatformTransactionManager)) {
    // Standard transaction demarcation with getTransaction and commit/rollback calls.
    TransactionInfo txInfo = createTransactionIfNecessary(tm, txAttr, joinpointIdentification);
    Object retVal = null;
    try {
        // This is an around advice: Invoke the next interceptor in the chain.
        // This will normally result in a target object being invoked.
        retVal = invocation.proceedWithInvocation(); ↑ 调用对象的方法 ②
    } catch (Throwable ex) {
        // target invocation exception
        completeTransactionAfterThrowing(txInfo, ex); ↑ 出现异常就回滚 ③
        throw ex;
    }
    finally {
        cleanupTransactionInfo(txInfo);
    }
    commitTransactionAfterReturning(txInfo); ↑ 提交事务 ④
    return retVal;
}
else {
```

调用方法前开启事务，调用方法后提交事务

## 1、方法执行前，开启事务

## 2、调用原方法

```
@Service
public class EmployeeService {
    @Autowired
    private EmployeeRepository employeeRepository;
    @Transactional
    public synchronized void addEmployee() {
        // 查出ID为8的记录，然后每次将年龄增加一
        Employee employee = employeeRepository.getOne(8);
        System.out.println(Thread.currentThread().getName() + employee);
        Integer age = employee.getAge();
        employee.setAge(age + 1);
        employeeRepository.save(employee);
    }
}
```

## 3、方法执行后，提交事务

在多线程环境下，就可能会出现：方法执行完了(synchronized 代码块执行完了)，事务还没提交，别的线程可以进入被 synchronized 修饰的方法，再读取的时候，读到的是还没提交事务的数据，这个数据不是最新的，所以就出现了这个问题。

## 1、方法执行前，开启事务

## 2、调用原方法

```
@Service
public class EmployeeService {
    @Autowired
    private EmployeeRepository employeeRepository;
    @Transactional
    public synchronized void addEmployee() {
        // 查出ID为8的记录，然后每次将年龄增加一
        Employee employee = employeeRepository.getOne(8);
        System.out.println(Thread.currentThread().getName() + employee);
        Integer age = employee.getAge();
        employee.setAge(age + 1);
        employeeRepository.save(employee);
    }
}
```

事务未提交，  
别的线程可以  
执行该方法(读  
取旧数据)

## 3、方法执行后(synchronized锁释放，别的线程再执行方法，读到旧数据)

### 三、解决问题

从上面我们可以发现，问题所在是因为`@Transactional`注解和`synchronized`一起使用了，加锁的范围没有包括到整个事务。所以我们可以这样做：

新建一个名叫`SynchronizedService`类，让其去调用`addEmployee()`方法，整个代码如下：

```
@RestController
public class EmployeeController {

    @Autowired
    private SynchronizedService synchronizedService;

    @RequestMapping("/add")
    public void addEmployee() {
        for (int i = 0; i < 1000; i++) {
            new Thread(() -> synchronizedService.synchronizedAddEmployee()).start();
        }
    }

    // 新建的 Service 类
    @Service
    public class SynchronizedService {

        @Autowired
        private EmployeeService employeeService;

        // 同步
        public synchronized void synchronizedAddEmployee() {
            employeeService.addEmployee();

        }
    }

    @Service
```

```

public class EmployeeService {

    @Autowired
    private EmployeeRepository employeeRepository;

    @Transactional
    public void addEmployee() {

        // 查出 ID 为 8 的记录，然后每次将年龄增加一
        Employee employee = employeeRepository.getOne(8);
        System.out.println(Thread.currentThread().getName() + employee);
        Integer age = employee.getAge();
        employee.setAge(age + 1);

