1. spring框架

# 初识Spring框架

## spring简介

1、Spring是一个开源的轻量级的应用开发框架，其目的是用于**简化**企业级应用程序**开发**，降低开发者的开发难度；

(简化开发：spring对常用的api，比如对JDBC做了封装，使用spring封装的jdbc访问数据库，就不用考虑获取连接、关闭连接等，极大的简化了代码)

2、Spring提供的IoC和AOP应用，可以将组件的耦合度降至最低（即**解耦）**，便于系统日后的维护和升级；

3、Spring为系统提供了一个整体的解决方案，开发者可以利用它本身提供的功能外，也可以**与第三方**框架和**技术整合**应用，可以自由选择采用哪种技术进行开发。（比如Spring整合SpringMVC、Spring整合MyBatis、Spring整合Struts2、Spring整合Hibernate、Spring整合Quartz[定时任务处理]）

总结：为什么要使用Spring？

1).方便解耦，简化开发

通过Spring提供的IoC容器，可以将对象之间的依赖关系交由Spring进行控制，避免硬编码所造成的过度程序耦合。

2).AOP编程的支持

通过Spring提供的AOP功能，方便进行面向切面的编程，如性能监测、事务管理、日志记录等。

3).声明式事务的支持

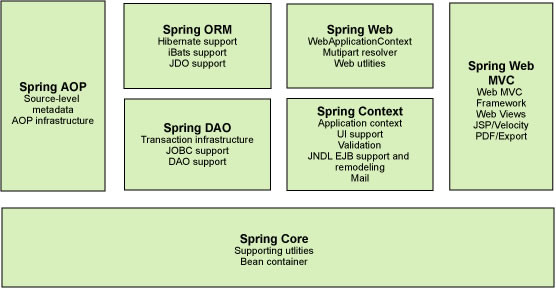
4).方便集成各种优秀框架

5).降低Java EE API的使用难度，如对JDBC，JavaMail，远程调用等提供了简便封装

Spring的本质是管理软件中的对象，即**创建对象**和**维护对象之间的关系**

## spring架构

Spring 最初的目标就是要整合一切优秀资源,然后对外提供一个统一的服务。Spring 模块构建在核心容器之上，核心容器定义了创建、配置和管理 bean 的方式，如下图所示：



组成 Spring 框架的每个模块（或组件）都可以单独存在，或者与其他一个或多个模块联合实现。每个模块的功能如下：

|  |  |  |
| --- | --- | --- |
|  | 模块 | 说明 |
|  | 核心容器Spring Core | 核心容器，提供Spring框架的基本功能。核心容器的主要组件是BeanFactory，它是工厂模式的实现。BeanFactory 使用控制反转（IOC）模式，将应用程序的配置和依赖性规范与实际的应用程序代码分开。 |
|  | Spring Context | Spring上下文，是一个配置文件，向 Spring 框架提供上下文信息。Spring 上下文包括企业服务，例如 JNDI、EJB、电子邮件、国际化、校验和调度功能。 |
|  | Spring AOP | 通过配置管理特性，Spring AOP 模块直接将面向切面的编程功能集成到了 Spring 框架中。可以很容易地使 Spring框架管理的任何对象支持AOP。Spring AOP模块为基于 Spring 的应用程序中的对象提供了事务管理服务。通过使用 Spring AOP，就可以将声明性事务管理集成到应用程序中。 |
|  | Spring DAO | JDBC DAO 抽象层提供了有意义的异常层次结构，可用该结构来管理异常处理和不同数据库供应商抛出的错误消息。异常层次结构简化了错误处理，并且极大地降低了需要编写的异常代码数量（例如打开和关闭连接）。Spring DAO 的面向 JDBC 的异常遵从通用的 DAO 异常层次结构。 |
|  | Spring ORM | Spring 框架插入了若干个 ORM 框架，从而提供了 ORM 的对象关系工具，其中包括JDO、Hibernate和iBatis SQL Map。所有这些都遵从 Spring 的通用事务和 DAO 异常层次结构。 |
|  | Spring Web | Web上下文模块建立在应用程序上下文模块之上，为基于 Web 的应用程序提供了上下文。所以Spring 框架支持与 Jakarta Struts的集成。Web模块还简化了处理多部分请求以及将请求参数绑定到域对象的工作。 |
|  | Spring MVC框架 | MVC 框架是一个全功能的构建 Web 应用程序的 MVC 实现。通过策略接口，MVC 框架变成为高度可配置的，MVC 容纳了大量视图技术，其中包括 JSP、Velocity、Tiles、iText 和 POI。 |

