Spring IoC

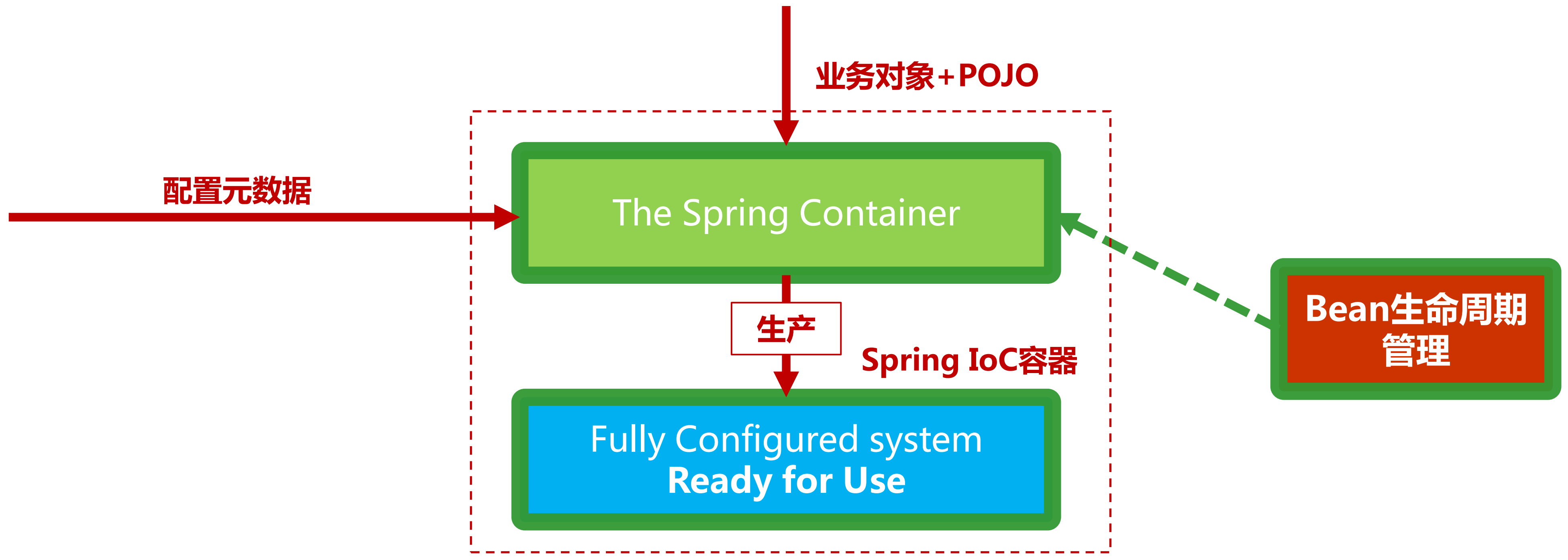
本章目标

- 1、掌握IOC和DI的概念
- 2、掌握Spring IOC的容器创建与使用
- 3、掌握Spring的Bean属性管理
- 4、掌握Spring容器的Bean生命周期管理
- 5、掌握基于注解的容器配置

Spring IoC

- Spring和JavaEE一样是基于容器+组件的管理模式。
- Spring是轻量级容器，EJB是重量级容器。

Spring容器与Bean生命周期管理



Spring容器与Bean生命周期管理

控制反转：Inversion of Control (IoC)
The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name Inversion of Control (IoC)

依赖注入：dependency injection (DI)

IOC概念示例

- 示例：服务层对象A依赖持久层对象B，正常的代码创建顺序是先创建A对象，然后A在调用B对象的时候，再去创建B对象。IOC的创建顺序正好相反，会先创建持久层对象B，再去创建服务层对象A

核心概念:IoC/DI

- 依赖注入的目标：
  - **提升组件重用的概率**
  - 为系统搭建一个**灵活、可扩展的平台**
    - 将USB接口和之前的串/并、PS2接口对比，就能明白其中的意味

Spring IoC

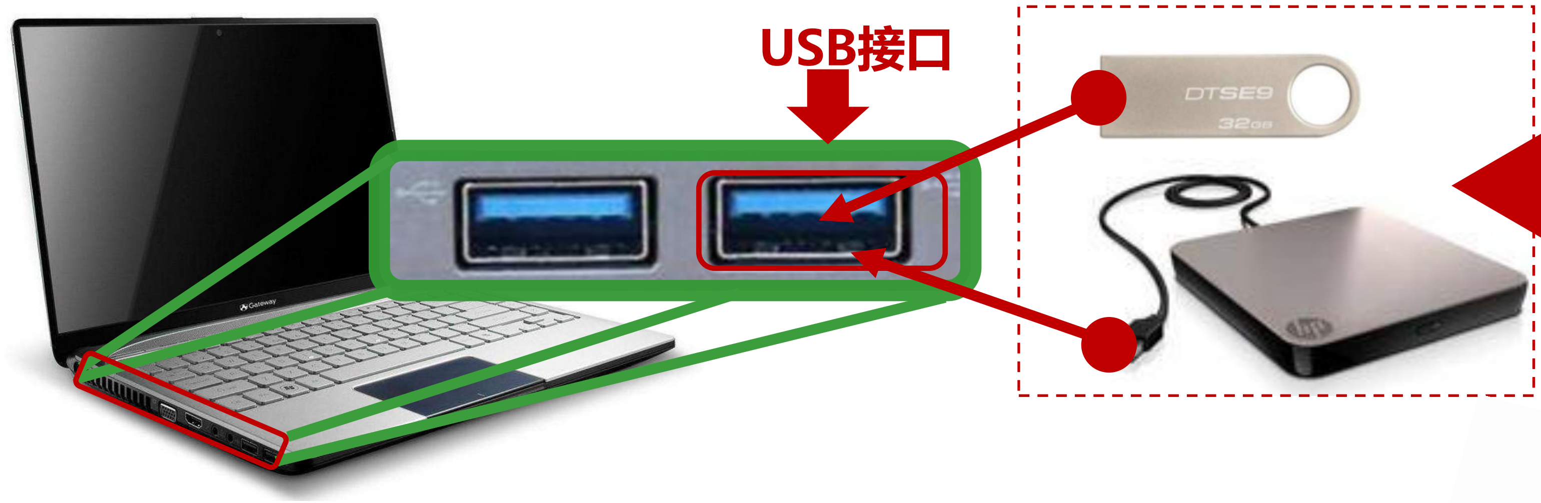
- 为什么要用IoC

选角时：
卖家秀

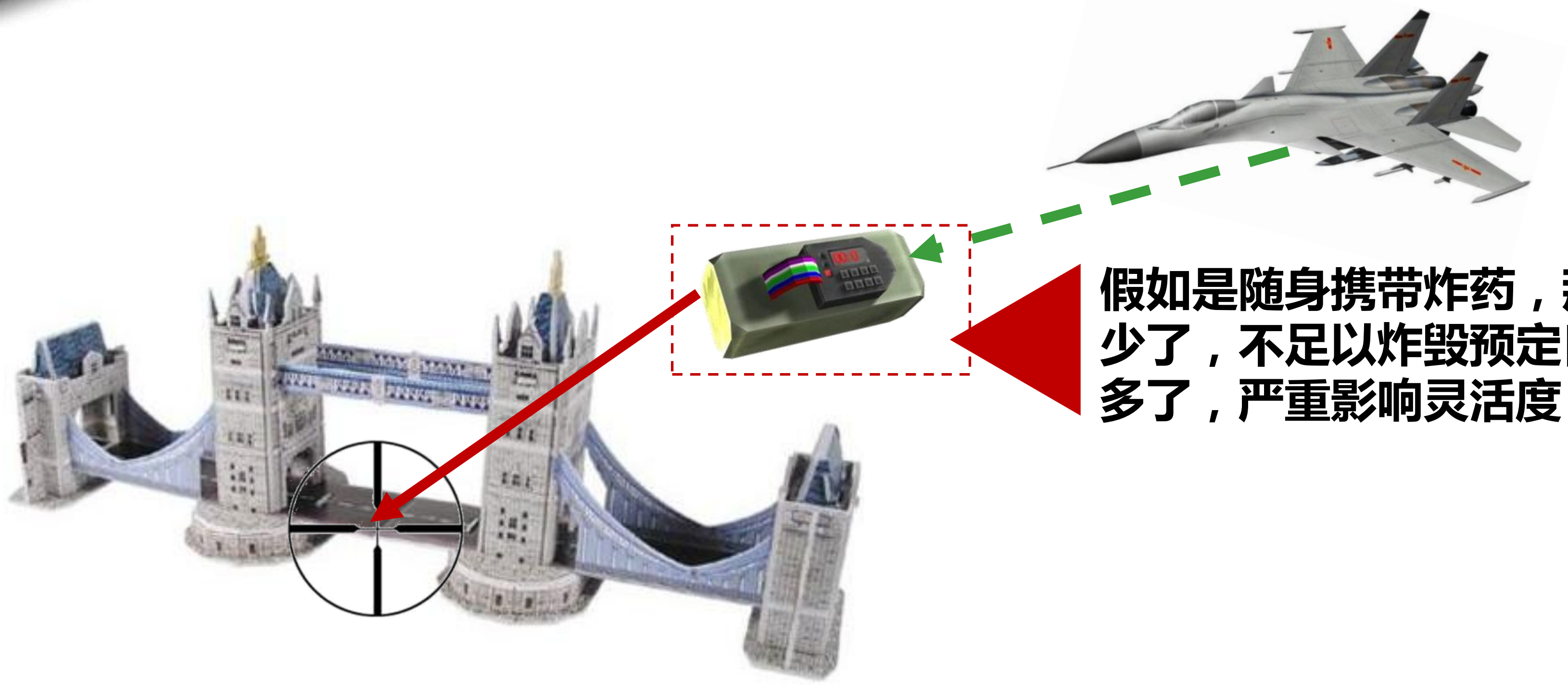
出演时：买家秀，没想到你是这么猥琐的Jerry

如果把角色和单一的演员硬编码绑定，求导演的心理阴影面积...

IoC生活实例（续）



U盘和移动硬盘都支持 USB 接口。当需要将数据复制到外围存储设备时，可以根据情况，选择是保存在U盘还是USB硬盘，下面的操作大家也都轻车熟路，无非接通USB接口，然后在资源浏览器中将选定的文件拖放到指定的盘符



假如是随身携带炸药，那么
少了，不足以炸毁预定目标
多了，严重影响灵活度，容易被发现并消灭

IOC容器与ApplicationContext

- 接口org.springframework.context.ApplicationContext代表了IOC容器，它同时负责配置、实例和装配bean。
- Spring Framework的 IoC 容器操作基于如下两个包：
  - org.springframework.beans 和 org.springframework.context

IOC容器创建

- 有三种方法可以对IOC容器进行创建

配置文件

  - 使用ClassPathXmlApplicationContext创建IOC容器

```
ApplicationContext context = new ClassPathXmlApplicationContext("services.xml", "daos.xml");
```

  - 使用FileSystemXmlApplicationContext创建容器

```
ApplicationContext ctx = new FileSystemXmlApplicationContext("classpath:conf/appContext.xml");
```

  - 使用XmlWebApplicationContext创建容器

```
XmlWebApplicationContext appContext = new XmlWebApplicationContext();
appContext.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");
```

使用容器读取bean

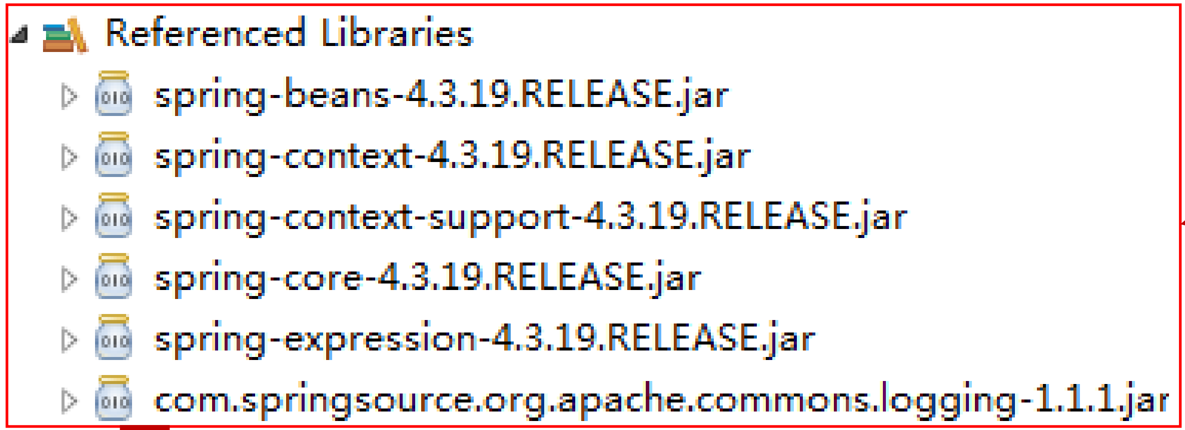
- 从IOC容器中读取Bean对象

获取Bean

```
ApplicationContext context = new
ClassPathXmlApplicationContext("services.xml", "daos.xml");

PetStoreService service = context.getBean("petStore",
PetStoreService.class);

List<String> userList = service.getUsernameList();
```

案例

- Bean声明

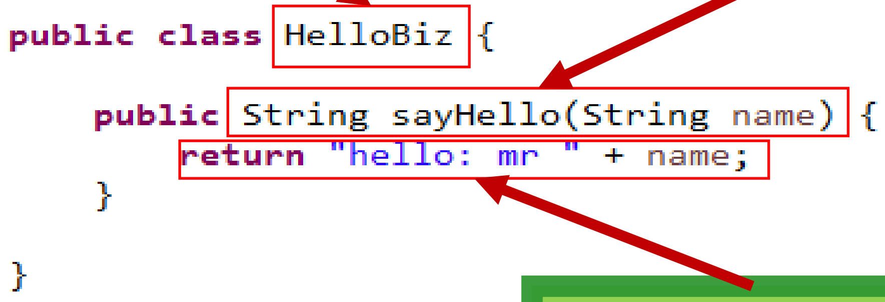


```
Referenced Libraries
  spring-beans-4.3.19.RELEASE.jar
  spring-context-4.3.19.RELEASE.jar
  spring-context-support-4.3.19.RELEASE.jar
  spring-core-4.3.19.RELEASE.jar
  spring-expression-4.3.19.RELEASE.jar
  com.springsource.org.apache.commons.logging-1.1.1.jar
```

导入Spring核心包和依赖包

创建一个HelloBiz类

创建sayHello方法

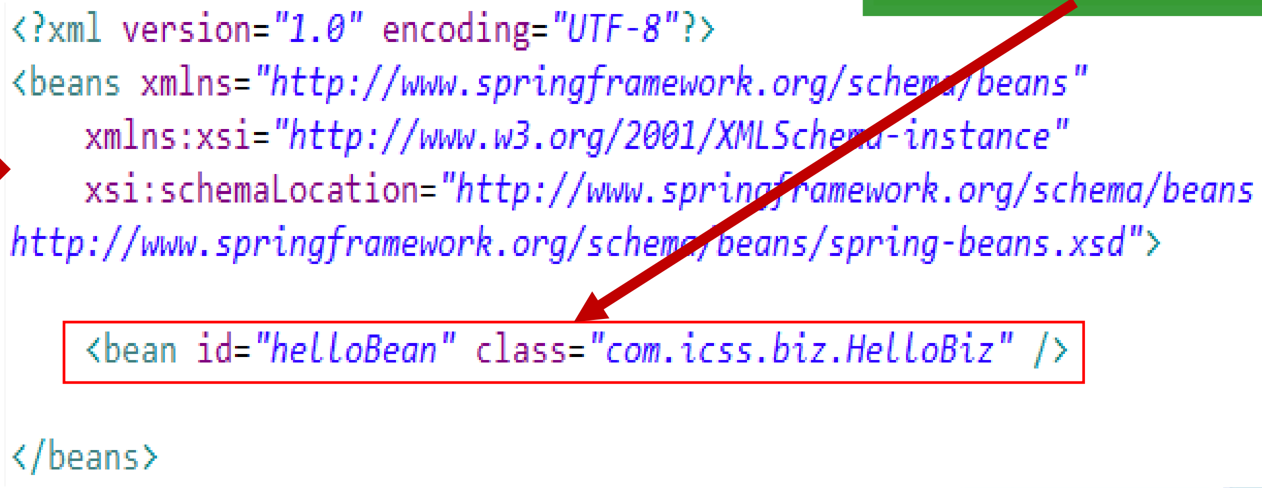


```java
public class HelloBiz {

    public String sayHello(String name) {
        return "hello: mr " + name;
    }

}
```

返回一个字符串

配置声明

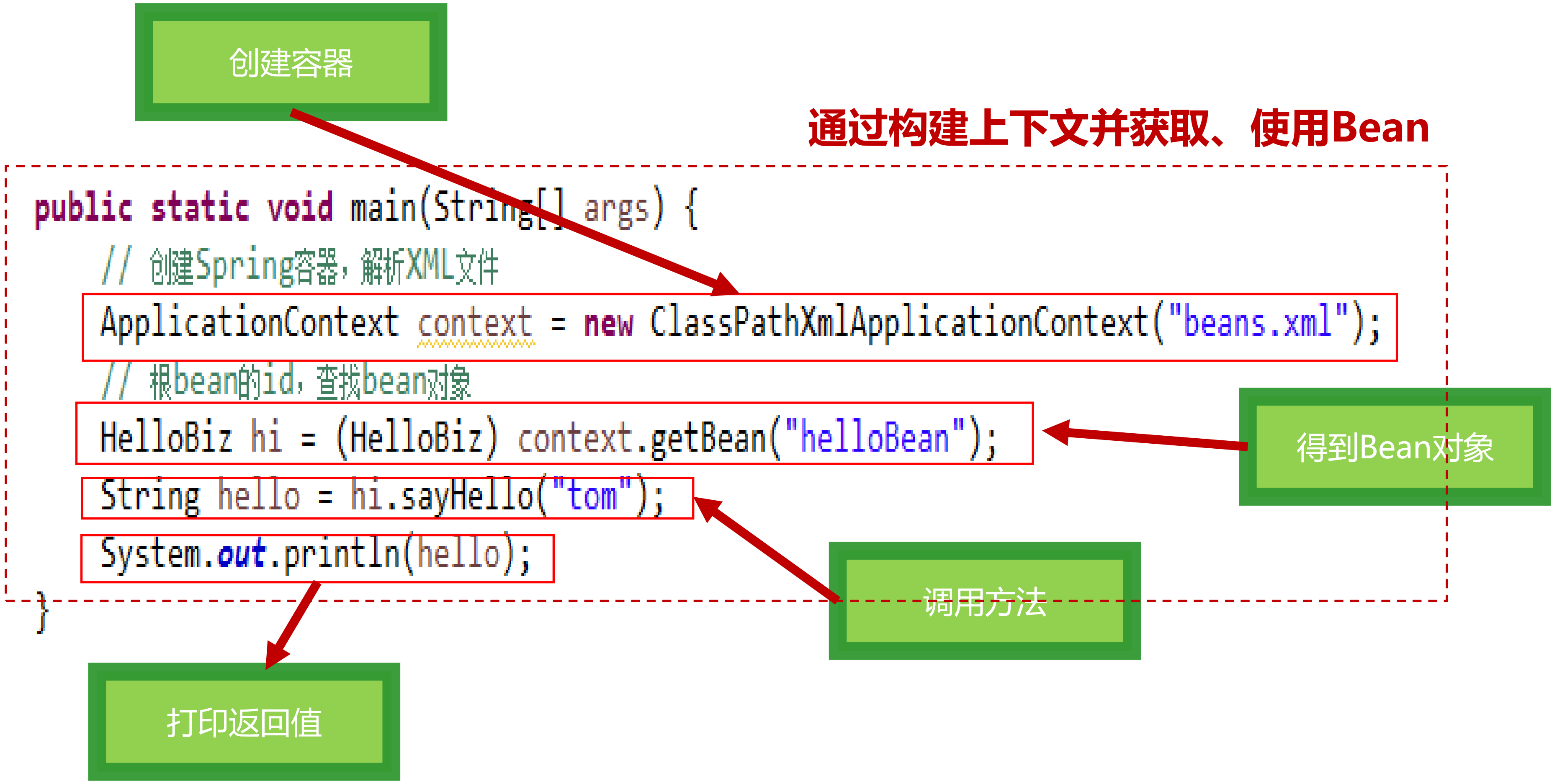
把HellBiz配置成id为helloBean的Bean实例



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

    <bean id="helloBean" class="com.icss.biz.HelloBiz" />

</beans>
```

案例

- 使用：



```
public static void main(String[] args) {
    // 创建Spring容器，解析XML文件
    ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
    // 根bean的id，查找bean对象
    HelloBiz hi = (HelloBiz) context.getBean("helloBean");
    String hello = hi.sayHello("tom");
    System.out.println(hello);
}
```

Spring Bean

- Spring bean的定义

```
<bean id="helloBean" class="com.icss.biz.HelloBiz"/>
```

- <bean>元素的常用属性

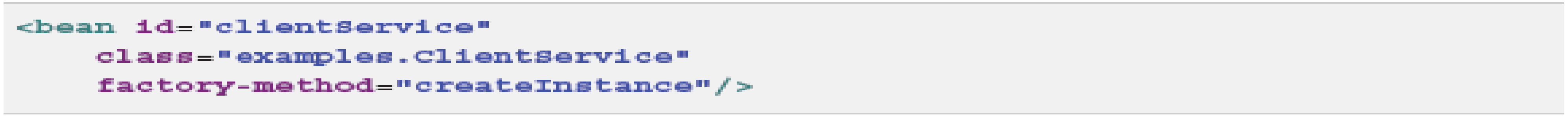
属性名称	描述
class	In Section Called “Instantiating beans”
name	Section 7.3.1, “Naming beans”
scope	Section 7.5, “Bean scopes”
constructor arguments	Section 7.4.1, “Dependency Injection”
properties	Section 7.4.1, “Dependency Injection”
autowiring mode	Section 7.4.5, “Autowiring collaborators”
lazy-initialization mode	Section 7.4.4, “Lazy-initialized beans”
initialization method	the section called “Initialization callbacks”
destruction method	the section called “Destruction callbacks”

Spring Bean

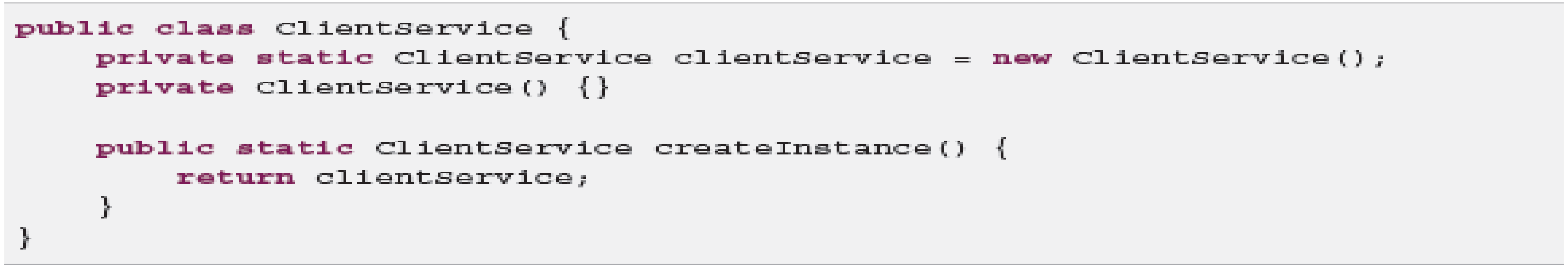
- 创建bean对象
  - 使用默认的构造函数创建bean对象

```
<bean id="exampleBean" class="examples.ExampleBean"/>
<bean name="anotherExample" class="examples.ExampleBeanTwo"/>
```

  - 静态factory-method创建bean对象



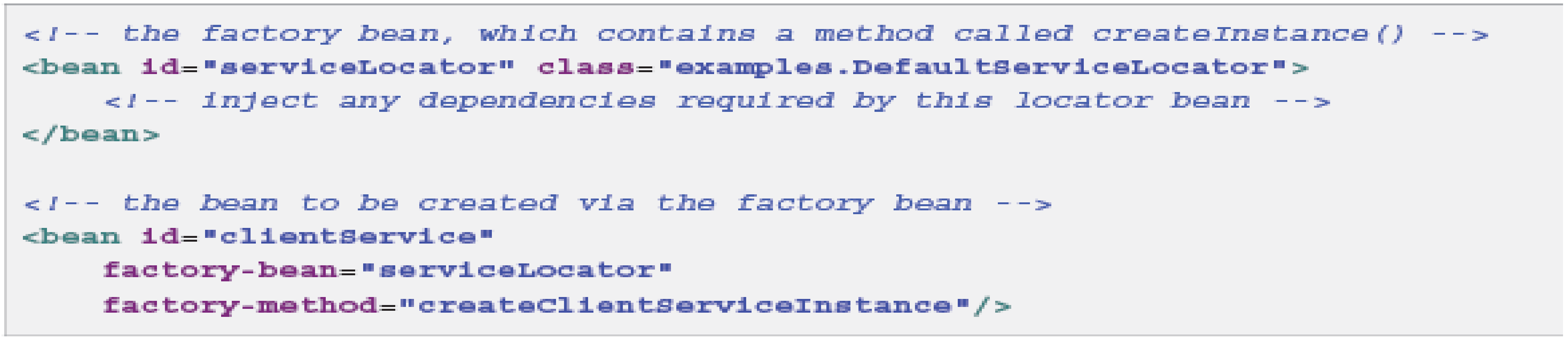
```
<bean id="clientService"
    class="examples.ClientService"
    factory-method="createInstance"/>
```



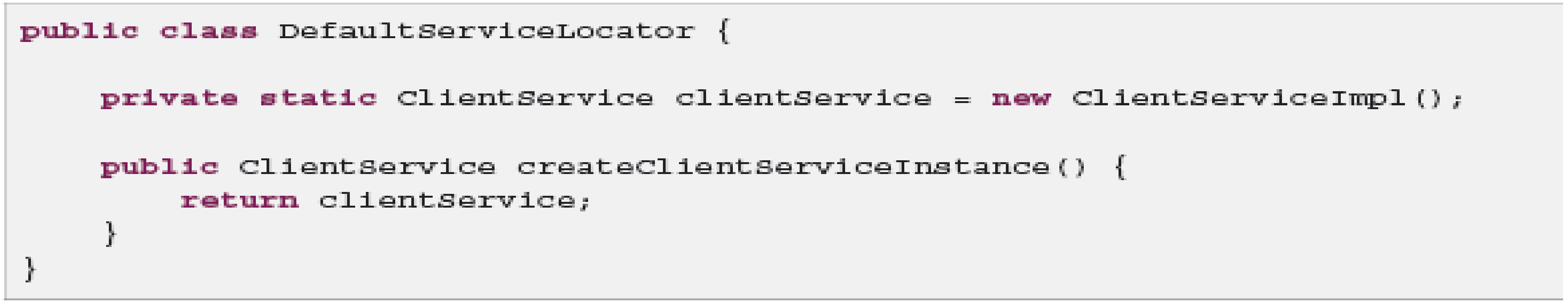
```
public class ClientService {
    private static ClientService clientService = new ClientService();
    private ClientService() {}

    public static ClientService createInstance() {
        return clientService;
    }
}
```

Spring Bean

- 非静态factory-method创建bean对象



```xml
<!-- the factory bean, which contains a method called createInstance() -->
<bean id="serviceLocator" class="examples.DefaultServiceLocator">
    <!-- inject any dependencies required by this locator bean -->
</bean>

<!-- the bean to be created via the factory bean -->
<bean id="clientService"
    factory-bean="serviceLocator"
    factory-method="createClientServiceInstance"/>
```



```java
public class DefaultServiceLocator {

    private static ClientService clientService = new ClientServiceImpl();

    public ClientService createClientServiceInstance() {
        return clientService;
    }
}
```

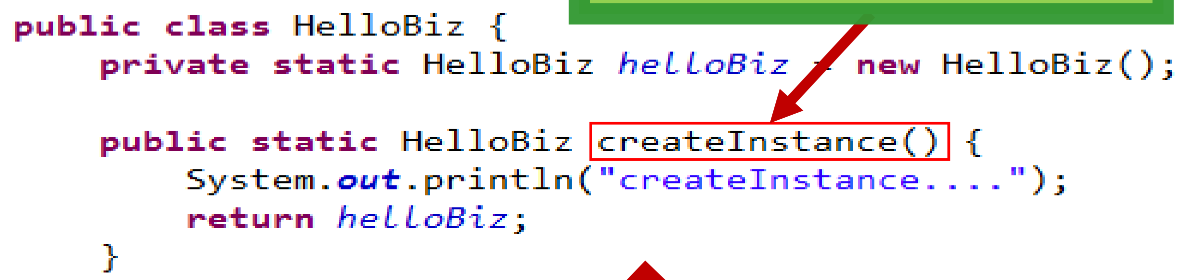
HelloFactoryStatic和HelloFactoryBean案例

- HelloFactoryStatic

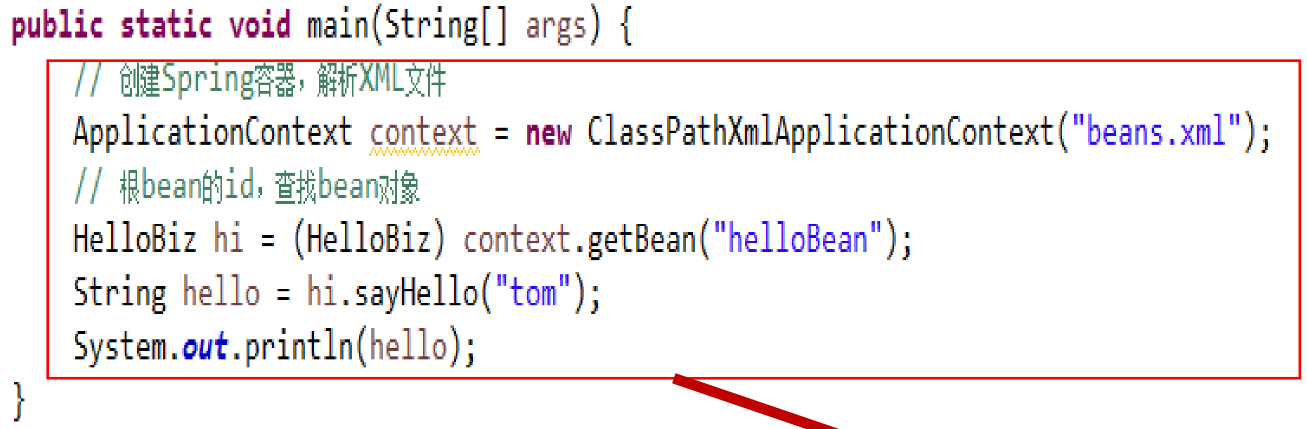
配置bean

```
<bean  id="helloBean" class="com.icss.biz.HelloBiz"  factory-method="createInstance" />
```

静态方法createInstance



```
public class HelloBiz {
    private static HelloBiz helloBiz = new HelloBiz();

    public static HelloBiz createInstance() {
        System.out.println("createInstance....");
        return helloBiz;
    }
```



```
public static void main(String[] args) {
    // 创建Spring容器，解析XML文件
    ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
    // 根bean的id，查找bean对象
    HelloBiz hi = (HelloBiz) context.getBean("helloBean");
    String hello = hi.sayHello("tom");
    System.out.println(hello);
}
```

代码测试

HelloFactoryStatic和HelloFactoryBean案例

- HelloFactoryBean

配置：当使用了factory-bean属性后，即使配置class属性，也不会有效

```
<bean  id="helloBean"
     factory-bean="helloFactory" factory-method="createInstance" />

 <bean id="helloFactory" class="com.icss.biz.HelloFactory"></bean>
```

```
public class HelloFactory {

    public HelloBiz createInstance() {
        System.out.println("HelloFactory-->>createInstance....");
        return new HelloBiz();
    }
```

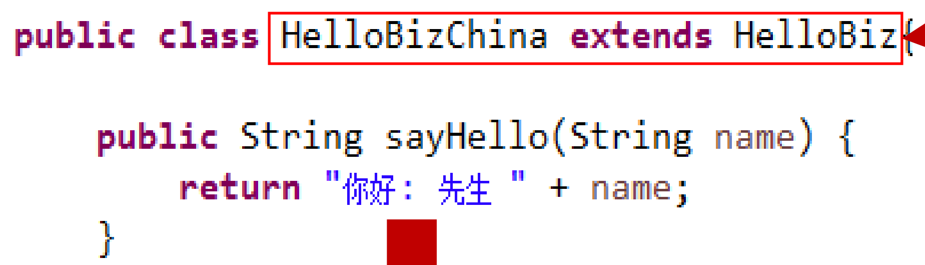
编写工厂类方法

代码测试

```
public static void main(String[] args) {
    // 创建Spring容器，解析XML文件
    ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml")
    // 根bean的id，查找bean对象
    HelloBiz hi = (HelloBiz) context.getBean("helloBean");
    String hello = hi.sayHello("tom");
    System.out.println(hello);
}
```

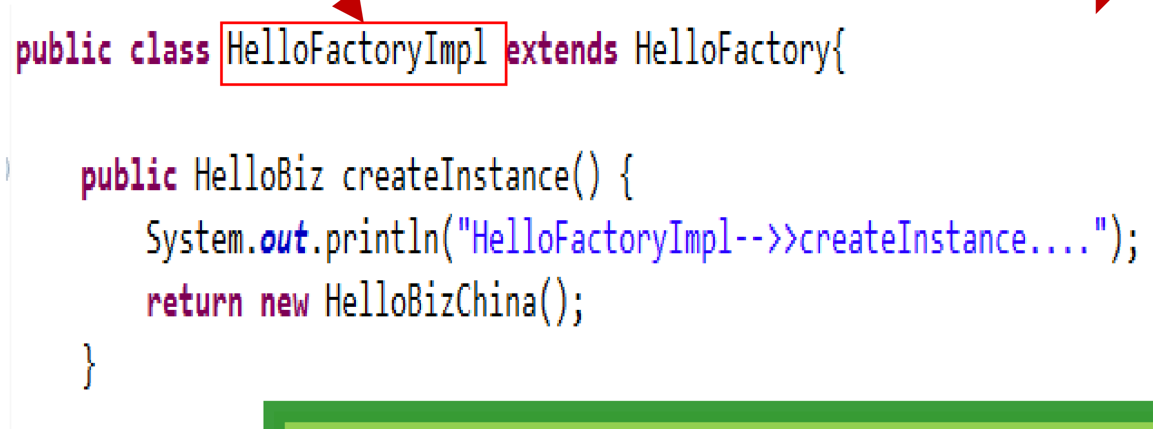
HelloFactoryBean扩展案例

延续之前的HelloFactoryBean案例，新增业务类HellBizChina，它继承HelloBiz



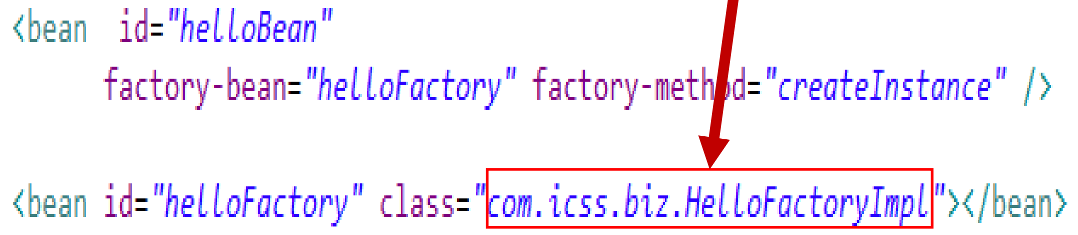
```java
public class HelloBizChina extends HelloBiz{

    public String sayHello(String name) {
        return "你好: 先生 " + name;
    }
```

新增工厂类HelloFactoryImpl

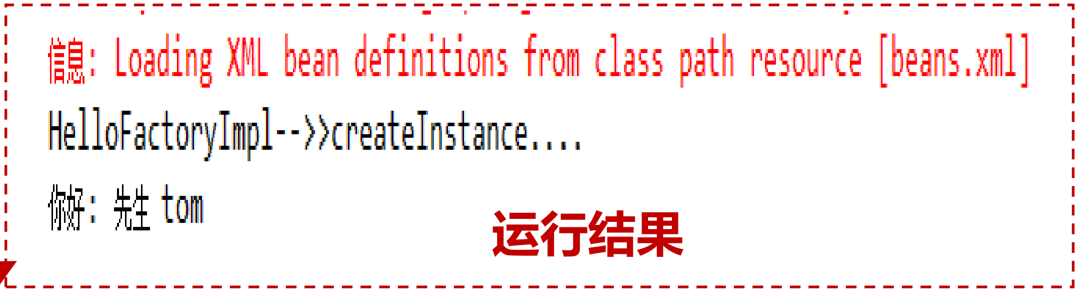