        employeeRepository.save(employee);
    }
}

```

我们将 synchronized 锁的范围包含到整个 Spring 事务上，这就不会出现线程安全的问题了。在测试的时候，我们可以发现 1000 个线程跑起来比之前要慢得多，当然我们的数据是正确的：

	<b>id</b>	<b>age</b>	<b>name</b>
1	8	1000	3y

## 最后

可以发现的是，虽然说 Spring 事务用起来我们是非常方便的，但如果不了解一些 Spring 事务的细节，很多时候出现 Bug 了就百思不得其解。还是得继续加油努力呀~~~





如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜 Java3y 公众号有我的联系方式。更多原创技术文章可关注我的 GitHub：

<https://github.com/ZhongFuCheng3y/3y>

## IOC 再回顾和面试题

本来想的是刷完《Spring 实战 (第 4 版)》和《精通 Spring4.x 企业应用开发实战》的 IOC 章节后来重新编写一篇 IOC 的文章的，看了一下之前已经写过的入门系列 [Spring 入门这一篇就够了](#) 和 [Spring【依赖注入】就是这么简单](#)。最主要的知识点都已经讲过了，所以感觉就没必要重新来编写这些知识点了...

这篇文章主要是补充和强化一些比较重要的知识点，并会把上面的两本书关于 IOC 的知识点整理出来。

那么接下来就开始吧，如果有错的地方希望能多多包涵，并不吝在评论区指正！

## 一、Spring IOC 全面认知

结合《Spring 实战 (第 4 版)》和《精通 Spring4.x 企业应用开发实战》两本书的 IOC 章节将其知识点整理起来~

### 1.1 IOC 和 DI 概述

在《精通 Spring4.x 企业应用开发实战》中对 IOC 的定义是这样的：

IoC(Inversion of Control)控制反转，包含了两个方面：一、控制。二、反转

我们可以简单认为：

- 控制指的是：当前对象对内部成员的控制权。
- 反转指的是：这种控制权不由当前对象管理了，由其他(类,第三方容器)来管理。

IOC 不够开门见山，于是 Martin Fowler 提出了 DI(dependency injection)来替代 IoC，即让调用类对某一接口实现类的依赖关系由第三方(容器或协作类)注入，以移除调用类对某一接口实现类的依赖。

在《Spring 实战 (第 4 版)》中并没有提及到 IOC，而是直接来说 DI 的：

通过 DI，对象的依赖关系将由系统中负责协调各对象的第三方组件在创建对象的时候进行设定，对象无需自行创建或管理它们的依赖关系，依赖关系将被自动注入到需要它们的对象当中去

从书上我们也可以发现：IoC 和 DI 的定义(区别)并不是如此容易就可以说得清楚的了。这里我就简单摘抄一下：

- IoC(思想，设计模式)主要的实现方式有两种：依赖查找，依赖注入。
- 依赖注入是一种更可取的方式(实现的方式)

对我们而言，其实也没必要分得那么清，混合一谈也不影响我们的理解...

再通过昨天写过的[工厂模式理解了没有？](#)，我们现在就可以很清楚的发现，其实所谓的 IOC 容器就是一个大工厂【第三方容器】(Spring 实现的功能很强大！比我们自己手写的工厂要好很多)。

使用 IOC 的好处([知乎@Intopass 的回答](#))：

1. 不用自己组装，拿来就用。
2. 享受单例的好处，效率高，不浪费空间。
3. 便于单元测试，方便切换 mock 组件。
4. 便于进行 AOP 操作，对于使用者是透明的。
5. 统一配置，便于修改。

参考资料：

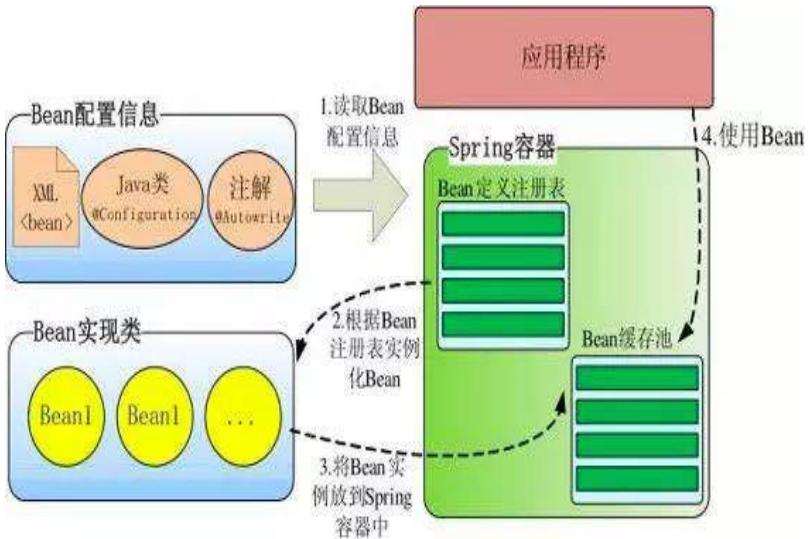
- <https://www.zhihu.com/question/23277575--Spring IoC 有什么好处呢？>

## 1.2 IOC 容器的原理

从上面就已经说了：IOC 容器其实就是一个大工厂，它用来管理我们所有的对象以及依赖关系。

- 原理就是通过 Java 的反射技术来实现的！通过反射我们可以获取类的所有信息(成员变量、类名等等等)！
- 再通过配置文件(xml)或者注解来描述类与类之间的关系
- 我们就可以通过这些配置信息和反射技术来构建出对应的对象和依赖关系了！

上面描述的技术只要学过点 Java 的都能说出来，这一下子可能就会被面试官问倒了，我们简单来看看实际 Spring IOC 容器是怎么实现对象的创建和依赖的：



1. 根据 Bean 配置信息在容器内部创建 Bean 定义注册表
2. 根据注册表加载、实例化 bean、建立 Bean 与 Bean 之间的依赖关系
3. 将这些准备就绪的 Bean 放到 Map 缓存池中，等待应用程序调用

Spring 容器(Bean 工厂)可简单分成两种：

- BeanFactory
  - 这是最基础、面向 Spring 的
- ApplicationContext
  - 这是在 BeanFactory 基础之上，面向使用 Spring 框架的开发者。提供了一系列的功能！

几乎所有的应用场合都是使用 ApplicationContext !

BeanFactory 的继承体系：

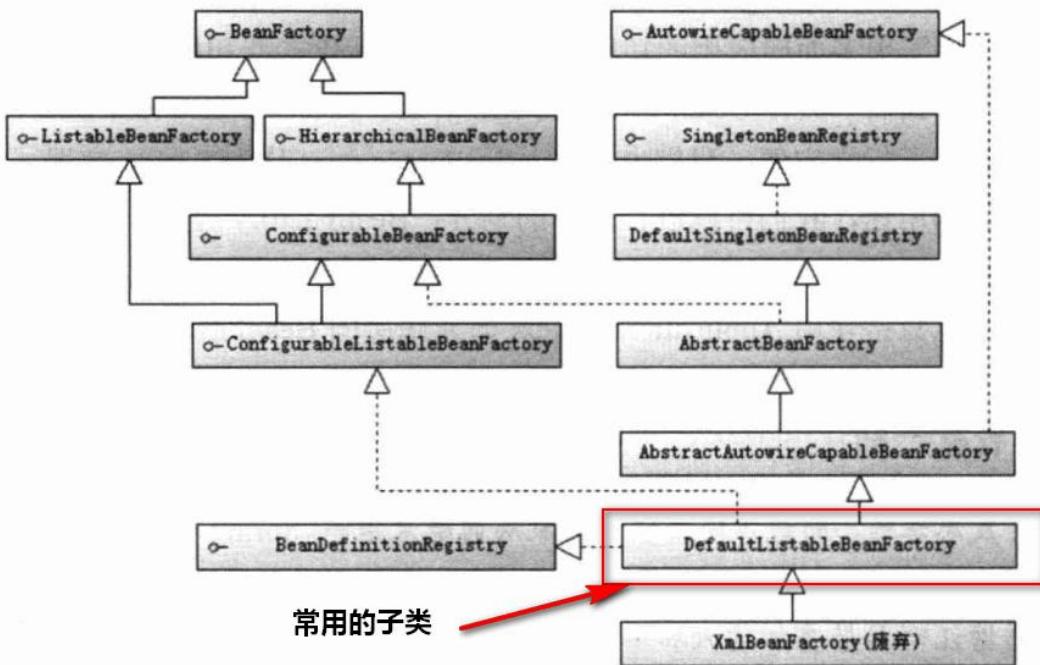
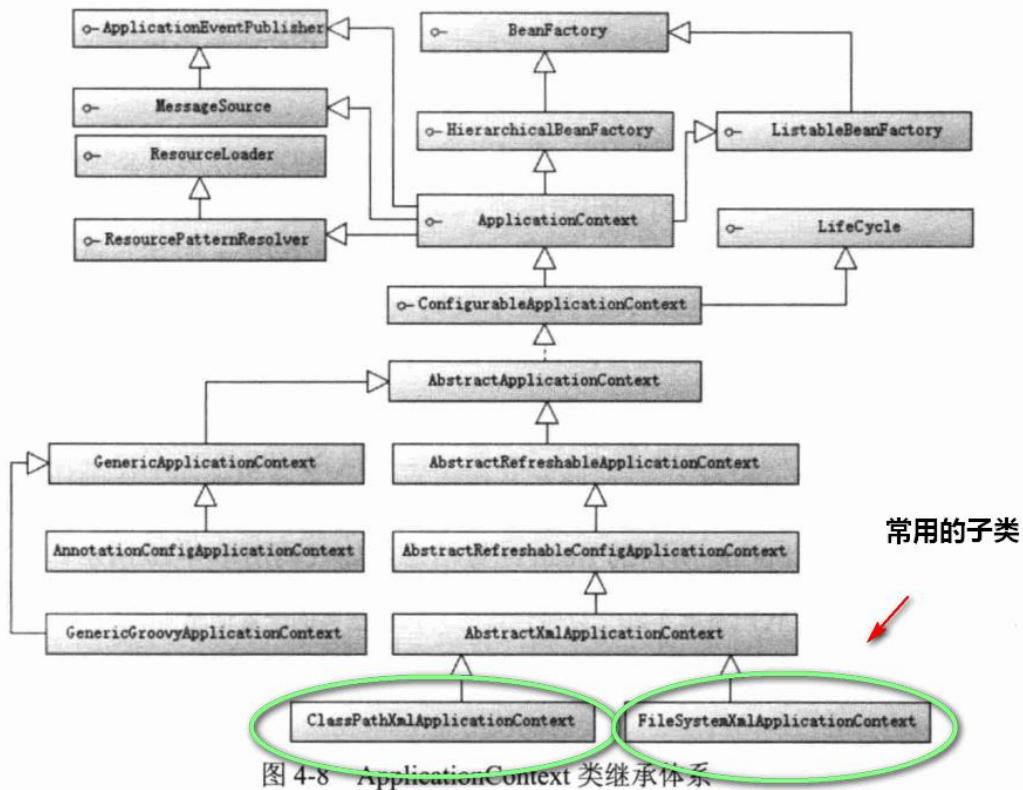


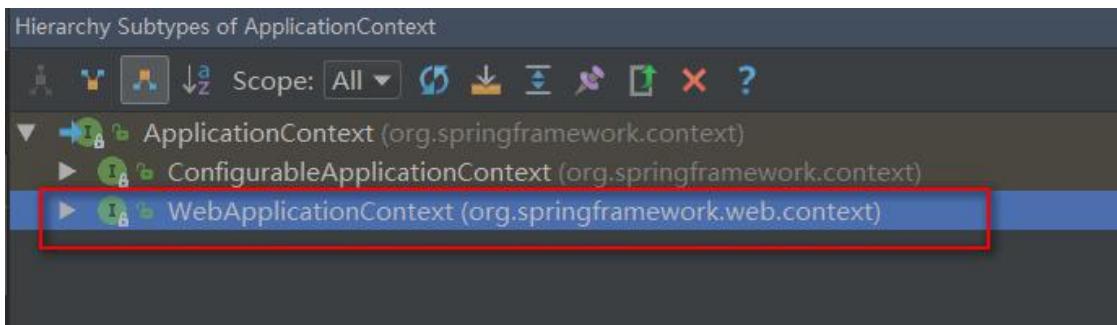
图 4-7 BeanFactory 类继承体系

ApplicationContext 的继承体系：



其中在 ApplicationContext 子类中又有一个比较重要的：WebApplicationContext

专门为 Web 应用准备的



Web 应用与 Spring 融合：

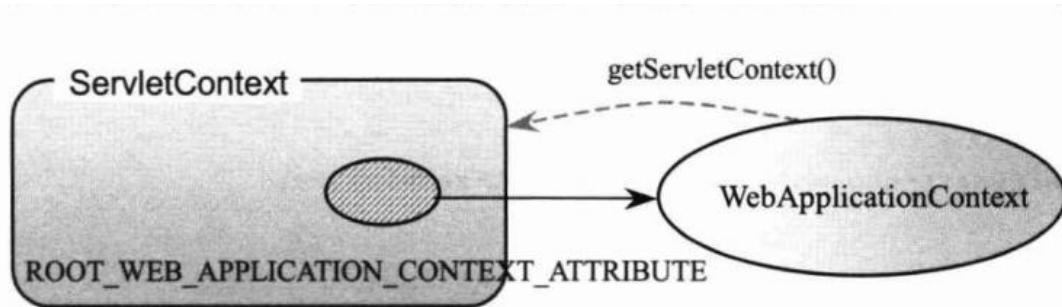


图 4-10 Spring 和 Web 应用的上下文融合

我们看看 BeanFactory 的生命周期：

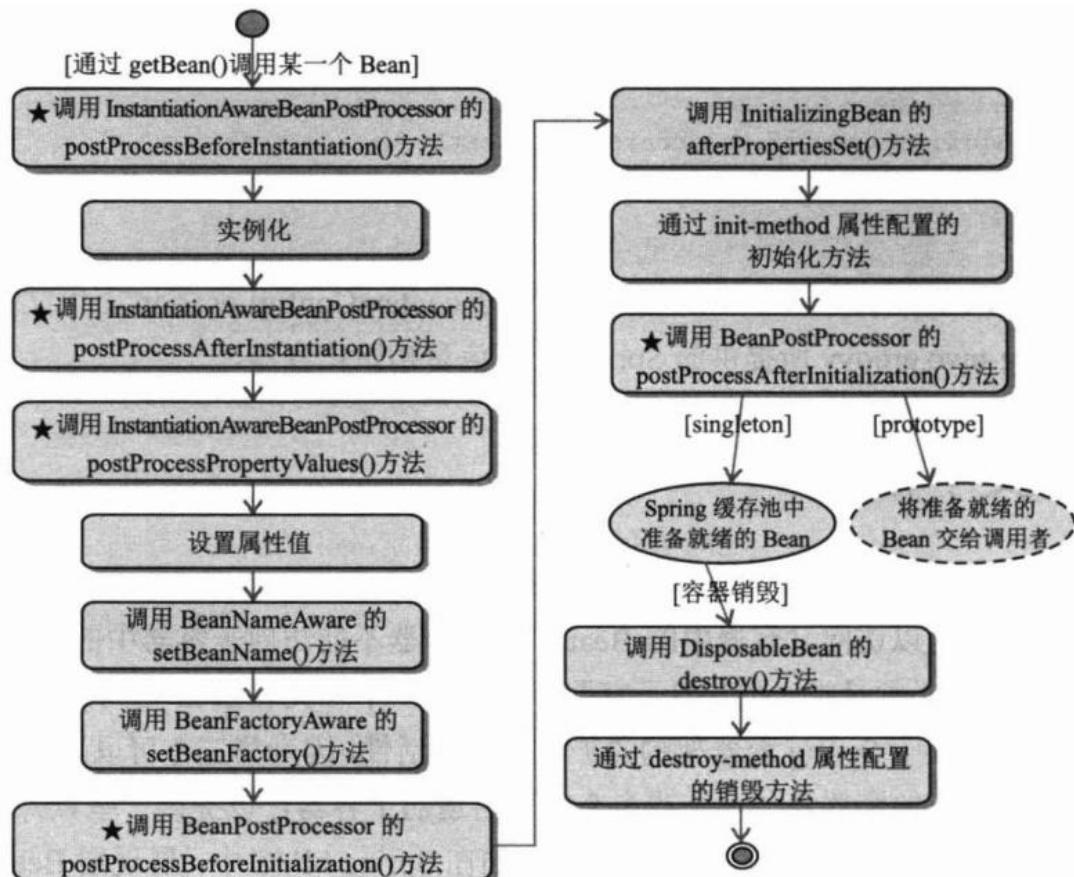


图 4-11 BeanFactory 中 Bean 的生命周期

接下来我们再看看 ApplicationContext 的生命周期：

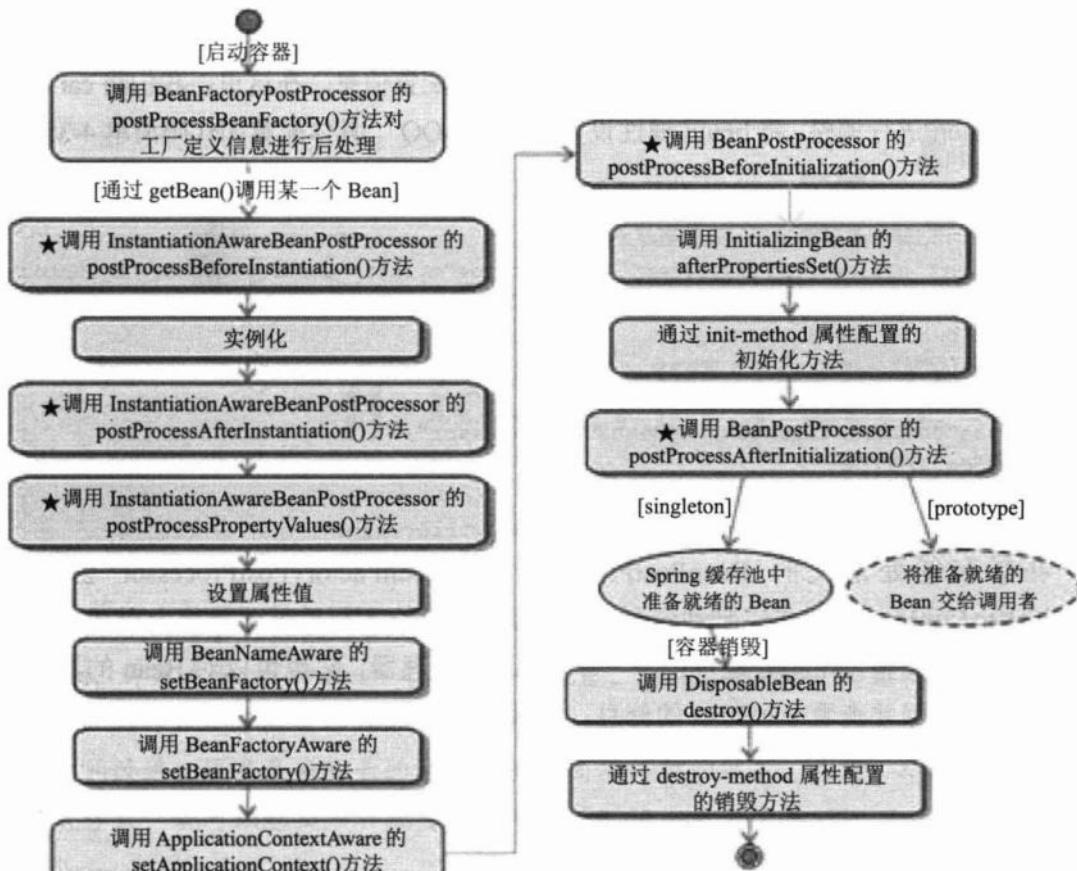


图 4-12 ApplicationContext 中 Bean 的生命周期

初始化的过程都是比较长，我们可以分类来对其进行解析：

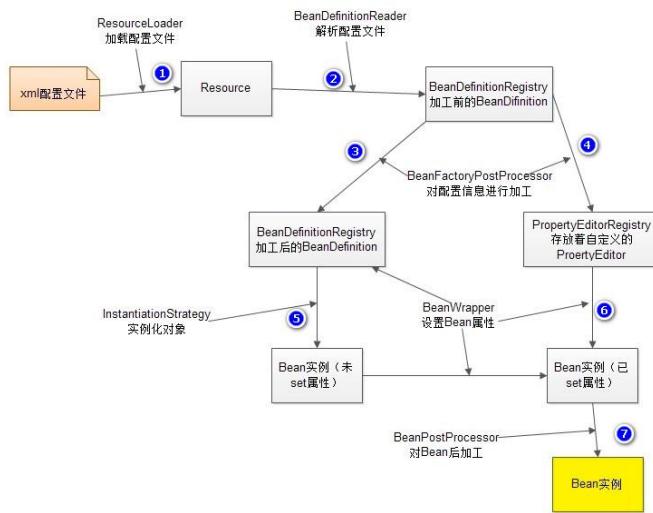
- **Bean 自身的方法**：如调用 Bean 构造函数实例化 Bean，调用 Setter 设置 Bean 的属性值以及通过的 init-method 和 destroy-method 所指定的方法；
- **Bean 级生命周期接口方法**：如 BeanNameAware、BeanFactoryAware、InitializingBean 和 DisposableBean，这些接口方法由 Bean 类直接实现；
- **容器级生命周期接口方法**：在上图中带“★”的步骤是由 InstantiationAwareBeanPostProcessor 和 BeanPostProcessor 这两个接口实现，一般称它们的实现类为“后处理器”。后处理器接口一般不由 Bean 本身实现，它们独立于 Bean，实现类以容器附加装置的形式注册到 Spring 容器中并通过接口反射为 Spring 容器预先识别。当 Spring 容器创建任何 Bean 的时候，这些后处理器都会发生作用，所以这些后处理器的影响是全局性的。

当然，用户可以通过合理地编写后处理器，让其仅对感兴趣 Bean 进行加工处理

ApplicationContext 和 BeanFactory 不同之处在于：

- ApplicationContext 会利用 Java 反射机制自动识别出配置文件中定义的 BeanPostProcessor、InstantiationAwareBeanPostProcesso 和 BeanFactoryPostProcessor 后置器，并自动将它们注册到应用上下文中。而 BeanFactory 需要在代码中通过手工调用 addBeanPostProcessor()方法进行注册
- ApplicationContext 在初始化应用上下文的时候就实例化所有单实例的 Bean。而 BeanFactory 在初始化容器的时候并未实例化 Bean，直到第一次访问某个 Bean 时才实例化目标 Bean。

有了上面的知识点了，我们再来详细地看看 Bean 的初始化过程：



简要总结：

- BeanDefinitionReader 读取 Resource 所指向的配置文件资源，然后解析配置文件。配置文件中每一个<bean>解析成一个 BeanDefinition 对象，并保存到 BeanDefinitionRegistry 中；
- 容器扫描 BeanDefinitionRegistry 中的 BeanDefinition；调用 InstantiationStrategy 进行 Bean 实例化的工作；使用 BeanWrapper 完成 Bean 属性的设置工作；

- 单例 Bean 缓存池：Spring 在 DefaultSingletonBeanRegistry 类中提供了一个用于缓存单实例 Bean 的缓存器，它是一个用 HashMap 实现的缓存器，单实例的 Bean 以 beanName 为键保存在这个 HashMap 中。

## 1.3 IOC 容器装配 Bean

### 1.3.1 装配 Bean 方式

Spring4.x 开始 IOC 容器装配 Bean 有 4 种方式：

- XML 配置
- 注解
- JavaConfig
- 基于 Groovy DSL 配置(这种很少见)

总的来说：我们以 XML 配置+注解来装配 Bean 得多，其中注解这种方式占大部分！

### 1.3.2 依赖注入方式

依赖注入的方式有 3 种方式：

- 属性注入--|通过 setter()方法注入
- 构造函数注入
- 工厂方法注入

总的来说使用属性注入是比较灵活和方便的，这是大多数人的选择！

### 1.3.3 对象之间关系

<bean>对象之间有三种关系：

- 依赖--|挺少用的(使用 depends-on 就是依赖关系了--|前置依赖【依赖的 Bean 需要初始化之后，当前 Bean 才会初始化】 )
- 继承--|可能会用到(指定 abstract 和 parent 来实现继承关系)

- 引用--最常见(使用 ref 就是引用关系了)

### 1.3.4 Bean 的作用域

Bean 的作用域：

- 单例 Singleton
- 多例 prototype
- 与 Web 应用环境相关的 Bean 作用域
  - request
  - session

使用到了 Web 应用环境相关的 Bean 作用域的话，是需要我们手动配置代理的~

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop" ①声明 aop Schema
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.
                           springframework.org/schema/beans/spring-beans-4.0.xsd
                           http://www.springframework.org/schema/aop http://www.springframework.org/
                           schema/aop/spring-aop-4.0.xsd">
    <bean name="car" class="com.smart.scope.Car" scope="request">
        <aop:scoped-proxy/> ② 创建代理
    </bean>
    <bean id="boss" class="com.smart.scope.Boss">
        <property name="car" ref="car"/> ③ 引用 Web 相关作用域的 car Bean
    </bean>
</beans>