Spring 框架的功能可以用在任何J2EE服务器中，大多数功能也适用于不受管理的环境。Spring 的核心要点是：支持不绑定到特定J2EE服务的可重用业务和数据访问对象。毫无疑问，这样的对象可以在不同J2EE环境（Web或EJB）、独立应用程序、测试环境之间重用。

**\*\*核心组件：**

|  |  |  |
| --- | --- | --- |
|  | 模块 | 说明 |
|  | **BeanFactory** | Spring内部使用，创建bean的工厂 |
|  | **ApplicationContext** | 外部应用程序调用，也成为spring容器的上下文 |
|  | **IoC控制反转**  Inversion of Control | 开发者在无需自己new对象，无需关心对象的创建过程  User user = new User(); 手动创建对象  User user = context.getBean(user); 容器创建对象 |
|  | **DI依赖注入**  Dependency Injection | 松耦合方式实现对象直接的依赖 |
|  | AOP面向切面编程 | 补充java面向对象的不足 |

## 主要jar组成

|  |  |  |
| --- | --- | --- |
|  | 模块 | 说明 |
|  | org.springframework.core | 核心工具包，其他包依赖此包 |
|  | org.springframework.beans | 核心，包括：配置文件，创建和管理bean等 |
|  | org.springframework.aop | 面向切面编程，提供AOP的实现 |
|  | org.springframework.context | 提供IoC功能上的扩展服务，此外还提供许多企业级服务的支持，邮件、任务调度、JNDI定位、EJB集成、远程访问、缓存以及多种视图层框架的支持 |
|  | org.springframework.web.mvc | 包含SpringMVC应用开发时所需的核心类 |
|  | org.springframework.transaction | 为JDBC、Hibernate、JDO、JPA提供一致的声明式和编程式事务管理 |
|  | org.springframework.web | 包含Web应用开发时所需支持类 |
|  | org.springframework.aspects | 提供对AspectJ框架的支持 |
|  | org.springframework.test | 对junit等测试框架的简单封装 |
|  | org.springframework.asm | 3.0后提供自己独立的，反编译 |
|  | org.springframework.context.support | Context的扩展支持，用于mvc方面 |
|  | org.springframework.expression | Spring表达式语言 |
|  | org.springframework.instument | 对服务器的代理接口 |
|  | org.springframework.jdbc | 对jdbc的简单封装 |
|  | org.springframework.jms | 为简化jms api的使用而做的简单封装 |
|  | org.springframework.orm | 整合第三方orm，如hibernate/mybatis |
|  | org.springframework.web.servlet | 增强servlet |

# Spring IOC控制反转

## 什么是控制反转

**IOC**（Inversion of Control），**控制反转**。

所谓的控制反转，就是指将对象的创建，对象的存储(map)，对象的管理(依赖查找,依赖注入)交给了spring容器。

（spring容器是spring中的一个核心模块，用于管理对象）

在此之前，当需要对象时，通常是利用new关键字创建一个对象：

//1.创建一个Hello对象

Hello hello = **new** Hello();

//2.调用hello对象的方法

hello.sayHi();

但由于new对象，会提高代码之间耦合性.

所谓的耦合指的是在软件开发中，在层与层之间产生了某种紧密的关系。这种关系可能会导致，在我们修改或者是替换某一层时会影响其他的层，像这种情况我们就称之为层与层之间产生了耦合。

由于耦合（层与层之间的紧密关系）可能会导致我们在修改某一层时影响到其他的层，而这严重违反了我们对软件进行分层的最初设想 -- 软件各层之间应该相互独立、互不干扰，在修改或者是替换某一层时，应该做到不影响其他层。

而使用spring框架，对象的创建可以交给spring来做：

//1.从spring容器中获取bean对象(而不是自己new)