```java
public class HelloFactoryImpl extends HelloFactory{

    public HelloBiz createInstance() {
        System.out.println("HelloFactoryImpl-->>createInstance....");
        return new HelloBizChina();
    }
```

修改配置文件



```xml
<bean  id="helloBean"
      factory-bean="helloFactory" factory-method="createInstance" />

<bean id="helloFactory" class="com.icss.biz.HelloFactoryImpl"></bean>
```



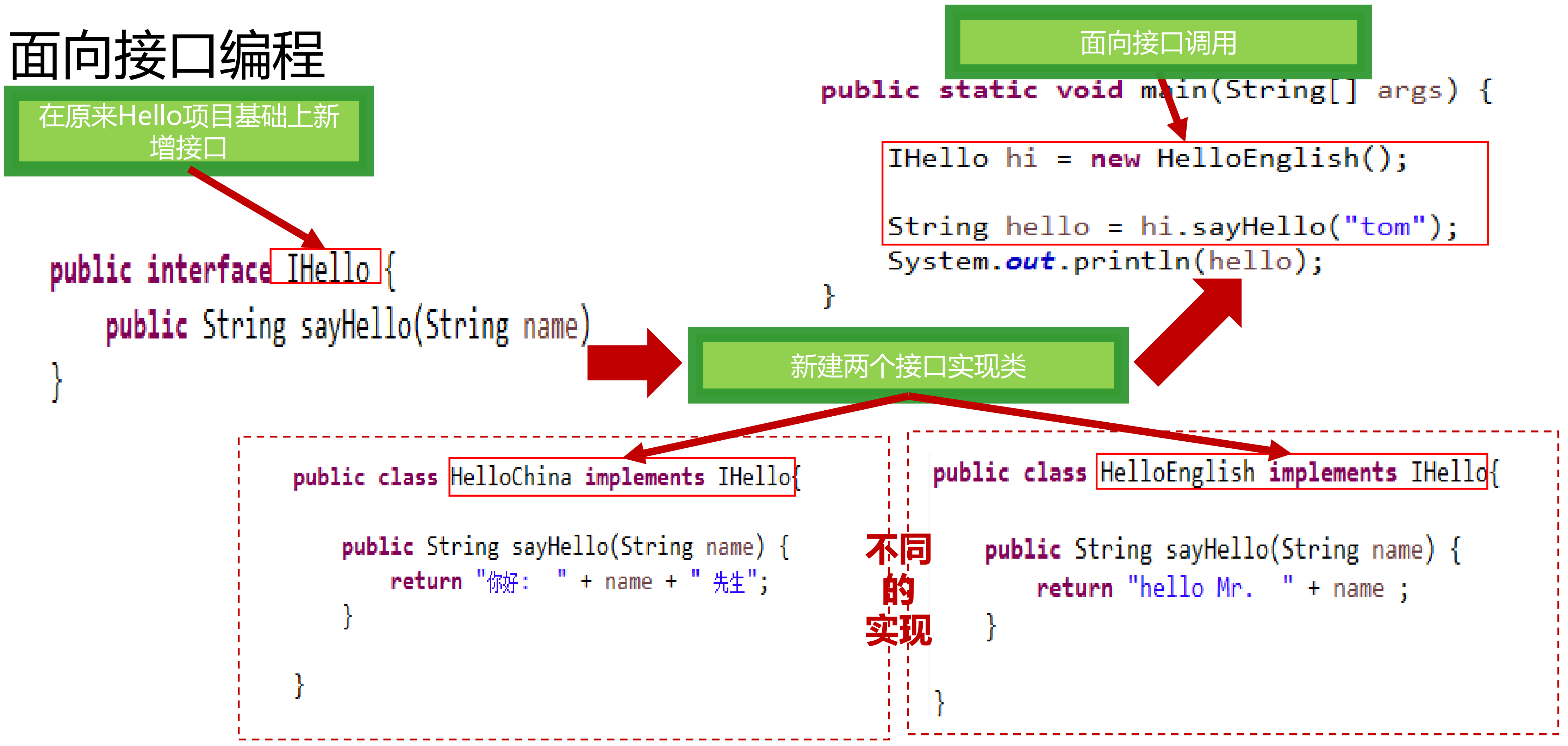
```
信息: Loading XML bean definitions from class path resource [beans.xml]
HelloFactoryImpl-->>createInstance....
你好: 先生 tom
```

运行结果

通过测试证明，Spring的factory-method配置，与设计模式的Fatory Method的思想完全一致

面向接口编程、XML+反射,实现IOC、Spring实现IOC相比较

- 面向接口编程



```java
public interface IHello {
    public String sayHello(String name)
}
```

```java
public static void main(String[] args) {
    IHello hi = new HelloEnglish();

    String hello = hi.sayHello("tom");
    System.out.println(hello);
}
```

```java
public class HelloChina implements IHello{

    public String sayHello(String name) {
        return "你好:  " + name + " 先生";
    }

}
```

```java
public class HelloEnglish implements IHello{

    public String sayHello(String name) {
        return "hello Mr.  " + name ;
    }

}
```

面向接口编程，增加了程序的灵活性和扩展性，但是如何解决硬编码问题？

IHello hi = new HelloEnglish();   //对象new后，无法再改变

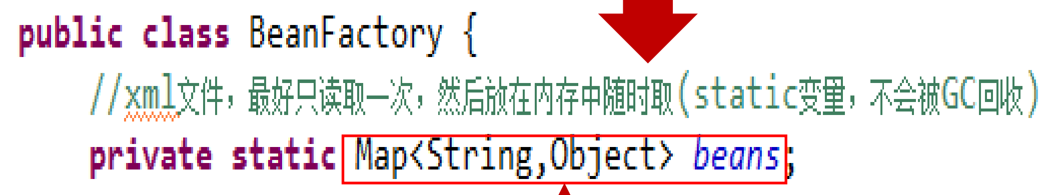
在程序runtime过程中，希望动态改变对象，需要IOC技术！

面向接口编程、XML+反射,实现IOC、Spring实现IOC相比较

- XML+反射,实现IOC

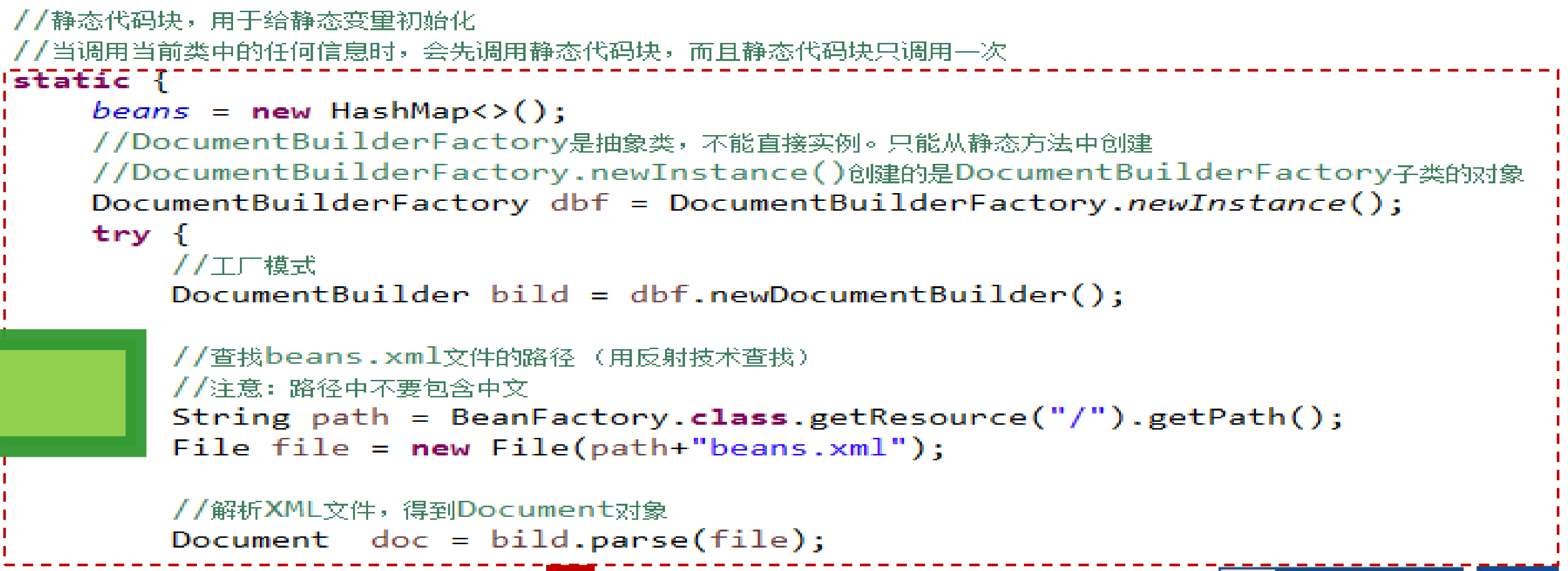
```
<bean  id="helloBean"  class="com.icss.biz.HelloEnglish" />
```

使用xml配置bean



```
public class BeanFactory {
    //xml文件，最好只读取一次，然后放在内存中随时取(static变量，不会被GC回收)
    private static Map<String,Object> beans;
```

定义Map接收所有的Bean对象

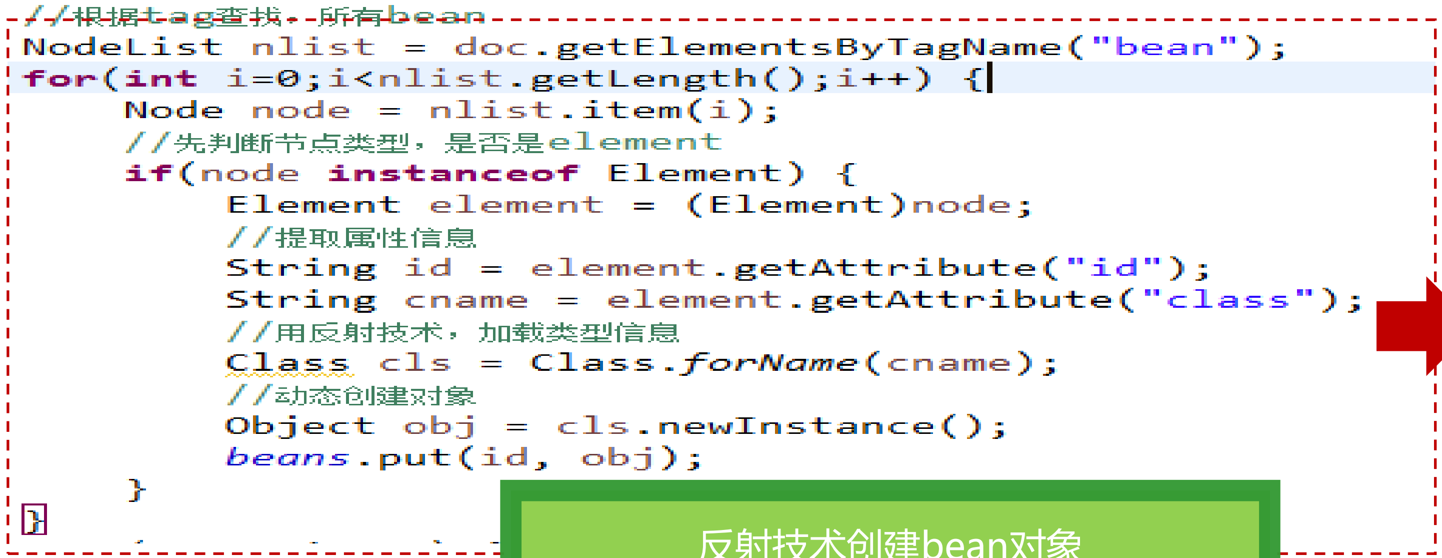


```
//静态代码块，用于给静态变量初始化
//当调用当前类中的任何信息时，会先调用静态代码块，而且静态代码块只调用一次
static {
    beans = new HashMap<>();
    //DocumentBuilderFactory是抽象类，不能直接实例。只能从静态方法中创建
    //DocumentBuilderFactory.newInstance()创建的是DocumentBuilderFactory子类的对象
    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    try {
        //工厂模式
        DocumentBuilder bild = dbf.newDocumentBuilder();

        //查找beans.xml文件的路径 （用反射技术查找）
        //注意：路径中不要包含中文
        String path = BeanFactory.class.getResource("/").getPath();
        File file = new File(path+"beans.xml");

        //解析XML文件，得到Document对象
        Document  doc = bild.parse(file);
```

DOM解析配置文件

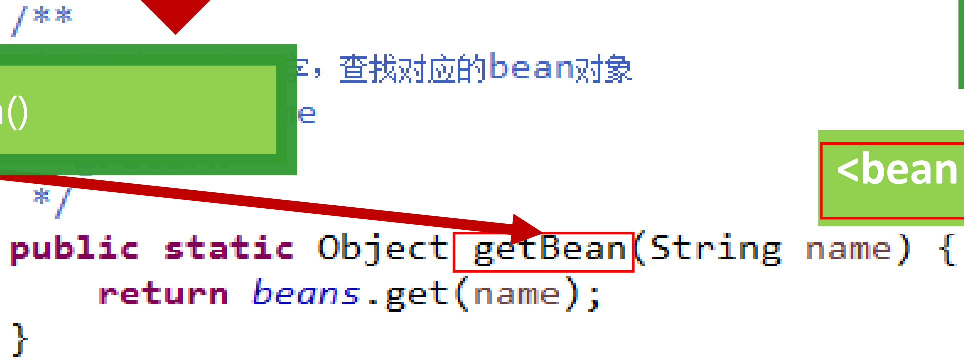
面向接口编程、XML+反射,实现IOC、Spring实现IOC相比较

- XML+反射,实现IOC



```
//根据tag查找，所有bean
NodeList nlist = doc.getElementsByTagName("bean");
for(int i=0;i<nlist.getLength();i++) {
    Node node = nlist.item(i);
    //先判断节点类型，是否是element
    if(node instanceof Element) {
        Element element = (Element)node;
        //提取属性信息
        String id = element.getAttribute("id");
        String cname = element.getAttribute("class");
        //用反射技术，加载类型信息
        Class cls = Class.forName(cname);
        //动态创建对象
        Object obj = cls.newInstance();
        beans.put(id, obj);
    }
}
```

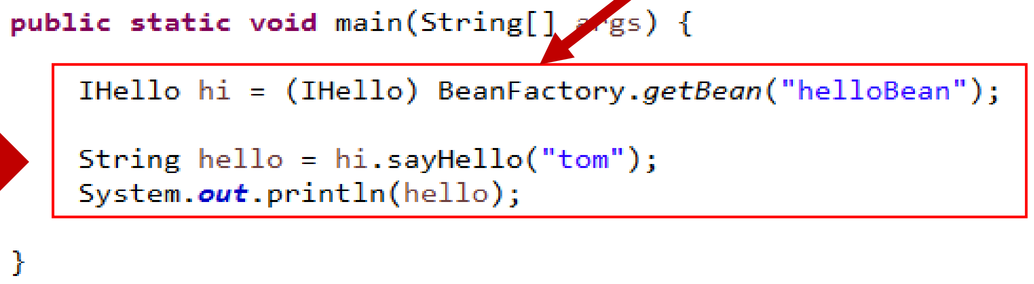
反射技术创建bean对象



```
/**
…，查找对应的bean对象
…e
 */
public static Object getBean(String name) {
    return beans.get(name);
}
```

封装getBean()

代码测试：输出结果：
hello Mr. tom



```
public static void main(String[] args) {
    IHello hi = (IHello) BeanFactory.getBean("helloBean");

    String hello = hi.sayHello("tom");
    System.out.println(hello);
}
```

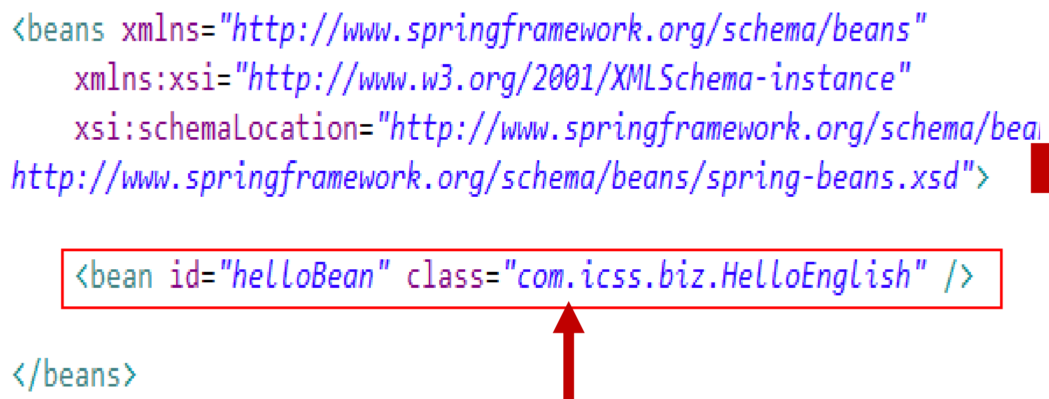
修改配置文件，测试代码不变，输出结果：你好:
tom 先生

<bean id="*helloBean*" class="*com.icss.biz.HelloChina*" />

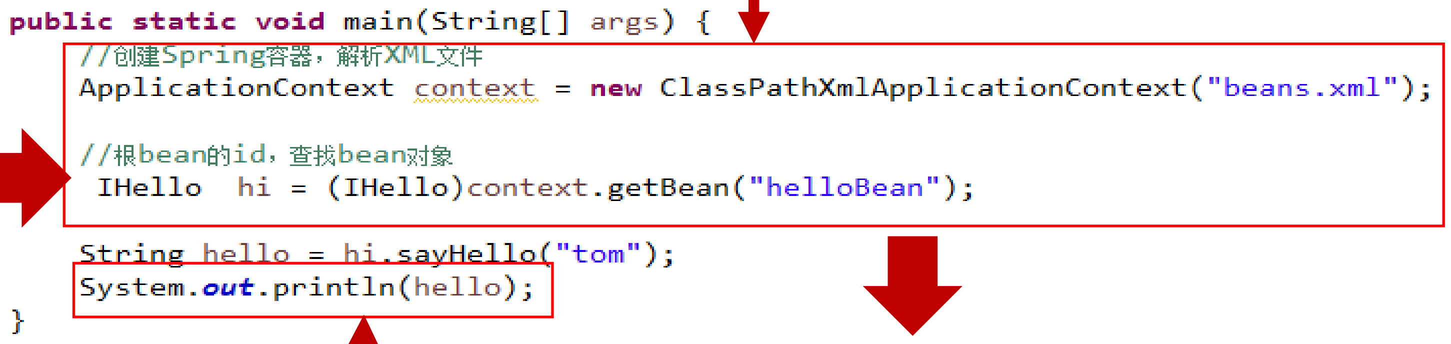
面向接口编程、XML+反射,实现IOC、Spring实现IOC相比较

• Spring实现IOC

实例IOC容器读取bean



```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/bea
http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="helloBean" class="com.icss.biz.HelloEnglish" />

</beans>
```



```
public static void main(String[] args) {
    //创建Spring容器，解析XML文件
    ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");

    //根bean的id，查找bean对象
     IHello  hi = (IHello)context.getBean("helloBean");

    String hello = hi.sayHello("tom");
    System.out.println(hello);
}
```

bean配置

输出结果：hello Mr. tom

修改配置文件，测试代码不变，输出结果：你好: tom 先生

<bean id="helloBean" class="com.icss.biz.HelloChina" />

依赖注入介绍

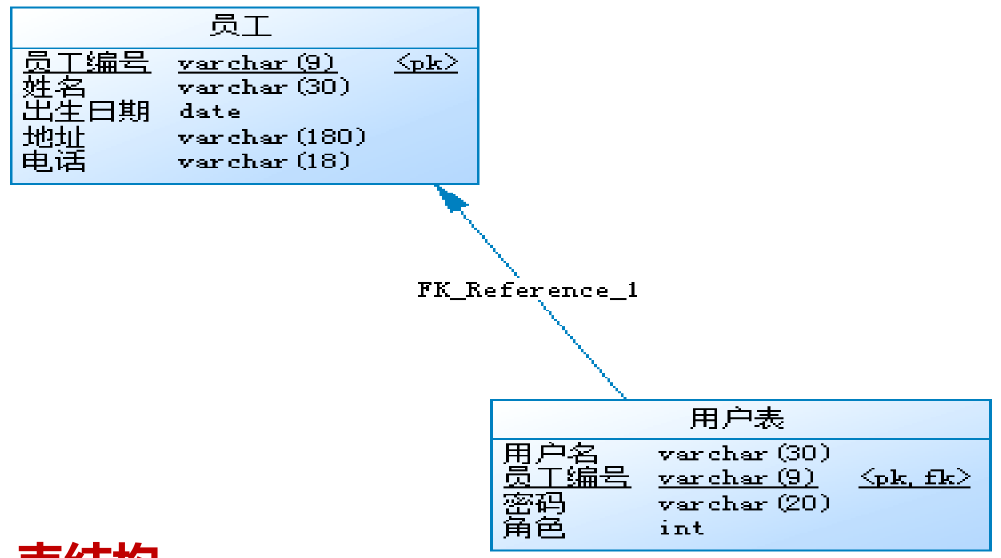
- 一个业务系统，由很多bean对象组成，bean之间存在调用关系，那么这些bean对象是如何协同工作的呢？
- 对象之间的调用，存在依赖关系

StaffUser系统案例

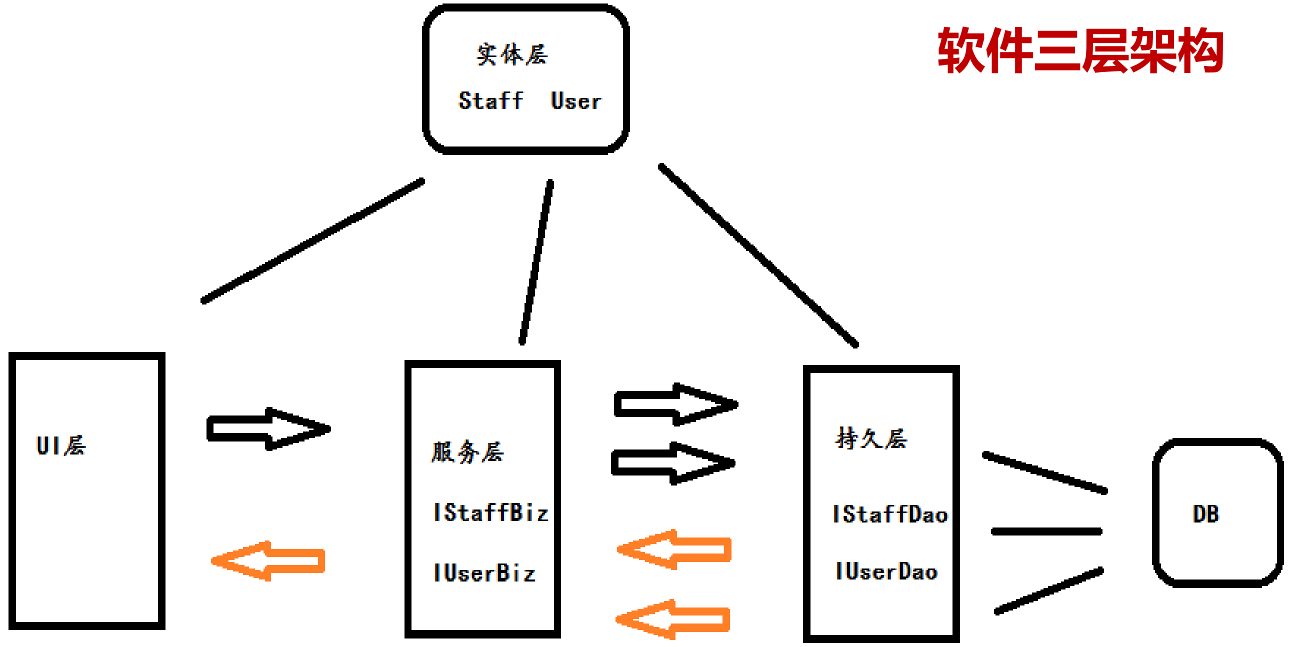
- StaffUser系统需求：
  - StaffUser是大企业的员工管理系统
  - StaffUser也可以看成是学生管理系统
  - 员工或学生的基本信息很多,当把员工基本信息录入系统后，系统需要帮助该员工自动生成一个用户。以后该员工就可以用这个用户登录该系统，做相关操作。
  - 默认用户属性(员工编号、用户名、密码、角色)
  - 用户使用员工编号登录系统，该编号唯一；用户名可以看成是员工的姓名或昵称；角色根据业务制定，默认为普通用户；
  - 首次登录，提示修改密码。初始用户的别名与员工编号一致，后面可以修改。
  - StaffUser要求同时持多种数据库，如oracle和mysql。

StaffUser系统案例

• 案例设计：



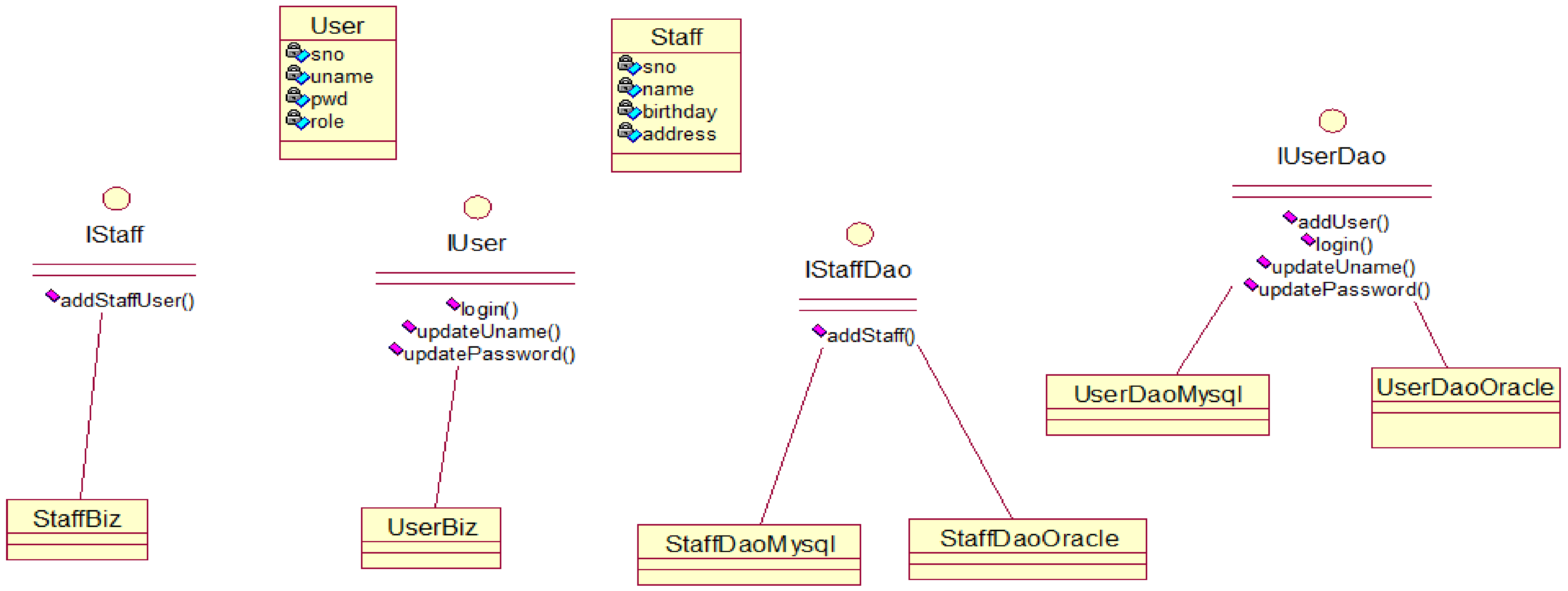
表结构



软件三层架构

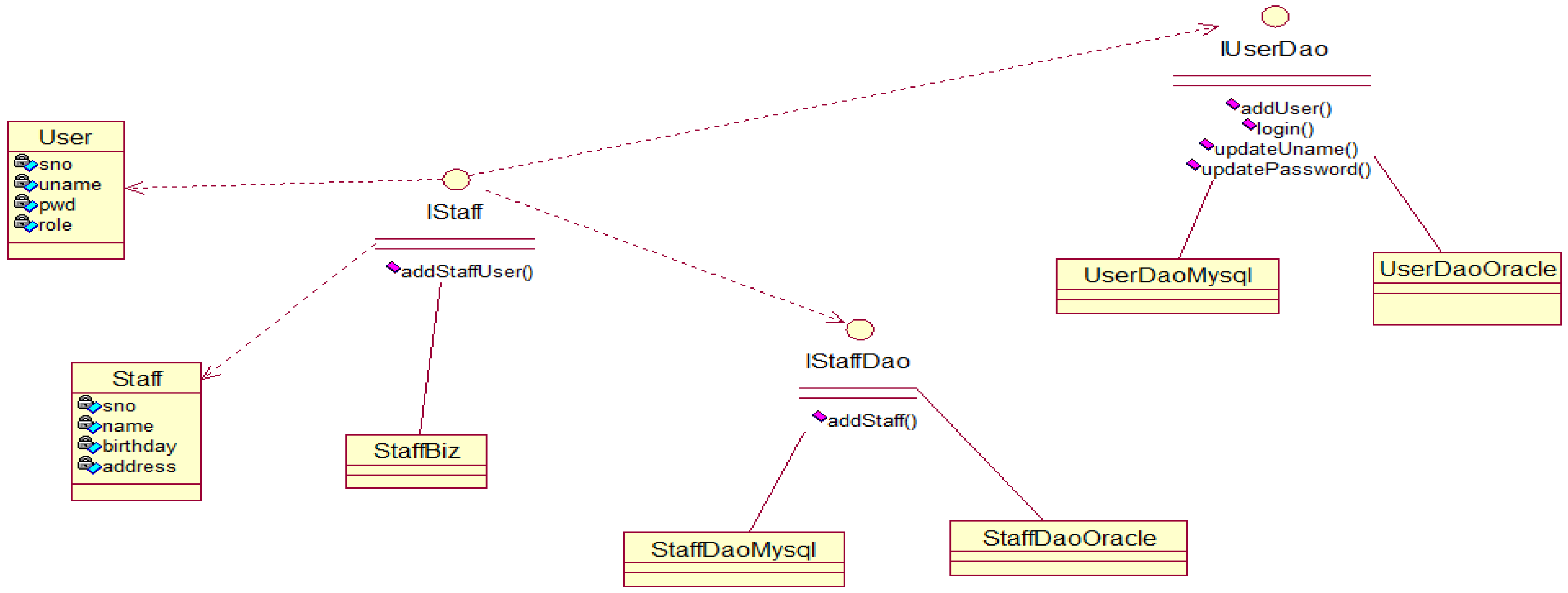
StaffUser系统案例

- StaffUser接口设计：



StaffUser系统案例

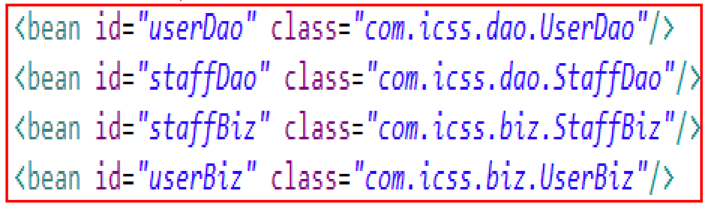
- StaffUser依赖关系：



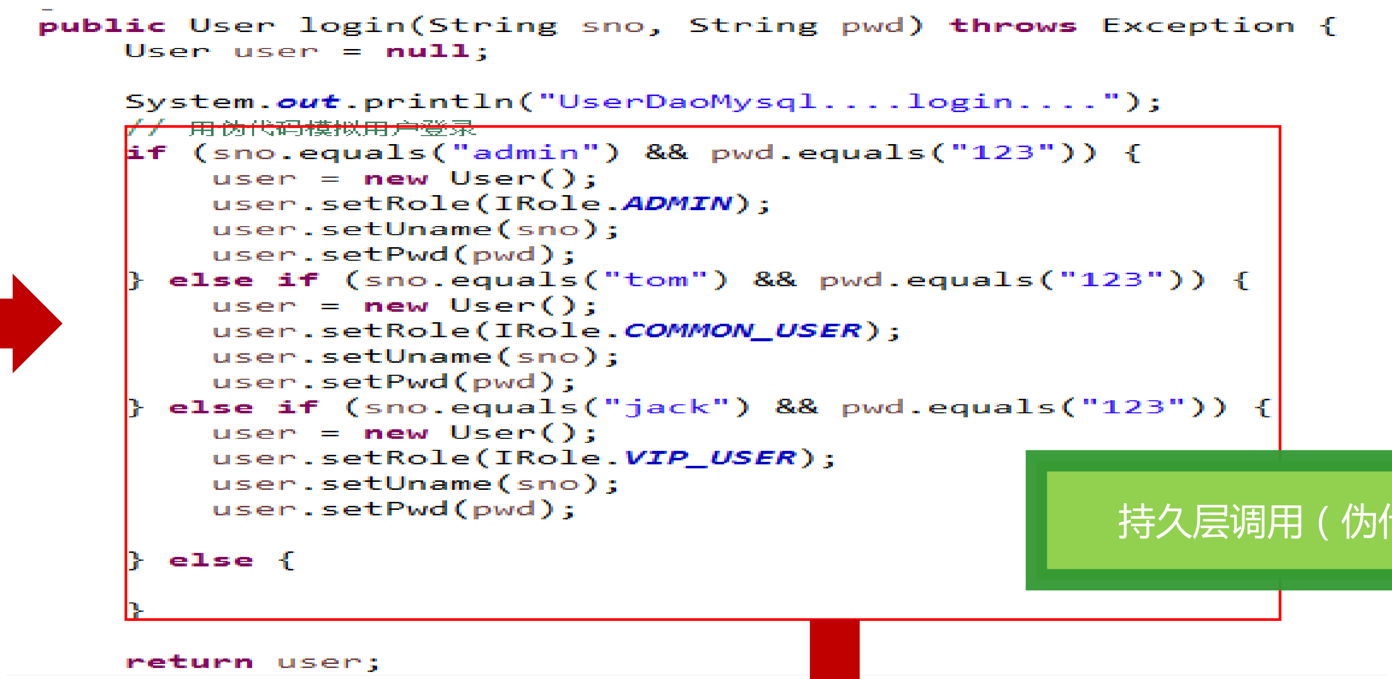
StaffUser系统案例

- StaffUser代码实现

beans.xml配置