```

原因也很简单：因为我们默认的 Bean 是单例的，为了适配 Web 应用环境相关的 Bean 作用域---|每个 request 都需要一个对象，此时我们返回一个代理对象出去就可以完成我们的需求了！

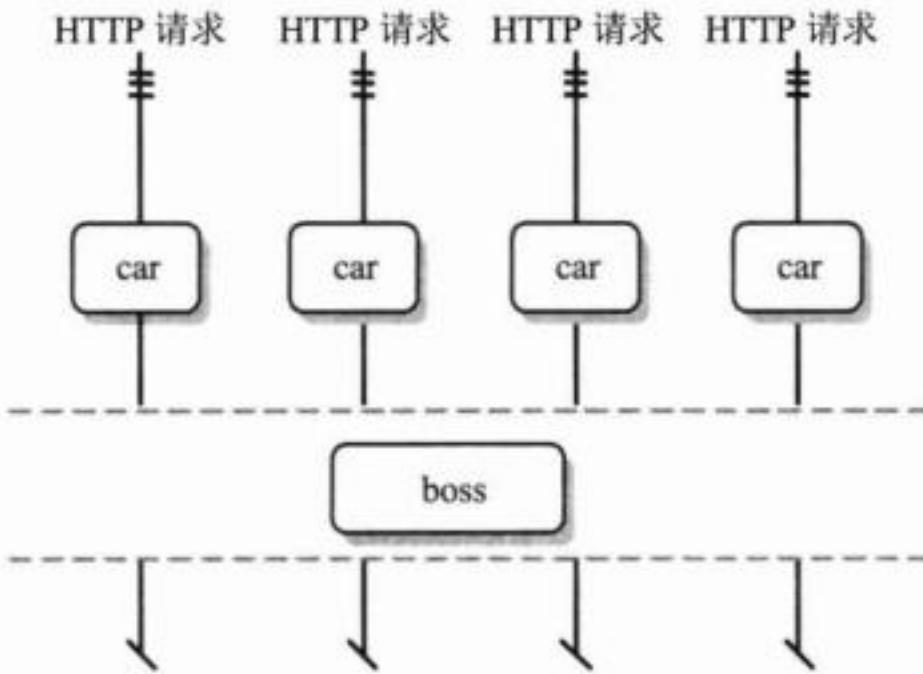


图 5-6 引用 Web 相关作用域的 Bean

将 Bean 配置单例的时候还有一个问题：

- 如果我们的 Bean 配置的是单例，而 Bean 对象里边的成员对象我们希望是多例的话。那怎么办呢？？
- 默认的情况下我们的 Bean 单例，返回的成员对象也默认是单例的(因为对象就只有那么一个)！

此时我们需要用到了 lookup 方法注入，使用也很简单，看看例子就明白了：

```
<!--① prototype 类型的 Bean -->
<bean id="car" class="com.smart.injectfun.Car"
      p: brand="红旗CA72" p: price="2000" scope="prototype"/>
<!--②实施方法注入-->
<bean id="magicBoss" class="com.smart.injectfun.MagicBoss">
    <lookup-method name="getCar" bean="car"/>
</bean>
```

单例的对象  
想返回多例  
的成员变量

### 1.3.6 处理自动装配的歧义性

昨天在刷书的时候刚好看到了有人在知乎邀请我回答这个问题：

#### 一个接口两个实现类怎么在注入的时候优先调用某个实现类？ [修改](#)

一个接口两个实现类怎么在注入的时候优先调用某个实现类？

▲ 162 次浏览

结合两本书的知识点，可以归纳成两种解决方案：

- 使用@Primary 注解设置为首选的注入 Bean
- 使用@Qualifier 注解设置特定名称的 Bean 来限定注入！
  - 也可以使用自定义的注解来标识

### 1.3.7 引用属性文件以及 Bean 属性

之前在写配置文件的时候都是直接将我们的数据库配置信息在里面写死的了：

```
#数据库配置
spring:
  datasource:
    username: xxx
    password: xxxxx
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/xxxxx?useUnicode=true&characterEncoding=utf-8
    &serverTimezone=UTC
    # JPA配置
    jpa:
```

其实我们有更优雅的做法：将这些配置信息写到配置文件上(因为这些配置信息很可能是会变的，而且有可能被多个配置文件引用)。

- 如此一来，我们改的时候就十分方便了。

```

resources
  com
    smart
      beanprop
        beans.xml
        beans1.xml
        jdbc.properties
      editor
      event
      i18n
      place
      placeholder
      log4j.properties
  test
  chapter6.ilm
  pom.xml
  external Libraries

7   http://www.springframework.org/schema/context
8   http://www.springframework.org/schema/context/spring-cc
9   <context:property-placeholder
10  location="classpath:com/smart/placeholder/jdbc.properti
11
12  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDa
13  destroy-method="close"
14  p:driverClassName="${driverClassName}"
15  p:url="${url}"
16  p:username="${userName}"
17  p:password="${password}"/>
18
19  <bean id="sysConfig" class="com.smart.beanprop.SysConfig"
20  init-method="initFromDB"
21  p:dataSource-ref="dataSource"/>
22

```

引用了这个配置文件的数据

引用配置文件的数据使用的是 {}

除了引用配置文件上的数据，我们还可以引用 Bean 的属性：

```

<bean id="sysConfig" class="com.smart.beanprop.SysConfig"
  init-method="initFromDB"
  p:dataSource-ref="dataSource"/>

<bean class="com.smart.beanprop.ApplicationManager"
  p:maxTabPageNum="#{sysConfig.maxTabPageNum}"
  p:sessionTimeout="#{sysConfig.sessionTimeout}"/>

```

```

public class SysConfig {
    private int sessionTimeout;
    private int maxTabPageNum;

    private DataSource dataSource;

    public void initFromDB() {
        //从数据库中获取属性配置值
        this.sessionTimeout = 30;
        this.maxTabPageNum = 10;
    }

    public int getSessionTimeout() { return sessionTimeout; }

    public int getMaxTabPageNum() { return maxTabPageNum; }

    public void setDataSource(DataSource dataSource) { this.dataSource = dataSource; }
}

```

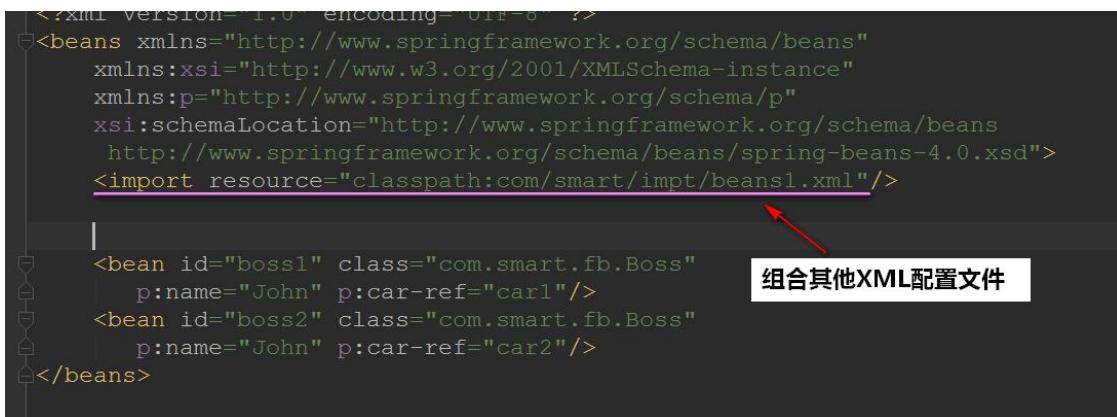
引用 Bean 的属性使用的是#{}

在这种技术在《Spring 实战 第四版》称之为 Spring EL，跟我们之前学过的 EL 表达式是类似的。主要的功能就是上面的那种，想要更深入了解可参考下面的链接：