Hello hello = (Hello) ac.getBean("hello");

//2.调用hello对象的方法

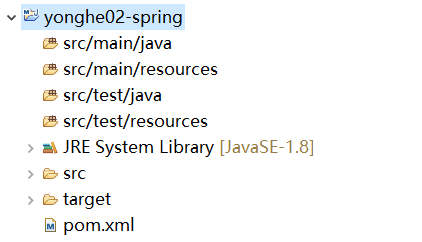
hello.sayHi();

只需要将类提前配置在spring配置文件中，就可以将对象的创建交给spring容器，当需要对象时，不需要自己创建，而是直接通过spring获取即可，省去了new对象，可以降低代码之间的耦合性。

## IOC入门案例

### 创建Maven工程，引入spring相关依赖包

1、创建Maven—Java工程



2、引入spring的jar包：在maven工程的pom.xml文件中添加如下配置：

<!-- 集中定义依赖版本号 -->

<properties>

<junit.version>4.10</junit.version>

<spring.version>4.1.3.RELEASE</spring.version>

</properties>

<dependencies>

<!-- 单元测试 -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>${junit.version}</version>

<scope>test</scope>

</dependency>

<!-- Spring -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>${spring.version}</version>

</dependency>

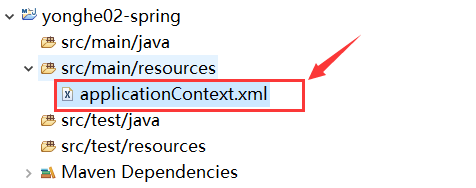
</dependencies>

导入后保存pom文件，项目如图所示：



### 创建spring核心配置文件—applicationContext.xml

1、在工程的java/resources目录下，创建applicationContext.xml文件:



2、在applicationContext.xml中添加如下内容（下面的配置不需要掌握，只是一个文件格式）：

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

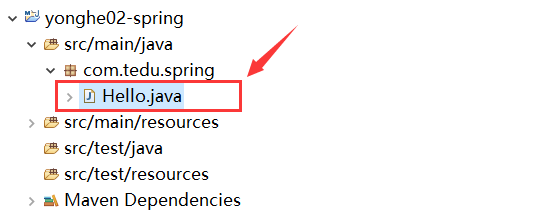
xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-4.0.xsd"*>

</beans>

### 创建实体类­—Hello，并将Hello对象的创建交给Spring管理

1、创建Hello实体类



2、在Hello类中添加方法如下：

**package** com.tedu.spring;

**public** **class** Hello {

**public** **void** sayHi(){

System.***out***.println("Hello.sayHi()...");

}

}

3、将Hello对象的创建交给Spring容器来管理，在核心配置文件中添加如下配置：

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-4.0.xsd"*>

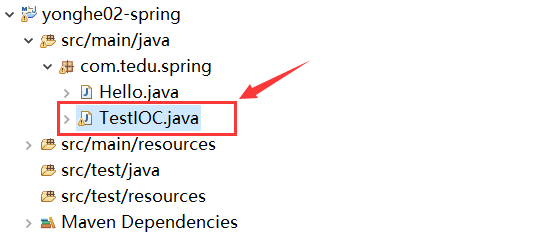
<!-- 声明bean对象，id是唯一标志，class：bean对象的全路径 -->

<bean id=*"hello"* class=*"com.tedu.spring.Hello"*></bean>

</beans>

### 创建测试类—TestIOC，创建Hello对象，并调用其中的方法。

1、创建TestIOC测试类



2、测试步骤及代码如下：

**package** com.tedu.spring;

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

**public** **class** TestIOC {

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

/\* 通过spring容器创建创建Hello对象,

\* 并调用Hello中的sayHi方法 \*/

//1.加载spring的核心配置文件

ClassPathXmlApplicationContext ac =

**new** ClassPathXmlApplicationContext(

"applicationContext.xml"

);

//2.从spring容器中获取bean对象(而不是自己new)

Hello hello = (Hello) ac.getBean("hello");

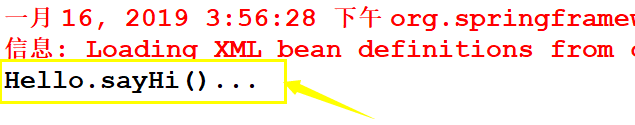
//3.调用hello对象的方法

hello.sayHi();