```xml
<bean id="userDao" class="com.icss.dao.UserDao"/>
<bean id="staffDao" class="com.icss.dao.StaffDao"/>
<bean id="staffBiz" class="com.icss.biz.StaffBiz"/>
<bean id="userBiz" class="com.icss.biz.UserBiz"/>
```

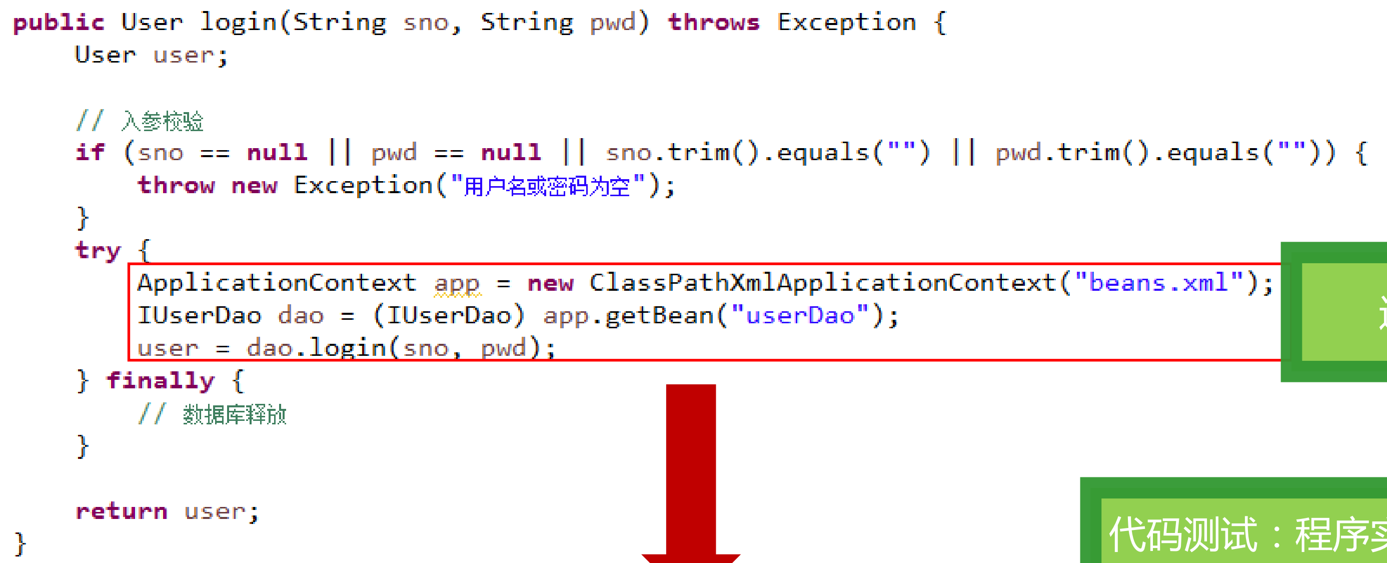


```java
public User login(String sno, String pwd) throws Exception {
    User user = null;

    System.out.println("UserDaoMysql....login....");
    // 用伪代码模拟用户登录
    if (sno.equals("admin") && pwd.equals("123")) {
        user = new User();
        user.setRole(IRole.ADMIN);
        user.setUname(sno);
        user.setPwd(pwd);
    } else if (sno.equals("tom") && pwd.equals("123")) {
        user = new User();
        user.setRole(IRole.COMMON_USER);
        user.setUname(sno);
        user.setPwd(pwd);
    } else if (sno.equals("jack") && pwd.equals("123")) {
        user = new User();
        user.setRole(IRole.VIP_USER);
        user.setUname(sno);
        user.setPwd(pwd);

    } else {

    }

    return user;
```

持久层调用（伪代码）

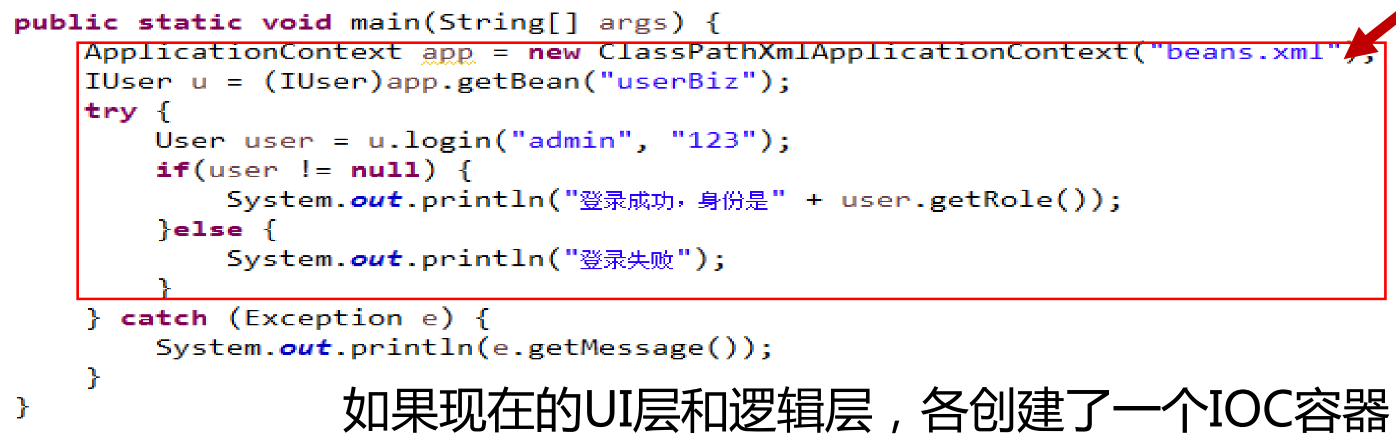
StaffUser系统案例

- StaffUser代码实现



```java
public User login(String sno, String pwd) throws Exception {
    User user;

    // 入参校验
    if (sno == null || pwd == null || sno.trim().equals("") || pwd.trim().equals("")) {
        throw new Exception("用户名或密码为空");
    }
    try {
        ApplicationContext app = new ClassPathXmlApplicationContext("beans.xml");
        IUserDao dao = (IUserDao) app.getBean("userDao");
        user = dao.login(sno, pwd);
    } finally {
        // 数据库释放
    }

    return user;
}
```

逻辑层调用



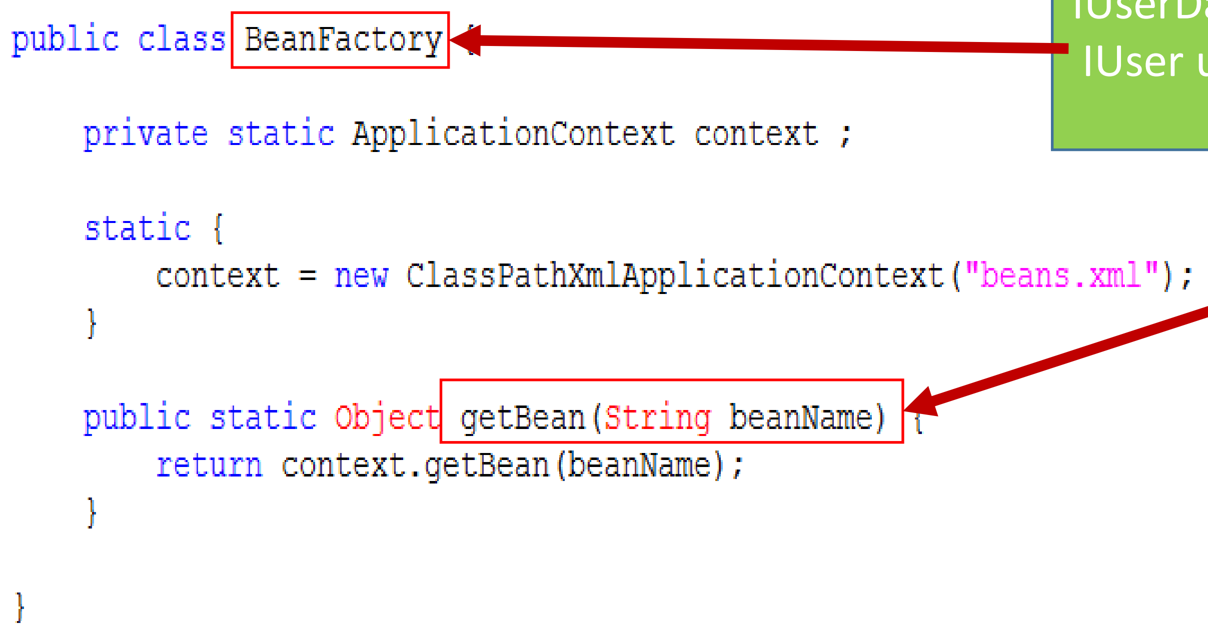
```java
public static void main(String[] args) {
    ApplicationContext app = new ClassPathXmlApplicationContext("beans.xml");
    IUser u = (IUser)app.getBean("userBiz");
    try {
        User user = u.login("admin", "123");
        if(user != null) {
            System.out.println("登录成功，身份是" + user.getRole());
        }else {
            System.out.println("登录失败");
        }
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
```

代码测试：程序实现了IOC，运行正常。

如果现在的UI层和逻辑层，各创建了一个IOC容器，**会存在什么问题？**

StaffUser系统案例

- 单例封装IOC容器



```java
public class BeanFactory {

    private static ApplicationContext context ;

    static {
        context = new ClassPathXmlApplicationContext("beans.xml");
    }

    public static Object getBean(String beanName) {
        return context.getBean(beanName);
    }

}
```

调用：
IUserDao dao = (IUserDao) BeanFactory.*getBean*("userDao");
IUser u = (IUser)BeanFactory.*getBean*("userBiz");

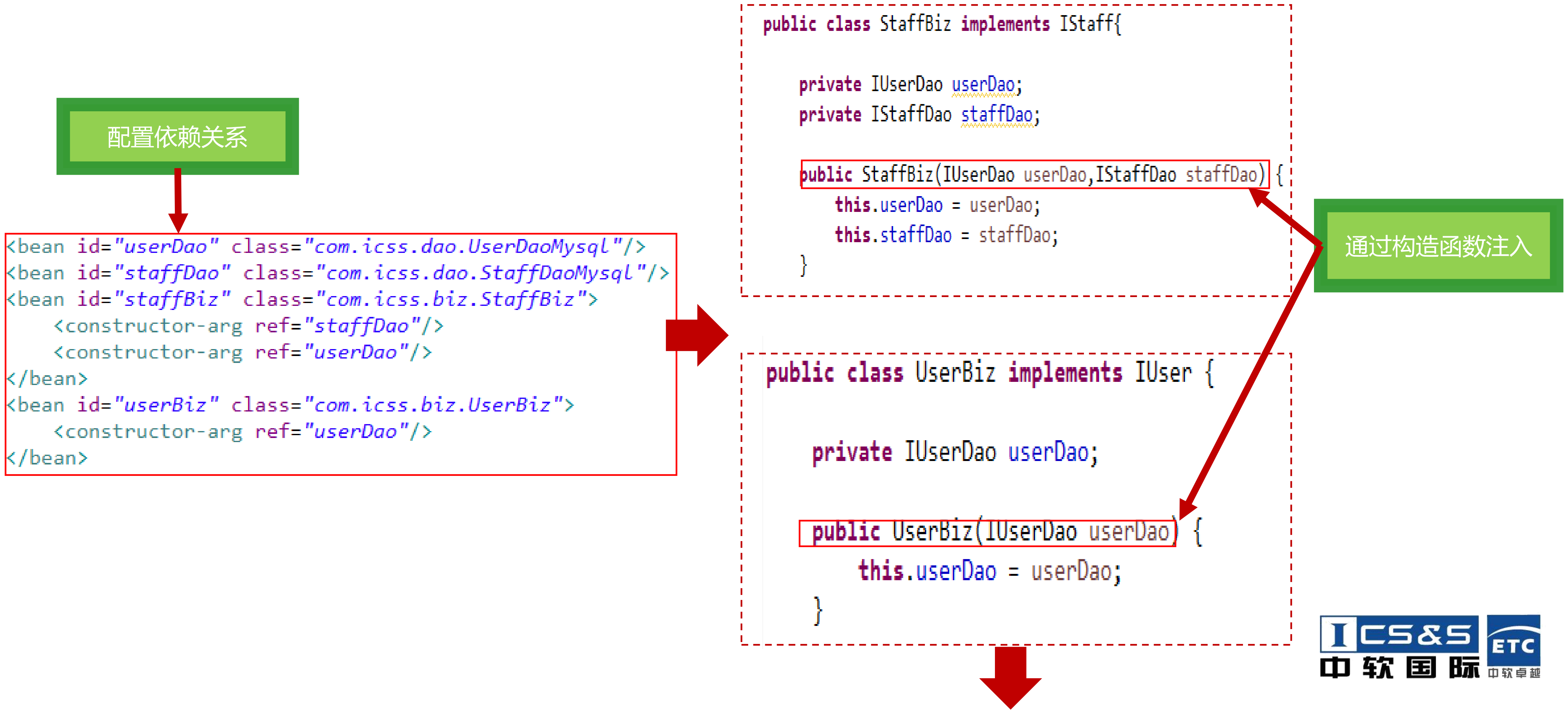
注意：**如果spring配置文件是一个，只需一个IOC容器**

构造器注入

- 构造器注入概念：
  - 在Bean的构造函数中，传入要依赖的其它Bean对象引用或简单类型参数
  - 构造器注入可以明确的表明依赖者与被依赖者的关系

构造器注入

- 构造器注入StaffUser持久层对象案例



```xml
<bean id="userDao" class="com.icss.dao.UserDaoMysql"/>
<bean id="staffDao" class="com.icss.dao.StaffDaoMysql"/>
<bean id="staffBiz" class="com.icss.biz.StaffBiz">
    <constructor-arg ref="staffDao"/>
    <constructor-arg ref="userDao"/>
</bean>
<bean id="userBiz" class="com.icss.biz.UserBiz">
    <constructor-arg ref="userDao"/>
</bean>
```

```java
public class StaffBiz implements IStaff{

    private IUserDao userDao;
    private IStaffDao staffDao;

    public StaffBiz(IUserDao userDao,IStaffDao staffDao) {
        this.userDao = userDao;
        this.staffDao = staffDao;
    }
```

```java
public class UserBiz implements IUser {

    private IUserDao userDao;

    public UserBiz(IUserDao userDao) {
        this.userDao = userDao;
    }
```

构造器注入

- 构造器注入StaffUser持久层对象案例

```
//IUserDao dao = (IUserDao) BeanFactory.getBean("userDao");
 user = userDao.login(sno, pwd);
```

代码调用： UserBiz中使用的持久层对象，不再从BeanFactory中查找，而是直接使用DI过来的userDao

```
IUser u = (IUser)BeanFactory.getBean("userBiz");
 User user = u.login("admin", "123");
```

代码调用： LoginTest中的逻辑对象，没有使用DI，因此仍旧从BeanFactory中提取

构造器注入

- 简单类型注入

前面使用的是bean对象的引用注入，bean的类型信息，Spring容器知道
对于非bean引用的简单类型，Spring容器无法知道，因此需要显示指定类型后注入

```xml
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg name="years" value="7500000" />
    <constructor-arg name="ultimateAnswer" value="42" />
</bean>
```

直接使用name赋值，也可以解决type赋值冲突的问题

```xml
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg type="int" value="7500000" />
    <constructor-arg type="java.lang.String" value="42" />
</bean>
```

```java
public class ExampleBean {

    // Number of years to calculate the Ultimate Answer
    private int years;

    // The Answer to Life, the Universe, and Everything
    private String ultimateAnswer;

    public ExampleBean(int years, String ultimateAnswer) {
        this.years = years;
        this.ultimateAnswer = ultimateAnswer;
    }
}
```

这个类的构造函数的参数为: int 和 String。
需要在配置文件中显示指定类型

```xml
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg index="0" value="7500000" />
    <constructor-arg index="1" value="42" />
</bean>
```

当构造函数的多个入参，使用的是相同类型时，按照type赋值的方式会出现问题。
因此就需要使用index，按照参数顺序给构造函数赋值

提问：**实体类也是引用类型，作为参数怎么注入？**

构造器注入

• 构造器注入User对象案例

```java
public class User {
    private String uname;
    private String  sno;
    private String   pwd;
    private int     role;

    public User() {

    }

    public User(String uname,String sno,String pwd,int role) {
        this.uname = uname;
        this.sno = sno;
        this.pwd = pwd;
        this.role = role;
    }
```

User类的构造函数

配置构造器注入

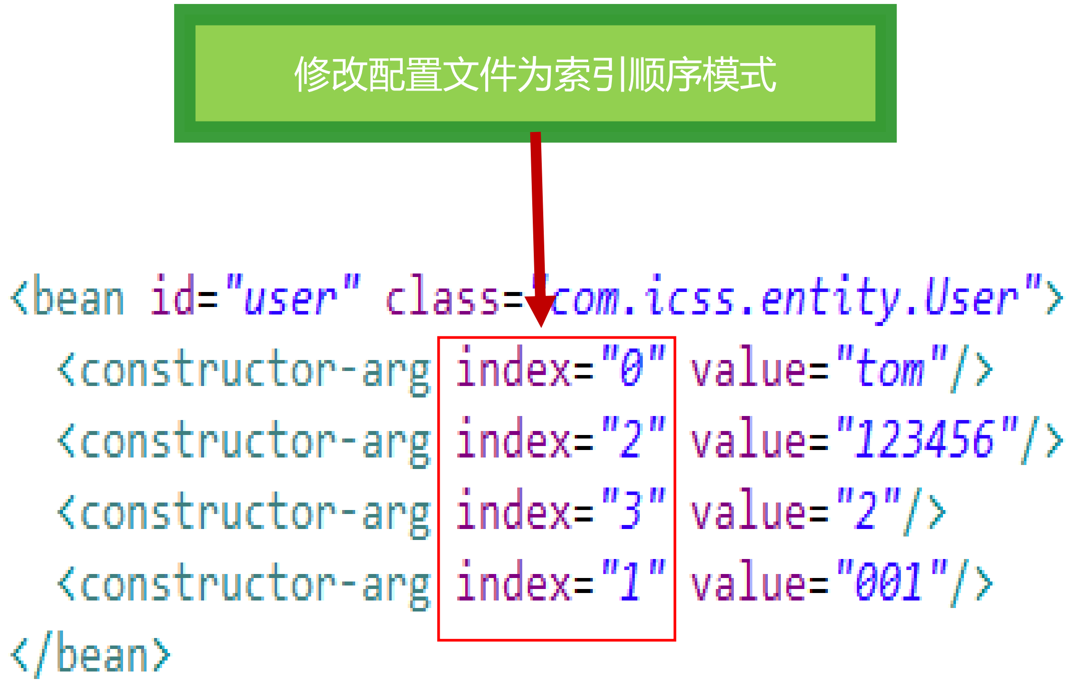
```xml
<bean id="user" class="com.icss.entity.User">
  <constructor-arg name="uname" value="tom"/>
  <constructor-arg name="pwd" value="123456"/>
  <constructor-arg name="role" value="2"/>
  <constructor-arg name="sno" value="001"/>
</bean>
```

```java
User user = (User)BeanFactory.getBean("user")
System.out.println(user.getSno());
System.out.println(user.getUname());
System.out.println(user.getPwd());
System.out.println(user.getRole());
```

```
信息: Loading XML bean definitions from class path resource [beans.xml]
001
tom
123456
2
```

构造器注入

- 构造器注入User对象案例



```
<bean id="user" class="com.icss.entity.User">
   <constructor-arg index="0" value="tom"/>
   <constructor-arg index="2" value="123456"/>
   <constructor-arg index="3" value="2"/>
   <constructor-arg index="1" value="001"/>
</bean>
```

Set方法注入

- Set注入StaffUser持久层对象案例

配置依赖关系

```xml
<bean id="userDao" class="com.icss.dao.UserDaoMysql"/>
<bean id="staffDao" class="com.icss.dao.StaffDaoMysql"/>

<bean id="staffBiz" class="com.icss.biz.StaffBiz">
   <property name="userDao" ref="userDao"></property>
   <property name="staffDao" ref="staffDao"></property>
</bean>
<bean id="userBiz" class="com.icss.biz.UserBiz">
    <property name="userDao" ref="userDao"></property>
</bean>
```

```java
public class StaffBiz implements IStaff{

    private IUserDao userDao;
    private IStaffDao staffDao;

    public void setUserDao(IUserDao userDao) {
        this.userDao = userDao;
    }

    public void setStaffDao(IStaffDao staffDao) {
        this.staffDao = staffDao;
    }
```

```java
public class UserBiz implements IUser {

    private IUserDao userDao;

    public void setUserDao(IUserDao userDao) {
        this.userDao = userDao;
    }
```

通过Set方法注入

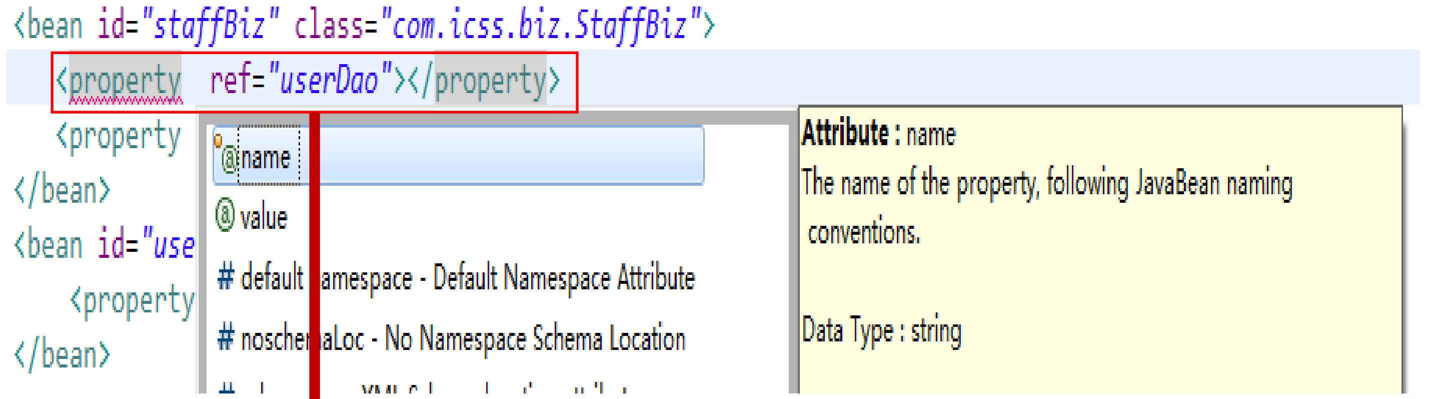
Set方法注入

- property配置说明

配置文件不变，UserBiz中的属性名修改为userDao2，参数名修改为userDao3

```
public class UserBiz implements IUser {
    private IUserDao userDao2;

    public void setUserDao(IUserDao userDao3) {
        this.userDao2 = userDao3;
    }
```

测试结果正常，并没有出错



```
<bean id="staffBiz" class="com.icss.biz.StaffBiz">
    <property  ref="userDao"></property>
    <property
</bean>
<bean id="use
     <property
</bean>
```

- @name
- @ value
- # default namespace - Default Namespace Attribute
- # noschemaLoc - No Namespace Schema Location

Attribute : name
The name of the property, following JavaBean naming conventions.

Data Type : string

property元素，如果没有name属性会报错

Set方法注入

修改配置文件如下

```
<bean id="userBiz" class="com.icss.biz.UserBiz">
    <property name="userDao2" ref="userDao"></property>
</bean>
```

测试出现异常

```
Caused by: org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'userBiz' defined in class path

 Invalid property 'userDao2' of bean class [com.icss.biz.UserBiz]: Bean property 'userDao2' is not writable or has an invalid setter
```

修改set方法后OK

```
public class UserBiz implements IUser {

    private IUserDao userDao;

    public void setUserDao2(IUserDao userDao) {
        this.userDao = userDao;
    }
```

```
<bean id="userBiz" class="com.icss.biz.UserBiz">
    <property name="userDao2" ref="userDao"></property>
</bean>
```

总结：

配置文件中的property的name，与set方法的名字对应不是与UserBiz中的userDao属性对应

Set方法注入

- Set注入与构造器注入对比
  - Set注入模式代码更加简洁
  - 构造器注入对依赖关系的表达更加清楚
  - Set注入可以避免循环依赖问题

依赖配置参数

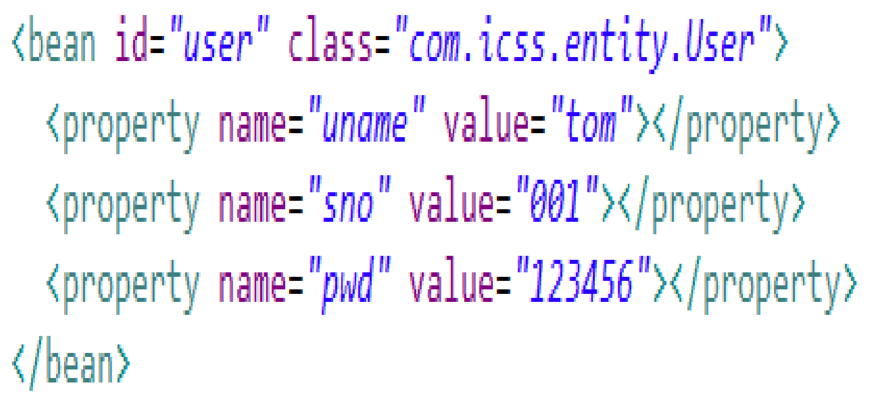
- 给Bean的属性或构造体参数直接赋值
  - 用property直接赋值

```xml
<bean id="myDataSource"
    class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName"
        value="com.mysql.jdbc.Driver" />
    <property name="url" value="jdbc:mysql://localhost:3306/mydb" />
    <property name="username" value="root" />
    <property name="password" value="masterkaoli" />
</bean>
```

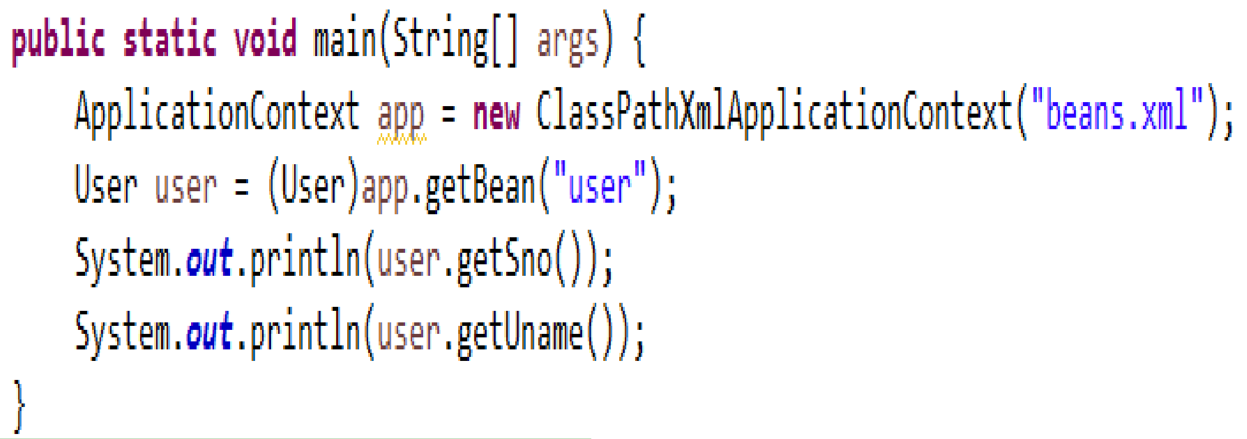
注意：对比set注入中的示例，property的值对应的是set方法，不是属性。property如何转换，靠的是Spring的转换服务机制。
Spring’s conversion service is used to convert these values from a String to the actual type of the property or argument。

依赖配置参数

- property给User属性赋值案例



```xml
<bean id="user" class="com.icss.entity.User">
  <property name="uname" value="tom"></property>
  <property name="sno" value="001"></property>
  <property name="pwd" value="123456"></property>
</bean>
```



```java
public static void main(String[] args) {
    ApplicationContext app = new ClassPathXmlApplicationContext("beans.xml");
    User user = (User)app.getBean("user");
    System.out.println(user.getSno());
    System.out.println(user.getUname());
}
```

把实体类User配置成bean，并用property赋值

依赖配置参数

- property给集合属性赋值

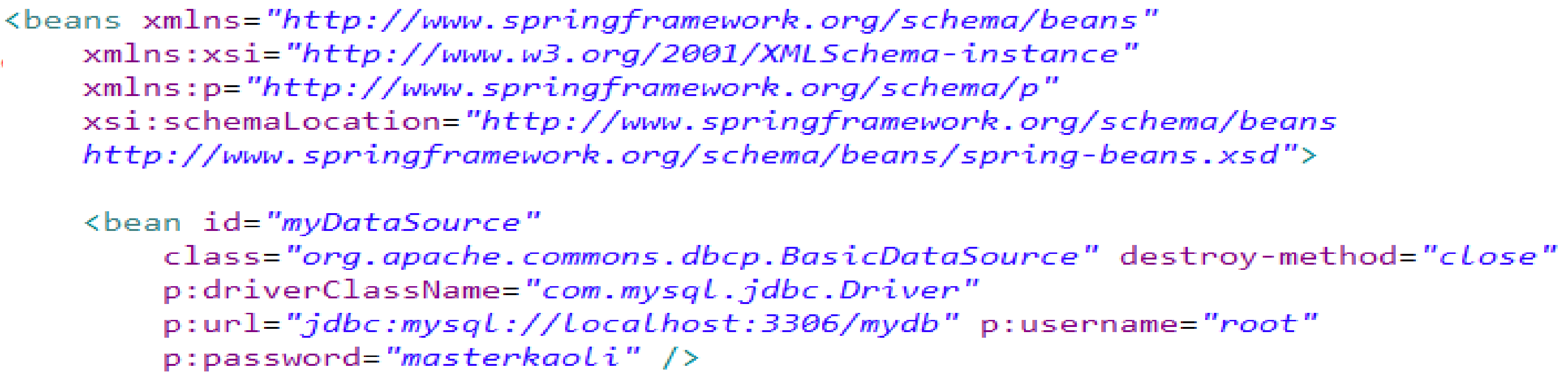
```java
public class EmailService implements ApplicationEventPublisherAware{

    private ApplicationEventPublisher publisher;
    private List<String> blackList;


    public void setBlackList(List<String> blackList) {
        this.blackList = blackList;
    }
```

集合属性赋值

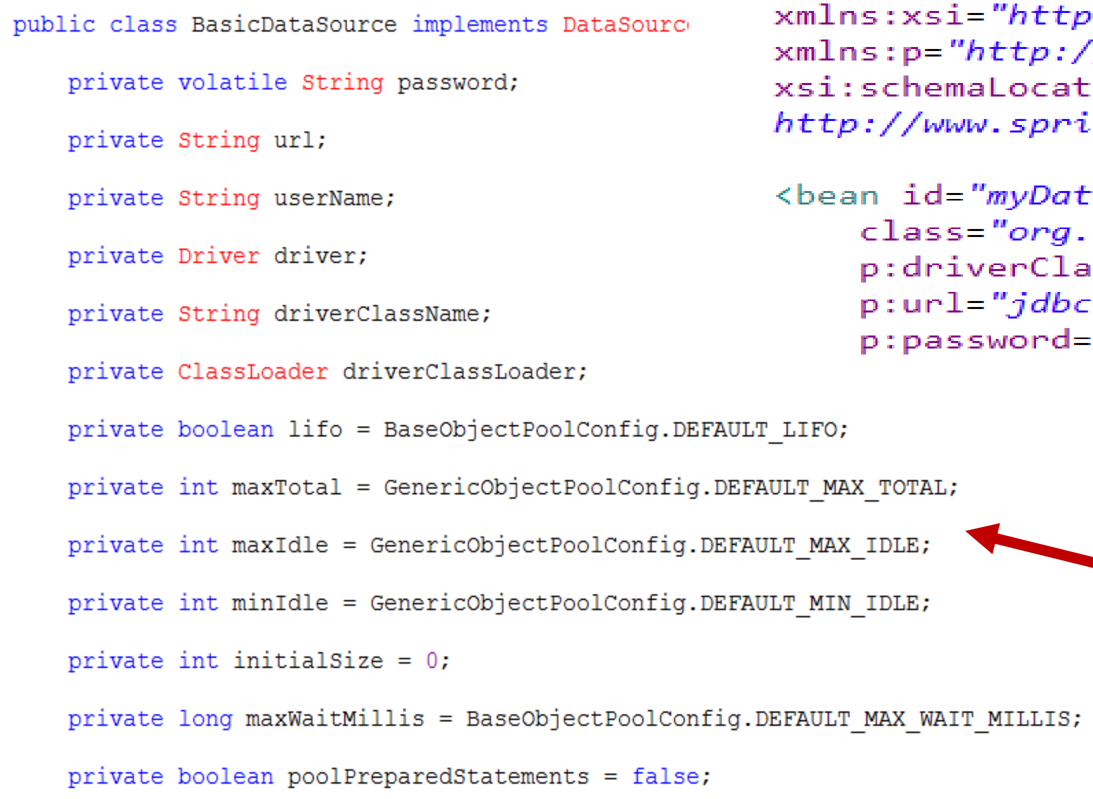
```xml
<bean id="emailService" class="com.icss.biz.EmailService">
     <property name="blackList">
       <list>
         <value>tom1@qq.com</value>
         <value>tom2@qq.com</value>
         <value>tom3@qq.com</value>
         <value>jack1@qq.com</value>
         <value>jack2@qq.com</value>
       </list>
     </property>
</bean>
```

依赖配置参数

- 用p:namespace直接赋值



```xml
<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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="myDataSource"
        class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close"
        p:driverClassName="com.mysql.jdbc.Driver"
        p:url="jdbc:mysql://localhost:3306/mydb" p:username="root"
        p:password="masterkaoli" />
```

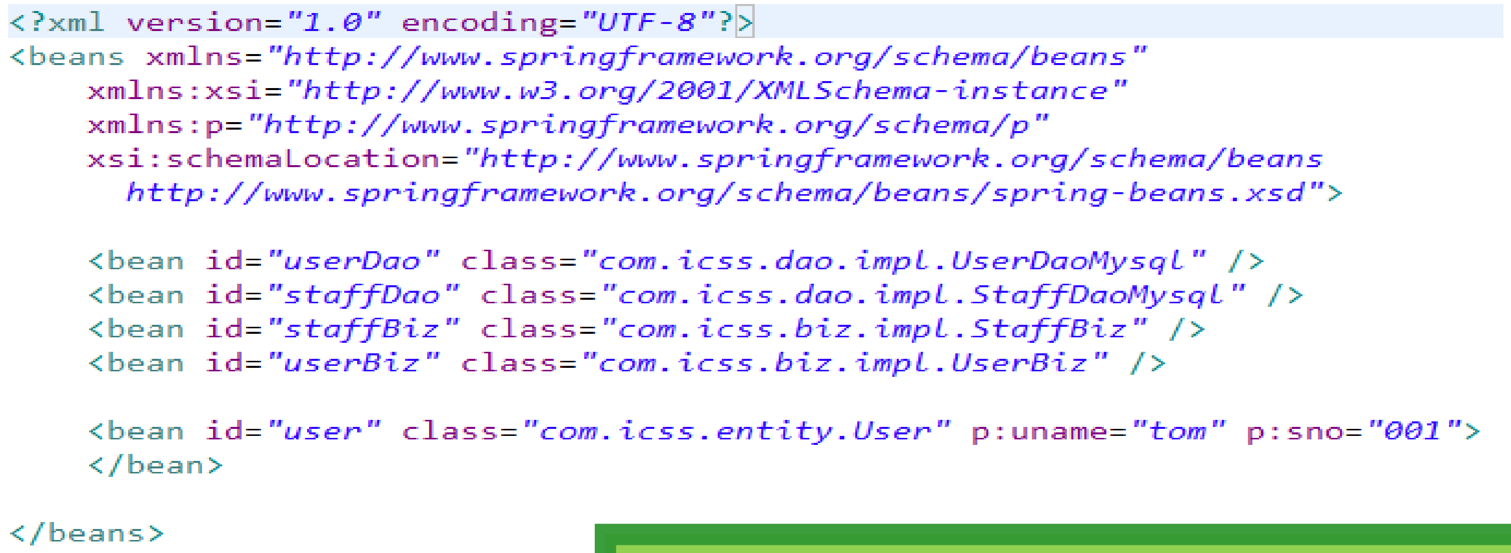