- <http://www.cnblogs.com/lei001ei/p/3543222.html>

### 1.3.8 组合配置文件

xml 文件之间组合：



xml 和 javaconfig 互相组合的方式：

```

public static void main(String[] args) {

    //1.通过构造函数加载配置类
    ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConf.c1
ass);

    //2.通过编码方式注册配置类
    AnnotationConfigApplicationContext ctx = new AnnotationConfigAppli
cationContext();
    ctx.register(DaoConfig.class);
    ctx.register(ServiceConfig.class);
    ctx.refresh();

    //3.通过 XML 组装@Configuration 配置类所提供的配置信息
    ApplicationContext ctx = new ClassPathXmlApplicationContext('com/sm
art/conf/beans2.xml');
  
```

//4.通过@Configuration 组装 XML 配置所提供的配置信息

```
ApplicationContext ctx = new AnnotationConfigApplicationContext(Lo  
gonAppConfig.class);
```

//5.@Configuration 的配置类相互引用

```
ApplicationContext ctx = new AnnotationConfigApplicationContext(Da  
oConfig.class,ServiceConfig.class);  
  
LogonService logonService = ctx.getBean(LogonService.class);  
System.out.println((logonService.getLogDao() !=null));  
logonService.printHello();  
}
```

第一种的例子：

```
@Configuration  
public class AppConf {  
    @Bean  
    public UserDao userDao() { return new UserDao(); }  
  
    @Bean  
    public LogDao logDao() { return new LogDao(); }  
  
    @Bean  
    public LogonService logonService(){  
        LogonService logonService = new LogonService();  
        logonService.setLogDao(logDao());  
        logonService.setUserDao(userDao());  
        return logonService;  
    }  
}
```

第二种的例子：

```
@Configuration
public class DaoConfig {

    @Bean(name="")
    public UserDao userDao() { return new UserDao(); }

    @Scope("prototype")
    @Bean
    public LogDao logDao() { return new LogDao(); }
}
```

```
@Configuration
@Import(DaoConfig.class)
public class ServiceConfig {

    @Autowired
    private DaoConfig daoConfig;

    @Bean
    public LogonService logonService() {
        LogonService logonService = new LogonService();
        System.out.println(daoConfig.logDao() == daoConfig.logDao());
        logonService.setLogDao(daoConfig.logDao());
        logonService.setUserDao(daoConfig.userDao());
        return logonService;
    }
}
```

第三种的例子：

```
<?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-4.0.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-4.0.xsd">
    <context:component-scan base-package="com.smart.conf"
        resource-pattern="AppConf.class" />
</beans>
```

第四种的例子：

```
@Configuration
@ImportResource("classpath:com/smart/conf/beans3.xml")
public class LogonAppConfig {

    @Bean
    @Autowired
    public LogonService logonService(UserDao userDao, LogDao logDao) {
        LogonService logonService = new LogonService();
        logonService.setUserDao(userDao);
        logonService.setLogDao(logDao);
        return logonService;
    }
}
```

### 1.3.9 装配 Bean 总结

总的来说，Spring IOC 容器就是在创建 Bean 的时候有很多的方式给了我们实现，其中也包括了很多关于 Bean 的配置~

对于 Bean 相关的注入教程代码和简化配置(p 和 c 名称空间)我就不一一说明啦，你们去看 [Spring 入门](#)这一篇就够了和 [Spring【依赖注入】](#)就是这么简单就行了。

总的对比图：

	基于 XML 配置	基于注解配置	基于 Java 类配置	基于 Groovy DSL 配置
Bean 定义	在 XML 文件中通过 <bean> 元素定义 Bean, 如: <bean class="com.smart.UserDao"/>	在 Bean 实现类处通过标注 @Compoent 或衍型类 (@Repository、@Service 及 @Controller) 定义 Bean	在标注了 @Configuration 的 Java 类中, 通过在类方法上标注 @Bean 定义一个 Bean。方法必须提供 Bean 的实例化逻辑	在 Groovy 文件中通过 DSL 定义 Bean, 如: userDao(UserDao)
Bean 名称	通过<bean>的 id 或 name 属性定义, 如: <bean id="userDao" class="com.smart.UserDao"/> 默认名称为 com.smart.UserDao#0	通过注解的 value 属性定义, 如@Component ("userDao")。默认名称为小写字母开头的类名(不带包名) userDao	通过@Bean 的 name 属性定义, 如@Bean("userDao")。默认名称为方法名	通过 Groovy 的 DSL 定义 Bean 的名称 (Bean 的类型, Bean 构建函数参数), 如: logonService(LogonService, userDao)
Bean 注入	通过<property>子元素或通过 p 命名空间的动态属性, 如 p:userDao-ref="userDao" 进行注入	通过在成员变更或方法入参处标注 @Autowired, 按类型匹配自动注入。还可以配合使用 @Qualifier 按名称匹配方式注入	比较灵活, 可以在方法处通过 @Autowired 使方法入参绑定 Bean, 然后在方法中通过代码进行注入; 还可通过调用配置类的 @Bean 方法进行注入	比较灵活, 可以在方法处通过 ref()方法进行注入, 如 ref("logDao")
Bean 生命过程方法	通过<bean>的 init-method 和 destory-method 属性指定 Bean 实现类的方法名。最多只能指定一个初始化方法和一个销毁方法	通过在目标方法上标注 @PostConstruct 和 @PreDestroy 注解指定初始化或销毁方法, 可以定义任意多个	通过@Bean 的 initMethod 或 destoryMethod 指定一个初始化或销毁方法。 对于初始化方法来说, 可以直接在方法内部通过代码的方式灵活定义初始化逻辑	通过 bean-> bean.initMethod 或 bean.destoryMethod 指定一个初始化或销毁方法
Bean 作用范围	通过<bean>的 scope 属性指定, 如: <bean class="com.smart.UserDao" scope="prototype"/>	通过在类定义处标注 @Scope 指定, 如 @Scope ("prototype")	通过在 Bean 方法定义处标注 @Scope 指定	通过 bean-> bean.scope = "prototype" 指定
Bean 延迟初始化	通过 <bean> 的 lazy-init 属性指定, 默认认为 default, 继承于 <beans> 的 default-lazy-init 设置, 该值默认认为 false	通过在类定义处标注 @Lazy 指定, 如 @Lazy (true)	通过在 Beam 方法定义处标注 @Lazy 指定	通过 bean-> bean.lazyInit = true 指定

分别的应用场景：

	基于 XML 配置	基于注解配置	基于 Java 类配置	基于 Groovy DSL 配置
适用场景	(1) Bean 实现类来源于第三方类库，如 DataSource、JdbcTemplate 等，因无法在类中标注注解，所以通过 XML 配置方式较好。 (2) 命名空间的配置，如 aop、context 等，只能采用基于 XML 的配置	Bean 的实现类是当前项目开发的，可以直接在 Java 类中使用基于注解的配置	基于 Java 类配置的优势在于可以通过代码方式控制 Bean 初始化的整体逻辑。如果实例化 Bean 的逻辑比较复杂，则比较适合基于 Java 类配置的方式	基于 Groovy DSL 配置的优势在于可以通过 Groovy 脚本灵活控制 Bean 初始化的过程。如果实例化 Bean 的逻辑比较复杂，则比较适合基于 Groovy DSL 配置的方式

至于一些小的知识点：

- 方法替换
  - 使用某个 Bean 的方法替换成另一个 Bean 的方法
- 属性编辑器
  - Spring 可以对基本类型做转换就归结于属性编辑器的功劳！
- 国际化
  - 使用不同语言(英语、中文)的操作系统去显式不同的语言
- profile 与条件化的 Bean
  - 满足了某个条件才初始化 Bean，这可以方便切换生产环境和开发环境~
- 容器事件
  - 类似于我们的 Servlet 的监听器，只不过它是在 Spring 中实现了~

上面这些小知识点比较少情况会用到，这也不去讲解啦。知道有这么一回事，到时候查查就会用啦~~~



加油！ 加油！ 加油！



如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜 Java3y 公众号有我的联系方式。更多原创技术文章可关注我的 GitHub：

<https://github.com/ZhongFuCheng3y/3y>

## 二、Spring IOC 相关面试题

将 SpringIOC 相关知识点整理了一遍，要想知道哪些知识点是比较重要的。很简单，我们去找找相关的面试题就知道了，如果该面试题是常见的，那么说明这个知识点还是相对比较重要的啦！

以下的面试题从各种博客上摘抄下来，摘抄量较大的会注明出处的~

### 2.1 什么是 spring?

什么是 spring ?

Spring 是个 java 企业级应用的开源开发框架。Spring 主要用来开发 Java 应用，但是有些扩展是针对构建 J2EE 平台的 web 应用。Spring 框架目标是简化 Java 企业级应用开发，并通过 POJO 为基础的编程模型促进良好的编程习惯。

### 2.2 使用 Spring 框架的好处是什么？

使用 Spring 框架的好处是什么？

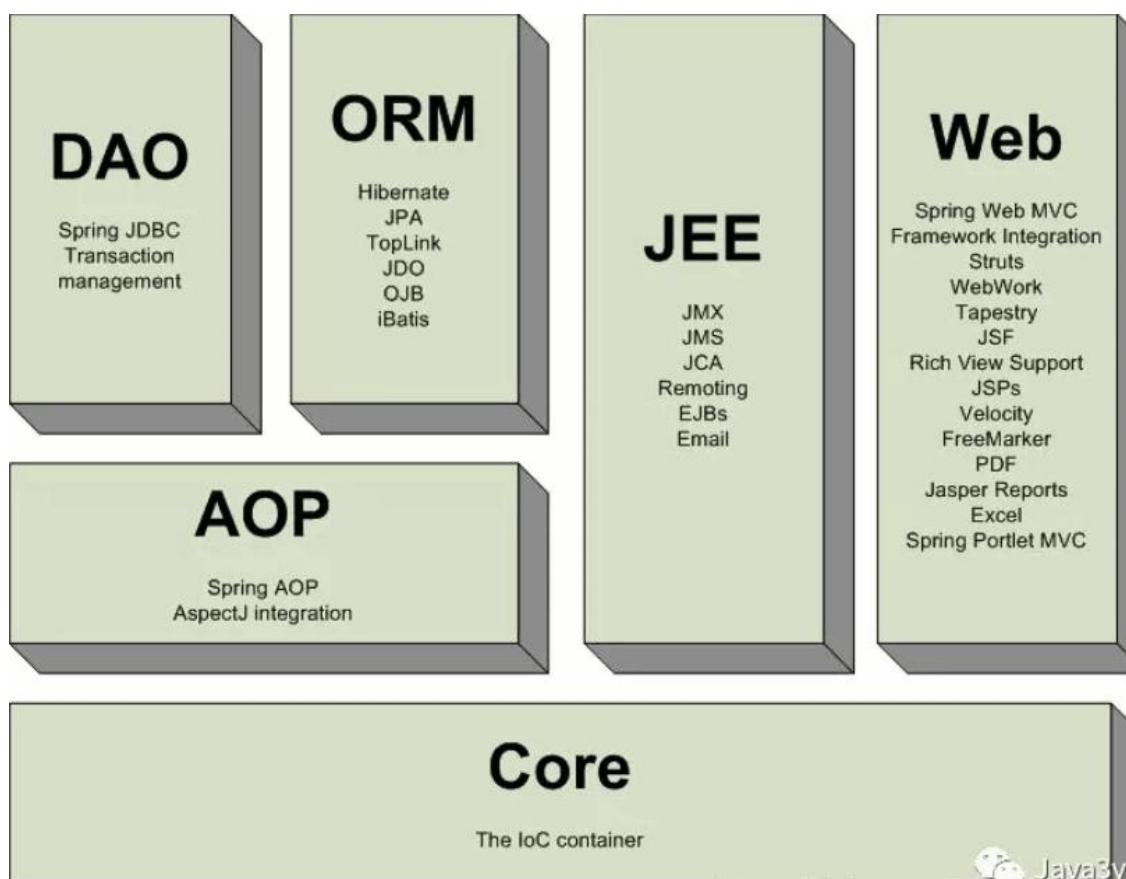
- **轻量**：Spring 是轻量的，基本的版本大约 2MB 。
- **控制反转**：Spring 通过控制反转实现了松散耦合，对象们给出它们的依赖，而不是创建或查找依赖的对象们。
- **面向切面的编程(AOP)**：Spring 支持面向切面的编程，并且把应用业务逻辑和系统服务分开。
- **容器**：Spring 包含并管理应用中对象的生命周期和配置。
- **MVC 框架**：Spring 的 WEB 框架是个精心设计的框架，是 Web 框架的一个很好的替代品。
- **事务管理**：Spring 提供一个持续的事务管理接口，可以扩展到上至本地事务下至全局事务（JTA）。
- **异常处理**：Spring 提供方便的 API 把具体技术相关的异常（比如由 JDBC，Hibernate or JDO 抛出的）转化为一致的 unchecked 异常。