}

}

3、运行结果：



## IOC小结

这就是spring框架的IOC——控制反转。之前我们自己new对象，例如：

User u = new User();

而现在，变成由一个初始化的xml配置文件来创建，也就是由spring容器来创建。

Hello hello = (Hello) ac.getBean("hello");

当程序运行，spring开始工作后，会加载整个xml核心配置文件，读取到<bean>，获取到class属性中类的全路径，利用反射创建该类的对象。

# Bean对象的单例和多例

## Bean对象的单例和多例概述

在Spring容器中管理的Bean对象的作用域，可以通过**scope属性**或用相关注解指定其作用域。

最常用是singleton（单例）或prototype（多例）。其含义如下：

**1) singleton**：单实例，是默认值。这个作用域标识的对象具备全局唯一性。

当把一个 bean 定义设置scope为singleton作用域时，那么Spring IOC容器只会创建该bean定义的唯一实例。也就是说，**整个Spring IOC容器中只会创建当前类的唯一一个对象**。

这个单一实例会被存储到单例缓存（singleton cache）中，并且所有针对该bean的后续请求和引用都 将返回被缓存的、唯一的这个对象实例。

singleton负责对象的创建、初始化、销毁。

**2) prototype**：多实例。这个作用域标识的对象每次获取都会创建新的对象。

当把一个 bean 定义设置scope为singleton作用域时，**Spring IOC容器会在每一次获取当前Bean时，都会产生一个新的Bean实例**（相当于new的操作）

prototype只负责对象的创建和初始化，不负责销毁。

## 为什么用单实例或多实例？

之所以用单实例，在没有线程安全问题的前提下，没必要每个请求都创建一个对象，这样子既浪费CPU又浪费内存；

之所以用多例，是为了防止并发问题；即一个请求改变了对象的状态(例如，可改变的成员变量)，此时对象又处理另一个请求，而之前请求对对象状态的改变导致了对象对另一个请求做了错误的处理；

用单例和多例的标准只有一个：当对象含有可改变的状态时（更精确的说就是在实际应用中该状态会改变），使用多实例，否则单实例；

在Spring中配置Bean实例是单例还是多例方法是：

**单例：**

<bean id="hello" scope="singleton" class="com.tedu.spring.Hello"></bean>

**多例：**

<bean id="hello" scope="prototype" class="com.tedu.spring.Hello"></bean>

## 测试spring的单实例和多实例

1、创建TestIOC2类，测试代码如下：

**package** com.tedu.spring;

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

**public** **class** TestIOC2 {

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

/\* 通过spring容器创建创建Hello对象,

\* 并调用Hello中的sayHi方法 \*/

//1.加载spring的核心配置文件

ClassPathXmlApplicationContext ac =

**new** ClassPathXmlApplicationContext(

"applicationContext.xml"

);

//2.从spring容器中获取bean对象(而不是自己new)

Hello hello1 = (Hello) ac.getBean("hello");

Hello hello2 = (Hello) ac.getBean("hello");

//3.测试

**if**(hello1 == hello2){

System.***out***.println("当前实例为单实例...");

}**else**{

System.***out***.println("当前实例为多实例...");

}

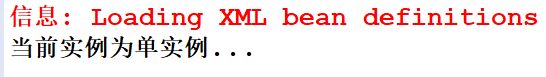
}

}

3、将applicationContext.xml中，Hello类bean标签的scope值设置为singleton：

<bean id="hello" scope="singleton" class="com.tedu.spring.Hello"></bean>

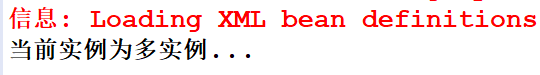
运行TestIOC2，运行结果为：



4、将applicationContext.xml中，Hello类bean标签的scope值修改为prototype

<bean id="hello" scope="prototype" class="com.tedu.spring.Hello"></bean>

再次运行TestIOC2，运行结果为：



# Spring DI依赖注入

## 两种注入方式介绍

DI（Dependency Injection）依赖注入 。

依赖注入，即组件之间的依赖关系由容器在应用系统运行期来决定，也就是由容器动态地将某种依赖关系的目标对象实例注入到应用系统中的各个关联的组件之中。

--------------------------------------------------------------------

简单来说，所谓的依赖注入其实就是，在创建对象的同时或之后，如何给对象的属性赋值。

如果对象由我们自己创建，这一切都变得很简单，例如：

User user = new User();

user.setName("韩少云");

user.setAge(18);