```java
public class BasicDataSource implements DataSourc

    private volatile String password;

    private String url;

    private String userName;

    private Driver driver;

    private String driverClassName;

    private ClassLoader driverClassLoader;

    private boolean lifo = BaseObjectPoolConfig.DEFAULT_LIFO;

    private int maxTotal = GenericObjectPoolConfig.DEFAULT_MAX_TOTAL;

    private int maxIdle = GenericObjectPoolConfig.DEFAULT_MAX_IDLE;

    private int minIdle = GenericObjectPoolConfig.DEFAULT_MIN_IDLE;

    private int initialSize = 0;

    private long maxWaitMillis = BaseObjectPoolConfig.DEFAULT_MAX_WAIT_MILLIS;

    private boolean poolPreparedStatements = false;
```

参考BasicDataSource的源代码，该类有几十个属性

依赖配置参数

- p给User属性赋值案例



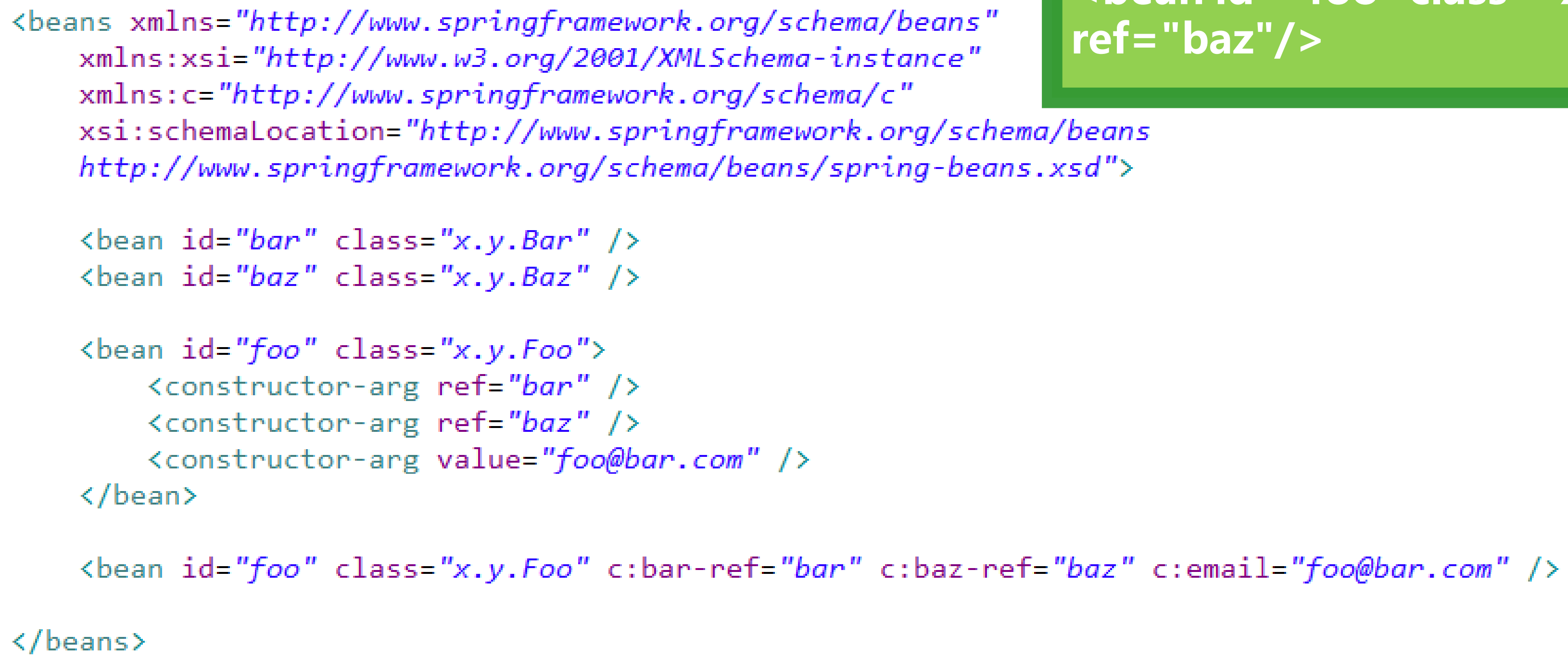
```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="userDao" class="com.icss.dao.impl.UserDaoMysql" />
    <bean id="staffDao" class="com.icss.dao.impl.StaffDaoMysql" />
    <bean id="staffBiz" class="com.icss.biz.impl.StaffBiz" />
    <bean id="userBiz" class="com.icss.biz.impl.UserBiz" />

    <bean id="user" class="com.icss.entity.User" p:uname="tom" p:sno="001">
    </bean>

</beans>
```

把实体类User配置成bean，并用p:namespace赋值

依赖配置参数

- c:namespace简化配置
  - **c:namespace作用**

使用如下格式命名c:xx也可以：
<bean id="foo" class="x.y.Foo" c:_0-ref="bar" c:_1-ref="baz"/>

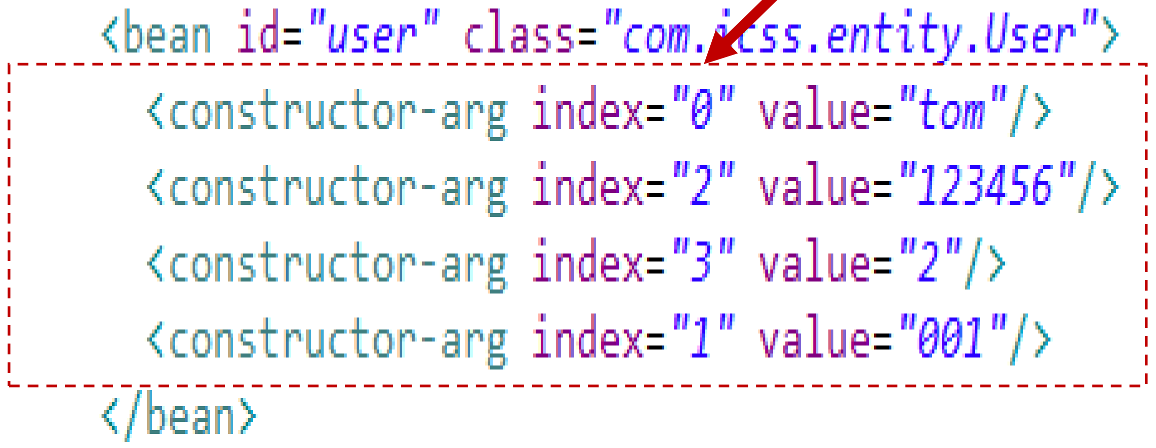


```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:c="http://www.springframework.org/schema/c"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="bar" class="x.y.Bar" />
    <bean id="baz" class="x.y.Baz" />

    <bean id="foo" class="x.y.Foo">
        <constructor-arg ref="bar" />
        <constructor-arg ref="baz" />
        <constructor-arg value="foo@bar.com" />
    </bean>

    <bean id="foo" class="x.y.Foo" c:bar-ref="bar" c:baz-ref="baz" c:email="foo@bar.com" />

</beans>
```

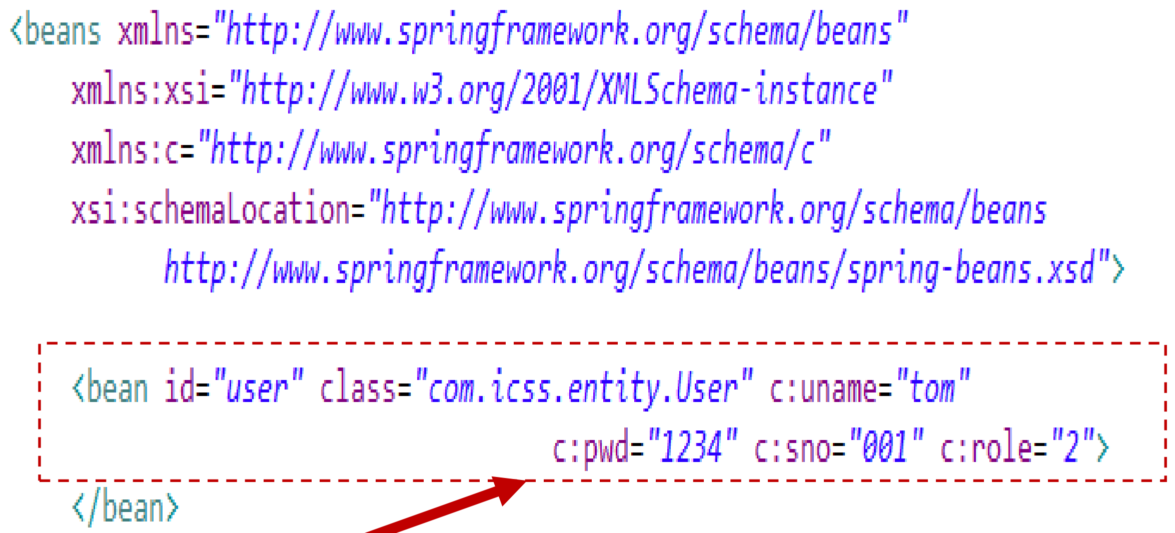
依赖配置参数

- 构造器注入User对象简化配置案例

原来的配置，参见 构造器注入User对象



```xml
<bean id="user" class="com.icss.entity.User">
    <constructor-arg index="0" value="tom"/>
    <constructor-arg index="2" value="123456"/>
    <constructor-arg index="3" value="2"/>
    <constructor-arg index="1" value="001"/>
</bean>
```



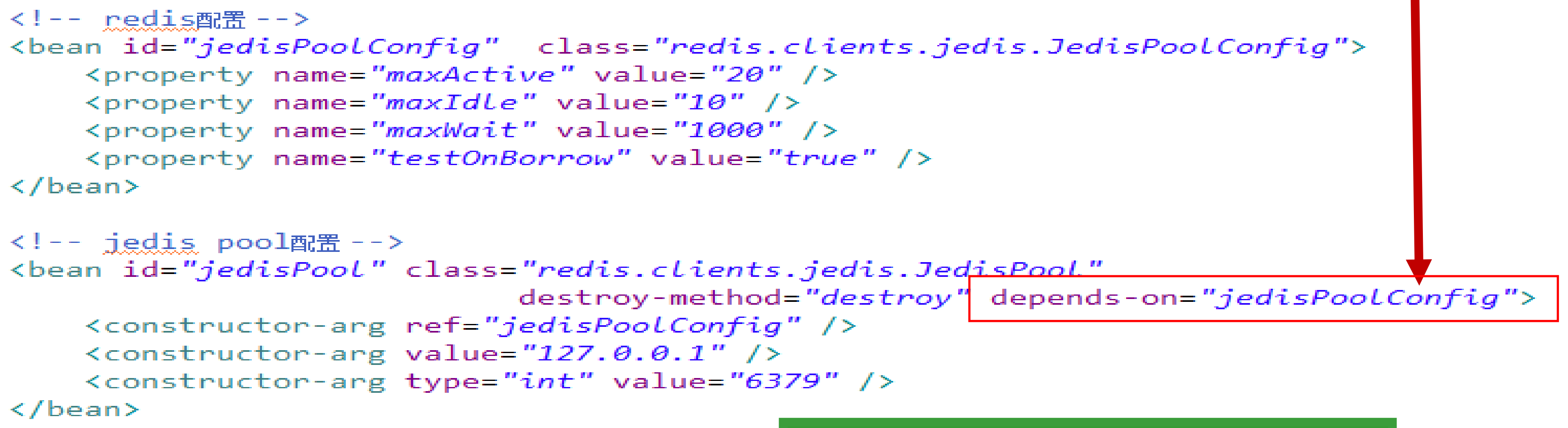
```xml
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:c="http://www.springframework.org/schema/c"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="user" class="com.icss.entity.User" c:uname="tom"
                          c:pwd="1234" c:sno="001" c:role="2">
    </bean>
```

用c:namespace简化配置

依赖配置参数

- depends-on
  - depends-on介绍

jedisPool定义depend-on="jedisPoolConfig"，这意味着Spring总会保证jedisPoolConfig在jedisPool之前实例化，总是在jedisPool之后销毁



```
<!-- redis配置 -->
<bean id="jedisPoolConfig"  class="redis.clients.jedis.JedisPoolConfig">
    <property name="maxActive" value="20" />
    <property name="maxIdle" value="10" />
    <property name="maxWait" value="1000" />
    <property name="testOnBorrow" value="true" />
</bean>

<!-- jedis pool配置 -->
<bean id="jedisPool" class="redis.clients.jedis.JedisPool"
                   destroy-method="destroy" depends-on="jedisPoolConfig">
    <constructor-arg ref="jedisPoolConfig" />
    <constructor-arg value="127.0.0.1" />
    <constructor-arg type="int" value="6379" />
</bean>
```

提问： depends-on与ref的区别？

依赖配置参数

- StaffUser服务层与持久层依赖案例

```xml
<bean id="userDao" class="com.icss.dao.impl.UserDaoMysql"
        init-method="init" destroy-method="destroy" />
```

配置userBiz依赖于depends-on

```xml
<bean id="userBiz" class="com.icss.biz.impl.UserBiz" init-method="init"
                destroy-method="destroy"  depends-on="userDao">
    <constructor-arg ref="userDao" />
</bean>
```

代码实现

```java
public class UserDaoMysql implements IUserDao {

    public UserDaoMysql() {
        System.out.println("UserDaoMysql 构造....");
    }

    public void init() {
        System.out.println("UserDaoMysql 初始化....");
    }

    public void destroy() {
        System.out.println("UserDaoMysql 析构....");
    }
```

```java
public class UserBiz implements IUser {

    private IUserDao userDao;

    public UserBiz(IUserDao userDao) {
        this.userDao = userDao;
        System.out.println("UserBiz 构造....");
    }

    public void init() {
        System.out.println("UserBiz 初始化....");
    }

    public void destroy() {
        System.out.println("UserBiz 析构....");
    }
```

依赖配置参数

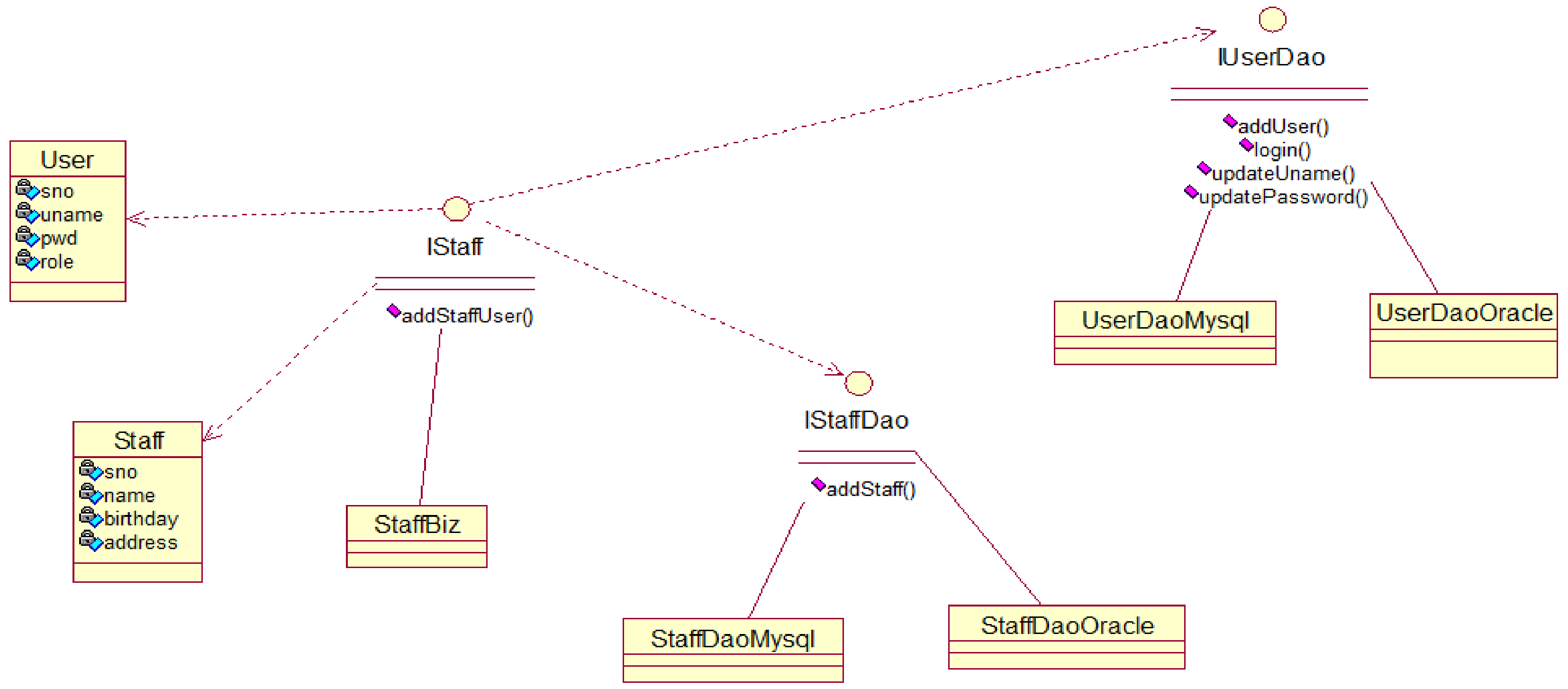
```java
ApplicationContext app = new ClassPathXmlApplicationContext("beans.xml");
IUser u = (IUser)app.getBean("userBiz");
try {
    User user = u.login("admin", "123");
    if(user != null) {
        System.out.println("登录成功，身份是" + user.getRole());
    }else {
        System.out.println("登录失败");
    }
} catch (Exception e) {
    System.out.println(e.getMessage());
}
//关闭IOC容器
((ClassPathXmlApplicationContext) app).close();
```

代码测试

```
信息: Loading XML bean definitions from class path resource [beans.xml]
UserDaoMysql 构造....
UserDaoMysql 初始化....
UserBiz 构造....
UserBiz 初始化....
UserDaoMysql....login....
登录成功，身份是1
十一月 05, 2019 1:15:07 下午 org.springframework.context.support.AbstractApplicationContext doClose
信息: Closing org.springframework.context.support.ClassPathXmlApplicationContext@817b38: startup
UserBiz 析构....
UserDaoMysql 析构....
```

依赖配置参数

- StaffUser服务层对User和Staff依赖案例



之前的案例中UserBiz和UserDao之间的关系，在UML中被称为聚合。这种聚合关系，是强依赖关系，即使删除depends-on配置,执行顺序也是一样的。

依赖配置参数

测试IStaff与User和Staff的依赖关系，未配置依赖关系

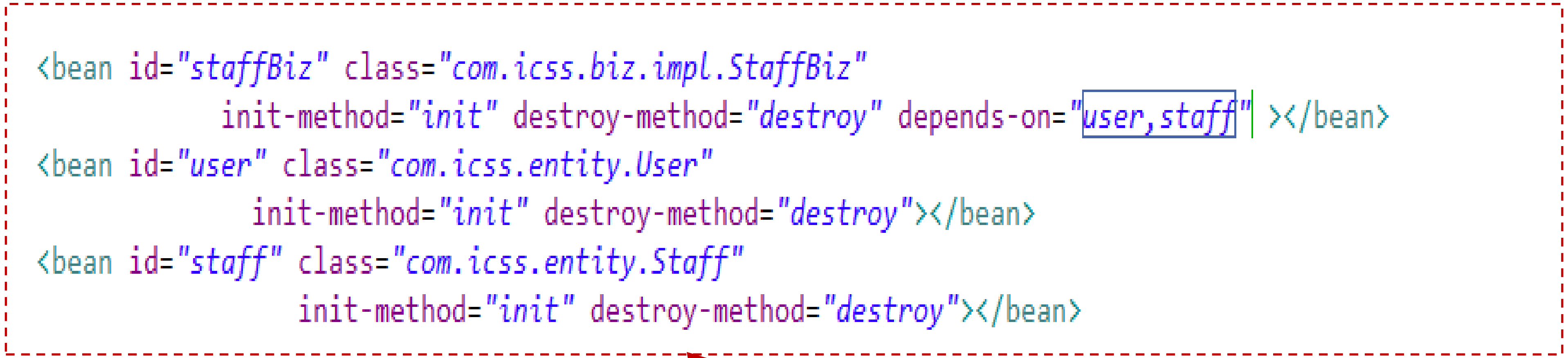
```xml
<bean id="staffBiz" class="com.icss.biz.impl.StaffBiz"
            init-method="init" destroy-method="destroy" ></bean>
<bean id="user" class="com.icss.entity.User"
             init-method="init" destroy-method="destroy"></bean>
<bean id="staff" class="com.icss.entity.Staff"
               init-method="init" destroy-method="destroy"></bean>
```

```java
ApplicationContext app = new ClassPathXmlApplicationContext("beans.xml")
IUser u = (IUser)app.getBean("userBiz");
try {
    u.login("admin", "123");
} catch (Exception e) {
    System.out.println(e.getMessage());
}
//关闭IOC容器
((ClassPathXmlApplicationContext) app).close();
```

测试结果： StaffBiz先与User和Staff对象构造成功

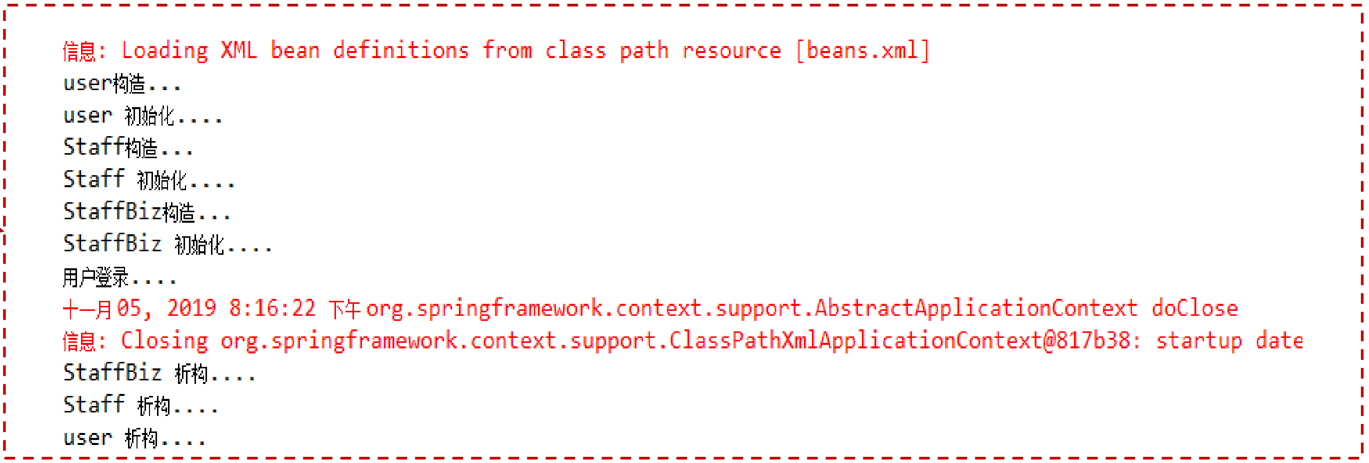
```
信息: Loading XML bean definitions from class path resource [beans.xml]
StaffBiz构造...
StaffBiz 初始化....
user构造...
user 初始化....
Staff构造...
Staff 初始化....
用户登录....
十一月05, 2019 8:13:16 下午 org.springframework.context.support.AbstractApplicationContext doClose
信息: Closing org.springframework.context.support.ClassPathXmlApplicationContext@817b38: startup date
Staff 析构....
user 析构....
StaffBiz 析构....
```

依赖配置参数



```
<bean id="staffBiz" class="com.icss.biz.impl.StaffBiz"
          init-method="init" destroy-method="destroy" depends-on="user,staff" ></bean>
<bean id="user" class="com.icss.entity.User"
           init-method="init" destroy-method="destroy"></bean>
<bean id="staff" class="com.icss.entity.Staff"
             init-method="init" destroy-method="destroy"></bean>
```

增加依赖配置后，执行结果



```
信息: Loading XML bean definitions from class path resource [beans.xml]
user构造...
user 初始化....
Staff构造...
Staff 初始化....
StaffBiz构造...
StaffBiz 初始化....
用户登录....
十一月05, 2019 8:16:22 下午 org.springframework.context.support.AbstractApplicationContext doClose
信息: Closing org.springframework.context.support.ClassPathXmlApplicationContext@817b38: startup date
StaffBiz 析构....
Staff 析构....
user 析构....
```

依赖配置参数

- lazy-init
  - lazy-init介绍

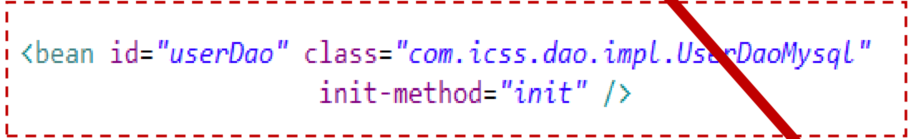
```
<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>
<bean name="not.lazy" class="com.foo.AnotherBean"/>
```

  - ApplicationContext的默认行为就是在创建IOC容器时将所有singleton 的bean提前进行实例化。
  - 系统**默认配置是lazy-init="false"**
  - 当**配置lazy-init="true"后，当第一次调用bean对象时，才进行实例**
  - A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.

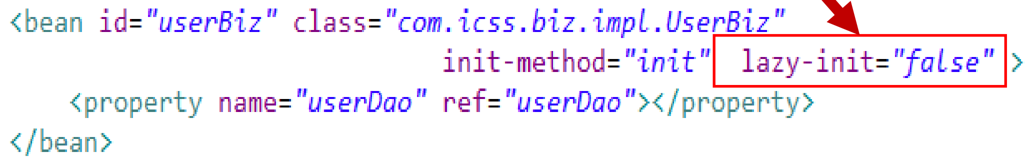
依赖配置参数

- 观察StaffUser系统bean的创建时间案例

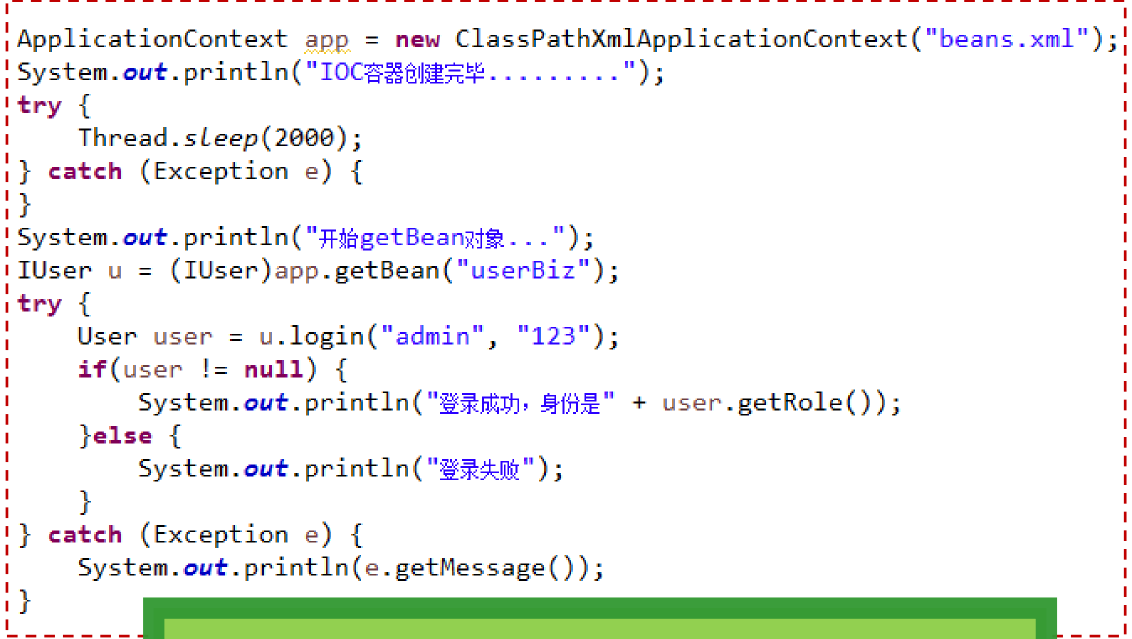
系统默认配置是lazy-init="false"，在容器启动时，就创建所有单例bean



```xml
<bean id="userDao" class="com.icss.dao.impl.UserDaoMysql"
                    init-method="init" />
```

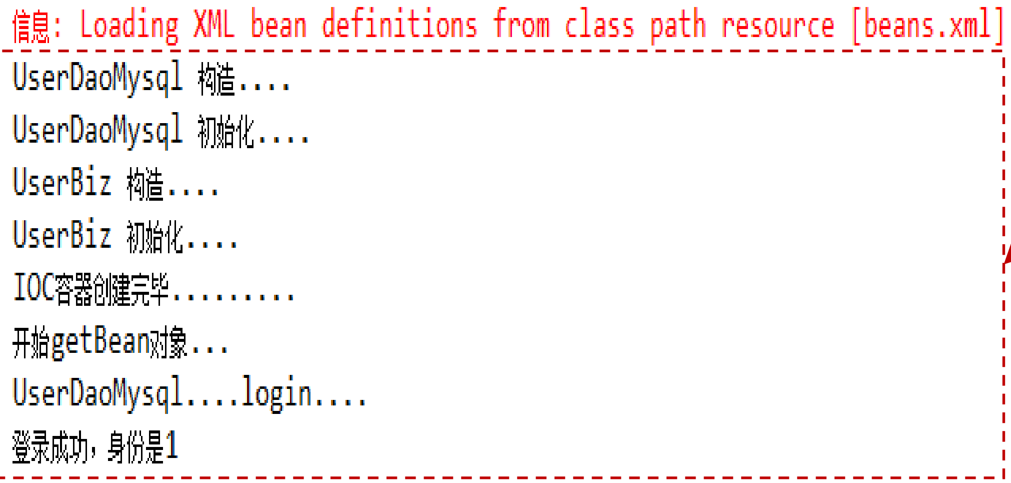


```xml
<bean id="userBiz" class="com.icss.biz.impl.UserBiz"
                              init-method="init"  lazy-init="false" >
    <property name="userDao" ref="userDao"></property>
</bean>
```



```java
ApplicationContext app = new ClassPathXmlApplicationContext("beans.xml");
System.out.println("IOC容器创建完毕.........");
try {
    Thread.sleep(2000);
} catch (Exception e) {
}
System.out.println("开始getBean对象...");
IUser u = (IUser)app.getBean("userBiz");
try {
    User user = u.login("admin", "123");
    if(user != null) {
        System.out.println("登录成功，身份是" + user.getRole());
    }else {
        System.out.println("登录失败");
    }
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

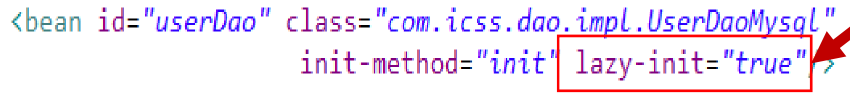
测试：容器创建完毕前，bean的构造和初始化完成



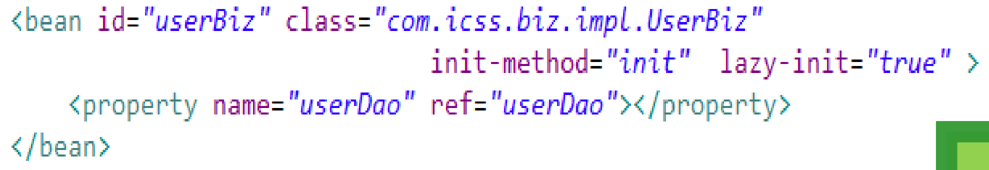
```
信息: Loading XML bean definitions from class path resource [beans.xml]
UserDaoMysql 构造....
UserDaoMysql 初始化....
UserBiz 构造....
UserBiz 初始化....
IOC容器创建完毕.........
开始getBean对象...
UserDaoMysql....login....
登录成功，身份是1
```

依赖配置参数

增加配置lazy-init="true"

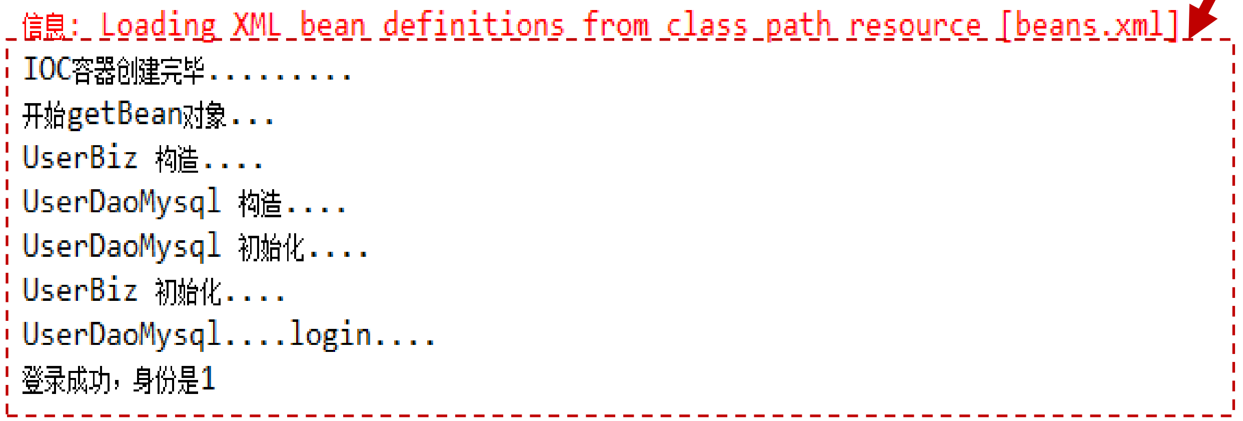


```
<bean id="userDao" class="com.icss.dao.impl.UserDaoMysql"
                    init-method="init" lazy-init="true" >
```



```
<bean id="userBiz" class="com.icss.biz.impl.UserBiz"
                          init-method="init"  lazy-init="true" >
    <property name="userDao" ref="userDao"></property>
</bean>
```

观察修改配置后的运行结果



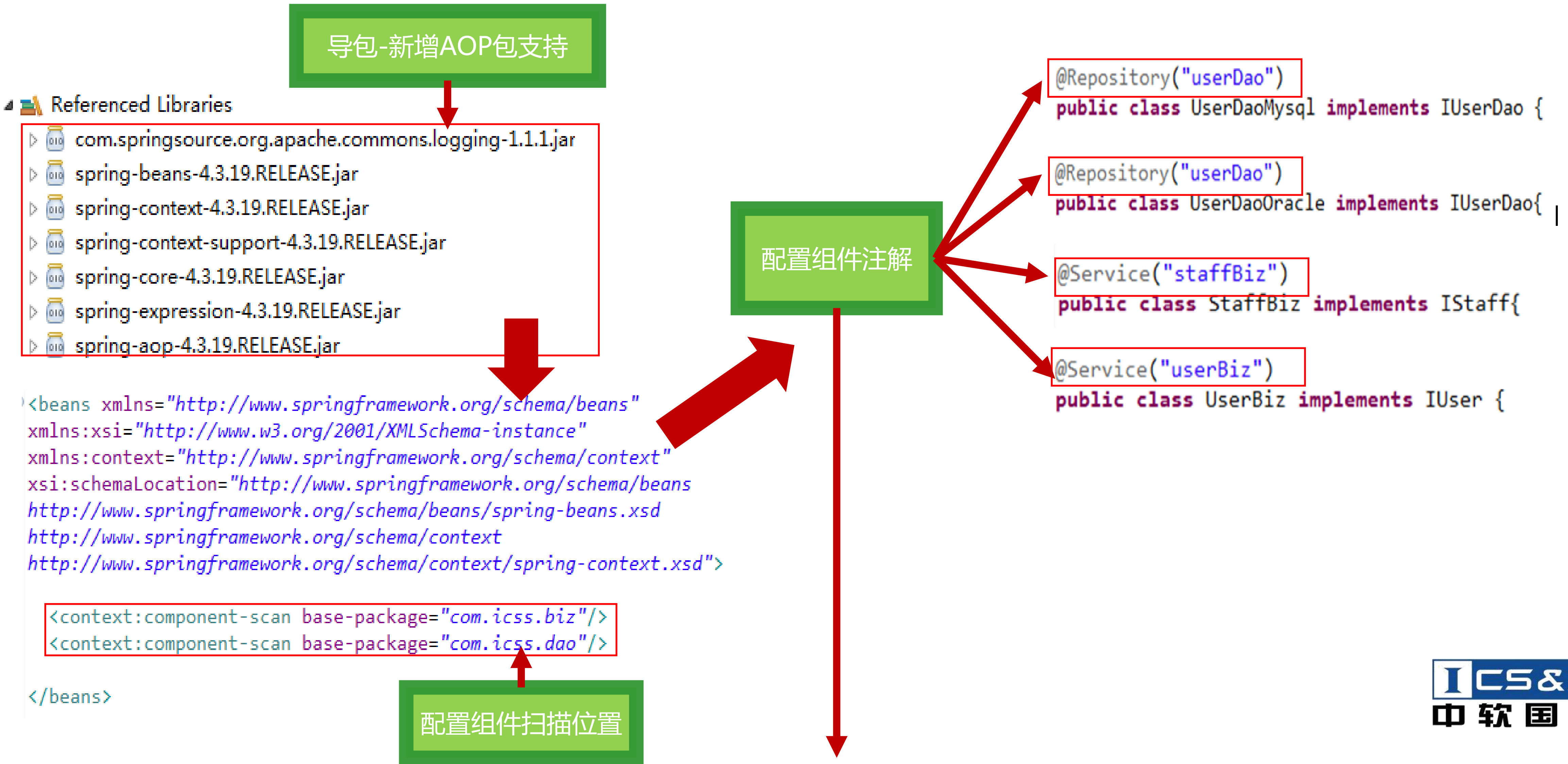
```
信息: Loading XML bean definitions from class path resource [beans.xml]
IOC容器创建完毕.........
开始getBean对象...
UserBiz 构造....
UserDaoMysql 构造....
UserDaoMysql 初始化....
UserBiz 初始化....
UserDaoMysql....login....
登录成功，身份是1
```