## 2.3 Spring 由哪些模块组成？

Spring 由哪些模块组成？

简单可以分成 6 大模块：

- Core
- AOP
- ORM
- DAO
- Web
- Spring EE



## 2.4 BeanFactory 实现举例

BeanFactory 实现举例

Bean 工厂是工厂模式的一个实现，提供了控制反转功能，用来把应用的配置和依赖从真正应用代码中分离。

在 spring3.2 之前最常用的是 XmlBeanFactory 的，但现在被废弃了，取而代之的是：XmlBeanDefinitionReader 和 DefaultListableBeanFactory

## 2.5 什么是 Spring 的依赖注入？

什么是 Spring 的依赖注入？

依赖注入，是 IOC 的一个方面，是个通常的概念，它有多种解释。这概念是说你不用创建对象，而只需要描述它如何被创建。你不在代码里直接组装你的组件和服务，但是要在配置文件里描述哪些组件需要哪些服务，之后一个容器（IOC 容器）负责把他们组装起来。

## 2.6 有哪些不同类型的 IOC ( 依赖注入 ) 方式？

有哪些不同类型的 IOC ( 依赖注入 ) 方式？

- **构造器依赖注入**：构造器依赖注入通过容器触发一个类的构造器来实现的，该类有一系列参数，每个参数代表一个对其他类的依赖。
- **Setter 方法注入**：Setter 方法注入是容器通过调用无参构造器或无参 static 工厂方法实例化 bean 之后，调用该 bean 的 setter 方法，即实现了基于 setter 的依赖注入。
- **工厂注入**：这个是遗留下来的，很少用的了！

## 2.7 哪种依赖注入方式你建议使用，构造器注入，还是 Setter 方法注入？

哪种依赖注入方式你建议使用，构造器注入，还是 Setter 方法注入？

你两种依赖方式都可以使用，构造器注入和 Setter 方法注入。最好的解决方案是用构造器参数实现强制依赖，setter 方法实现可选依赖。

## 2.8 什么是 Spring beans?

什么是 Spring beans ?

Spring beans 是那些形成 Spring 应用的主干的 java 对象。它们被 Spring IOC 容器初始化，装配，和管理。这些 beans 通过容器中配置的元数据创建。比如，以 XML 文件中<bean/>的形式定义。

这里有四种重要的方法给 Spring 容器提供配置元数据。

- XML 配置文件。
- 基于注解的配置。
- 基于 java 的配置。
- Groovy DSL 配置

## 2.9 解释 Spring 框架中 bean 的生命周期

解释 Spring 框架中 bean 的生命周期

- Spring 容器从 XML 文件中读取 bean 的定义，并实例化 bean。
- Spring 根据 bean 的定义填充所有的属性。
- 如果 bean 实现了 BeanNameAware 接口，Spring 传递 bean 的 ID 到 setBeanName 方法。
- 如果 Bean 实现了 BeanFactoryAware 接口，Spring 传递 beanfactory 给 setBeanFactory 方法。
- 如果有任何与 bean 相关联的 BeanPostProcessors，Spring 会在 postProcesserBeforeInitialization()方法内调用它们。
- 如果 bean 实现 IntializingBean 了，调用它的 afterPropertySet 方法，如果 bean 声明了初始化方法，调用此初始化方法。
- 如果有 BeanPostProcessors 和 bean 关联，这些 bean 的 postProcessAfterInitialization() 方法将被调用。
- 如果 bean 实现了 DisposableBean，它将调用 destroy()方法。

## 2.10 解释不同方式的自动装配

解释不同方式的自动装配

- no：默认的方式是不进行自动装配，通过显式设置 ref 属性来进行装配。

- byName：通过参数名自动装配，Spring 容器在配置文件中发现 bean 的 autowire 属性被设置成 byname，之后容器试图匹配、装配和该 bean 的属性具有相同名字的 bean。
- byType:：通过参数类型自动装配，Spring 容器在配置文件中发现 bean 的 autowire 属性被设置成 byType，之后容器试图匹配、装配和该 bean 的属性具有相同类型的 bean。如果有多个 bean 符合条件，则抛出错误。
- constructor：这个方式类似于 byType，但是要提供给构造器参数，如果没有确定的带参数的构造器参数类型，将会抛出异常。
- autodetect：首先尝试使用 constructor 来自动装配，如果无法工作，则使用 byType 方式。

只用注解的方式时，注解默认是使用 byType 的！

## 2.11 IOC 的优点是什么？

IOC 的优点是什么？

IOC 或 依赖注入把应用的代码量降到最低。它使应用容易测试，单元测试不再需要单例和 JNDI 查找机制。最小的代价和最小的侵入性使松散耦合得以实现。IOC 容器支持加载服务时的饿汉式初始化和懒加载。

## 2.12 哪些是重要的 bean 生命周期方法？你能重载它们吗？

哪些是重要的 bean 生命周期方法？你能重载它们吗？

有两个重要的 bean 生命周期方法，第一个是 setup，它是在容器加载 bean 的时候被调用。第二个方法是 teardown 它是在容器卸载类的时候被调用。

The bean 标签有两个重要的属性（init-method 和 destroy-method）。用它们你可以自己定制初始化和注销方法。它们也有相应的注解（@PostConstruct 和 @PreDestroy）。

## 2.13 怎么回答面试官：你对 Spring 的理解？

怎么回答面试官：你对 Spring 的理解？

来源：

- <https://www.zhihu.com/question/48427693?sort=created>

下面我就截几个答案：

一、



飞龙

卑鄙是卑鄙者的通行证，高尚是高尚者的墓志铭。

76 人赞同了该回答

农业社会中，你需要一个对象，你必须亲手把它 new 出来。

工业社会中，你需要一个对象，可以去工厂那里获取。

共产主义社会中，你需要一个对象，只需要凭 ID 去对象池里面领。

这是何等的共产主义精神！

编辑于 2017-10-30

▲ 76

▼

添加评论

分享

收藏

感谢

...

二、



暗灭

夜是谁

59 人赞同了该回答

其实这种问题考核的就是前因后果。

spring最初诞生的时候是依赖注入，控制反转和非侵入式。  
这三点讲清楚就很好了。

后续的发展如果讲的出来，就是你对开源社区的关注度。

spring cloud，spring boot等都是可以聊一聊的。

确切的说，在没有spring的时候大家怎么写代码？spring是为了解决什么问题呢。

发布于 2018-01-08

▲ 59

▼

● 1条评论

分享

收藏

感谢

...

## 2.14 Spring 框架中的单例 Beans 是线程安全的么？

Spring 框架中的单例 Beans 是线程安全的么？

Spring 框架并没有对单例 bean 进行任何多线程的封装处理。关于单例 bean 的线程安全和并发问题需要开发者自行去搞定。但实际上，大部分的 Spring bean 并没有可变的状态(比如 Serview 类和 DAO 类)，所以在某种程度上说 Spring 的单例 bean 是线程安全的。如果你的 bean 有多种状态的话(比如 View Model 对象)，就需要自行保证线程安全。

最浅显的解决办法就是将多态 bean 的作用域由“singleton”变更为“prototype”

## 2.15 FileSystemResource 和 ClassPathResource 有何区别？

FileSystemResource 和 ClassPathResource 有何区别？

在 FileSystemResource 中需要给出 spring-config.xml 文件在你项目中的相对路径或者绝对路径。在 ClassPathResource 中 spring 会在 ClassPath 中自动搜寻配置文件，所以要把 ClassPathResource 文件放在 ClassPath 下。

如果将 spring-config.xml 保存在了 src 文件夹下的话，只需给出配置文件的名称即可，因为 src 文件夹是默认。

简而言之，ClassPathResource 在环境变量中读取配置文件，FileSystemResource 在配置文件中读取配置文件。





如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜 Java3y 公众号有我的联系方式。更多原创技术文章可关注我的 GitHub：

<https://github.com/ZhongFuCheng3y/3y>

## AOP 再回顾

这篇文章主要是补充和强化一些比较重要的知识点

### 一、Spring AOP 全面认知

结合《Spring 实战 (第 4 版)》和《精通 Spring4.x 企业应用开发实战》两本书的 AOP 章节将其知识点整理起来~

## 1.1 AOP 概述

AOP 称为面向切面编程，那我们怎么理解面向切面编程？？

我们可以先看看下面这段代码：

```
public class ForumService {  
    private TransactionManager transManager;  
    private PerformanceMonitor pmonitor;  
    private TopicDao topicDao;  
    private ForumDao forumDao;  
  
    public void removeTopic(int topicId) {  
        pmonitor.start();  
        transManager.beginTransaction();  
        topicDao.removeTopic(topicId); //①  
        transManager.commit();  
        pmonitor.end();  
    }  
    public void createForum(Forum forum) {  
        pmonitor.start();  
        transManager.beginTransaction();  
        forumDao.create(forum); //②  
        transManager.commit();  
        pmonitor.end();  
    }  
    ...  
}
```

标红的是业务代码，蓝色的是非业务代码（重复！）

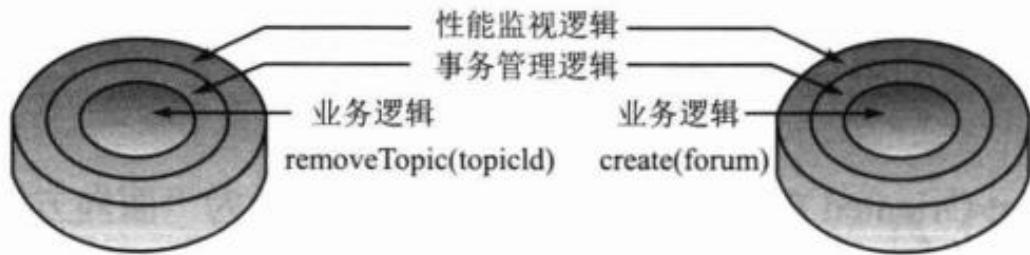
我们学 Java 面向对象的时候，如果代码重复了怎么办啊？？可以分成下面几个步骤：

- 1：抽取成方法
- 2：抽取类

抽取成类的方式我们称之为：纵向抽取

- 通过继承的方式实现纵向抽取

但是，我们现在的办法不行：即使抽取成类还是会出现重复的代码，因为这些逻辑（开始、结束、提交事务）依附在我们业务类的方法逻辑中！



现在纵向抽取的方式不行了，AOP 的理念：就是将分散在各个业务逻辑代码中相同的代码通过横向切割的方式抽取到一个独立的模块中！

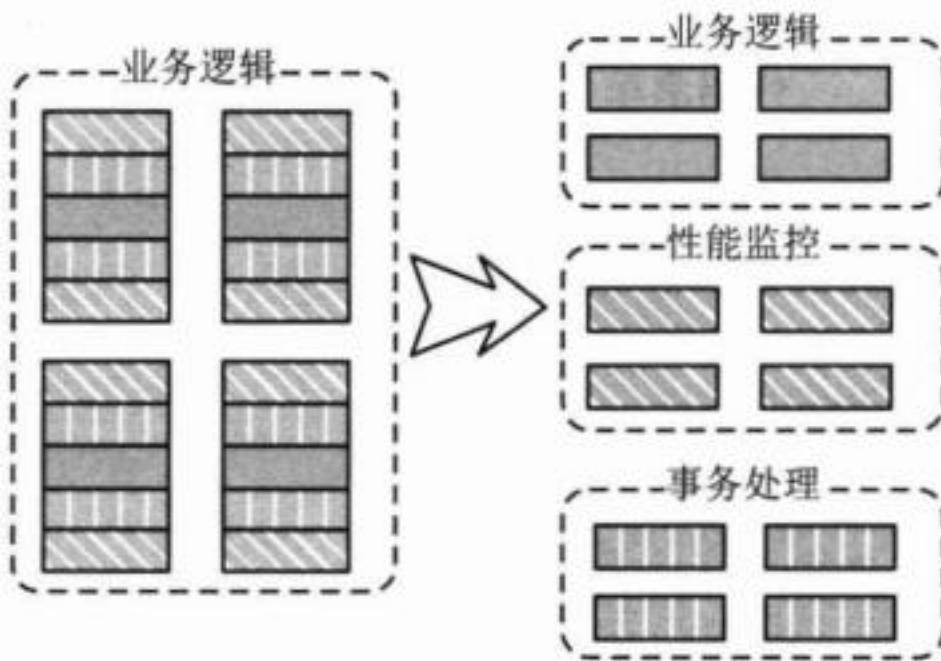


图 7-2 横向抽取

上面的图也很清晰了，将重复性的逻辑代码横切出来其实很容易(我们简单可认为就是封装成一个类就好了)，但我们要将这些被我们横切出来的逻辑代码融合到业务逻辑中，来完成和之前(没抽取前)一样的功能！这就是 AOP 首要解决的问题了！

## 1.2 Spring AOP 原理

被我们横切出来的逻辑代码融合到业务逻辑中，来完成和之前(没抽取前)一样的功能

没有学 Spring AOP 之前，我们就可以使用代理来完成。

- 如果看过我写的[给女朋友讲解什么是代理模式](#)这篇文章的话，一定就不难理解上面我说的那句话了
- 代理能干嘛？代理可以帮我们增强对象的行为！使用动态代理实质上就是调用时拦截对象方法，对方法进行改造、增强！

其实 Spring AOP 的底层原理就是动态代理！

来源《精通 Spring4.x 企业应用开发实战》一段话：

Spring AOP 使用纯 Java 实现，它不需要专门的编译过程，也不需要特殊的类装载器，它在运行期通过代理方式向目标类织入增强代码。在 Spring 中可以无缝地将 Spring AOP、IoC 和 AspectJ 整合在一起。

来源《Spring 实战 (第 4 版)》一句话：

Spring AOP 构建在动态代理基础之上，因此，Spring 对 AOP 的支持局限于方法拦截。

在 Java 中动态代理有两种方式：

- JDK 动态代理
- CGLib 动态代理

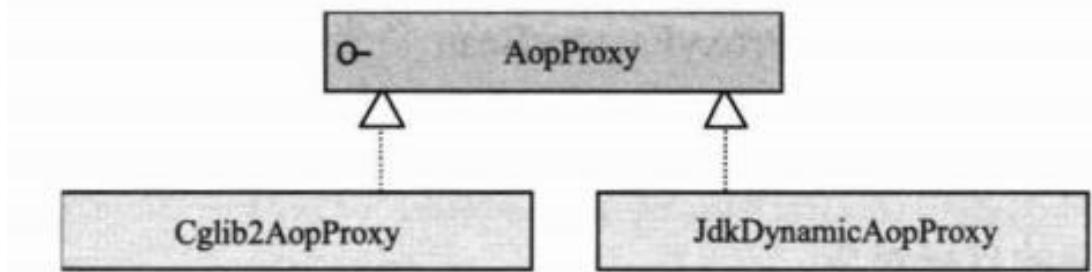


图 7-5 AopProxy 类结构

JDK 动态代理是需要实现某个接口了，而我们类未必全部会有接口，于是 CGLib 代理就有了~~

- CGLib 代理其生成的动态代理对象是目标类的子类
- Spring AOP 默认是使用 JDK 动态代理，如果代理的类没有接口则会使用 CGLib 代理。

那么 JDK 代理和 CGLib 代理我们该用哪个呢？？在《精通 Spring4.x 企业应用开发实战》给出了建议：

- 如果是单例的我们最好使用 CGLib 代理，如果是多例的我们最好使用 JDK 代理

原因：

- JDK 在创建代理对象时的性能要高于 CGLib 代理，而生成代理对象的运行性能却比 CGLib 的低。
- 如果是单例的代理，推荐使用 CGLib

看到这里我们就应该知道什么是 Spring AOP(面向切面编程)了：将相同逻辑的重复代码横向抽取出来，使用动态代理技术将这些重复代码织入到目标对象方法中，实现和原来一样的功能。

- 这样一来，我们就在写业务时只关心业务代码，而不用关心与业务无关的代码

## 1.3AOP 的实现者

AOP 除了有 Spring AOP 实现外，还有著名的 AOP 实现者：AspectJ，也有可能大家没听说过的实现者：JBoss AOP~~

我们下面来说说 AspectJ 扩展一下知识面：

AspectJ 是语言级别的 AOP 实现，扩展了 Java 语言，定义了 AOP 语法，能够在编译期提供横切代码的织入，所以它有专门的编译器用来生成遵守 Java 字节码规范的 Class 文件。

而 Spring 借鉴了 AspectJ 很多非常有用的做法，融合了 AspectJ 实现 AOP 的功能。但 Spring AOP 本质上底层还是动态代理，所以 Spring AOP 是不需要有专门的编辑器的~

## 1.4 AOP 的术语

嗯，AOP 搞了好几个术语出来~~两本书都有讲解这些术语，我会尽量让大家看得明白的：

**连接点(Join point)：**

- 能够被拦截的地方：Spring AOP 是基于动态代理的，所以是方法拦截的。每个成员方法都可以称之为连接点~

**切点(Poincut)：**

- 具体定位的连接点：上面也说了，每个方法都可以称之为连接点，我们具体定位到某一个方法就成为切点。

**增强/通知(Advice)：**

- 表示添加到切点的一段逻辑代码，并定位连接点的方位信息。
  - 简单来说就定义了是干什么的，具体是在哪干
  - Spring AOP 提供了 5 种 Advice 类型给我们：前置、后置、返回、异常、环绕给我们使用！

**织入(Weaving)：**

- 将增强/通知添加到目标类的具体连接点上的过程。

**引入/引介(Introduction)：**

- 引入/引介允许我们向现有的类添加新方法或属性。是一种特殊的增强！

**切面(Aspect)：**

- 切面由切点和增强/通知组成，它既包括了横切逻辑的定义、也包括了连接点的定义。

在《Spring 实战 (第 4 版)》给出的总结是这样子的：

通知/增强包含了需要用于多个应用对象的横切行为；连接点是程序执行过程中能够应用通知的所有点；切点定义了通知/增强被应用的具体位置。其中关键的是切点定义了哪些连接点会得到通知/增强。

总的来说：

- 这些术语可能翻译过来不太好理解，但对我们正常使用 AOP 的话影响并没有那么大~~看多了就知道它是什么意思了。

## 1.5 Spring 对 AOP 的支持

Spring 提供了 3 种类型的 AOP 支持：

- 基于代理的经典 SpringAOP
  - 需要实现接口，手动创建代理
- 纯 POJO 切面
  - 使用 XML 配置，aop 命名空间
- @AspectJ 注解驱动的切面
  - 使用注解的方式，这是最简洁和最方便的！





如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜 Java3y 公众号有我的联系方式。更多原创技术文章可关注我的 GitHub：

<https://github.com/ZhongFuCheng3y/3y>

## 二、基于代理的经典 SpringAOP

这部分配置比较麻烦，用起来也很麻烦，这里我就主要整理一下书上的内容，大家看看了解一下吧，我们实际上使用 Spring AOP 基本不用这种方式了！

首先，我们来看一下增强接口的继承关系图：

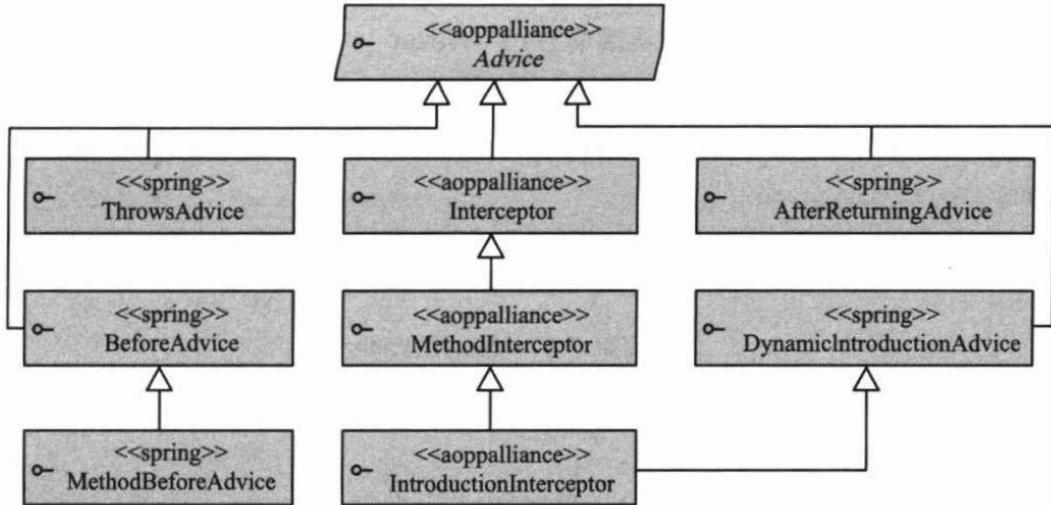


图 7-4 增强接口继承关系图

可以分成五类增强的方式：

- 前置增强：org.springframework.aop.BeforeAdvice 代表前置增强。因为 Spring 只支持方法级的增强，所以 MethodBeforeAdvice 是目前可用的前置增强，表示在目标方法执行前实施增强，而 BeforeAdvice 是为了将来版本扩展需要而定义的。
- 后置增强：org.springframework.aop.AfterReturningAdvice 代表后置增强，表示在目标方法执行后实施增强。
- 环绕增强：org.aopalliance.intercept.MethodInterceptor 代表环绕增强，表示在目标方法执行前后实施增强。
- 异常抛出增强：org.springframework.aop.ThrowsAdvice 代表抛出异常增强，表示在目标方法抛出异常后实施增强。
- 引介增强：org.springframework.aop.IntroductionInterceptor 代表引介增强，表示在目标类中添加一些新的方法和属性。

Spring 提供了六种的切点类型：

- ① 静态方法切点: `org.springframework.aop.support.StaticMethodMatcherPointcut` 是静态方法切点的抽象基类, 默认情况下它匹配所有的类。`StaticMethodMatcherPointcut` 包括两个主要的子类, 分别是 `NameMatchMethodPointcut` 和 `AbstractRegexpMethodPointcut`, 前者提供简单字符串匹配方法签名, 而后者使用正则表达式匹配方法签名。
- ② 动态方法切点: `org.springframework.aop.support.DynamicMethodMatcherPointcut` 是动态方法切点的抽象基类, 默认情况下它匹配所有的类。
- ③ 注解切点: `org.springframework.aop.support.annotation.AnnotationMatchingPointcut` 实现类表示注解切点。使用 `AnnotationMatchingPointcut` 支持在 Bean 中直接通过 Java 5.0 注解标签定义的切点。
- ④ 表达式切点: `org.springframework.aop.support.ExpressionPointcut` 接口主要是为了支持 AspectJ 切点表达式语法而定义的接口。 **重点!**
- ⑤ 流程切点: `org.springframework.aop.support.ControlFlowPointcut` 实现类表示控制流程切点。`ControlFlowPointcut` 是一种特殊的切点, 它根据程序执行堆栈的信息查看目标方法是否由某一个方法直接或间接发起调用, 以此判断是否为匹配的连接点。
- ⑥ 复合切点: `org.springframework.aop.support.ComposablePointcut` 实现类是为创建多个切点而提供的方便操作类。它所有的方法都返回 `ComposablePointcut` 类, 这样就可以使用链接表达式对切点进行操作, 形如: `Pointcut pc=new ComposablePointcut().union(classFilter).intersection(methodMatcher).intersection(pointcut)`。

切面类型主要分成了三种:

- 一般切面
- 切点切面
- 引介/引入切面



图 7-7 切面类继承关系

一般切面，切点切面，引介/引入切面介绍：

- **Advisor:** 代表一般切面，仅包含一个 **Advice**。因为 **Advice** 包含了横切代码和连接点信息，所以 **Advice** 本身就是一个简单的切面，只不过它代表的横切的连接点是所有目标类的所有方法，因为这个横切面太宽泛，所以一般不会直接使用。
- **PointcutAdvisor:** 代表具有切点的切面，包含 **Advice** 和 **Pointcut** 两个类，这样就可以通过类、方法名及方法方位等信息灵活地定义切面的连接点，提供更具适用性的切面。
- **IntroductionAdvisor:** 代表引介切面。7.3.6 节介绍了引介增强类型，引介切面是对应引介增强的特殊的切面，它应用于类层面上，所以引介切点使用 **ClassFilter** 进行定义。

对于切点切面我们一般都是直接用就好了，我们来看看引介/引入切面是怎么一回事：

- 引介/引入切面是引介/引入增强的封装器，通过引介/引入切面，可以更容易地为现有对象添加任何接口的实现！

继承关系图：

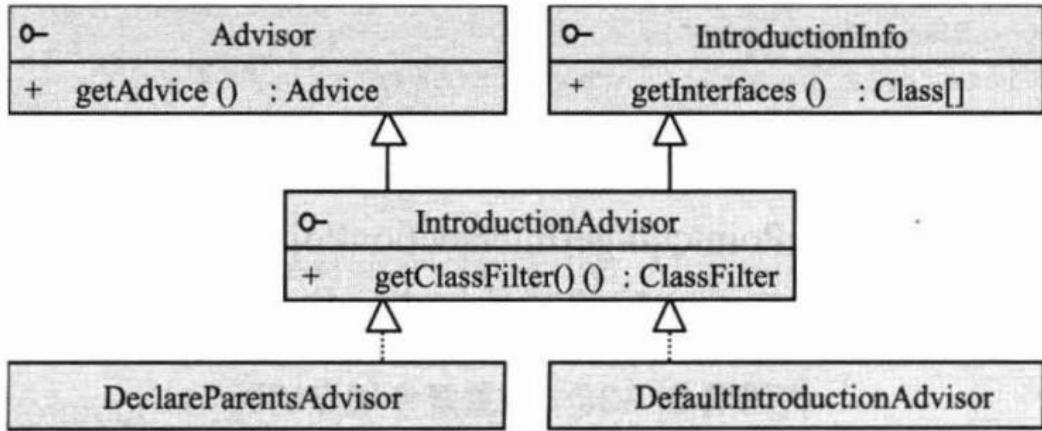


图 7-9 引介切面类继承关系图

引介/引入切面有两个实现类：

- `DefaultIntroductionAdvisor`：常用的实现类
- `DeclareParentsAdvisor`：用于实现 AspectJ 语言的 `DeclareParent` 注解表示的引介/引入切面

实际上，我们使用 AOP 往往是 Spring 内部使用 BeanPostProcessor 帮我们创建代理。

这些代理的创建器可以分成三类：

- 基于 Bean 配置名规则的自动代理创建器：`BeanNameAutoProxyCreator`
- 基于 Advisor 匹配机制的自动代理创建器：它会对容器所有的 Advisor 进行扫描，实现类为 `DefaultAdvisorAutoProxyCreator`
- 基于 Bean 中的 AspectJ 注解标签的自动代理创建器：  
`AnnotationAwareAspectJAutoProxyCreator`

对应的类继承图：

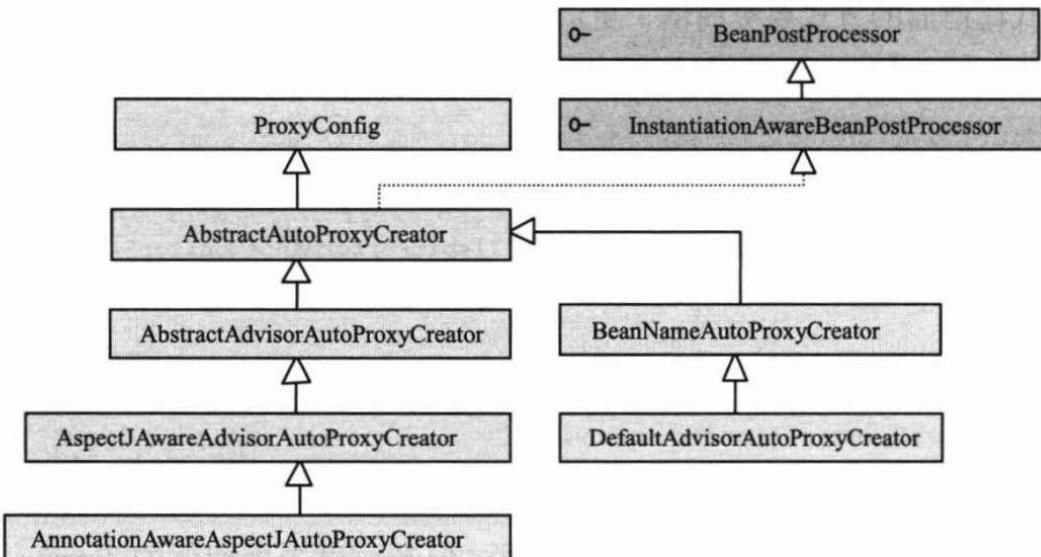


图 7-10 自动代理创建器实现类的类继承图

嗯，基于代理的经典 SpringAOP 就讲到这里吧，其实我是不太愿意去写这个的，因为已经几乎不用了，在《Spring 实战 第 4 版》也没有这部分的知识点了。

- 但是通过这部分的知识点可以更加全面地认识 Spring AOP 的各种接口吧~

### 三、拥抱基于注解和命名空间的 AOP 编程

Spring 在新版本中对 AOP 功能进行了增强，体现在这么几个方面：

- 在 XML 配置文件中为 AOP 提供了 aop 命名空间
- 增加了 AspectJ 切点表达式语言的支持
- 可以无缝地集成 AspectJ

那我们使用@AspectJ 来玩 AOP 的话，学什么？？其实也就是上面的内容，学如何设置切点、创建切面、增强的内容是什么...

类 别	函 数	入 参	说 明
方法切点 函数	execution()	方法匹配模式串	表示满足某一匹配模式的所有目标类方法连接点。如 execution(* greetTo(..))表示所有目标类中的 greetTo()方法
	@annotation()	方法注解类名	表示标注了特定注解的目标类方法连接点。如@annotation(com.smart.anno.NeedTest)表示任何标注了@NeedTest 注解的目标类方法
方法入参 切点函数	args()	类名	通过判别目标类方法运行时入参对象的类型定义指定连接点。如 args(com.smart.Waiter)表示所有有且仅有一个按类型匹配于 Waiter 入参的方法
	@args()	类型注解类名	通过判别目标类方法运行时入参对象的类是否标注特定注解来指定连接点。如@args(com.smart.Monitorable)表示任何这样的一个目标方法：它有一个入参且入参对象的类标注@Monitorable 注解
目标类 切点函数	within()	类名匹配串	表示特定域下的所有连接点。如 within(com.smart.service.*)表示 com.smart.service 包中的所有连接点，即包中所有类的所有方法；而 within(com.smart.service.*Service)表示在 com.smart.service 包中所有以 Service 结尾的类的所有连接点
	target()	类名	假如目标类按类型匹配于指定类，则目标类的所有连接点匹配这个切点。如通过 target(com.smart.Waiter)定义的切点、Waiter 及 Waiter 实现类 NaiveWaiter 中的所有连接点都匹配该切点
	@within()	类型注解类名	假如目标类按类型匹配于某个类 A，且类 A 标注了特定注解，则目标类的所有连接点匹配这个切点。如@within(com.smart.Monitorable)定义的切点，假如 Waiter 类标注了@Monitorable 注解，则 Waiter 及 Waiter 实现类 NaiveWaiter 的所有连接点都匹配这个切点
	@target()	类型注解类名	假如目标类标注了特定注解，则目标类的所有连接点都匹配该切点。如@target (com.smart.Monitorable)，假如 NaiveWaiter 标注了@Monitorable，则 NaiveWaiter 的所有连接点都匹配这个切点
代理类 切点函数	this()	类名	代理类按类型匹配于指定类，则被代理的目标类的所有连接点都匹配该切点。这个函数比较难以理解，这里暂不举例，留待后面详解

具体的切点表达式使用还是前往：[Spring【AOP 模块】就这么简单看吧~~](#)

对应的增强注解：

## 1. @Before

前置增强，相当于 BeforeAdvice。Before 注解类拥有两个成员。

- value: 该成员用于定义切点。
- argNames: 由于无法通过 Java 反射机制获取方法入参名，所以如果在 Java 编译时未启用调试信息，或者需要在运行期解析切点，就必须通过这个成员指定注解所标注增强方法的参数名（注意二者名字必须完全相同），多个参数名用逗号分隔。

## 2. @AfterReturning

后置增强，相当于 AfterReturningAdvice。AfterReturning 注解类拥有 4 个成员。

- value: 该成员用于定义切点。
- pointcut: 表示切点的信息。如果显式指定 pointcut 值，那么它将覆盖 value 的设置值，可以将 pointcut 成员看作 value 的同义词。
- returning: 将目标对象方法的返回值绑定给增强的方法。
- argNames: 如前所述。

## 3. @Around

环绕增强，相当于 MethodInterceptor。Around 注解类拥有两个成员。

- value: 该成员用于定义切点。
- argNames: 如前所述。

## 4. @AfterThrowing

抛出增强，相当于 ThrowsAdvice。AfterThrowing 注解类拥有 4 个成员。

- value: 该成员用于定义切点。
- pointcut: 表示切点的信息。如果显式指定 pointcut 值，那么它将覆盖 value 的设置值。可以将 pointcut 成员看作 value 的同义词。

- throwing: 将抛出的异常绑定到增强方法中。
- argNames: 如前所述。

### 5. @After

Final 增强，不管是抛出异常还是正常退出，该增强都会得到执行。该增强没有对应的增强接口，可以把它看成 ThrowsAdvice 和 AfterReturningAdvice 的混合物，一般用于释放资源，相当于 try{}finally{} 的控制流。After 注解类拥有两个成员。

- value: 该成员用于定义切点。
- argNames: 如前所述。

### 6. @DeclareParents

引介增强，相当于 IntroductionInterceptor。DeclareParents 注解类拥有两个成员。

- value: 该成员用于定义切点，它表示在哪个目标类上添加引介增强。
- defaultImpl: 默认的接口实现类。

## 3.1 使用引介/引入功能实现为 Bean 引入新方法

其实前置啊、后置啊这些很容易就理解了，整篇文章看下来就只有这个引介/引入切面有点搞头。于是我们就来玩玩吧~

我们来看一下具体的用法吧，现在我有个服务员的接口：

```
public interface Waiter {  
  
    // 向客人打招呼  
    void greetTo(String clientName);  
  
    // 服务  
    void serveTo(String clientName);  
}
```

一位年轻服务员实现类：

```
public class NaiveWaiter implements Waiter {  
    public void greetTo(String clientName) {  
        System.out.println('NaiveWaiter:greet to ' + clientName + '...');  
    }  
}
```

```
@NeedTest
public void serveTo(String clientName) {
    System.out.println('NaiveWaiter:serving ' + clientName + '...');
}

}
```

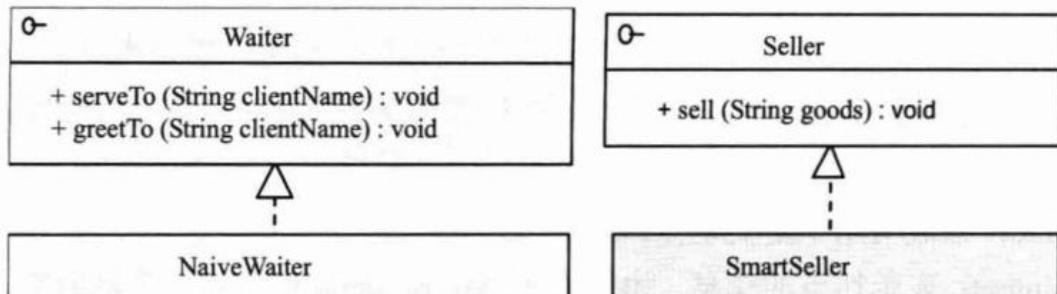
现在我想做的就是：想这个服务员可以充当售货员的角色，可以卖东西！当然了，我肯定不会加一个卖东西的方法到 Waiter 接口上啦，因为这个是暂时的~ 所以，我搞了一个售货员接口：

```
public interface Seller {
    // 卖东西
    int sell(String goods, String clientName);
}
```

一个售货员实现类：

```
public class SmartSeller implements Seller {
    // 卖东西
    public int sell(String goods, String clientName) {
        System.out.println('SmartSeller: sell ' + goods + ' to ' + clientName + '...');
        return 100;
    }
}
```

此时，我们的类图是这样子的：



现在我想干的就是：借助 AOP 的引入/引介切面，来让我们的服务员也可以卖东西！

我们的引入/引介切面具体是这样干的：

```

@Aspect
public class EnableSellerAspect {

    @DeclareParents(value = 'com.smart.NaiveWaiter', // 指定服务员具体的实现
                   defaultImpl = SmartSeller.class) // 售货员具体的实现
    public Seller seller; // 要实现的目标接口

}

```

写了这个切面类会发生什么？？

- 切面技术将 SmartSeller 融合到 NaiveWaiter 中，这样 NaiveWaiter 就实现了 Seller 接口！！！！

是不是很神奇？？我也觉得很神奇啊，我们来测试一下：

我们的 bean.xml 文件很简单：

```

<?xml version='1.0' encoding='UTF-8' ?!
<beans xmlns='http://www.springframework.org/schema/beans'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xmlns:aop='http://www.springframework.org/schema/aop'
    xsi:schemaLocation='http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.0.xsd'

```

```
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd'

<aop:aspectj-autoproxy>

    <bean id='waiter' class='com.smart.NaiveWaiter'/>
    <bean class='com.smart.aspectj.basic.EnableSellerAspect'/>

</beans>
```

测试一下：

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

        ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("com/smart/aspectj/basic/beans.xml");
        Waiter waiter = (Waiter) ctx.getBean("waiter");

        // 调用服务员原有的方法
        waiter.greetTo("Java3y");
        waiter.serveTo("Java3y");

        // 通过引介/引入切面已经将 waiter 服务员实现了 Seller 接口，所以可以强制转换
        Seller seller = (Seller) waiter;
        seller.sell("水军", "Java3y");

    }
}
```

```
NaiveWaiter:greet to Java3y...
NaiveWaiter:serving Java3y...
SmartSeller: sell 水军 to Java3y...
```

具体的调用过程是这样子的：

当引入接口方法被调用时，代理对象会把此调用委托给实现了新接口的某个其他对象。实际上，一个 Bean 的实现被拆分到多个类中

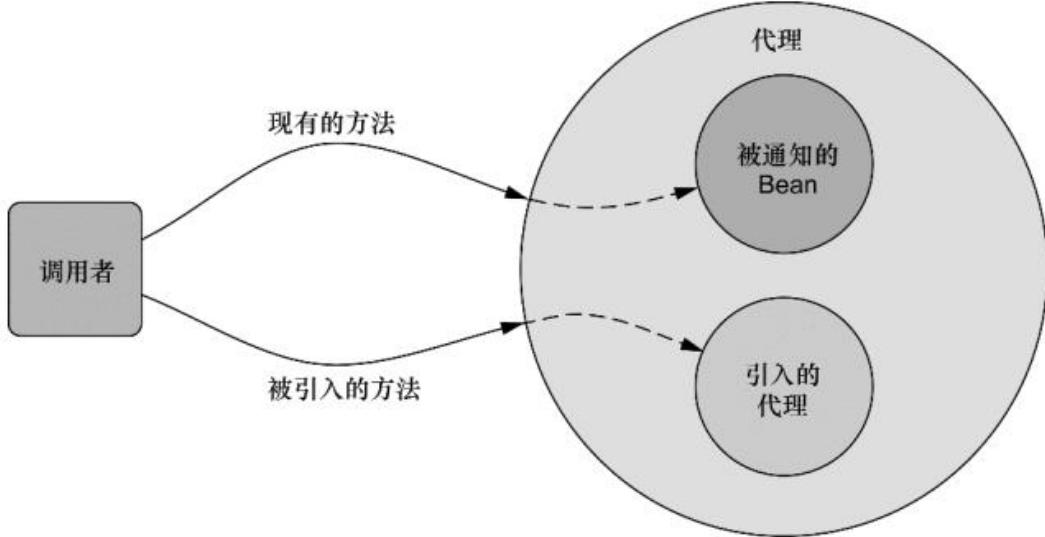


图4.7 使用Spring AOP, 我们可以为bean引入新的方法。  
代理拦截调用并委托给实现该方法的其他对象

### 3.2 在 XML 中声明切面

我们知道注解很方便，但是，要想使用注解的方式使用 Spring AOP 就必须要有源码(因为我们要在切面类上添加注解)。如果没有源码的话，我们就得使用 XML 来声明切面了~

其实就跟注解差不多的功能：

AOP配置元素	用    途
<aop:advisor>	定义AOP通知器
<aop:after>	定义AOP后置通知(不管被通知的方法是否执行成功)
<aop:after-returning>	定义AOP返回通知
<aop:after-throwing>	定义AOP异常通知
<aop:around>	定义AOP环绕通知
<aop:aspect>	定义一个切面
<aop:aspectj-autoproxy>	启用@AspectJ注解驱动的切面
<aop:before>	定义一个AOP前置通知
<aop:config>	顶层的AOP配置元素。大多数的<aop:>元素必须包含在<aop:config>元素内
<aop:declare-parents>	以透明的方式为被通知的对象引入额外的接口
<aop:pointcut>	定义一个切点

我们就直接来个例子终结掉它吧：

首先我们来测试一下与传统的 SpringAOP 结合的 advisor 是怎么用的：

实现类：

```

public class TestBeforeAdvice implements MethodBeforeAdvice {
    public void before(Method method, Object[] args, Object target)
        throws Throwable {
        System.out.println("-----TestBeforeAdvice-----");
        System.out.println("clientName:" + args[0]);
        System.out.println("-----TestBeforeAdvice-----");
    }
}

```

实现了接口

xml 配置文件：

```

<bean id="testAdvice" class="com.smart.schema.TestBeforeAdvice"/>
<aop:config proxy-target-class="true"> 指定cglib代理
    <aop:advisor advice-ref="testAdvice" pointcut="execution(* com..*.Waiter.greetTo(..))" /> 指定切入点表达式
        <aop:aspect ref="adviceMethods">

```

.....

一个一个来讲解还是太花时间了，我就一次性用图的方式来讲啦：

```

<aop:config proxy-target-class="true">
    <aop:aspect ref="adviceMethods">
        <aop:before method="preGreeting"
            pointcut="target(com.smart.NaiveWaiter) and args(name)"
            arg-names="name" /> 前置增强，绑定参数
        <aop:after-returning method="afterReturning"
            pointcut="target(com.smart.SmartSeller)" returning="retVal" /> 绑定返回值
        <aop:around method="aroundMethod"
            pointcut="execution(* serveTo(..)) and within(com.smart.Waiter)" /> 环绕增强
        <aop:after-throwing method="afterThrowingMethod"
            pointcut="target(com.smart.SmartSeller) and execution(* checkBill(..))"
            throwing="iae" /> 绑定抛出的异常
        <aop:after method="afterMethod"
            pointcut="execution(* com..*.Waiter.greetTo(..))" /> 后置增强
    <aop:declare-parents
        implement-interface="com.smart.Seller"
        default-impl="com.smart.SmartSeller"
        types-matching="com.smart.Waiter" /> 为Waiter实现Seller接口
        <aop:before method="bindParams"
            pointcut="target(com.smart.NaiveWaiter) and args(name,num,...)" /> 增强方法接收连接点的参数
    </aop:aspect>
</aop:config>

```

最后还有一个切面类型总结图，看完就几乎懂啦：

		@AspectJ	<aop:aspect>	Advisor	<aop:advisor>
增强 类型	前置增强	@Before	<aop:before>	MethodBeforeAdvice	同 Advisor
	后置增强	@AfterReturning	<aop:after-returning>	AfterReturningAdvice	同 Advisor
	环绕增强	@Around	<aop:around>	MethodInterceptor	同 Advisor
	抛出异常增强	@AfterThrowing	<aop:after-throwing>	ThrowsAdvice	同 Advisor
	final 增强	@After	<aop:after>	无对应接口	同 Advisor
	引介增强	@DeclareParents	<aop:declare-parents>	IntroductionInterceptor	同 Advisor
切点定义	支持 AspectJ 切点表达式语法，可以通过@Pointcut 注解定义命名切点	支持 AspectJ 切点表达式语法，可以通过<aop:pointcut> 定义命名切点	直接通过基于 Pointcut 的实现类定义切点	基本上和<aop:aspect>相同，不过切点函数不能绑定参数	
连接点方法入参绑定	(1) 使用 JoinPoint、Proceeding JoinPoint 连接点对象； (2) 使用切点函数指定参数名绑定	同@AspectJ <aop:after-returning>	通过增强接口方法入参绑定	同 Advisor	
连接点方法返回值或 抛出异常绑定	(1) 在后置增强中，使用 @AfterReturning 的 returning 成员绑定方法返回值； (2) 在抛出异常增强中，使用@AfterThrowing 的 throwing 成员绑定方法抛出的异常	(1) 在后置增强中，使用 <aop:after-returning> 的 returning 属性绑定方法返回值； (2) 在抛出异常增强中，使用 <aop:after-throwing> 的 throwing 属性绑定方法抛出的异常	通过增强接口方法入参绑定	同 Advisor	

## 四、总结

看起来 AOP 有很多很多的知识点，其实我们只要记住 AOP 的核心概念就行啦。

下面是我的简要总结 AOP：

- AOP 的底层实际上是动态代理，动态代理分成了 JDK 动态代理和 CGLib 动态代理。如果被代理对象没有接口，那么就使用的是 CGLIB 代理(也可以直接配置使用 CGLib 代理)
- 如果是单例的话，那我们最好使用 CGLib 代理，因为 CGLib 代理对象运行速度要比 JDK 的代理对象要快
- AOP 既然是基于动态代理的，那么它只能对方法进行拦截，它的层面上是方法级别的

- 无论经典的方式、注解方式还是 XML 配置方式使用 Spring AOP 的原理都是一样的，只不过形式变了而已。一般我们使用注解的方式使用 AOP 就好了。
- 注解的方式使用 Spring AOP 就了解几个切点表达式，几个增强/通知的注解就完事了，是不是贼简单... 使用 XML 的方式和注解其实没有很大的区别，很快就可以上手啦。
- 引介/引入切面也算是一个比较亮的地方，可以用代理的方式为某个对象实现接口，从而能够使用借口下的方法。这种方式是非侵入式的~
- 要增强的方法还可以接收与被代理方法一样的参数、绑定被代理方法的返回值这些功能...





如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜 **Java3y** 公众号有我的联系方式。更多原创技术文章可关注我的 GitHub：

<https://github.com/ZhongFuCheng3y/3y>