或者：

User user = new User("韩少云", 18);

如果对象由spring创建，那么spring是怎么给属性赋值的？spring提供两种方式为属性赋值：

(1).Set方式注入

(2).构造方法注入（spring大都使用这种方式）

## set方式注入

### 普通属性注入

**需求：通过Spring创建User实例，并为User实例的name和age属性（普通属性）赋值**

1、创建User类，声明name和age属性，并添加对应的setter和getter方法，以及toString方法

**package** com.tedu.spring;

**public** **class** User {

**private** String name;

**private** Integer age;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** Integer getAge() {

**return** age;

}

**public** **void** setAge(Integer age) {

**this**.age = age;

}

@Override

**public** String toString() {

**return** "User [name=" + name + ", age=" + age + "]";

}

}

2、在applicationContext.xml中声明User类的bean实例

<!-- 声明User类的bean实例 -->

<bean id=*"user"* class=*"com.tedu.spring.User"*></bean>

3、创建测试类—TestDI

**package** com.tedu.spring;

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

**public** **class** TestDI {

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

//1.加载applicationContext.xml核心配置文件

ClassPathXmlApplicationContext ac =

**new** ClassPathXmlApplicationContext(

"applicationContext.xml"

);

//2.获取User类的实例

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

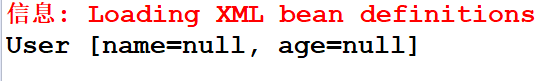
//3.输出User实例

System.***out***.println( user );

}

}

由于这里没有为User对象的属性赋值，所以此时运行测试，结果为：



4、修改applicationContext.xml中User实例的声明，为User实例注入属性

<!-- 声明User类的bean实例 -->

<bean id=*"user"* class=*"com.tedu.spring.User"*>

<!-- 通过set方式为普通属性赋值 -->

<property name=*"name"* value=*"韩少云"*></property>

<property name=*"age"* value=*"20"*></property>

</bean>

其中name属性的值，必须要和User类中所注入属性对应的get方法的名字去掉get后首字母变为小写的名字相同。

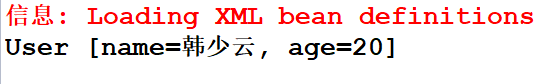
例如：为 User类中的age属性赋值，由于name属性对应的get方法名字为 getAge，当去调用get和 首字母变为小写后的名称为age，因此为age属性注入的配置内容为：

<property name=*"****age****"* value=*"20"*></property>

普通属性直接通过value注入即可。

5、运行测试类TestDI，结果为：

上面通过spring提供的set方式对User对象的属性进行了赋值赋值，所以此时运行测试，结果为：



### 对象属性注入

**需求：通过Spring创建User实例，并为User对象的userInfo属性(对象属性)赋值**

1、创建UserInfo类

**package** com.tedu.spring;

**public** **class** UserInfo {

}

2、在applicationContext.xml中，声明UserInfo类的bean实例

<!-- 声明UserInfo类的bean实例 -->

<bean id=*"userInfo"* class=*"com.tedu.spring.UserInfo"*></bean>

3、修改User类，声明userInfo属性，添加对应的setter和getter方法，并重新生成toString方法

**public** **class** User {

...

**private** UserInfo userInfo;

**public** UserInfo getUserInfo() {

**return** userInfo;

}

**public** **void** setUserInfo(UserInfo userInfo) {

**this**.userInfo = userInfo;

}

...