总结：

lazy-init="true"，所有为singleton的bean，在第一次调用getBean()时被实例

Autowire适配注入

- Autowire注入StaffUser持久层对象案例

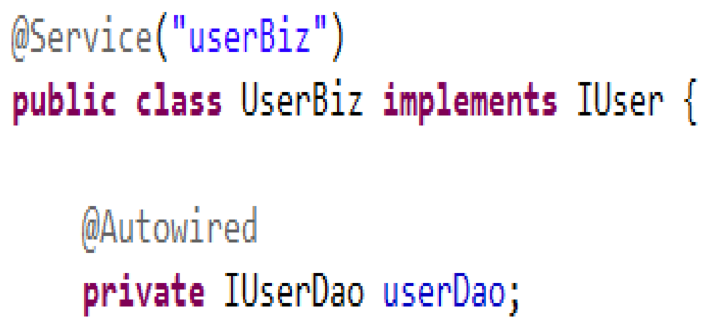


Referenced Libraries
- com.springsource.org.apache.commons.logging-1.1.1.jar
- spring-beans-4.3.19.RELEASE.jar
- spring-context-4.3.19.RELEASE.jar
- spring-context-support-4.3.19.RELEASE.jar
- spring-core-4.3.19.RELEASE.jar
- spring-expression-4.3.19.RELEASE.jar
- spring-aop-4.3.19.RELEASE.jar

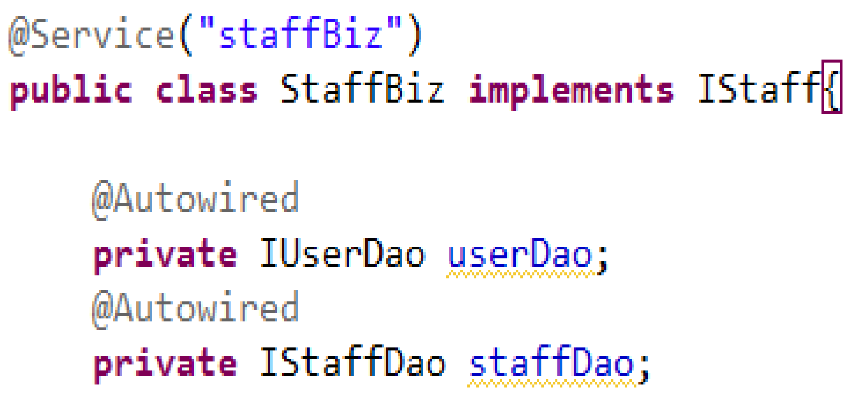
```xml
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

  <context:component-scan base-package="com.icss.biz"/>
  <context:component-scan base-package="com.icss.dao"/>

</beans>
```

```java
@Repository("userDao")
public class UserDaoMysql implements IUserDao {

@Repository("userDao")
public class UserDaoOracle implements IUserDao{

@Service("staffBiz")
public class StaffBiz implements IStaff{

@Service("userBiz")
public class UserBiz implements IUser {
```

Autowire适配注入



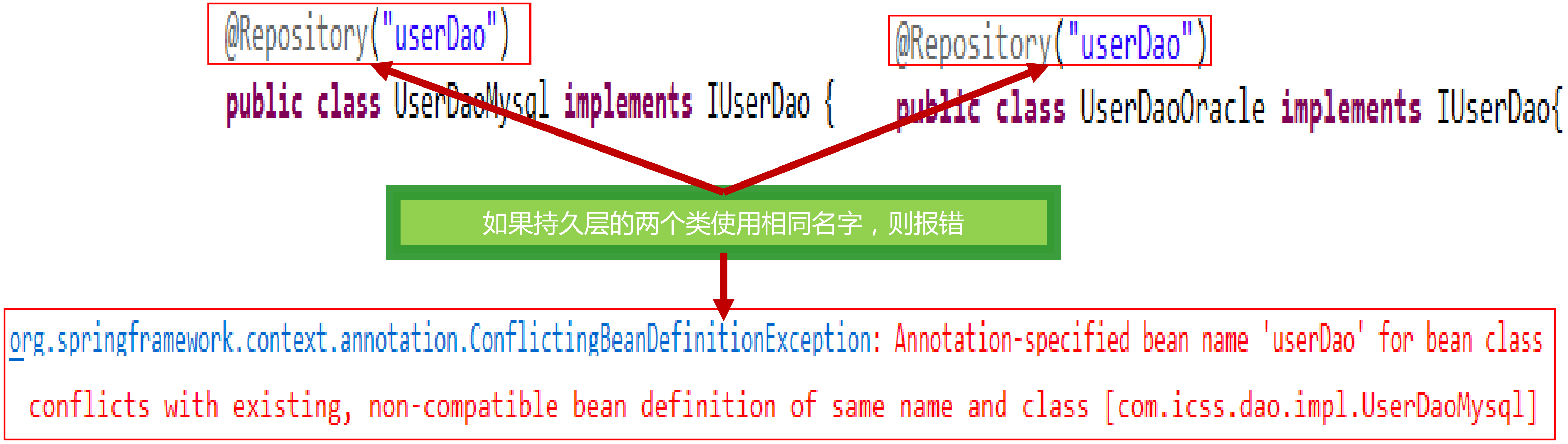
```
@Service("userBiz")
public class UserBiz implements IUser {

    @Autowired
    private IUserDao userDao;
```



```
@Service("staffBiz")
public class StaffBiz implements IStaff{

    @Autowired
    private IUserDao userDao;
    @Autowired
    private IStaffDao staffDao;
```

Autowire注入持久层对象

Autowire适配注入



```java
@Repository("userDao")
public class UserDaoMysql implements IUserDao {
```

```java
@Repository("userDao")
public class UserDaoOracle implements IUserDao{
```

如果持久层的两个类使用相同名字，则报错

```
org.springframework.context.annotation.ConflictingBeanDefinitionException: Annotation-specified bean name 'userDao' for bean class
 conflicts with existing, non-compatible bean definition of same name and class [com.icss.dao.impl.UserDaoMysql]
```

解决：**上面的两个实现类，选一个使用@Repository("userDao")**

AutoWire注入模式

- 注入模式

Mode	Explanation
no	(Default) No autowiring. Bean references must be defined via a ref element. Changing the default setting is not recommended for larger deployments, because specifying collaborators explicitly gives greater control and clarity. To some extent, it documents the structure of a system.
byName	Autowiring by property name. Spring looks for a bean with the same name as the property that needs to be autowired. For example, if a bean definition is set to autowire by name, and it contains a master property (that is, it has a setMaster(..) method), Spring looks for a bean definition named master, and uses it to set the property.
byType	Allows a property to be autowired if exactly one bean of the property type exists in the container. If more than one exists, a fatal exception is thrown, which indicates that you may not use byType autowiring for that bean. If there are no matching beans, nothing happens; the property is not set.
constructor	Analogous to byType, but applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.

Autowire适配注入

- byName与byType

持久层对象的名字为userDao，使用autowire注入时，优先使用byType模式注入

```
@Repository("userDao")
public class UserDaoMysql implements IUserDao {
```

```
@Service("userBiz")
public class UserBiz implements IUser {
```

```
@Autowired
private IUserDao userDao;
```

修改注入对象的名字为userDao2，然后调用测试

```
@Service("userBiz")
public class UserBiz implements IUser {
```

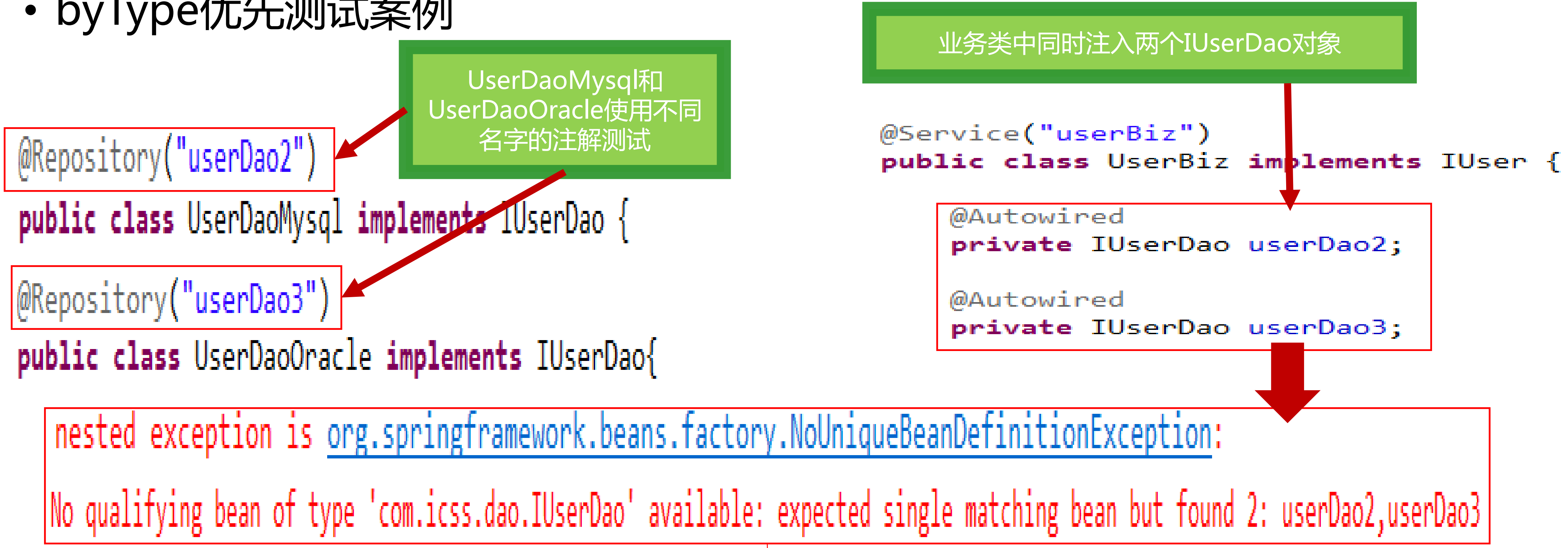
```
@Autowired
private IUserDao userDao2;
```

```
信息: Loading XML bean definitions from class path resource [beans.xml]
UserDaoMysql....login....
登录成功，身份是1
```

总结：**优先使用byType进行类型匹配，当同一类型下有多个bean时，用byName区别**

Autowire适配注入

- byType优先测试案例


UserDaoMysql和UserDaoOracle使用不同名字的注解测试
业务类中同时注入两个IUserDao对象

```
@Repository("userDao2")
public class UserDaoMysql implements IUserDao {

@Repository("userDao3")
public class UserDaoOracle implements IUserDao{
```

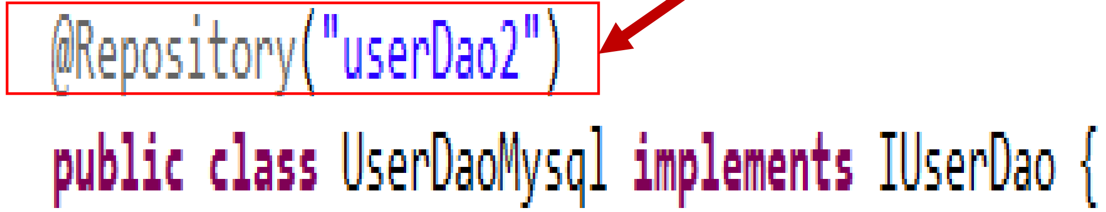
```
@Service("userBiz")
public class UserBiz implements IUser {

    @Autowired
    private IUserDao userDao2;

    @Autowired
    private IUserDao userDao3;
```

```
nested exception is org.springframework.beans.factory.NoUniqueBeanDefinitionException:

No qualifying bean of type 'com.icss.dao.IUserDao' available: expected single matching bean but found 2: userDao2,userDao3
```

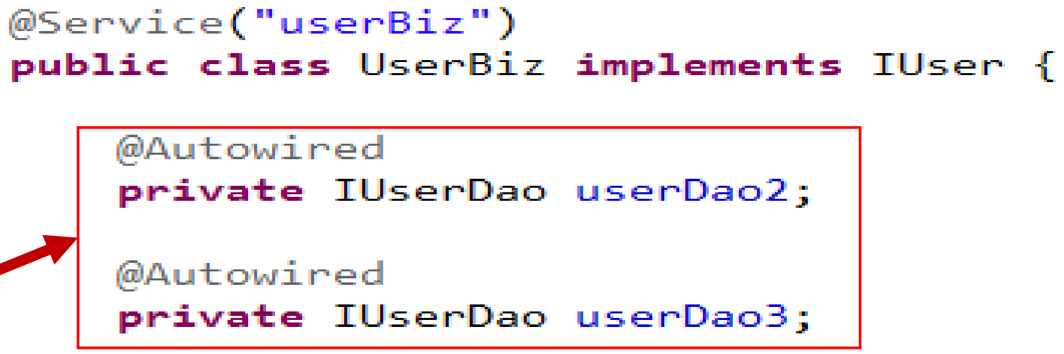
运行结果：**因为是byType优先，Spring无法区分IUser的两个实现类对象，因此报错。如果是byName优先，不应该报错**

Autowire适配注入

删除UserDaoOracle的注解，只保留userDao2

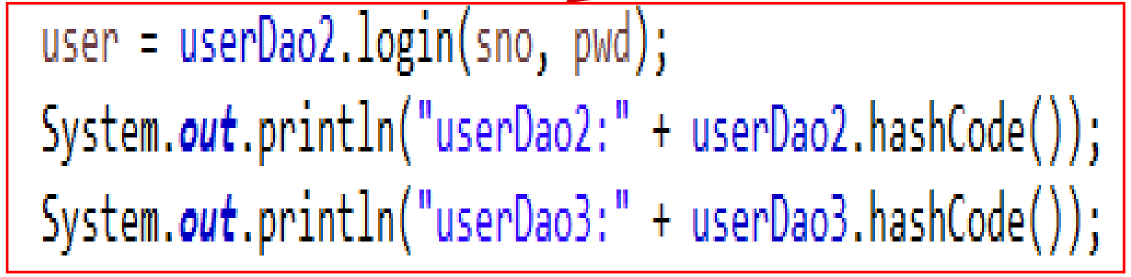


```java
@Repository("userDao2")
public class UserDaoMysql implements IUserDao {
```

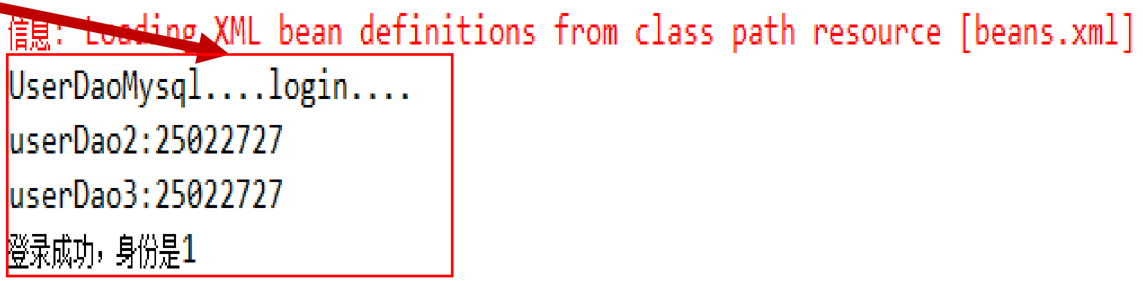


```java
@Service("userBiz")
public class UserBiz implements IUser {

    @Autowired
    private IUserDao userDao2;

    @Autowired
    private IUserDao userDao3;
```

通过byType模式注入，userDao2和userDao3指向了相同对象



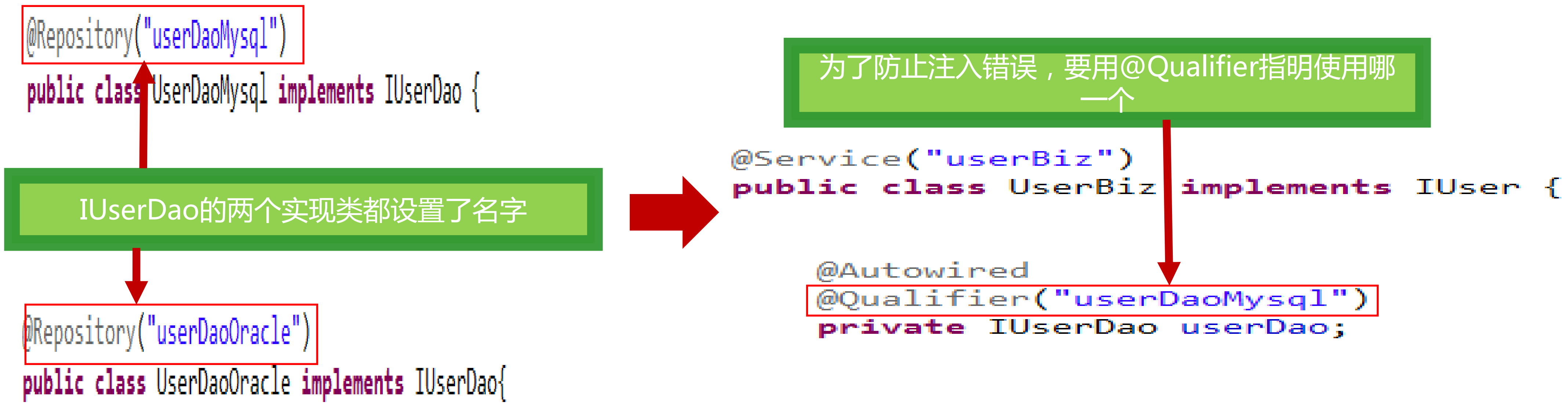
```java
user = userDao2.login(sno, pwd);
System.out.println("userDao2:" + userDao2.hashCode());
System.out.println("userDao3:" + userDao3.hashCode());
```



```
信息: Loading XML bean definitions from class path resource [beans.xml]
UserDaoMysql....login....
userDao2:25022727
userDao3:25022727
登录成功，身份是1
```

Autowire适配注入

- byName测试案例



```
@Repository("userDaoMysql")
public class UserDaoMysql implements IUserDao {
```

IUserDao的两个实现类都设置了名字

```
@Repository("userDaoOracle")
public class UserDaoOracle implements IUserDao{
```

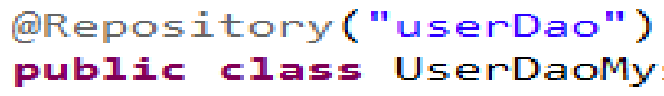
为了防止注入错误，要用@Qualifier指明使用哪一个

```
@Service("userBiz")
public class UserBiz implements IUser {

    @Autowired
    @Qualifier("userDaoMysql")
    private IUserDao userDao;
```

总结：**当存在同一接口的多个实现类注入时，byType无法区分使用哪个，这时就需要用@Qualifier调用byName模式**

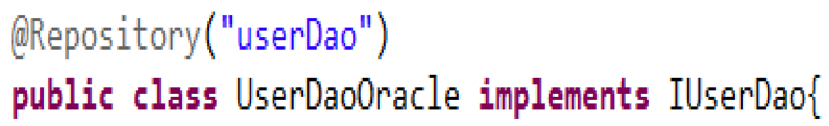
Autowire适配注入

- Autowire模式的缺陷
  - A. 对于java基本类型和String等简单类型，无法使用Autowire方式注入
  - B. 业务变化，注入的配置项必须改变时，没有xml配置修改容易。例如：

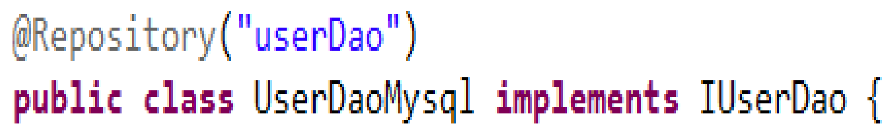


```
@Repository("userDao")
public class UserDaoMysql implements IUserDao {
```

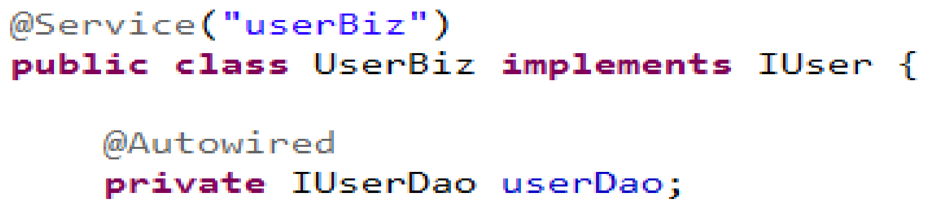
  - C. 同一接口的多个实现类同时使用时，容易引发冲突。例如：



```
@Repository("userDao")
public class UserDaoOracle implements IUserDao{
```



```
@Repository("userDao")
public class UserDaoMysql implements IUserDao {
```



```
@Service("userBiz")
public class UserBiz implements IUser {

    @Autowired
    private IUserDao userDao;
```

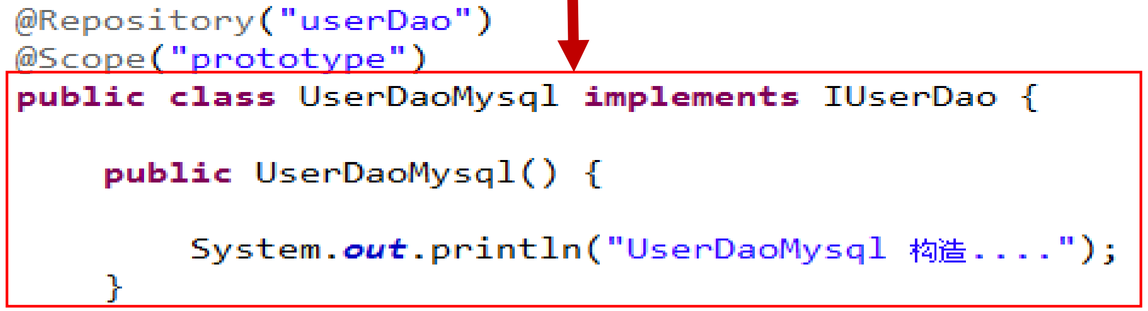
方法注入

- 在绝大多数的业务场景，IOC容器中的bean都是单例模式。
- 一个单例bean通常依赖其它单例bean，或者一个非单例bean依赖其它非单例bean.
- 有个严重问题是，如果依赖与被依赖的bean对象，生命周期不一致怎么办？
- 假如单例beanA依赖非单例beanB，会出什么事？IOC容器创建beanA只有一次，然后设置它的依赖对象beanB。在以后beanA的方法调用中，beanA对于beanB的对象依赖不会发生变化，即始终使用同一个beanB对象。

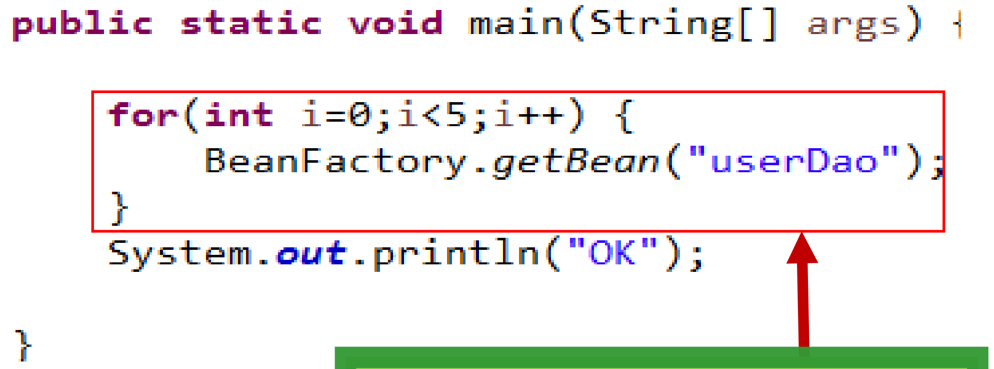
方法注入

- UserBiz与UserDao生命期不一致问题

UserDaoMysql配置成非单例模式



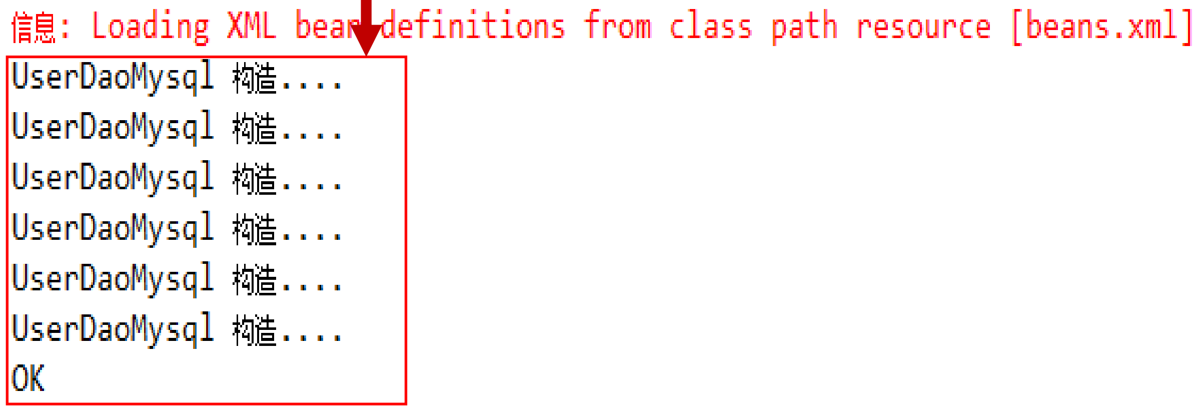
```
@Repository("userDao")
@Scope("prototype")
public class UserDaoMysql implements IUserDao {

    public UserDaoMysql() {

        System.out.println("UserDaoMysql 构造....");
    }
```



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

    for(int i=0;i<5;i++) {
        BeanFactory.getBean("userDao");
    }
    System.out.println("OK");

}
```

多次getBean("userDao")

结果显示每次getBean("userDao")都会创建新对象



```
信息: Loading XML bean definitions from class path resource [beans.xml]
UserDaoMysql 构造....
UserDaoMysql 构造....
UserDaoMysql 构造....
UserDaoMysql 构造....
UserDaoMysql 构造....
UserDaoMysql 构造....
OK
```

方法注入

服务层注入userDao

```
@Service("userBiz")
public class UserBiz implements IUser {

    @Autowired
     private IUserDao userDao;

    public UserBiz() {
        System.out.println("UserBiz构造...");
    }
```

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

    for(int i=0;i<5;i++) {
        IUser u = (IUser)BeanFactory.getBean("userBiz");
        try {
            u.login("admin", "123");
        } catch (Exception e) {
        }
    }
}
```

多次调用userBiz的login()方法

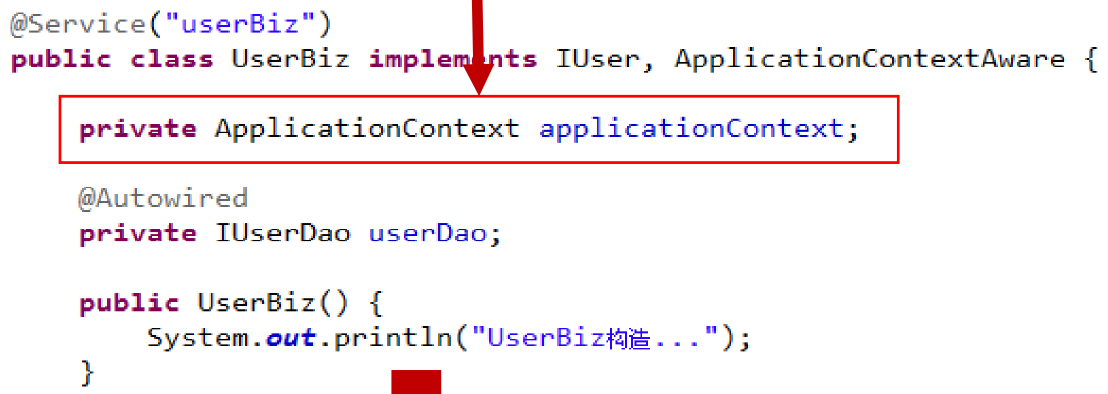
运行结果如下，userDao只被实例了一次

```
信息: Loading XML bean definitions from class path resource [beans.xml]
UserBiz构造...
UserDaoMysql 构造....
login......
login......
login......
login......
login......
```

方法注入

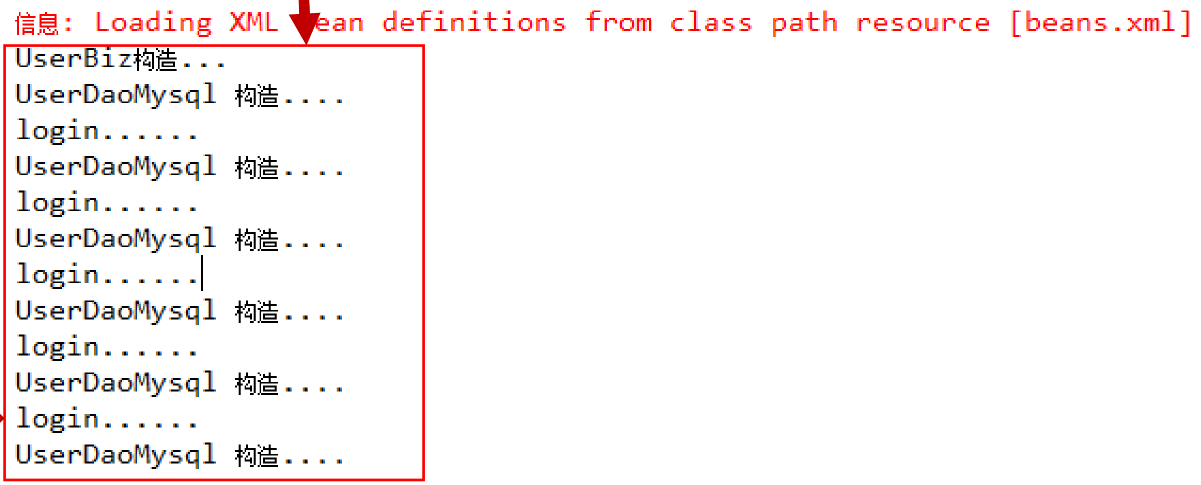
- 抛弃DI注入解决UserBiz与UserDao生命期不一致问题

UserBiz中注入ApplicationContext环境



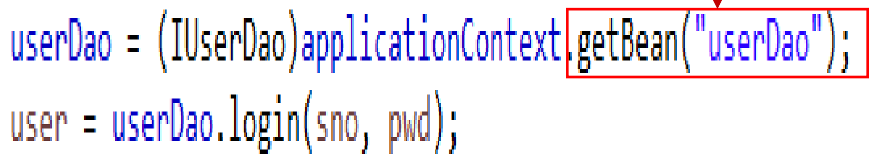
```
@Service("userBiz")
public class UserBiz implements IUser, ApplicationContextAware {

    private ApplicationContext applicationContext;

    @Autowired
    private IUserDao userDao;

    public UserBiz() {
        System.out.println("UserBiz构造...");
    }
```

运行结果如下



```
信息: Loading XML bean definitions from class path resource [beans.xml]
UserBiz构造...
UserDaoMysql 构造....
login......
UserDaoMysql 构造....
login......
UserDaoMysql 构造....
login......
UserDaoMysql 构造....
login......
UserDaoMysql 构造....
login......
UserDaoMysql 构造....
```

不使用注入的userDao，而采用
getBean("userDao")



```
userDao = (IUserDao)applicationContext.getBean("userDao");
user = userDao.login(sno, pwd);
```

这种方法非常简单

其实就是把Autowire注入的对象抛弃了

方法注入

- Lookup解决UserBiz与UserDao生命期不一致问题

新增如下接口

```
@Component("lookDao")
public interface ILookDao {

    @Lookup
    public IUserDao lookUserDao();

}
```

注入lookDao，而不是userDao

```
@Service("userBiz")
public class UserBiz implements IUser {

    @Autowired
    private ILookDao lookDao;

    public UserBiz() {
        System.out.println("UserBiz构造...");
    }
```

```
IUserDao userDao = lookDao.lookUserDao();
user = userDao.login(sno, pwd);
```

调用业务方法，测试结果如下：

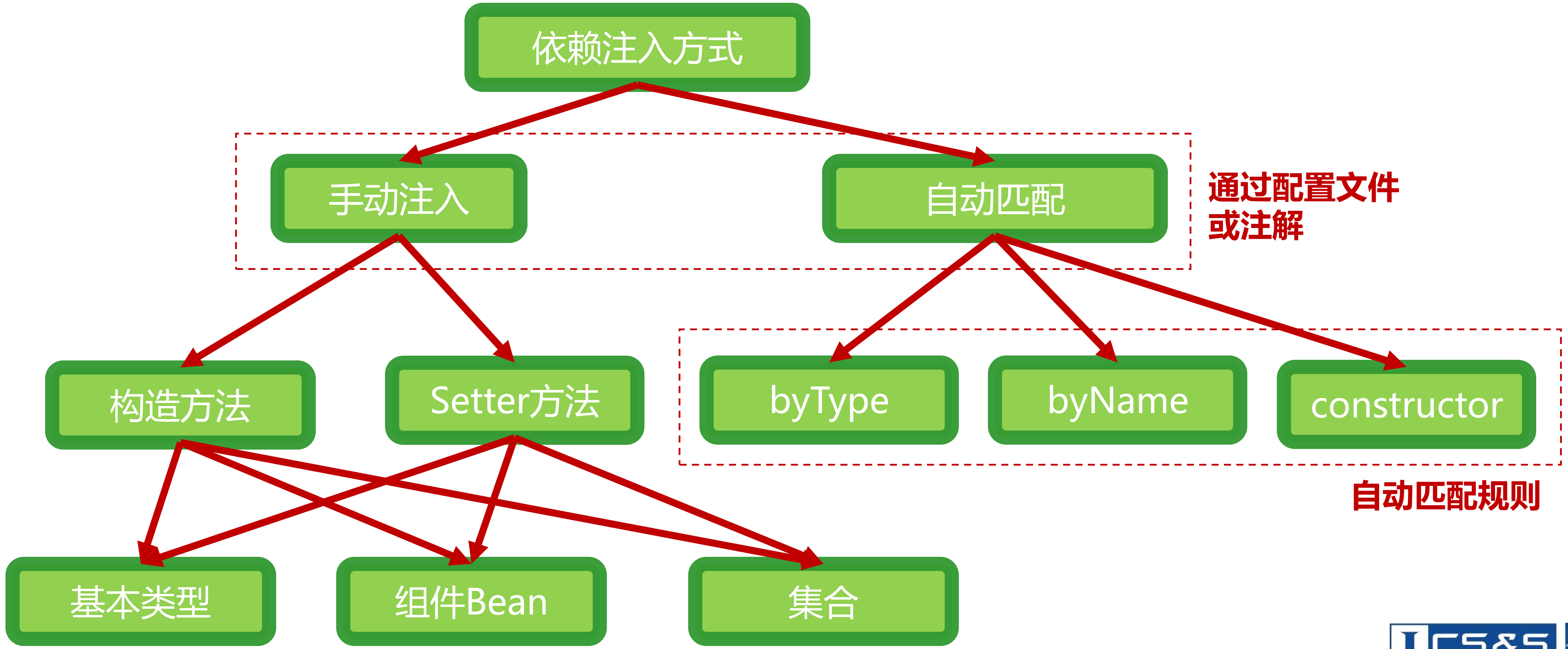
```
信息: Loading XML bean definitions from class path resource [beans.xml]
UserBiz构造...
login......
UserDaoMysql 构造....
login......
UserDaoMysql 构造....
login......
UserDaoMysql 构造....
login......
UserDaoMysql 构造....
login......
UserDaoMysql 构造....
```

依赖注入总结

- DI与IOC的关系
  - IOC = 反射创建对象 + 依赖(Dependency) + Inject(注入)

依赖注入总结

- 注入方式总结



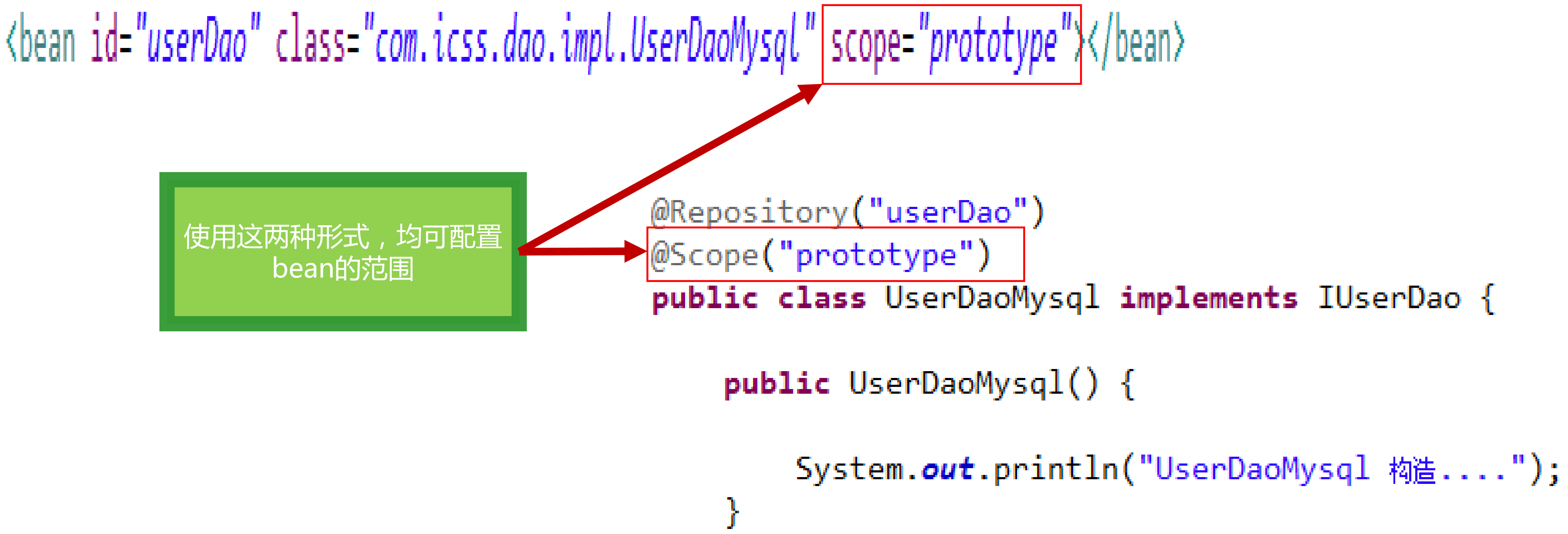
依赖注入总结

- 注入循环引用问题
  - 如何解决BeanA依赖BeanB，同时BeanB要依赖注入BeanA的问题？
    - 当循环引用注入无法避免时，使用set注入模式，不要使用构造函数注入

依赖注入总结

- 实体类需要注入吗?
  - 实体都是有属性信息的，因此每个实体对象都有自己的特性，不能使用singleton配置。
  - EJB中实体对象的管理，使用EntityBean。每个实体bean与数据库中的一条表记录形成映射关系。EntityBean是轻易不释放的，生命期相当长。
  - EJB是重量级容器，有应用服务器支持，因此可以管理实体bean。而Spring是轻量级容器，无法管理大量有状态的实体对象。
  - 如果在Spring中，把实体配置成bean，且scope="prototype"，是可以的，但是价值并不高。

Bean配置时Scope选项



```
<bean id="userDao" class="com.icss.dao.impl.UserDaoMysql" scope="prototype"></bean>
```

使用这两种形式，均可配置bean的范围

```
@Repository("userDao")
@Scope("prototype")
public class UserDaoMysql implements IUserDao {

    public UserDaoMysql() {

        System.out.println("UserDaoMysql 构造....");
    }
```

Bean配置时Scope选项

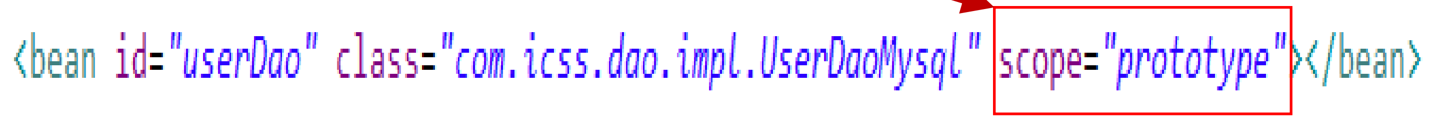
- Bean有如下scope选项

Scope	Description
singleton	(Default) Scopes a single bean definition to a single object instance per Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
request	Scopes a single bean definition to the lifecycle of a single HTTP request; that is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.
session	Scopes a single bean definition to the lifecycle of an HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.
globalSession	Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a Portlet context. Only valid in the context of a web-aware Spring ApplicationContext.
application	Scopes a single bean definition to the lifecycle of a ServletContext. Only valid in the context of a web-aware Spring ApplicationContext.
websocket	Scopes a single bean definition to the lifecycle of a WebSocket. Only valid in the context of a web-aware Spring ApplicationContext.

singleton和prototype的区别

- 当定义一个bean定义并且它的作用域是一个singleton时，Spring IoC容器创建由该bean定义的对象的一个实例。
- 这个单实例存储在这个单例bean的缓存中，该bean的所有后续请求和引用都返回从缓存中获得对象。

当scope="prototype"时，每次调用getBean("userDao")，都会生成一个新的对象。



```
<bean id="userDao" class="com.icss.dao.impl.UserDaoMysql" scope="prototype"></bean>
```

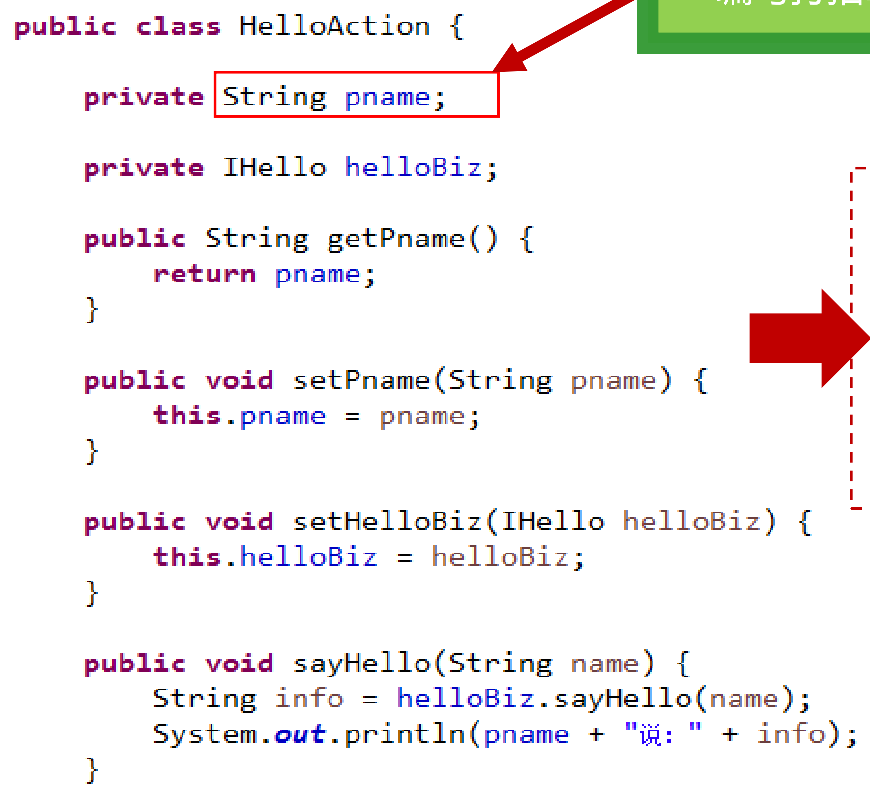
HelloSpringAction案例

- 功能说明
  - 在原来的HelloSpringIOC项目基础上，新增一个HelloAction类。即允许多人同时打招呼，而且把打招呼的人和被打招呼的人都描述清楚。如：张三说，你好，李四

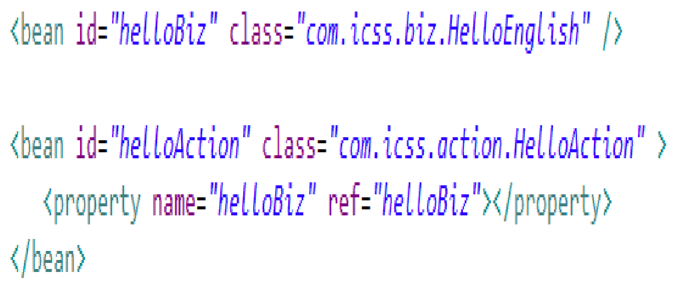
HelloSpringAction案例

- 带属性的HelloAction

编写打招呼人的类HelloAction，有一个pname的属性



```java
public class HelloAction {

    private String pname;

    private IHello helloBiz;

    public String getPname() {
        return pname;
    }

    public void setPname(String pname) {
        this.pname = pname;
    }

    public void setHelloBiz(IHello helloBiz) {
        this.helloBiz = helloBiz;
    }

    public void sayHello(String name) {
        String info = helloBiz.sayHello(name);
        System.out.println(pname + "说: " + info);
    }
```



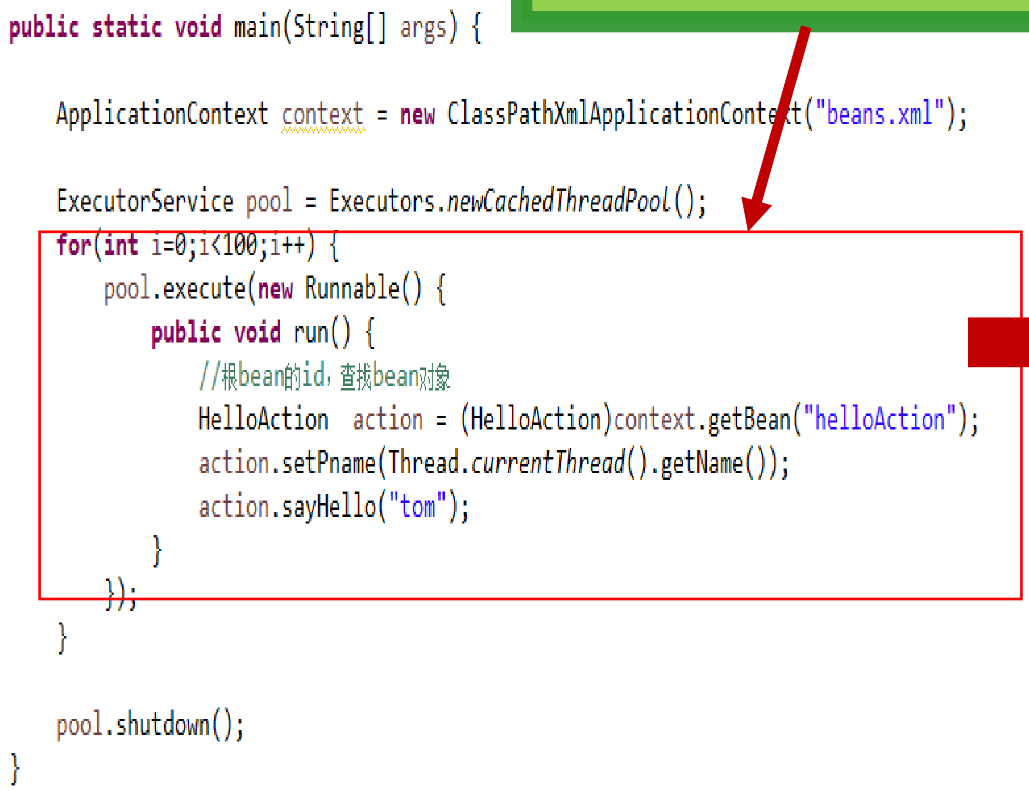
```xml
<bean id="helloBiz" class="com.icss.biz.HelloEnglish" />

<bean id="helloAction" class="com.icss.action.HelloAction" >
   <property name="helloBiz" ref="helloBiz"></property>
</bean>
```

配置bean

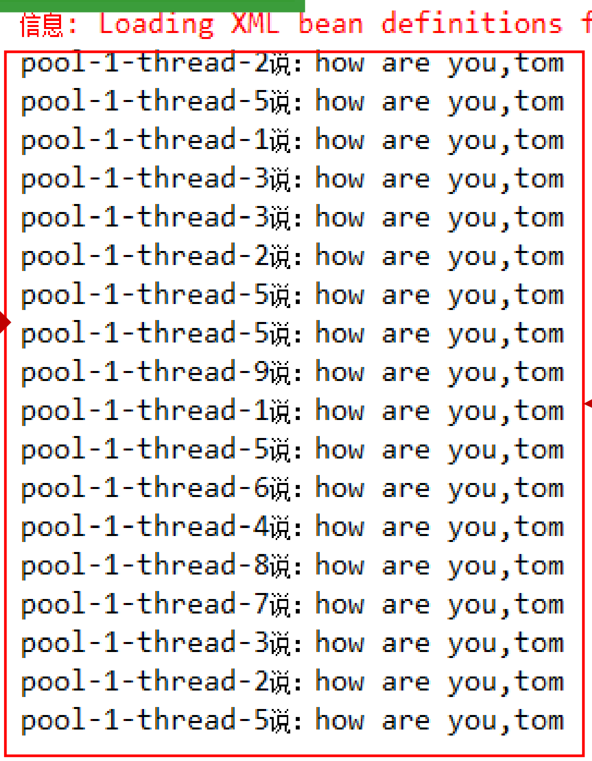
HelloSpringAction案例

• 多用户并发环境测试

并发测试，每个线程模拟一个打招呼的人



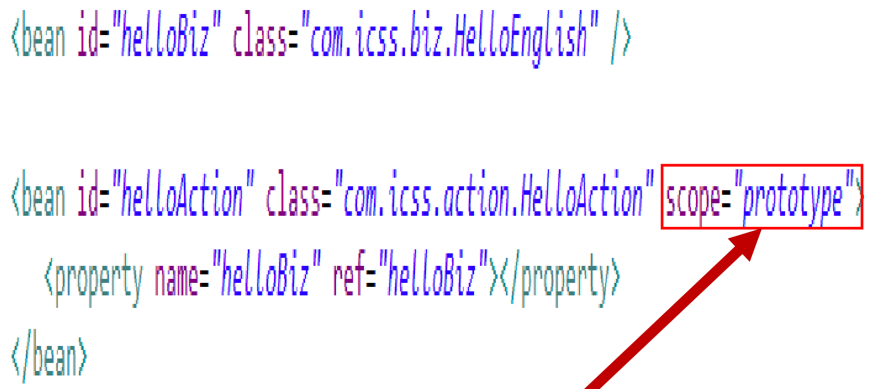
```java
public static void main(String[] args) {

    ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");

    ExecutorService pool = Executors.newCachedThreadPool();
    for(int i=0;i<100;i++) {
        pool.execute(new Runnable() {
            public void run() {
                //根bean的id，查找bean对象
                HelloAction  action = (HelloAction)context.getBean("helloAction");
                action.setPname(Thread.currentThread().getName());
                action.sayHello("tom");
            }
        });
    }

    pool.shutdown();
}
```



```
信息: Loading XML bean definitions from class path resource [beans.xml]
pool-1-thread-2说: how are you,tom
pool-1-thread-5说: how are you,tom
pool-1-thread-1说: how are you,tom
pool-1-thread-3说: how are you,tom
pool-1-thread-3说: how are you,tom
pool-1-thread-2说: how are you,tom
pool-1-thread-5说: how are you,tom
pool-1-thread-5说: how are you,tom
pool-1-thread-9说: how are you,tom
pool-1-thread-1说: how are you,tom
pool-1-thread-5说: how are you,tom
pool-1-thread-6说: how are you,tom
pool-1-thread-4说: how are you,tom
pool-1-thread-8说: how are you,tom
pool-1-thread-7说: how are you,tom
pool-1-thread-3说: how are you,tom
pool-1-thread-2说: how are you,tom
pool-1-thread-5说: how are you,tom
```

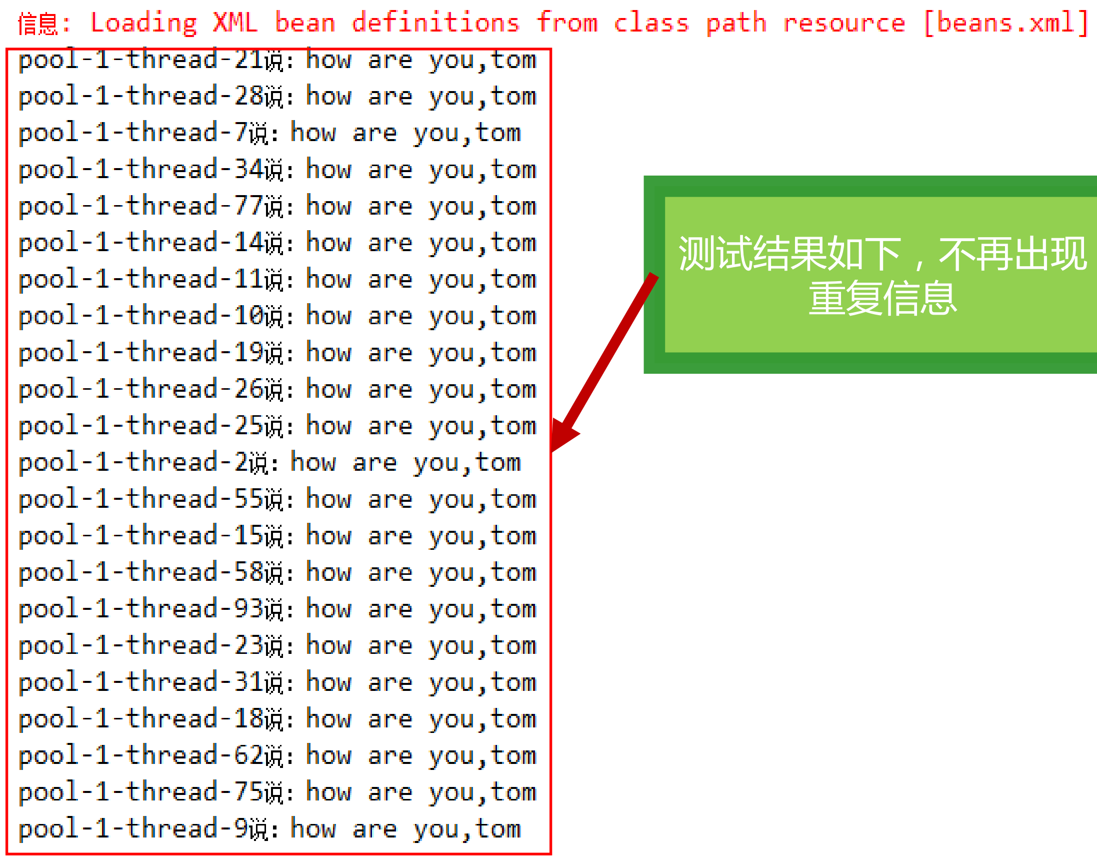
因为helloAction是单例的，当并发调用时，后面线程的赋值会替换前面线程的值，因此测试结果中有很多相同的打招呼信息

HelloSpringAction案例



```xml
<bean id="helloBiz" class="com.icss.biz.HelloEnglish" />

<bean id="helloAction" class="com.icss.action.HelloAction" scope="prototype">
   <property name="helloBiz" ref="helloBiz"></property>
</bean>
```

修改配置文件，增加scope="prototype"



```
信息: Loading XML bean definitions from class path resource [beans.xml]
pool-1-thread-21说: how are you,tom
pool-1-thread-28说: how are you,tom
pool-1-thread-7说: how are you,tom
pool-1-thread-34说: how are you,tom
pool-1-thread-77说: how are you,tom
pool-1-thread-14说: how are you,tom
pool-1-thread-11说: how are you,tom
pool-1-thread-10说: how are you,tom
pool-1-thread-19说: how are you,tom
pool-1-thread-26说: how are you,tom
pool-1-thread-25说: how are you,tom
pool-1-thread-2说: how are you,tom
pool-1-thread-55说: how are you,tom
pool-1-thread-15说: how are you,tom
pool-1-thread-58说: how are you,tom
pool-1-thread-93说: how are you,tom
pool-1-thread-23说: how are you,tom
pool-1-thread-31说: how are you,tom
pool-1-thread-18说: how are you,tom
pool-1-thread-62说: how are you,tom
pool-1-thread-75说: how are you,tom
pool-1-thread-9说: how are you,tom
```

测试结果如下，不再出现重复信息

Bean的生命周期回调处理

- 使用JSR-250 *@PostConstruct*和 *@PreDestroy*是bean对象生命周期callback的最好方式。
- 使用Spring 的*InitializingBean*和 *DisposableBean*接口也可以。使用接口的弊端是callback管理与Spring的代码产生了耦合，带来了不必要的麻烦。

Bean的生命周期回调处理

- Bean对象的初始化
  - init-method与destroy-method

编写初始化和析构方法

```java
public class UserBiz implements IUser {

    public UserBiz() {
        System.out.println("UserBiz构造...");
    }

    public void init() {
        System.out.println("UserBiz 初始化...");
    }

    public void destroy() {
        System.out.println("UserBiz 析构...");
    }
```

配置信息

```xml
<bean id="userDao" class="com.icss.dao.impl.UserDaoMysql"
        init-method="init" destroy-method="destroy"></bean>

<bean id="userBiz" class="com.icss.biz.impl.UserBiz"
        init-method="init" destroy-method="destroy"></bean>
```

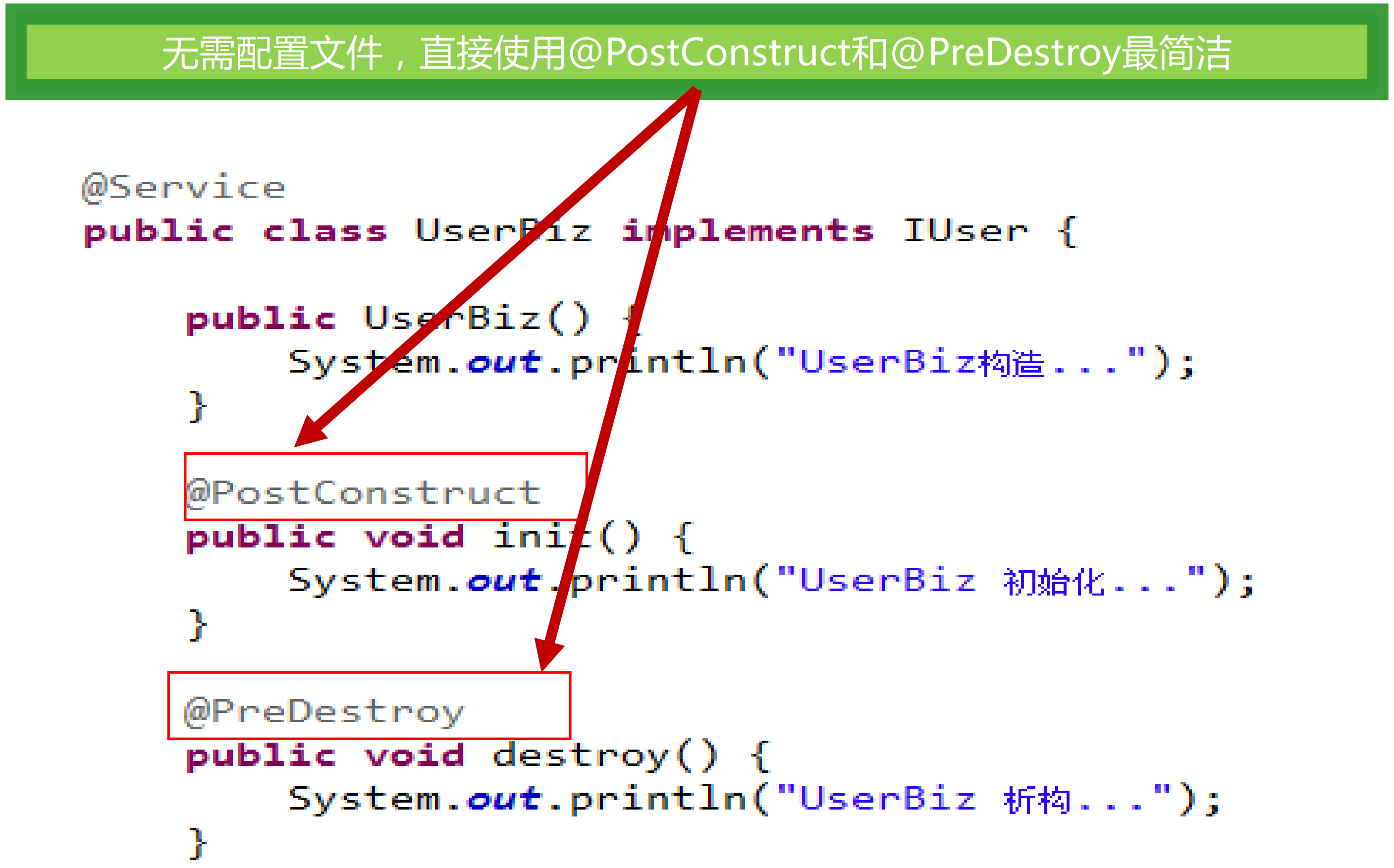
代码测试

```java
ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");

System.out.println("run....");

context.close();
```

```
信息: Overriding bean definition for bean 'userDao' with a different definition: replacing
UserDaoMysql 构造....
UserDaoMysql 初始化...
UserBiz构造...
UserBiz 初始化...
run....
十一月 07, 2019 6:05:37 上午 org.springframework.context.support.AbstractApplicationContext dc
信息: Closing org.springframework.context.support.ClassPathXmlApplicationContext@13c675d: :
UserBiz 析构...
UserDaoMysql 析构...
```

Bean的生命周期回调处理

- @PostConstruct和@PreDestroy



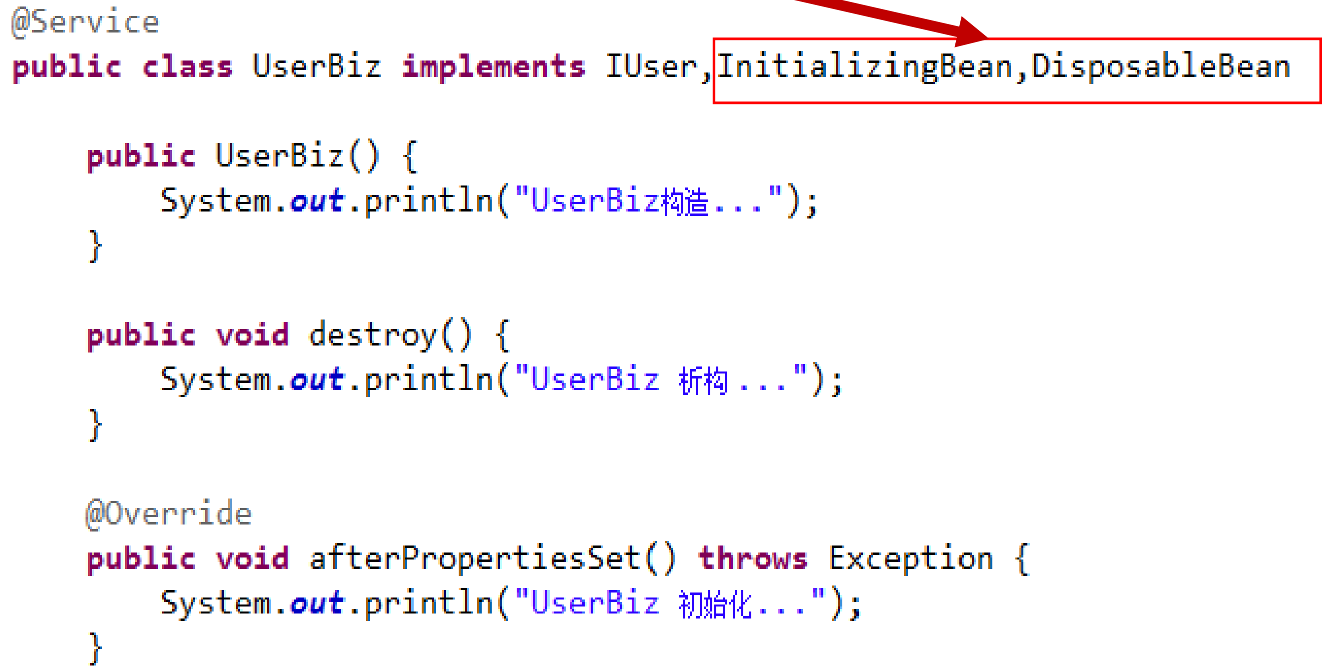
无需配置文件，直接使用@PostConstruct和@PreDestroy最简洁

```
@Service
public class UserBiz implements IUser {

    public UserBiz() {
        System.out.println("UserBiz构造...");
    }

    @PostConstruct
    public void init() {
        System.out.println("UserBiz 初始化...");
    }

    @PreDestroy
    public void destroy() {
        System.out.println("UserBiz 析构...");
    }
```

Bean的生命周期回调处理

- InitializingBean和DisposableBean

使用接口InitializingBean和DisposableBean，也可以进行bean的初始化和析构。这种模式对Spring的代码产生了耦合，不推荐使用



```
@Service
public class UserBiz implements IUser,InitializingBean,DisposableBean

    public UserBiz() {
        System.out.println("UserBiz构造...");
    }

    public void destroy() {
        System.out.println("UserBiz 析构 ...");
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("UserBiz 初始化...");
    }
```

Bean的生命周期回调处理

- 测试数据库连接的有效性案例
  - 初始化与析构的作用
    - 前面讲了如何配置Bean对象的初始化和析构方法，但是在什么场景使用呢？
    - 简单信息，如属性的初始值，一般在构造函数中进行。
    - 复杂信息、需要消耗较多时间的处理、可能出异常的处理，一般建议在初始化方法中进行。如Servlet可以在初始化方法中读取启动参数，在构造函数中无法读取。EJB作为重量级组件，更是需要初始化和钝化操作。

Bean的生命周期回调处理

- 案例说明
  - 编写一个Bean管理数据库，在系统启动时校验数据库连接是否有效。如配置错误，及时提醒

Bean的生命周期回调处理

- 配置文件及测试代码

数据库配置信息

```xml
<bean id="dbFactory" class="com.icss.util.DbFactory">
    <property name="driver" value="com.mysql.cj.jdbc.Driver"></property>
    <property name="url" value="jdbc:mysql://localhost:3306/staff?useSSL=false&amp;serverTimezone=UTC"></property>
    <property name="username" value="root"></property>
    <property name="password" value="123456"></property>
</bean>
```

```java
public class DbFactory {
    private String driver;
    private String url;
    private String username;
    private String password;

    @PostConstruct
    public void init() {
        try {
            Class.forName(this.driver);
            DriverManager.getConnection(url, username, password);
            System.out.println("数据库配置信息正确................");
        }catch(ClassNotFoundException e) {
            System.out.println("..............加载数据库驱动错误，请检查............");
        } catch (SQLException e) {
            System.out.println("..............数据库连接失败，请检查..............");
        }
    }
```

测试数据库连接

Bean的生命周期回调处理

- 使用钩子关闭IOC容器
  - 钩子介绍
    - 使用钩子，在非WEB环境下，可以优雅的关闭IOC容器。
    - 如富客户端的桌面环境，可以向JVM注册一个钩子。即使程序非正常退出，钩子函数也会被执行，这样在钩子函数中做环境清理工作，如关闭非托管资源，就是非常有效的方法。
    - 注册一个shutdown hook，需要调用ConfigurableApplicationContext接口中的registerShutdownHook()方法。

Bean的生命周期回调处理

- HelloSpringHook案例

添加钩子处理函数

```java
Runtime.getRuntime().addShutdownHook(new Thread(){
    public void run() {
        System.out.println("通过hook方法清除垃圾...");
    }
});
```

注册shutdown hook

```java
ConfigurableApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");

context.registerShutdownHook();

IHello  hi = (IHello)context.getBean("helloBean");
String hello = hi.sayHello("tom");
System.out.println(hello);
```

```java
IHello  hi = (IHello)context.getBean("helloBean");
String hello = hi.sayHello("tom");
System.out.println(hello);
try {
    Thread.sleep(1000);
    System.exit(1);             //非正常退出，不影响hook
} catch (Exception e) {
}
System.out.println("系统退出....");
```

```
信息: Loading XML bean definitions from class path resource [beans.xml]
how are you,tom
通过hook方法清除垃圾...
```

程序运行中，强制退出

非正常退出，不影响hook方法的调用

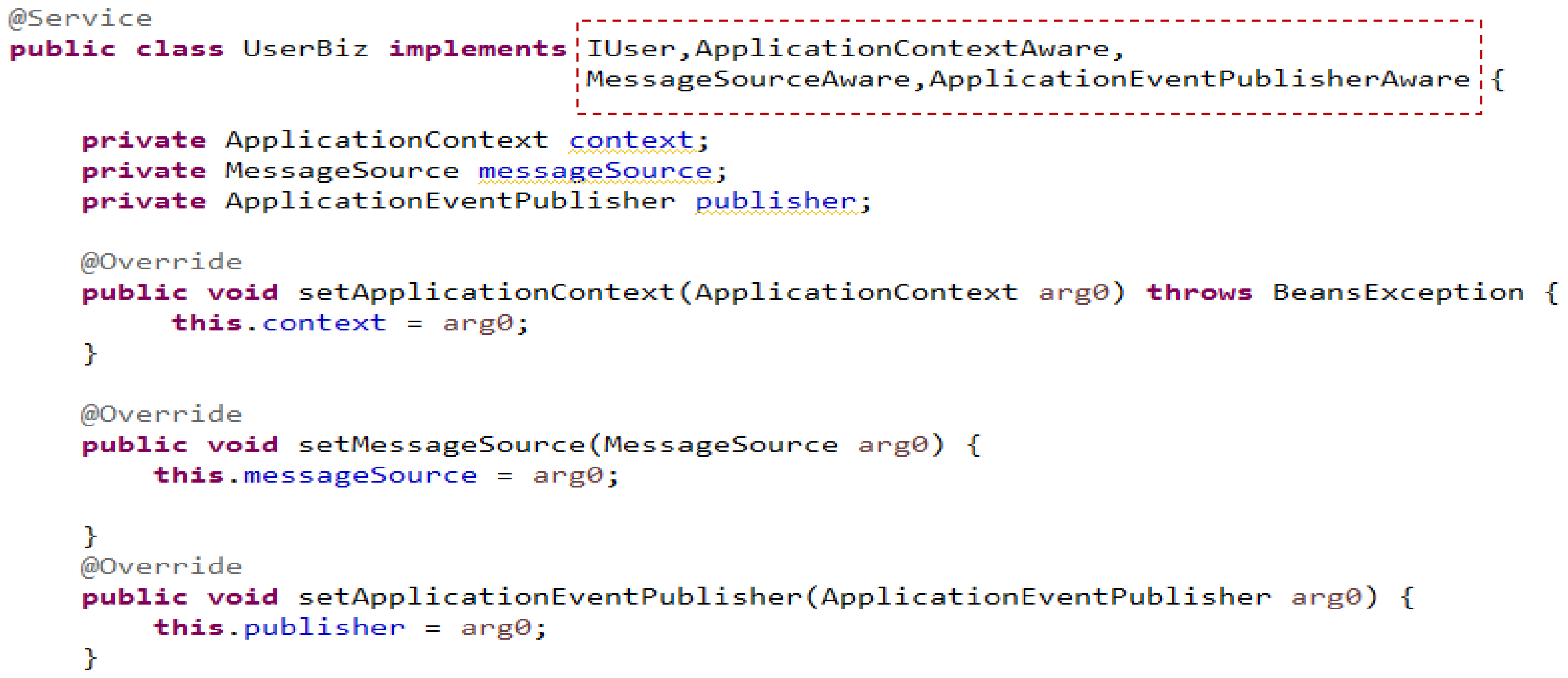
Aware接口

- Spring提供了很多aware接口，用于向bean对象提供IOC容器下的基础环境依赖。即在bean对象内获取各种环境信息数据。

Name	Injected Dependency
ApplicationContextAware	Declaring ApplicationContext
ApplicationEventPublisherAware	Event publisher of the enclosing ApplicationContext
BeanClassLoaderAware	Class loader used to load the bean classes.
BeanFactoryAware	Declaring BeanFactory
BeanNameAware	Name of the declaring bean
BootstrapContextAware	Resource adapter BootstrapContext the container runs in. Typically available only in JCA aware ApplicationContexts
LoadTimeWeaverAware	Defined weaver for processing class definition at load time
MessageSourceAware	Configured strategy for resolving messages (with support for parametrization and internationalization)
NotificationPublisherAware	Spring JMX notification publisher
PortletConfigAware	Current PortletConfig the container runs in. Valid only in a web-aware Spring ApplicationContext
PortletContextAware	Current PortletContext the container runs in. Valid only in a web-aware Spring ApplicationContext
ResourceLoaderAware	Configured loader for low-level access to resources
ServletConfigAware	Current ServletConfig the container runs in. Valid only in a web-aware Spring ApplicationContext
ServletContextAware	Current ServletContext the container runs in. Valid only in a web-aware Spring ApplicationContext

Aware接口

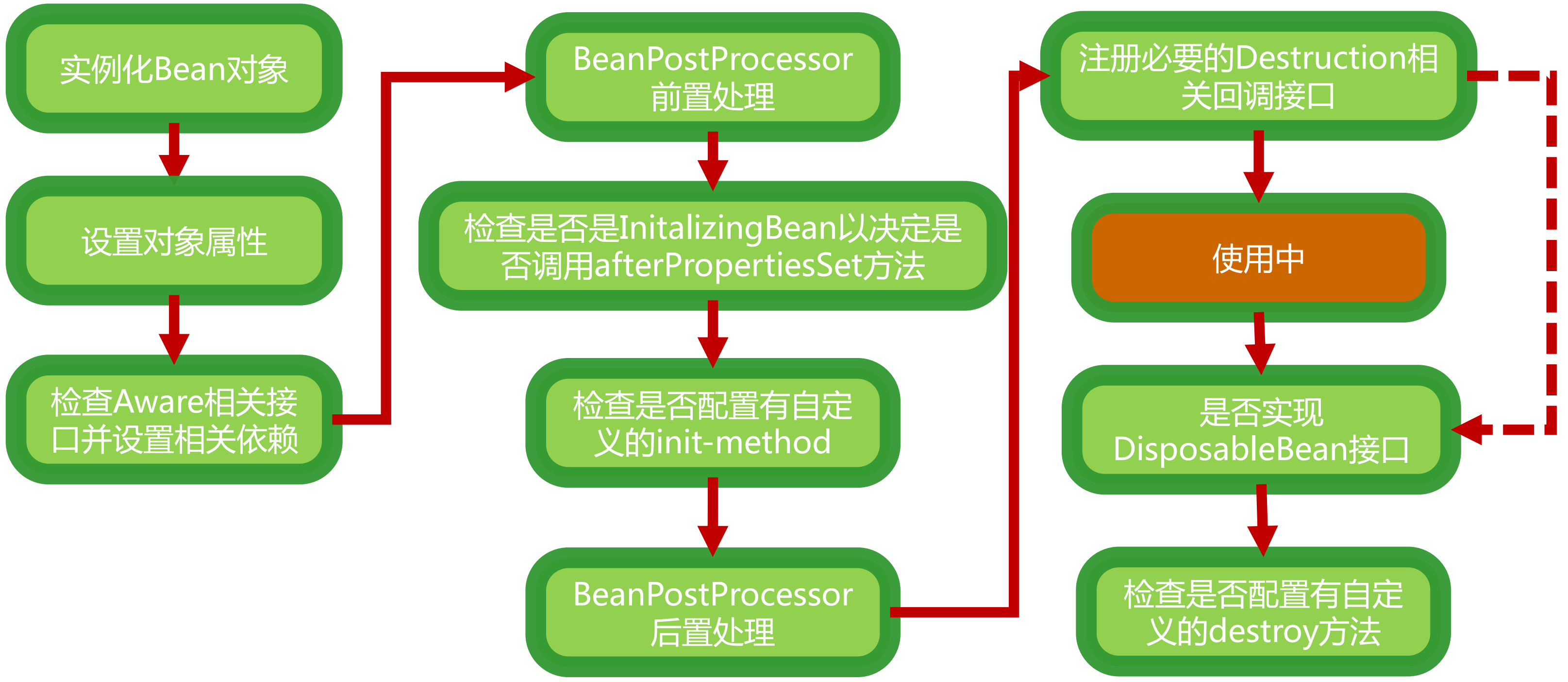
- UserBiz中注入基础环境案例



```
@Service
public class UserBiz implements IUser,ApplicationContextAware,
                                MessageSourceAware,ApplicationEventPublisherAware {

    private ApplicationContext context;
    private MessageSource messageSource;
    private ApplicationEventPublisher publisher;

    @Override
    public void setApplicationContext(ApplicationContext arg0) throws BeansException {
         this.context = arg0;
    }

    @Override
    public void setMessageSource(MessageSource arg0) {
        this.messageSource = arg0;

    }
    @Override
    public void setApplicationEventPublisher(ApplicationEventPublisher arg0) {
        this.publisher = arg0;
    }
```

BeanPostProcessor接口

- BeanPostProcessor接口介绍
  - IOC容器生成bean对象后，在bean初始化前后，你可以通过BeanPostProcessor接口定制你的业务逻辑，如日志跟踪等。
  - 配置BeanPostProcessor后Bean的使用过程如下：

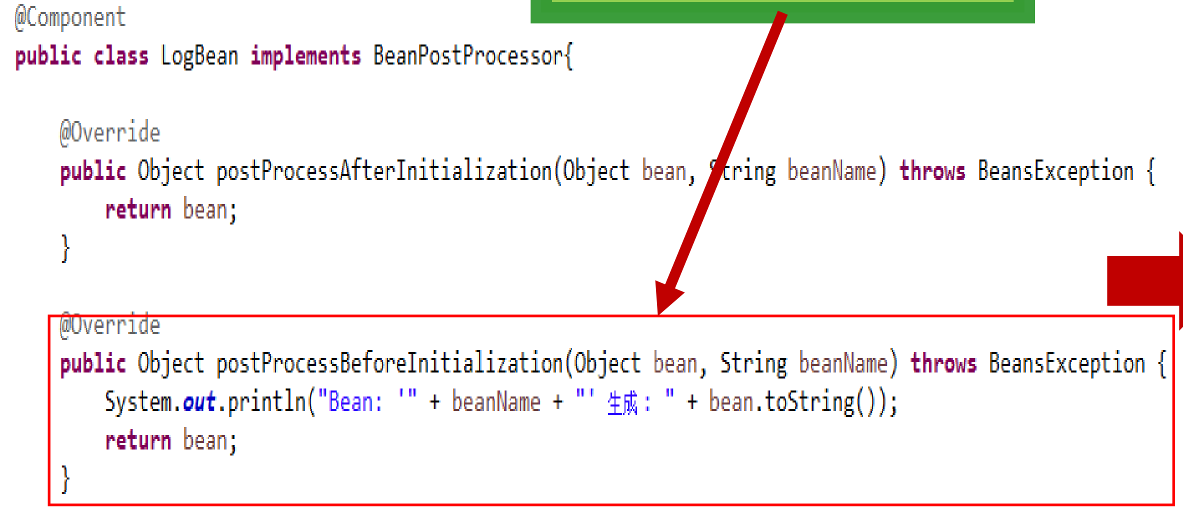


实例化Bean对象
设置对象属性
检查Aware相关接口并设置相关依赖
BeanPostProcessor前置处理
检查是否是InitalizingBean以决定是否调用afterPropertiesSet方法
检查是否配置有自定义的init-method
BeanPostProcessor后置处理
注册必要的Destruction相关回调接口
使用中
是否实现DisposableBean接口
检查是否配置有自定义的destroy方法

BeanPostProcessor接口

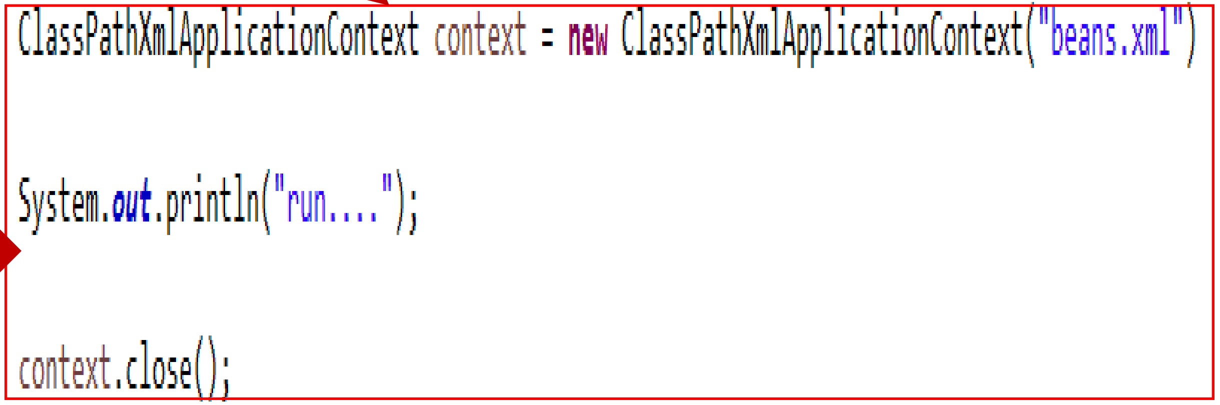
- Bean对象实例日志跟踪案例

在所有的bean对象初始化前打日志



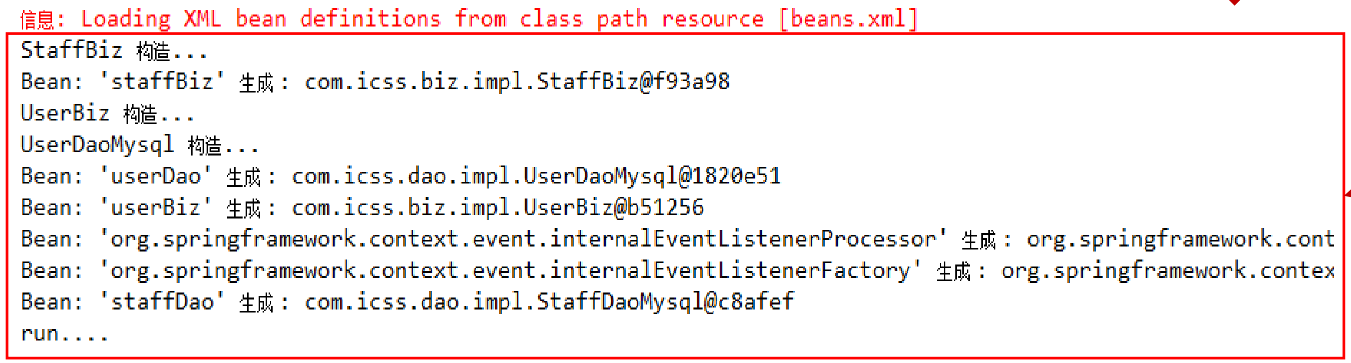
```java
@Component
public class LogBean implements BeanPostProcessor{

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        return bean;
    }

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
        System.out.println("Bean: '" + beanName + "' 生成: " + bean.toString());
        return bean;
    }
```

代码测试



```java
ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("beans.xml")

System.out.println("run....");

context.close();
```

测试结果



```
信息: Loading XML bean definitions from class path resource [beans.xml]
StaffBiz 构造...
Bean: 'staffBiz' 生成: com.icss.biz.impl.StaffBiz@f93a98
UserBiz 构造...
UserDaoMysql 构造...
Bean: 'userDao' 生成: com.icss.dao.impl.UserDaoMysql@1820e51
Bean: 'userBiz' 生成: com.icss.biz.impl.UserBiz@b51256
Bean: 'org.springframework.context.event.internalEventListenerProcessor' 生成: org.springframework.cont
Bean: 'org.springframework.context.event.internalEventListenerFactory' 生成: org.springframework.contex
Bean: 'staffDao' 生成: com.icss.dao.impl.StaffDaoMysql@c8afef
run....
```

FactoryBean接口

- FactoryBean介绍
  - FactoryBean就是对一个复杂Bean的包装，可以在FactoryBean中进行初始化，然后把初始化的值传给它包装的对象。
  - FactoryBean接口在Spring framework框架自身，有大量的实现，如用于创建动态代理对象的ProxyFactoryBean。
  - 实现FactoryBean中的getObject()方法，返回真正需要的对象。

FactoryBean接口

- FactoryBean包装DbFactory案例

配置数据源

```xml
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.cj.jdbc.Driver" />
    <property name="url"
        value="jdbc:mysql://localhost:3306/staff?useSSL=false&amp;serverTimezone=UTC" />
    <property name="username" value="root" />
    <property name="password" value="123456" />
 </bean>
```

```java
@Override
public void afterPropertiesSet() throws Exception {
    dataSource.getConnection();              //打开数据库测试
    dbFactory = new DbFactory();
    dbFactory.setUrl(this.dataSource.getUrl());
    dbFactory.setPassword(this.dataSource.getPassword());
    dbFactory.setUsername(this.dataSource.getUsername());
}
```

在初始化方法中，创建DbFactory并赋值

```java
@Component("dbFactoryBean")
public class DbFactoryBean implements InitializingBean,FactoryBean<DbFactory> {

    @Autowired
    private org.springframework.jdbc.datasource.DriverManagerDataSource dataSource;
    private DbFactory dbFactory;

    @Override
    public DbFactory getObject() throws Exception {
        return this.dbFactory;
    }

    @Override
    public Class<?> getObjectType() {
        return DbFactory.class;
    }

    @Override
    public boolean isSingleton() {
        return true;
    }
```

编写FactoryBean，封装DbFactory

```java
ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
DbFactory db = (DbFactory)context.getBean("dbFactoryBean");
System.out.println(db.getUsername());
System.out.println(db.getUrl());

context.close();
```

```
信息: Loaded JDBC driver: com.mysql.cj.jdbc.Driver
root
jdbc:mysql://localhost:3306/staff?useSSL=false&serverTimezone=UTC
```

代码测试

JSR相关注解

- 如下注解，在Spring Framework的任何地方都被支持：
  - @Autowired
  - @Qualifier
  - @Resource (javax.annotation) *if JSR-250 is present*
  - @ManagedBean (javax.annotation) *if JSR-250 is present*
  - @Inject (javax.inject) *if JSR-330 is present*
  - @Named (javax.inject) *if JSR-330 is present*
  - @PersistenceContext (javax.persistence) *if JPA is present*
  - @PersistenceUnit (javax.persistence) *if JPA is present*
  - @Required
  - @Transactional

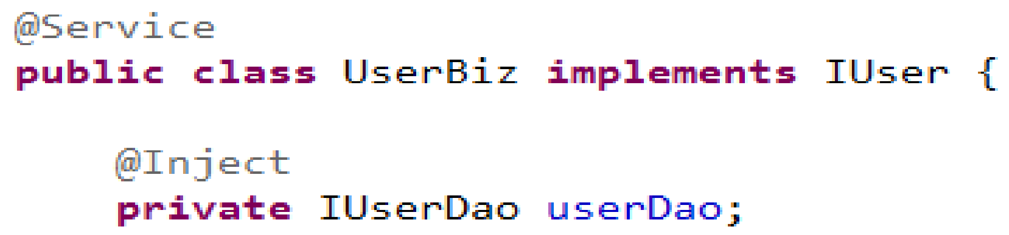
JSR相关注解

- Spring与JSR330对应注解

Spring	javax.inject.*	javax.inject restrictions / comments
@Autowired	@Inject	@Inject has no 'required' attribute; can be used with Java 8’s Optional instead.
@Component	@Named / @ManagedBean	JSR-330 does not provide a composable model, just a way to identify named components.
@Scope("singleton")	@Singleton	The JSR-330 default scope is like Spring’s prototype. However, in order to keep it consistent with Spring’s general defaults, a JSR-330 bean declared in the Spring container is a singleton by default. In order to use a scope other than singleton, you should use Spring’s @Scope annotation. javax.inject also provides a @Scope annotation. Nevertheless, this one is only intended to be used for creating your own annotations.
@Qualifier	@Qualifier / @Named	javax.inject.Qualifier is just a meta-annotation for building custom qualifiers. Concrete String qualifiers (like Spring’s @Qualifier with a value) can be associated through javax.inject.Named.
@Value	-	no equivalent
@Required	-	no equivalent
@Lazy	-	no equivalent
ObjectFactory	Provider	javax.inject.Provider is a direct alternative to Spring’s ObjectFactory, just with a shorter get() method name. It can also be used in combination with Spring’s @Autowired or with non-annotated constructors and setter methods.

JSR相关注解

- @Inject
  - @Inject 是JSR 330的注解，在使用@Autowired地方，可以使用 @Inject代替
  - 案例测试：
    - A、导入javaee-api-7.0.jar
    - B、原来使用@Autowired注入的userDao，修改为@Inject

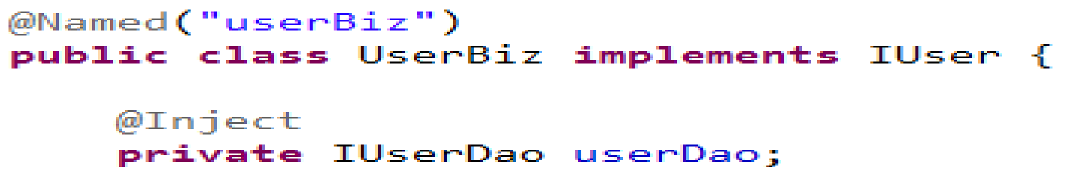


```
@Service
public class UserBiz implements IUser {

    @Inject
    private IUserDao userDao;
```

    - C、测试ok

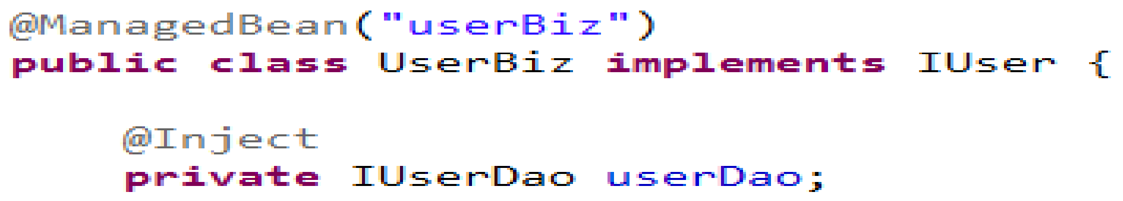
JSR相关注解

- @Name与@ManagedBean
  - 使用@Named 或 @ManagedBean，可以代替@Component。
  - managed bean概念，源于javaEE的JSF部分。
    - A、用@Named("userBiz")替代原来的@Service



```
@Named("userBiz")
public class UserBiz implements IUser {

    @Inject
    private IUserDao userDao;
```

    - B、测试ok
    - C、修改为@ManagedBean("userBiz") ，效果相同



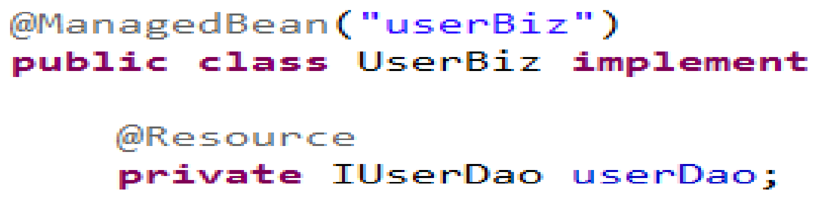
```
@ManagedBean("userBiz")
public class UserBiz implements IUser {

    @Inject
    private IUserDao userDao;
```

JSR相关注解

- @PostConstruct和@PreDestroy
  - javax.annotation.PostConstruct 和 javax.annotation.PreDestroy ，这两个注解前面已经测试过，在bean的初始化和析构时，优先推荐。

JSR相关注解说明

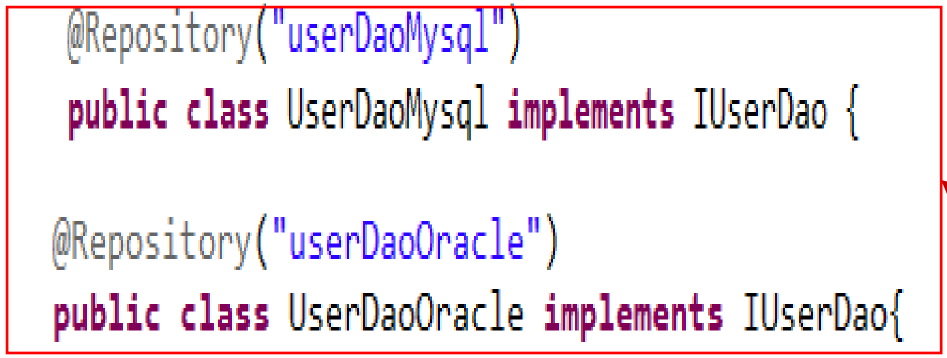
- **@Resource**
  - Spring支持使用JSR-250 @Resource注入数据。
  - @Resource可以应用在属性、Set方法上，注入数据。
  - 使用@Resource代替@Inject、@Autowired
  - 与@Autowired相反，@Resource默认的装配方式是byName



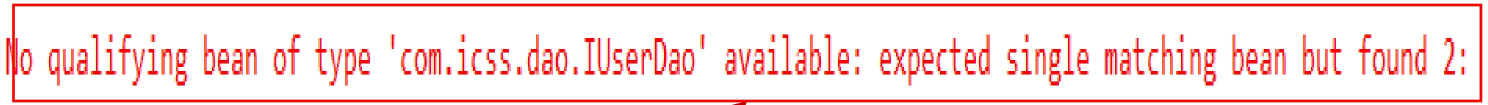
```
@ManagedBean("userBiz")
public class UserBiz implements IUser {

    @Resource
    private IUserDao userDao;
```

JSR相关注解说明

- @Resource解决冲突案例



```
@Repository("userDaoMysql")
public class UserDaoMysql implements IUserDao {

@Repository("userDaoOracle")
public class UserDaoOracle implements IUserDao{
```

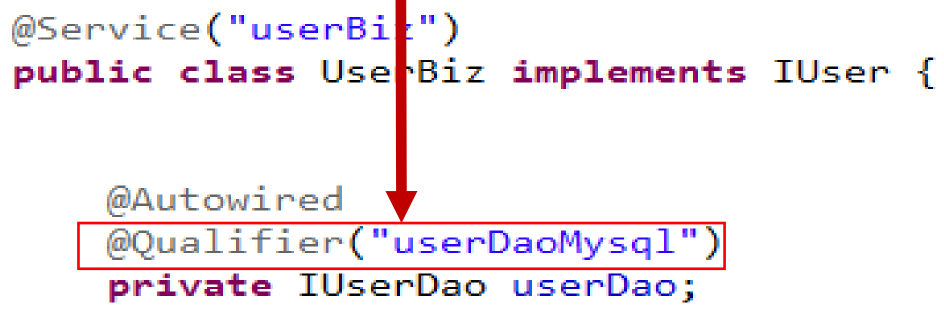


```
No qualifying bean of type 'com.icss.dao.IUserDao' available: expected single matching bean but found 2:
```

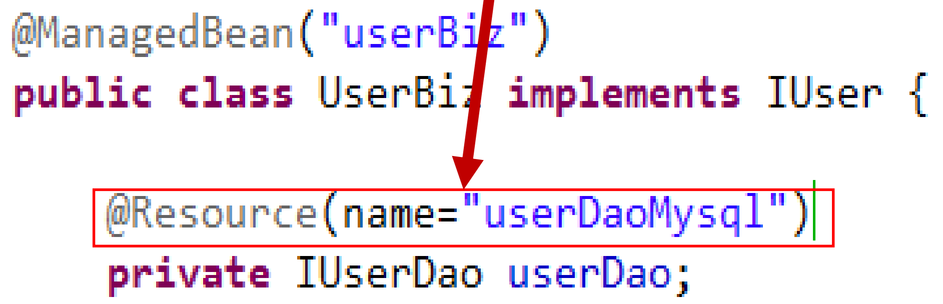
当同时定义了两个IUserDao类型时，注入时会出错

解决方法：@Qualifier("userDaoMysql")指明使用哪一个

解决方法：用@Resource(name="userDaoMysql")也可解决



```
@Service("userBiz")
public class UserBiz implements IUser {

    @Autowired
    @Qualifier("userDaoMysql")
    private IUserDao userDao;
```

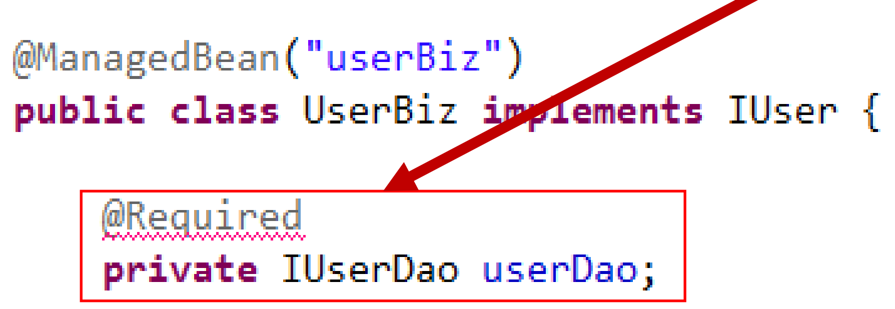


```
@ManagedBean("userBiz")
public class UserBiz implements IUser {

    @Resource(name="userDaoMysql")
    private IUserDao userDao;
```

Spring相关注解

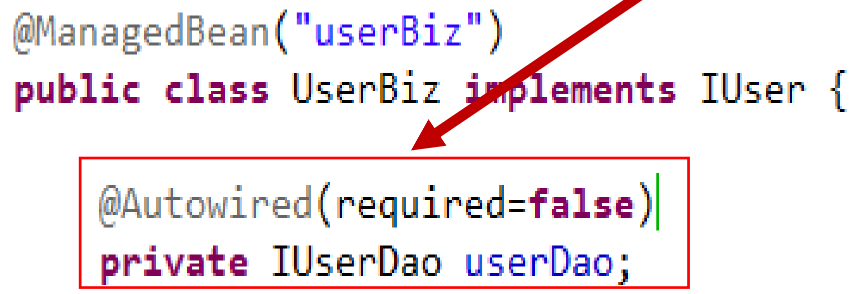
- @Required

@Required注解，只能应用在set方法上，不能应用于属性



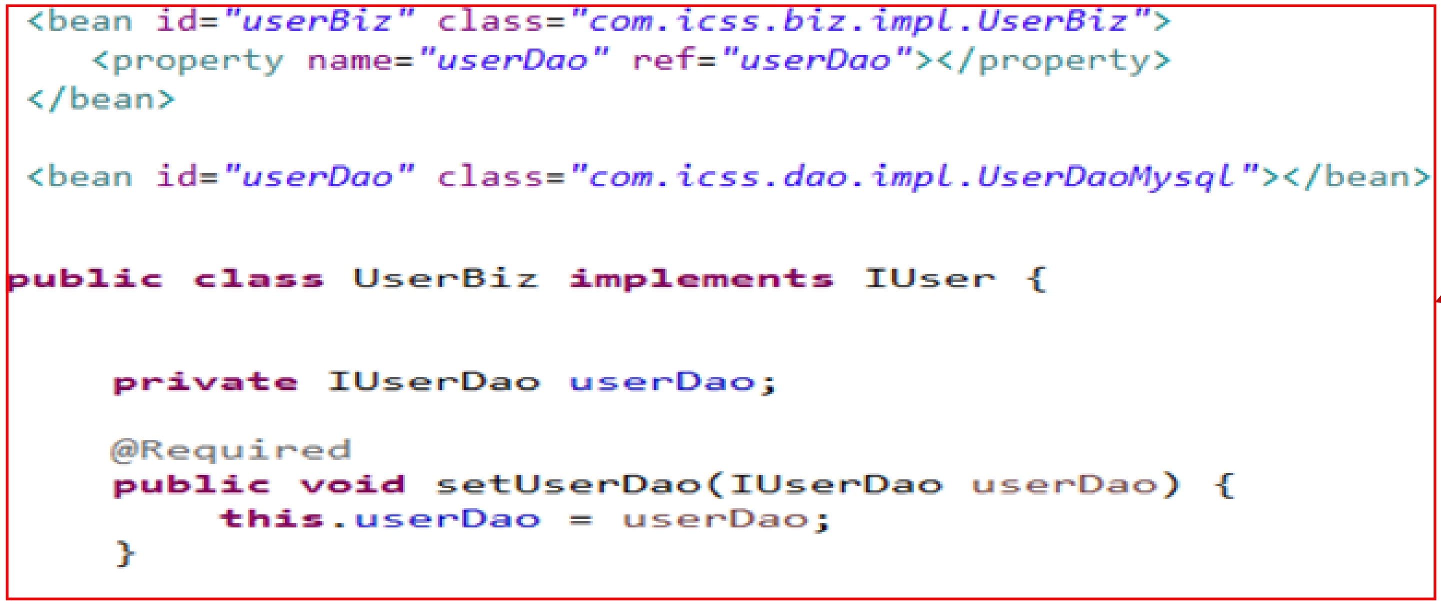
```
@ManagedBean("userBiz")
public class UserBiz implements IUser {

    @Required
    private IUserDao userDao;
```

当使用@Autowired(required=false)时，即使找不到合适的IUserDao对象注入，也可以



```
@ManagedBean("userBiz")
public class UserBiz implements IUser {

    @Autowired(required=false)
    private IUserDao userDao;
```

@Required强制要求使用此方式配置才可以
如果使用注解方式创建bean对象，报错

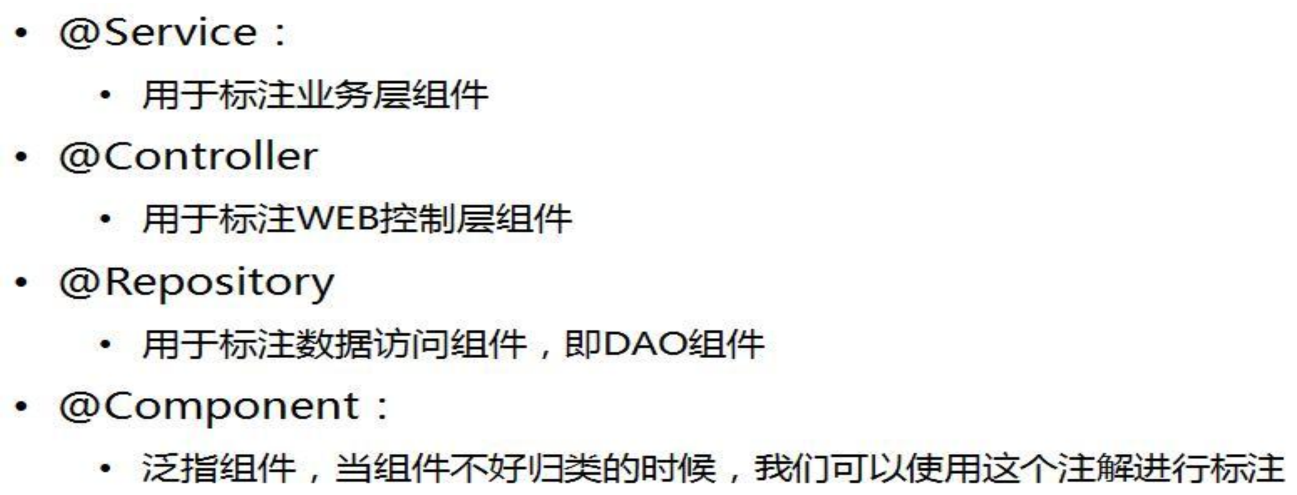


```
<bean id="userBiz" class="com.icss.biz.impl.UserBiz">
    <property name="userDao" ref="userDao"></property>
</bean>

<bean id="userDao" class="com.icss.dao.impl.UserDaoMysql"></bean>

public class UserBiz implements IUser {

    private IUserDao userDao;

    @Required
    public void setUserDao(IUserDao userDao) {
        this.userDao = userDao;
    }
```

结论：**@Required使用很不灵活，不推荐使用**

Spring相关注解

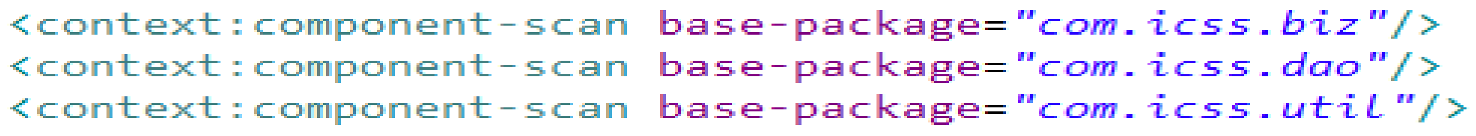
- @Component
  - @Component是Bean的通用注解，在不同的层，也可以使用具体的注解。
  - @Controller、@Service、@Repository与@Component等效



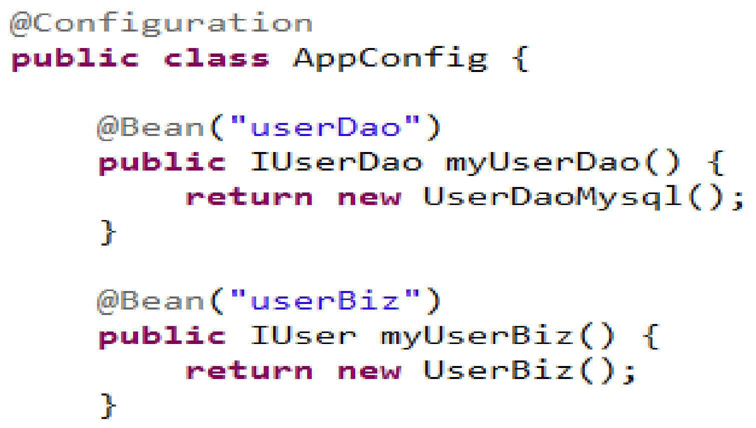
- @Service：
  - 用于标注业务层组件
- @Controller
  - 用于标注WEB控制层组件
- @Repository
  - 用于标注数据访问组件，即DAO组件
- @Component：
  - 泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注

Spring相关注解

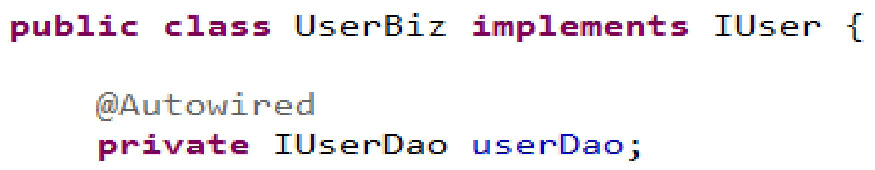
- @Bean和@Configuration
  - @Bean注解方法，即通知IOC容器，把方法返回的对象当成bean



```
<context:component-scan base-package="com.icss.biz"/>
<context:component-scan base-package="com.icss.dao"/>
<context:component-scan base-package="com.icss.util"/>
```



```
@Configuration
public class AppConfig {

    @Bean("userDao")
    public IUserDao myUserDao() {
        return new UserDaoMysql();
    }

    @Bean("userBiz")
    public IUser myUserBiz() {
        return new UserBiz();
    }
```



```
public class UserBiz implements IUser {

    @Autowired
    private IUserDao userDao;
```

  - 用这种方式创建bean对象，与@Component 和 <bean id="" class=""/>效果相同

Spring相关注解

- @Primary
  - 当使用Autowire注入对象时，可能会遇到同一个类型多个对象存在的情况

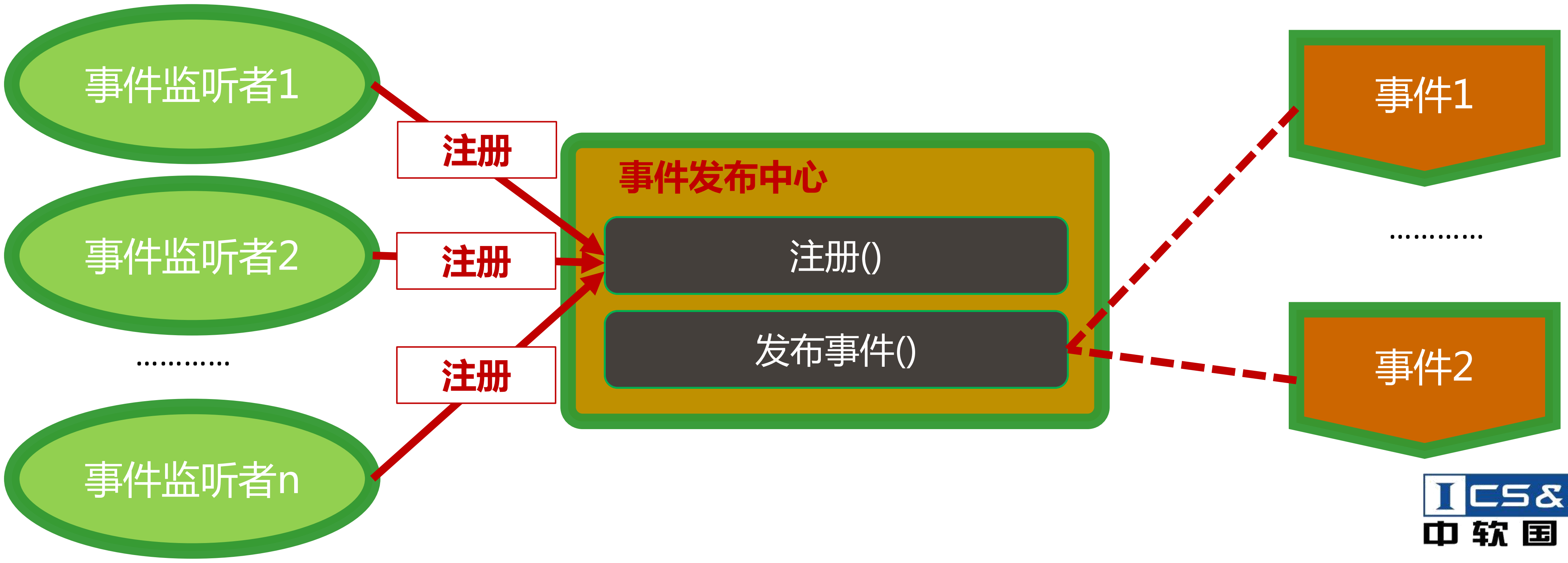
```
@Repository("userDaoMysql")
public class UserDaoMysql implements IUserDao {

@Repository("userDaoOracle")
public class UserDaoOracle implements IUserDao{

@Service
public class UserBiz implements IUser

    @Autowired
    private IUserDao userDao;
```

  - 这时，可以使用@Primary表示这个bean被优先注入

```
@Repository("userDaoMysql")
@Primary
public class UserDaoMysql implements IUserDao {
```

标准事件与自定义事件

- ApplicationContext基于Observer模式提供了针对Bean的事件传播功能。
- 通过ApplicationContext的publishEvent方法，可以将事件通知系统内所有的ApplicationListener，标准自定义事件结构描述如下：



标准事件

- Spring Framework提供了如下标准事件：

标准事件

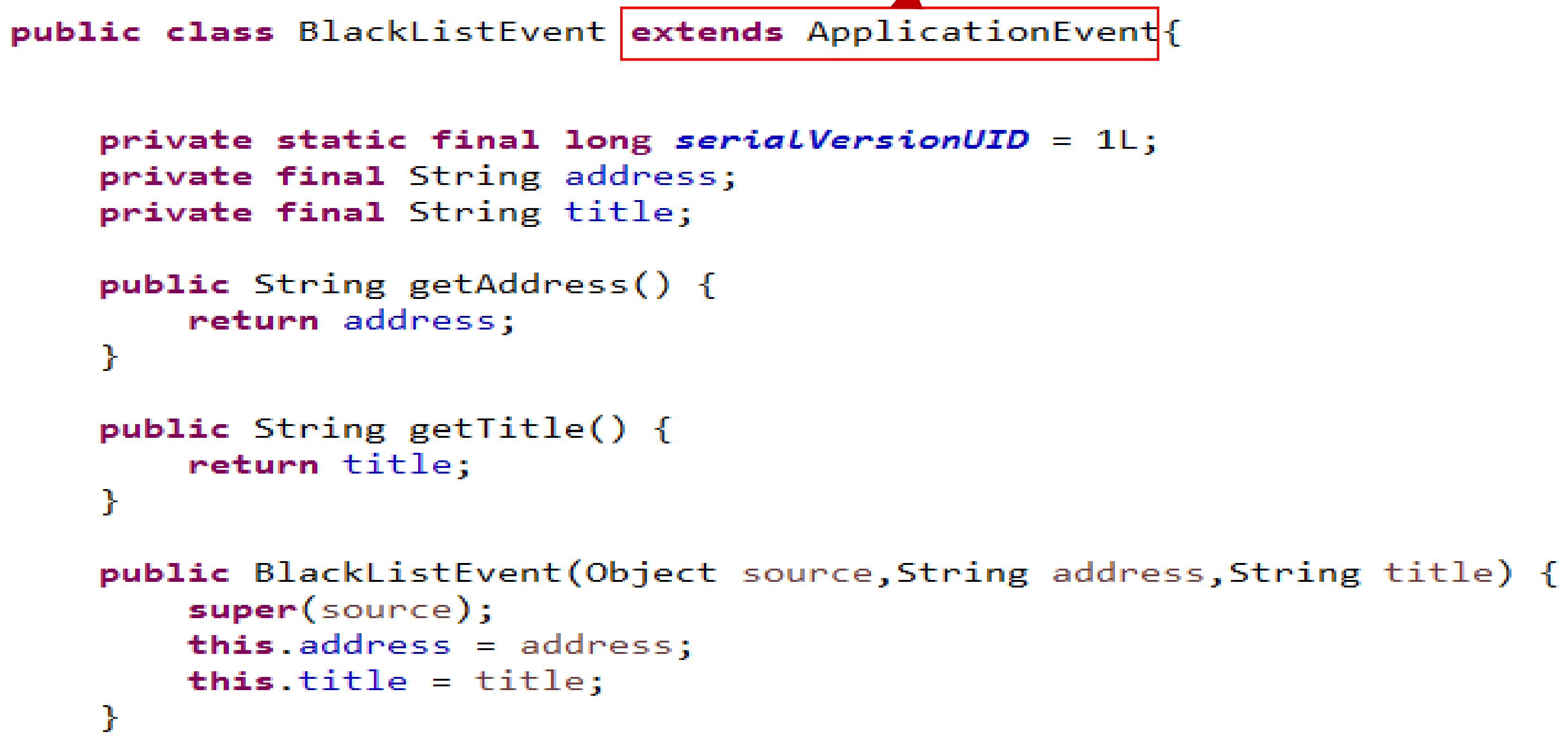
- Spring Framework提供了如下标准事件：

事件	解释说明
ContextRefreshedEvent	当ApplicationContext初始化或刷新时发送的事件。这里的初始化意味着：所有的bean被装载，singleton被预实例化，以及ApplicationContext已就绪可用
ContextStartedEvent	当容器调用ConfigurableApplicationContext的Start()方法开始/重新开始容器时触发该事件
ContextStoppedEvent	当容器调用ConfigurableApplicationContext的Stop()方法停止容器时触发该事件
ContextClosedEvent	当使用ApplicationContext的close()方法结束上下文时发送的事件。这里的结束意味着：singleton bean 被销毁
RequestHandledEvent	一个与web相关的事件，告诉所有的bean一个HTTP请求已经被响应了（也就是在一个请求结束后会发送该事件）。注意，只有在Spring中使用了DispatcherServlet的web应用才能使用

自定义事件邮件通知案例

消息类必须继承ApplicationEvent

- 步骤一：定义消息



```
public class BlackListEvent extends ApplicationEvent{

    private static final long serialVersionUID = 1L;
    private final String address;
    private final String title;

    public String getAddress() {
        return address;
    }

    public String getTitle() {
        return title;
    }

    public BlackListEvent(Object source,String address,String title) {
        super(source);
        this.address = address;
        this.title = title;
    }
```

自定义事件邮件通知案例

• 步骤二：发送消息

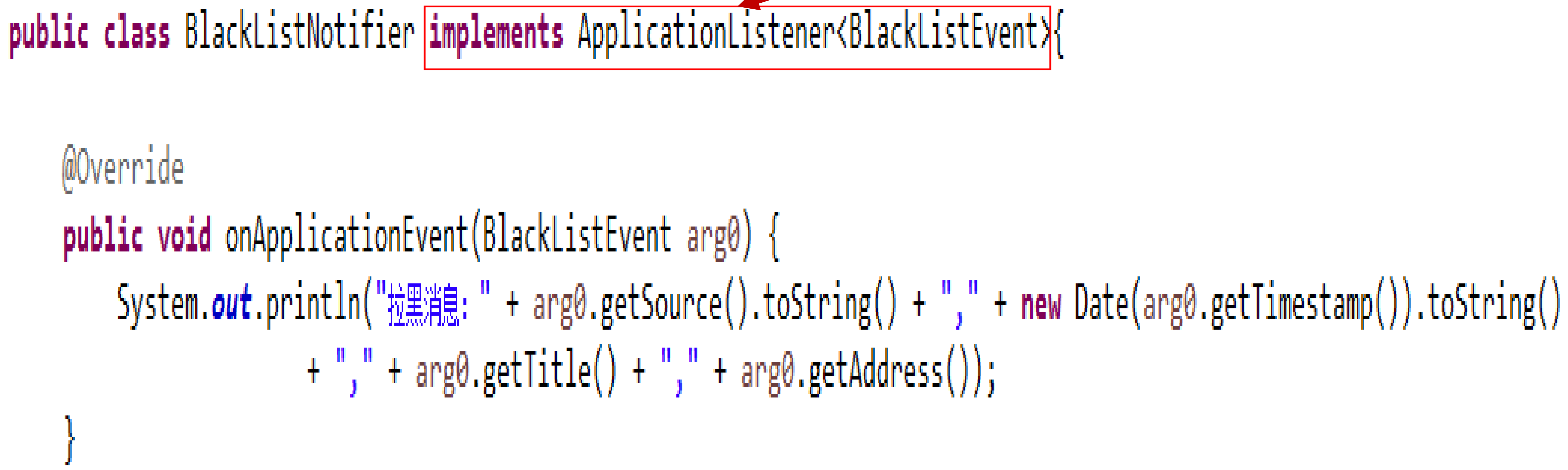
注入ApplicationEventPublisherAware，用于发送消息

```java
public class EmailService implements ApplicationEventPublisherAware{
    private ApplicationEventPublisher publisher;
    private List<String> blackList;


    public void setBlackList(List<String> blackList) {
        this.blackList = blackList;
    }

    @Override
    public void setApplicationEventPublisher(ApplicationEventPublisher arg0) {
        this.publisher = arg0;
    }

    public void sendEmail(String address, String content,String title) {
        if (blackList.contains(address)) {
            publisher.publishEvent(new BlackListEvent(this, address,title));
            return;
        }
        // send email...
        System.out.println("发送邮件，目标: " + address + ",内容: "  + content);
    }
```

自定义事件邮件通知案例

- 步骤三：接收消息

实现ApplicationListener接口，接收消息



```
public class BlackListNotifier implements ApplicationListener<BlackListEvent>{

    @Override
    public void onApplicationEvent(BlackListEvent arg0) {
        System.out.println("拉黑消息: " + arg0.getSource().toString() + "," + new Date(arg0.getTimestamp()).toString()
                + "," + arg0.getTitle() + "," + arg0.getAddress());
    }
```

自定义事件邮件通知案例

- 步骤四：测试

```xml
<bean  class="com.icss.biz.BlackListNotifier"> </bean>

<bean id="emailService" class="com.icss.biz.EmailService">
     <property name="blackList">
        <list>
          <value>tom1@qq.com</value>
          <value>tom2@qq.com</value>
          <value>tom3@qq.com</value>
          <value>jack1@qq.com</value>
          <value>jack2@qq.com</value>
        </list>
     </property>
</bean>
```

```java
public static void main(String[] args) {
    ApplicationContext app = new ClassPathXmlApplicationContext("beans.xml");

    EmailService e = app.getBean(EmailService.class);

    e.sendEmail("aa@sina.com","下午三点，全体开会", "开会通知");

    e.sendEmail("tom2@qq.com","下午三点，全体开会", "开会通知");
}
```

```
信息: Loading XML bean definitions from class path resource [beans.xml]
发送邮件，目标：aa@sina.com,内容：下午三点，全体开会
拉黑消息：com.icss.biz.EmailService@19f99ea,Fri Nov 08 09:16:56 CST 2019,开会通知,tom2@qq.com
```

接收多消息案例

定义消息接收者

```java
public class BlackListNotifier implements ApplicationListener<ApplicationEvent>{

    @Override
    public void onApplicationEvent(ApplicationEvent event) {
        if(event instanceof BlackListEvent) {
            BlackListEvent arg0 = (BlackListEvent)event;
            System.out.println("拉黑消息: " + arg0.getSource().toString() + ","
                    + new Date(arg0.getTimestamp()).toString()
                    + "," + arg0.getTitle() + "," + arg0.getAddress());
        }else if(event instanceof ContextRefreshedEvent) {
            ContextRefreshedEvent arg0 = (ContextRefreshedEvent)event;
            System.out.println(" 服务器刷新...." + new Date(arg0.getTimestamp()).toString() );

        }else if(event instanceof ContextClosedEvent) {
            ContextClosedEvent arg0 = (ContextClosedEvent)event;
            System.out.println(" 服务器关闭...." + new Date(arg0.getTimestamp()).toString() );
        }
```

发送消息

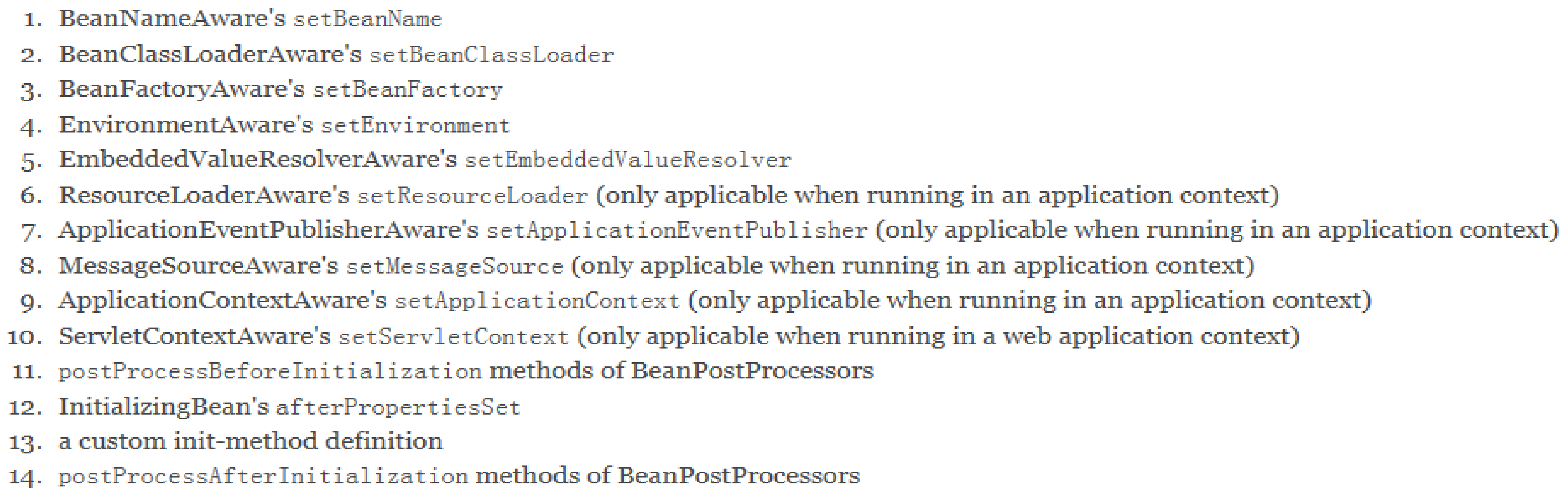
```java
public static void main(String[] args) {
    ConfigurableApplicationContext app = new ClassPathXmlApplicationContext("beans.xml");

    EmailService e = app.getBean(EmailService.class);

    e.sendEmail("aa@sina.com","下午三点,全体开会", "开会通知");

    e.sendEmail("tom2@qq.com","下午三点,全体开会", "开会通知");

    app.close();
}
```

运行结果

```
信息: Loading XML bean definitions from class path resource [beans.xml]
服务器刷新....Fri Nov 08 10:09:08 CST 2019
发送邮件,目标: aa@sina.com,内容: 下午三点,全体开会
拉黑消息: com.icss.biz.EmailService@1caeb3e,Fri Nov 08 10:09:08 CST 2019,开会通知,tom2@qq.com
十一月 08, 2019 10:09:08 上午 org.springframework.context.support.AbstractApplicationContext doClose
信息: Closing org.springframework.context.support.ClassPathXmlApplicationContext@13c675d: startup d
服务器关闭....Fri Nov 08 10:09:08 CST 2019
```

BeanFactory接口

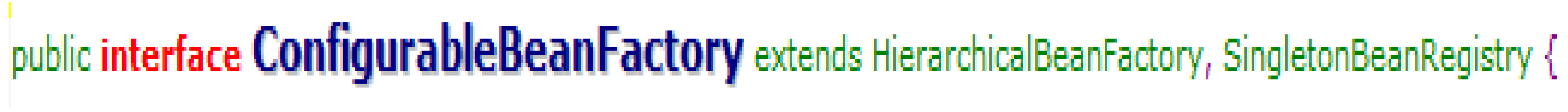
- BeanFactory的实现类，需要管理一群bean definition，每个bean definition有一个唯一的识别ID。
- BeanFactory是spring组件的注册中心和配置中心。
- 配置Spring bean对象时，最好使用DI方式，而不是从BeanFactory从查找配置。
- BeanFactory的实现类，应该尽可能的实现bean生命期接口，如下列表



1. BeanNameAware's `setBeanName`
2. BeanClassLoaderAware's `setBeanClassLoader`
3. BeanFactoryAware's `setBeanFactory`
4. EnvironmentAware's `setEnvironment`
5. EmbeddedValueResolverAware's `setEmbeddedValueResolver`
6. ResourceLoaderAware's `setResourceLoader` (only applicable when running in an application context)
7. ApplicationEventPublisherAware's `setApplicationEventPublisher` (only applicable when running in an application context)
8. MessageSourceAware's `setMessageSource` (only applicable when running in an application context)
9. ApplicationContextAware's `setApplicationContext` (only applicable when running in an application context)
10. ServletContextAware's `setServletContext` (only applicable when running in a web application context)
11. `postProcessBeforeInitialization` methods of BeanPostProcessors
12. InitializingBean's `afterPropertiesSet`
13. a custom init-method definition
14. `postProcessAfterInitialization` methods of BeanPostProcessors

HierarchicalBeanFactory接口

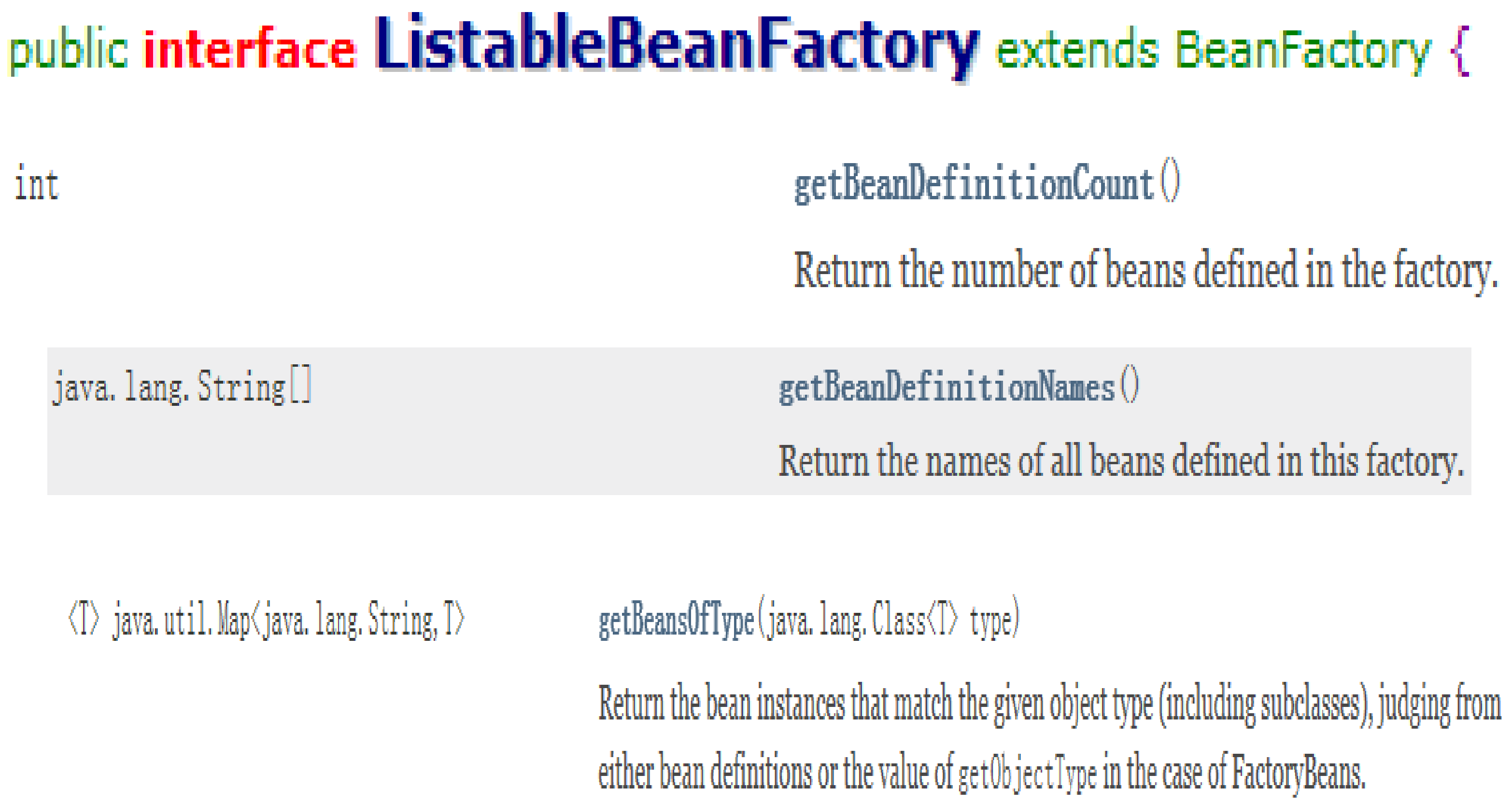
- IOC容器ApplicationContext的父接口。
- IOC容器是树状结构，这个结构源于HierarchicalBeanFactory的树结构。
- 使用getBean()查找bean对象时，如果在当前实例中未找到，则马上到父工厂中去查找。



```
public interface ConfigurableBeanFactory extends HierarchicalBeanFactory, SingletonBeanRegistry {
```

ListableBeanFactory接口

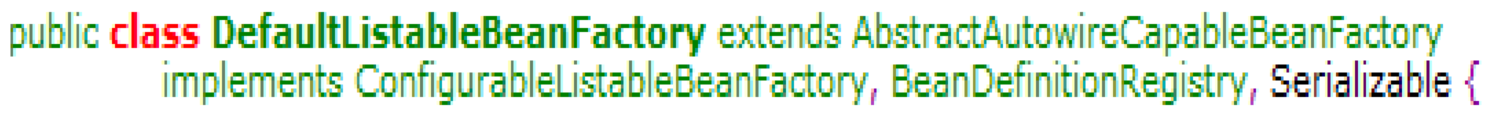
- 不仅可以用getBean()的方式查找bean对象。还可以使用迭代方式，找到Bean工厂中的所有Bean实例



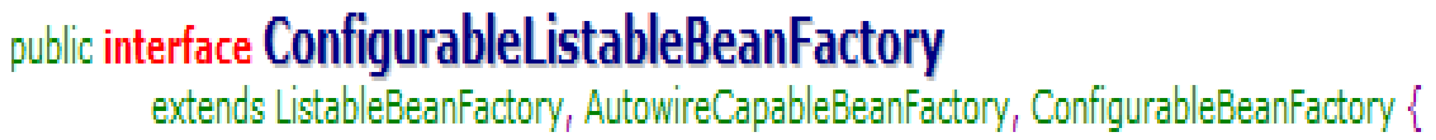
```
public interface ListableBeanFactory extends BeanFactory {
```

int	getBeanDefinitionCount() Return the number of beans defined in the factory.
java.lang.String[]	getBeanDefinitionNames() Return the names of all beans defined in this factory.
<T> java.util.Map<java.lang.String,T>	getBeansOfType(java.lang.Class<T> type) Return the bean instances that match the given object type (including subclasses), judging from either bean definitions or the value of getObjectType in the case of FactoryBeans.

实现类DefaultListableBeanFactory

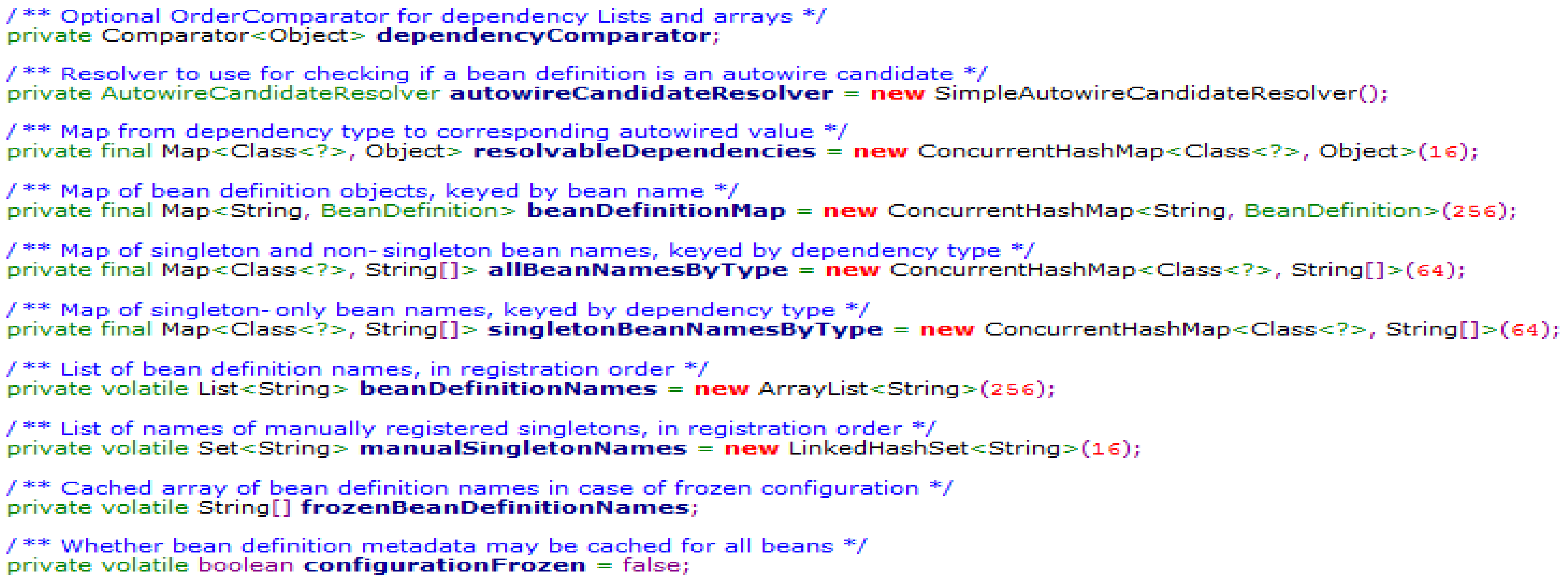


```java
public class DefaultListableBeanFactory extends AbstractAutowireCapableBeanFactory
        implements ConfigurableListableBeanFactory, BeanDefinitionRegistry, Serializable {
```



```java
public interface ConfigurableListableBeanFactory
        extends ListableBeanFactory, AutowireCapableBeanFactory, ConfigurableBeanFactory {
```

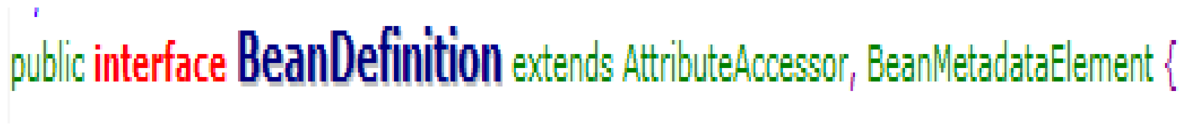
DefaultListableBeanFactory成员信息如下：



```java
/** Optional OrderComparator for dependency Lists and arrays */
private Comparator<Object> dependencyComparator;

/** Resolver to use for checking if a bean definition is an autowire candidate */
private AutowireCandidateResolver autowireCandidateResolver = new SimpleAutowireCandidateResolver();

/** Map from dependency type to corresponding autowired value */
private final Map<Class<?>, Object> resolvableDependencies = new ConcurrentHashMap<Class<?>, Object>(16);

/** Map of bean definition objects, keyed by bean name */
private final Map<String, BeanDefinition> beanDefinitionMap = new ConcurrentHashMap<String, BeanDefinition>(256);

/** Map of singleton and non-singleton bean names, keyed by dependency type */
private final Map<Class<?>, String[]> allBeanNamesByType = new ConcurrentHashMap<Class<?>, String[]>(64);

/** Map of singleton-only bean names, keyed by dependency type */
private final Map<Class<?>, String[]> singletonBeanNamesByType = new ConcurrentHashMap<Class<?>, String[]>(64);

/** List of bean definition names, in registration order */
private volatile List<String> beanDefinitionNames = new ArrayList<String>(256);

/** List of names of manually registered singletons, in registration order */
private volatile Set<String> manualSingletonNames = new LinkedHashSet<String>(16);

/** Cached array of bean definition names in case of frozen configuration */
private volatile String[] frozenBeanDefinitionNames;

/** Whether bean definition metadata may be cached for all beans */
private volatile boolean configurationFrozen = false;
```

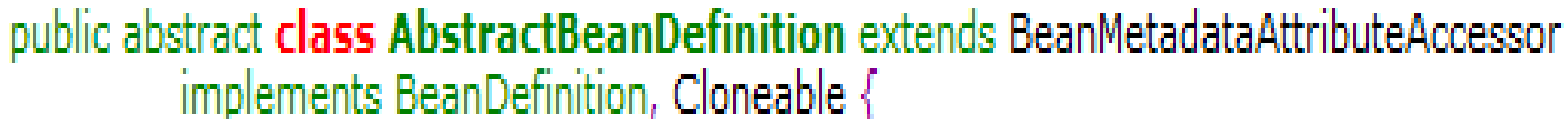
Bean与BeanFactory

- 使用BeanDefinition接口描述Bean:



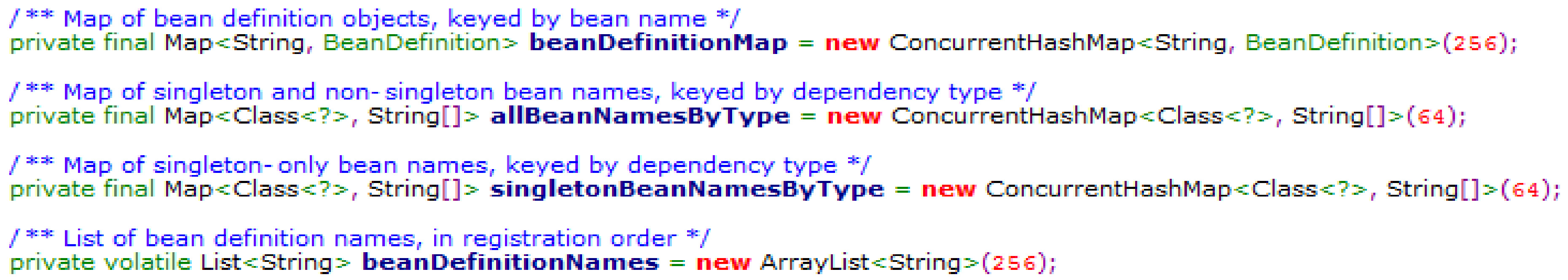
```java
public interface BeanDefinition extends AttributeAccessor, BeanMetadataElement {
```

- AbstractBeanDefinition是BeanDefinition的唯一实现类：



```java
public abstract class AbstractBeanDefinition extends BeanMetadataAttributeAccessor
        implements BeanDefinition, Cloneable {
```

- BeanFactory通过如下成员管理Bean:



```java
/** Map of bean definition objects, keyed by bean name */
private final Map<String, BeanDefinition> beanDefinitionMap = new ConcurrentHashMap<String, BeanDefinition>(256);

/** Map of singleton and non-singleton bean names, keyed by dependency type */
private final Map<Class<?>, String[]> allBeanNamesByType = new ConcurrentHashMap<Class<?>, String[]>(64);

/** Map of singleton-only bean names, keyed by dependency type */
private final Map<Class<?>, String[]> singletonBeanNamesByType = new ConcurrentHashMap<Class<?>, String[]>(64);

/** List of bean definition names, in registration order */
private volatile List<String> beanDefinitionNames = new ArrayList<String>(256);
```

IOC容器与BeanFactory

- ApplicationContext与BeanFactory的功能对比如下：

Feature	BeanFactory	ApplicationContext
Bean instantiation/wiring	Yes	Yes
Integrated lifecycle management	No	Yes
Automatic BeanPostProcessor registration	No	Yes
Automatic BeanFactoryPostProcessor registration	No	Yes
Convenient MessageSource access (for internalization)	No	Yes
Built-in ApplicationEvent publication mechanism	No	Yes

ICS&S 中软国际 ETC 中软卓越