**public** String toString() {

**return** "User [name=" + name + ", age=" + age + ", userInfo=" + userInfo + "]";

}

}

4、在applicationContext.xml中，将UserInfo对象作为值，赋值给User对象的userInfo属性

<!-- 声明User类的bean实例 -->

<bean id=*"user"* class=*"com.tedu.spring.User"*>

<!-- 通过set方式为普通属性赋值 -->

<property name=*"name"* value=*"韩少云"*></property>

<property name=*"age"* value=*"20"*></property>

<!-- 通过set方式为对象属性赋值 -->

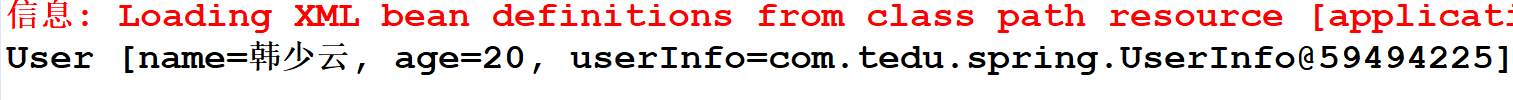
<property name=*"userInfo"* ref=*"userInfo"*></property>

</bean>

由于此处是将UserInfo对象作为值赋值给另一个对象的属性，因此ref属性的值，为UserInfo对象bean标签的id值。

对象属性通过ref属性注入。

5、运行测试类TestDI，结果为：



## 构造方法注入

**需求：通过Spring创建User对象，并为User对象的属性（name、age、UserInfo属性）赋值**

1、为User类声明构造函数

//声明无参构造函数

**public** User() {

}

//声明有参构造函数

**public** User(String name, Integer age, UserInfo userInfo) {

**super**();

**this**.name = name;

**this**.age = age;

**this**.userInfo = userInfo;

}

2、修改applicationContext.xml文件，将set方式修改为构造方法注入。

<bean id="user" class="com.tedu.spring.User">

<!-- 通过set方式为普通属性赋值

<property name="name" value="韩少云"></property>

<property name="age" value="20"></property>

<property name="userInfo" ref="userInfo"></property>

-->

<!-- 通过构造器中参数为属性赋值 -->

<constructor-arg name="name" value="马云"></constructor-arg>

<constructor-arg name="age" value="35"></constructor-arg>

<constructor-arg name="userInfo" ref="userInfo"></constructor-arg>

</bean>

<!-- 声明UserInfo类的bean实例 -->

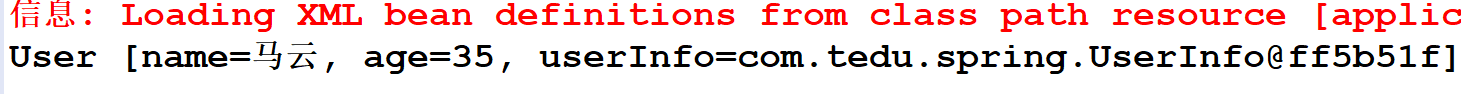
<bean id="userInfo" class="com.tedu.spring.UserInfo"></bean>

其中，constructor-arg标签name属性的值必须和构造函数中参数的名字相同!

同样的，普通属性直接通过value注入即可；

对象属性通过ref属性注入。

2、运行测试类TestDI，结果为：



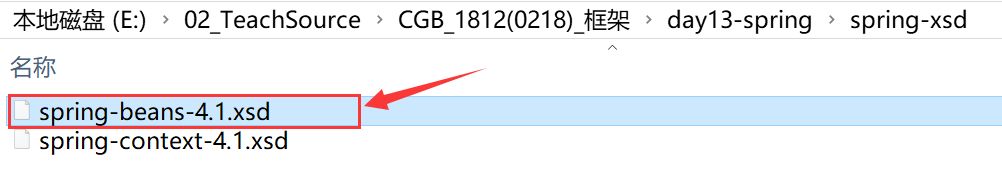
# 扩展内容

## applicationContext.xml没有提示的解决办法

### applicationContext.xml没有提示的解决办法

1、配置spring-beans-4.1.xsd文件

(1)找到spring-beans-4.1.xsd的文件的位置，例如：

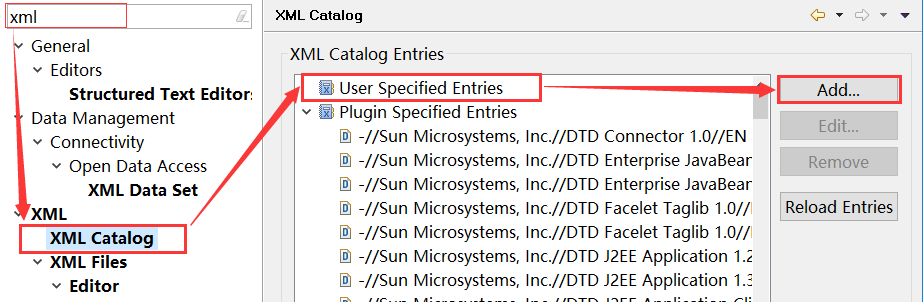


(2)复制下面的url地址：

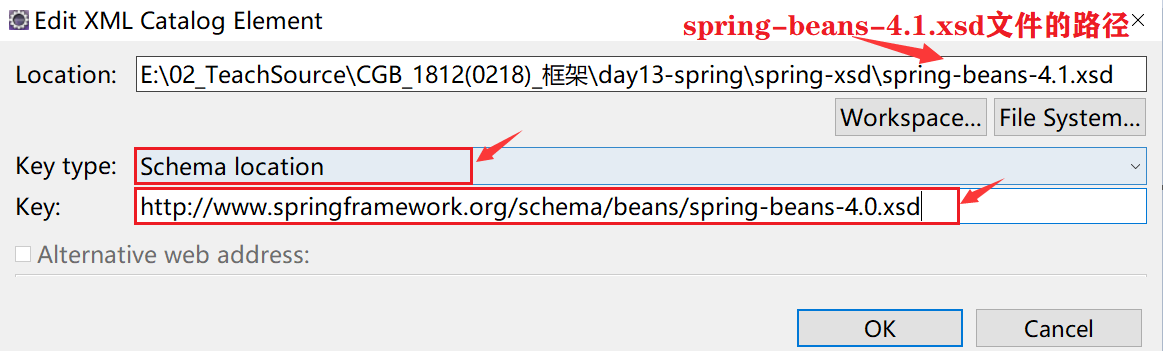
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd

(3)在eclipse菜单栏中: window --> Preferences --> 在搜索框中搜索 [ xml ]

XML --> XML Catalog --> User Specified Entries --> Add…

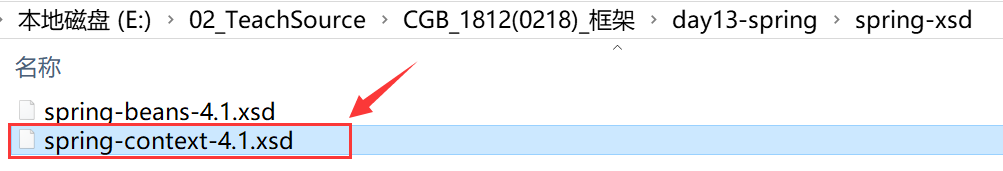


(4)在弹出的窗口中:



2、配置spring-context-4.0.xsd文件

(1)找到spring-context-4.0.xsd的文件的位置，例如：

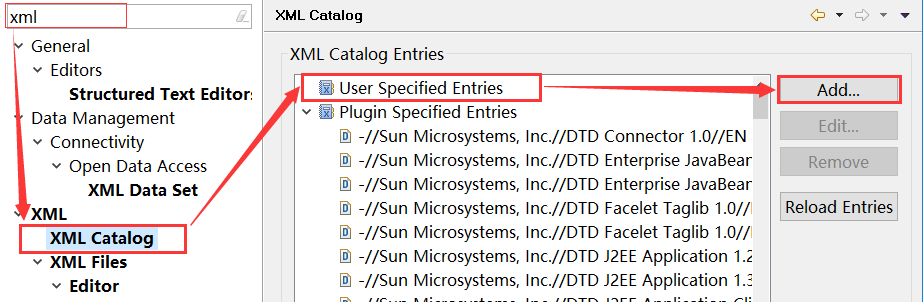


(2)复制下面的url地址：

http://www.springframework.org/schema/context/spring-context-4.0.xsd

(3)在eclipse菜单栏中: window --> Preferences --> 在搜索框中搜索 [ xml ]

XML --> XML Catalog --> User Specified Entries --> Add…



(4)在弹出的窗口中:

