框架概述

2018年10月12日 9:11

1. 框架概述

所谓的框架其实就是程序的架子，在这个程序的架子中，搭建起程序的基本的骨架，针对程序的通用问题给出了便捷的解决方案，可以使开发人员 基于框架快速开发具体的应用程序。

2. 常见的框架

SSH

- Struts2
- Spring
- Hibernate

SSM

- SpringMVC
- Spring
- MyBatis

Spring框架概述

2018年10月12日　　9:39

1. Spring框架概述

Spring是一个Service层的框架，可以整合许多其它框架进行工作。
Spring的主要技术是 IOC(DI) AOP
　　IOC(DI) - 控制反转(依赖注入)
　　AOP - 面向切面编程

2. 为MyEclipse配置Spring的约束

Spring本身是基于xml配置来工作的，在使用Spring的过程中不可避免的要编写大量xml配置，Spring官方提供了这些xml文件的编写规范，这是通过提供xml的约束文件来实现的。

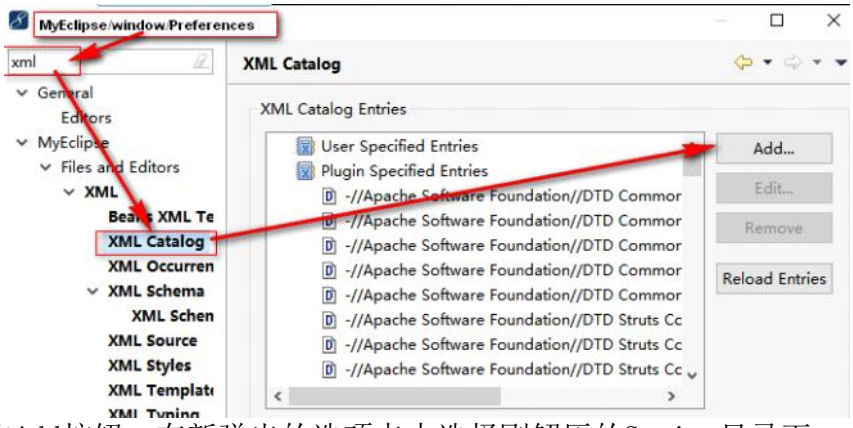
所谓的xml的约束其实是一种限定xml文件写法的技术，主要分为两种：
　　DTD，通常文件的后缀.dtd
　　Schema,通常文件的后缀为.xsd

Spring提供了Schema格式的约束，来限定Spring配置文件的写法。

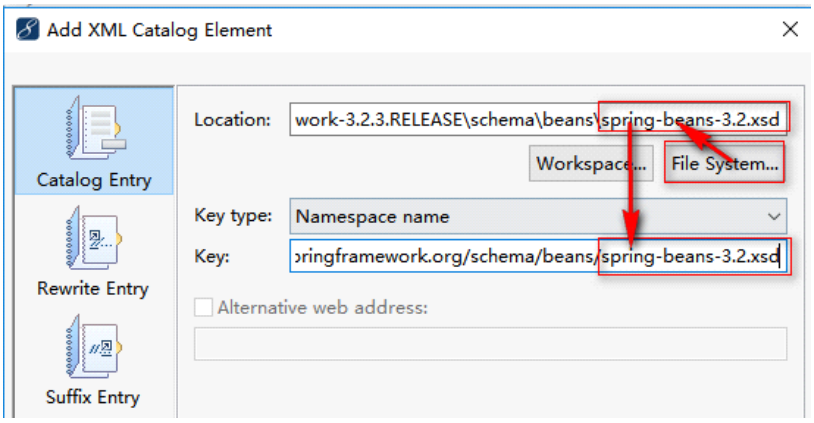
开发人员可以通过阅读Spring提供的约束文件来了解Sprinig的xml配置的写法。
但是这个过程比较麻烦，通常我们会将约束文件交给开发工具管理，开发工具可以通过解析约束文件了解xml的写法，并在需要时为开发者提供标签提示。

a. 将Spring的约束文件交给MyEclipse管理：

1) 将spring的压缩包解压一份放置到到一个固定目录中，要注意的是路径中不可以有中文或空格。
2) 打开MyEclipse的/window/preferences,配置其中的XML Catalog

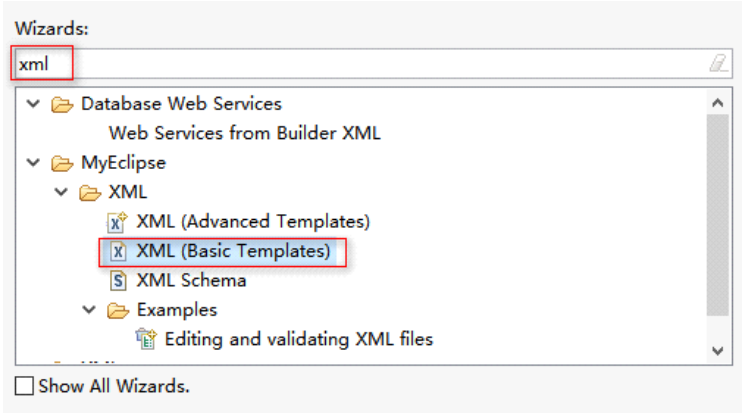


3) 点击Add按钮，在新弹出的选项卡中选择刚解压的Spring目录下schema目录中想要导入的.xsd文件，并且设定好名称空间，通常就是在自动识别的名称空间之后加上/文件名，点击确定就可以使MyEclipse管理该约束文件了

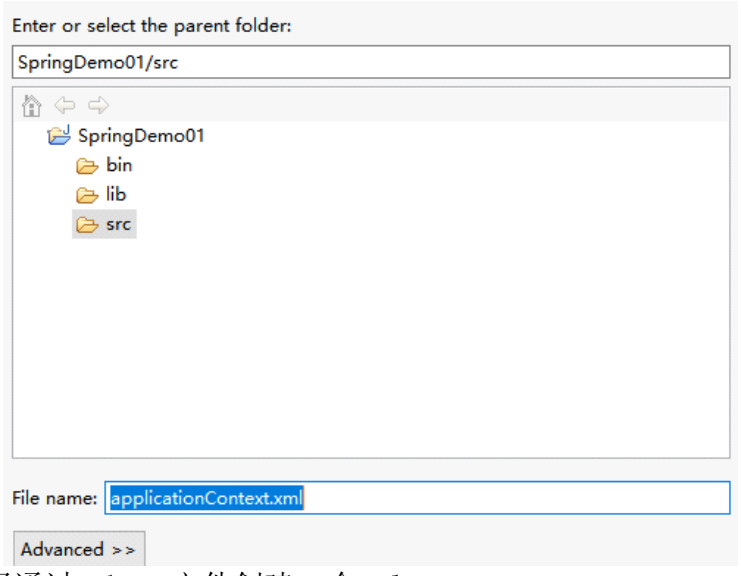


b. 通过xml约束文件自动生成符合约束格式的xml：

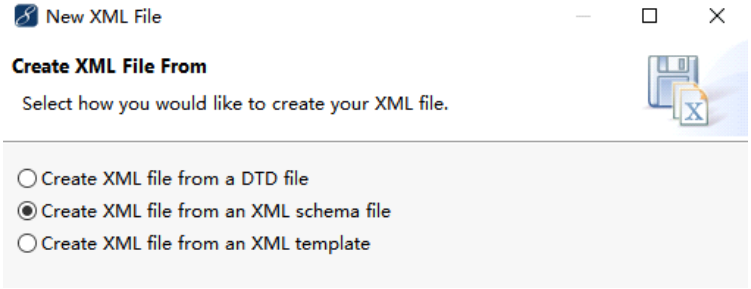
1) 新建xml文件，选择BasicTemplates方式



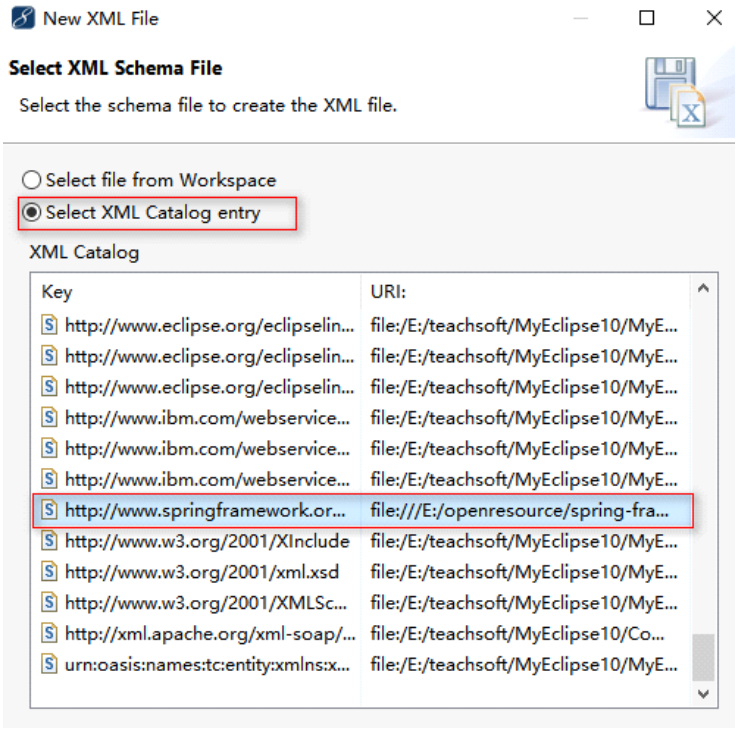
2) 为当前xml起一个名字



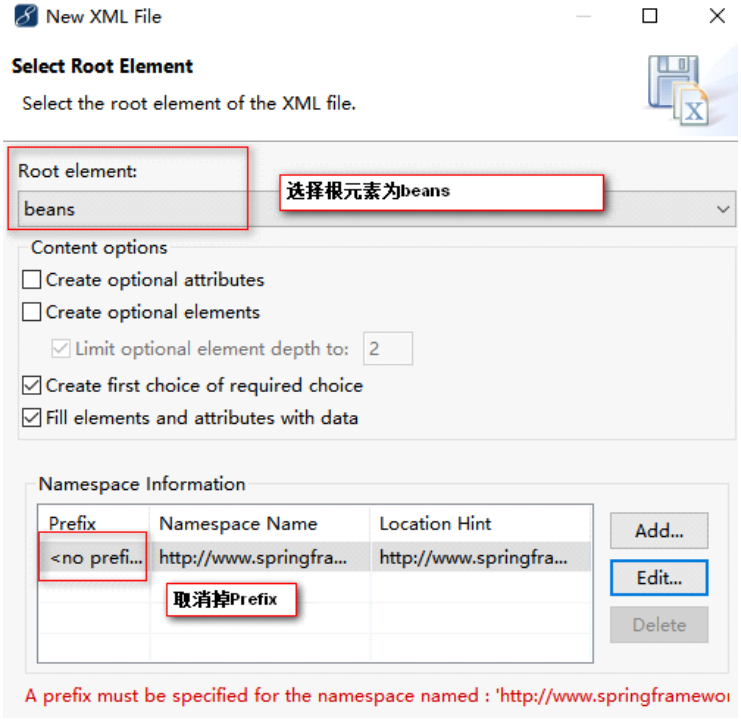
3) 选择通过schema文件创建一个xml



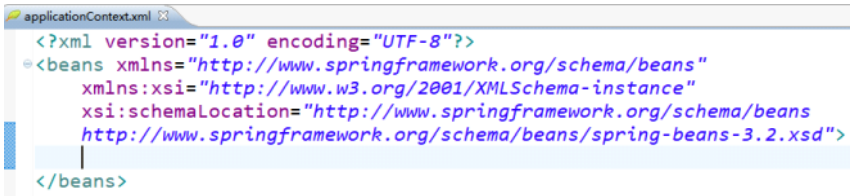
4) 选择之前导入的 schema文件



5) 指定根标签，及标签前缀，通常可以将前缀置为空，方便后续使用

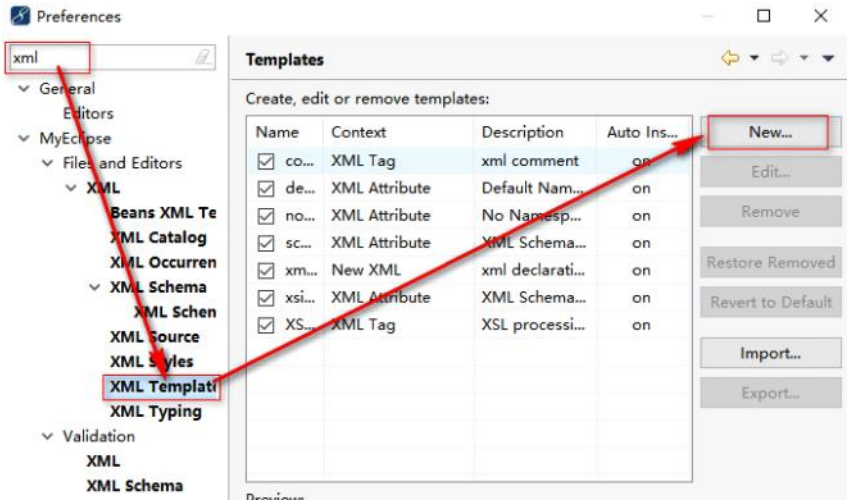


6) 确定之后即可产生符合格式的xml



c. 配置xml模版，用来快速生成xml:

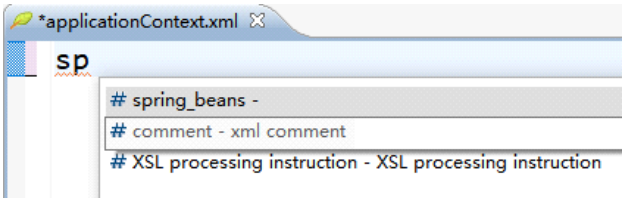
1) 在MyEclipse的window/preferences中配置新模版



2) 设定模版

3) 使用模版

在xml文件中输入模版名称，可以通过自动提示插入模版内容



Spring IOC

2018年10月12日 10:56

1. IOC(DI) - 控制反转(依赖注入)

所谓的IOC称之为控制反转，简单来说就是将对象的创建的权利及对象的生命周期的管理过程交由Spring框架来处理，从此在开发过程中不再需要关注对象的创建和生命周期的管理，而是在需要时由Spring框架提供，这个由spring框架管理对象创建和生命周期的机制称之为控制反转。而在 创建对象的过程中Spring可以依据配置对对象的属性进行设置，这个过称之为依赖注入,也即DI。

2. IOC的入门案例

a. 下载Spring

访问Spring官网,下载Spring相关的包

b. 解压下载好的压缩包

其中包含着Spring的依赖包

c. 创建一个java项目

spring并不是非要在javaweb环境下才可以使用，一个普通的java程序中也可以使用Spring。

d. 导入Spring的libs目录下IOC相关的jar包

e. 创建Spring的配置文件

Spring采用xml文件作为配置文件，xml文件名字任意，但通常都取名为applicationContext.xml，通常将该文件放置在类加载的目录里下(src目录)，方便后续使用。

f. 创建bean类，并在spring中进行配置交由spring来管理

```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-3.2.xsd">

        <bean id="person" class="cn.tedu.beans.Person"></bean>

</beans>
```

g. 在程序中通过Spring容器获取对象并使用

```java
public class Person_Test {
        @Test
        /**
         * SpringIOC的入门案例
         */
        public void  test1(){
                ApplicationContext context =
                                new ClassPathXmlApplicationContext("applicationContext.xml");
                Person p = (Person) context.getBean("person");
                p.say();
        }
}
```

3. IOC的实现原理

在初始化一个Spring容器时，Spring会去解析指定的xml文件，当解析到其中的<bean>标签时，

会根据该标签中的class属性指定的类的全路径名，通过反射创建该类的对象，并将该对象存入内置的Map中管理。其中键就是该标签的id值，值就是该对象。
之后，当通过getBean方法来从容器中获取对象时，其实就是根据传入的条件在内置的Map中寻找是否有匹配的键值，如果有则将该键值对中保存的对象返回，如果没有匹配到则抛出异常。

由此可以推测而知：

默认情况下，多次获取同一个id的bean，得到的将是同一个对象。

即使 是同一个类，如果配置过多个<bean>标签具有不同的id，每个id都会在内置Map中有一个键值对，其中的值是这个类创建的不同的对象

同一个<beans>标签下不允许配置多个同id的<bean>标签，如果配置则启动抛异常

4. IOC获取对象的方式

通过context.getBeans()方法获取bean时，可以通过如下两种方式获取：

传入id值

传入class类型

通过class方式获取bean时，如果同一个类配置过多个bean，则在获取时因为无法确定到底要获取哪个bean会抛出异常。
而id是唯一的，不存在这样的问题，所以建议大家尽量使用id获取bean。

```
@Test
/**
 * SpringIOC获取bean的方式
 */
public void  test3(){
        /*
         <bean id="person" class="cn.tedu.beans.Person"></bean>
         */
        /*
         <bean id="person" class="cn.tedu.beans.Person"></bean>
         <bean id="personx" class="cn.tedu.beans.Person"></bean>
         */
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
        //--通过id获取
        Person p1 = (Person) context.getBean("person");
        p1.say();
        //--通过class获取
        Person p2 = context.getBean(Person.class);
        p2.say();
}

```

5. 别名标签

在 Spring中提供了别名标签<alias>可以为配置的<bean>起一个别名，要注意的是这仅仅是对指定的<bean>起的一个额外的名字，并不会额外的创建对象存入map。

<alias name="要起别名的bean的id" alias="要指定的别名"/>

```
@Test
/**
 * SpringIOC中bean别名
 */
public void  test4(){
        /*
                <bean id="person" class="cn.tedu.beans.Person"></bean>
                <alias name="person" alias="personx"/>
         */
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
        //--通过id获取
        Person p1 = (Person) context.getBean("personx");
        p1.say();
}

```

6. Spring创建对象的方式

a. 通过类的无法构造方法创建对象

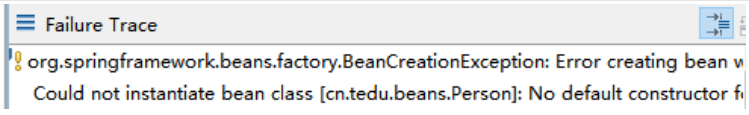
在入门案例中使用的就是这种方式。当用最普通方式配置一个<bean>时,默认就是采用类的无参构造创建对象。在Spring容器初始化时，通过<bean>上配置的class属性反射得到字节

码对象，通过newInstance()创建对象

```
Class c = Class .forName("类的全路径名称")
Object obj = c.newInstance()
```

这种方式下spring创建对象，要求类必须有无参的构造，否则无法通过反射创建对象，会抛出异常。

```
package cn.tedu.beans;


public class Person {
        public Person(String arg) {
                System.out.println("Person的无参构造执行了。。。");
        }
        public void say(){
                System.out.println("person hello spring~");
        }
}
@Test
/**
 * SpringIOC 创建对象方式 1 - 通过无参构造方法创建对象
 */
public void test5(){
        /*
                <bean id="person" class="cn.tedu.beans.Person"></bean>
         */
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
        Person p  = (Person) context.getBean("person");
        p.say();
}
```



b. 通过静态工厂创建对象

很多的时候，我们面对的类是无法通过无参构造去创建的，例如该类没有无参构造、是一抽象类 等等情况 ，此时无法要求spring通过无参构造创建对象，此时可以使用静态工厂 方式创建对象。

```
public class CalendarStaticFactory {
        public static Calendar getCalendar(){
                return Calendar.getInstance();
        }
}
```

```
<?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-3.2.xsd">

        <bean id="calendar"  class="cn.tedu.factory.CalendarStaticFactory" factory-
method="getCalendar"></bean>


</beans>
```

```
@Test
/**
 * SpringIOC 创建对象方式 2 - 静态工厂
 */
public void test6(){
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
        Calendar calendar = (Calendar) context.getBean("calendar");
        System.out.println(calendar);
}
```

c. 实例工厂创建对象

实例工厂也可以解决类是无法通过无参构造创建的问题，解决的思路和静态 工厂类似，只不过实例工厂提供的方法不是静态的。spring需要先创建出实例工厂的对象，在调用实例工厂对象上指定的普通方法来创建对象。所以实例工厂也需要配置到Spring中管理。

```
package cn.tedu.factory;


import java.util.Calendar;

```

```java
public class CalendarFactory {
        public Calendar getCalendar(){
                return Calendar.getInstance();
        }
}
```

```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-3.2.xsd">

        <bean id="calendarFactory"  class="cn.tedu.factory.CalendarFactory"></bean>
        <bean id="calendar" factory-bean="calendarFactory" factory-method="getCalendar"/>


</beans>
```

```java
@Test
/**
 * SpringIOC 创建对象方式 3 - 实例工厂
 */
public void test7(){
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
        Calendar calendar = (Calendar) context.getBean("calendar");
        System.out.println(calendar);
}
```

d. Spring工厂创建对象

Spring内置了工厂接口，也可以通过实现这个接口来开发Spring工厂，通过这个工厂创建对象。

```java
package cn.tedu.factory;


import java.util.Calendar;


import org.springframework.beans.factory.FactoryBean;


public class CalendarSpringFactory implements FactoryBean<Calendar>{


        /**
         * Spring工厂生产对象的方法
         */
        @Override
        public Calendar getObject() throws Exception {
                return Calendar.getInstance();
        }


        /**
         * 获取当前工厂生产的对象的类型的方法
         */
        @Override
        public Class<?> getObjectType() {
                return Calendar.class;
        }


        /**
         * Spring工厂生产对象时是否采用单例模式
         * 如果返回true，则在spring中该对象只创建一次 后续 重复使用
         * 如果返回false，则每次获取该bean 都重新 创建对象
         */
        @Override
        public boolean isSingleton() {
                return true;
        }


}
```

```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-3.2.xsd">

        <bean id="calendar" class="cn.tedu.factory.CalendarSpringFactory"></bean>

</beans>
```

```java
@Test
/**
 * SpringIOC 创建对象方式 3 - spring工厂
 */
public void test8(){
```

```
public void test6(){
        /*
                <bean id="calendar" class="cn.tedu.factory.CalendarSpringFactory"></bean>
        */
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
        Calendar calendar = (Calendar) context.getBean("calendar");
        System.out.println(calendar);
}
```

7. 单例和多例

Spring容器管理的bean在默认情况下是单例的，也即，一个bean只会创建一个对象，存在内置map中，之后无论获取多少次该bean，都返回同一个对象。

Spring默认采用单例方式，减少了对象的创建，从而减少了内存的消耗。
但是在实际开发中是存在多例的需求的，Spring也提供了选项可以将bean设置为多例模式。

```
<?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-3.2.xsd">

        <!--
                scope属性控制当前bean的创建模式：
                        singleton:则当前bean处在单例模式中,默认就是此模式
                        prototype:则当前bean处在多例模式中
         -->
        <bean id="cart" class="cn.tedu.beans.Cart" scope="prototype"></bean>

</beans>

```

bean在单例模式下的生命周期：

bean在单例模式下，spring容器启动时解析xml发现该bean标签后，直接创建该bean的对象存入内部map中保存，此后无论调用多少次getBean()获取该bean都是从map中获取该对象返回，一直是一个对象。此对象一直被Spring容器持有，直到容器退出时，随着容器的退出对象被销毁。

bean在多例模式下的生命周期：

bean在多例模式下，spring容器启动时解析xml发现该bean标签后，只是将该bean进行管理，并不会创建对象，此后每次使用 getBean()获取该bean时，spring都会重新创建该对象返回，每次都是一个新的对象。这个对象spring容器并不会持有，什么销毁取决于使用该对象的用户自己什么时候销毁该对象。

- 实验：通过断点调试模式，观察spring单例和多例的bean执行构造的过程

```
略
```

8. 懒加载机制

Spring默认会在容器初始化的过程中，解析xml，并将单例的bean创建并保存到map中，这样的机制在bean比较少时问题不大，但一旦bean非常多时，spring需要在启动的过程中花费大量的时间来创建bean 花费大量的空间存储bean，但这些bean可能很久都用不上，这种在启动时在时间和空间上的浪费显得非常的不值得。
所以Spring提供了懒加载机制。所谓的懒加载机制就是可以规定指定的bean不在启动时立即创建，而是在后续第一次用到时才创建，从而减轻在启动过程中对时间和内存的消耗。
懒加载机制只对单例bean有作用，对于多例bean设置懒加载没有意义。

懒加载的配置方式：

为指定bean配置懒加载

```
<?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-3.2.xsd"
        >

        <bean id="cart" class="cn.tedu.beans.Cart" lazy-init="true"></bean>

</beans>
```

为全局配置懒加载

```
<?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-3.2.xsd"
        default-lazy-init="true"
        >

        <bean id="cart" class="cn.tedu.beans.Cart"></bean>

</beans>
```

**如果同时设定全局和指定bean的懒加载机制，且配置不相同，则对于该bean局部配置覆盖全局配置。

- 实验：通过断点调试，验证懒加载机制的执行过程

```
package cn.tedu.beans;


public class Cart {
        public Cart() {
                System.out.println("Cart init...");
        }
}
```

```
<?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-3.2.xsd"
        >

        <bean id="cart" class="cn.tedu.beans.Cart" lazy-init="true"></bean>

</beans>
```

```
@Test
/**
 * SpringIOC 懒加载机制
 */
public void test10(){
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
        Cart cart1 = (Cart) context.getBean("cart");
        Cart cart2 = (Cart) context.getBean("cart");
        System.out.println(cart1 == cart2);
}
```

9. 配置初始化和销毁的方法

在Spring中如果某个bean在初始化之后 或 销毁之前要做一些 额外操作可以为该bean配置初始化和销毁的方法 ，在这些方法中完成要功能。

- 实验：通过断点调试模式，测试初始化方法 和 销毁方法的执行

```
package cn.tedu.beans;


public class ProdDao {

        public ProdDao() {
                System.out.println("ProdDao 被创建。。。");
        }

        public void init(){
                System.out.println("init。。连接数据库。。。。。");
        }


        public void destory(){
                System.out.println("destory。。断开数据库。。。。。");
        }

        public void addProd(){
```

```
18          public void  addProd(){
19                  System.out.println("增加商品。。");
20          }
21          public void  updateProd(){
22                  System.out.println("修改商品。。");
23          }
24          public void  delProd(){
25                  System.out.println("删除商品。。");
26          }
27          public void  queryProd(){
28                  System.out.println("查询商品。。");
29          }
30  }
```

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <beans xmlns="http://www.springframework.org/schema/beans"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://www.springframework.org/schema/beans
5           http://www.springframework.org/schema/beans/spring-beans-3.2.xsd"
6           >
7
8           <bean id="prodDao" class="cn.tedu.beans.ProdDao"
9                   init-method="init" destroy-method="d y"></bean>
10
11  </beans>
```

```
1   @Test
2   /**
3    * SpringIOC 初始化和 销毁方法
4    */
5   public void test11(){
6           ClassPathXmlApplicationContext context = new
7   ClassPathXmlApplicationContext("applicationContext.xml");
8           ProdDao prodDao = (ProdDao) context.getBean("prodDao");
9           prodDao.addProd();
10          context.close();
    }
```

**Spring中关键方法的执行顺序：

在Spring创建bean对象时，先创建对象(通过无参构造或工厂)，之后立即调用init方法来执行初始化操作，之后此bean就可以哪来调用其它普通方法,而在对象销毁之前，spring容器调用其destory方法来执行销毁操作。

Spring DI

2018年10月12日 16:26

1. IOC(DI) - 控制反转(依赖注入)

所谓的IOC称之为控制反转，简单来说就是将对象的创建的权利及对象的生命周期的管理过程交由Spring框架来处理，从此在开发过程中不再需要关注对象的创建和生命周期的管理，而是在需要时由Spring框架提供，这个由spring框架管理对象创建和生命周期的机制称之为控制反转。而在 创建对象的过程中Spring可以依据配置对对象的属性进行设置，这个过称之为依赖注入,也即DI。

2. set方法注入

通常的javabean属性都会私有化，而对外暴露setXxx()getXxx()方法，此时spring可以通过这样的setXxx()方法将属性的值注入对象。

a. Spring内置的可直接注入类型的注入：

```java
package cn.tedu.beans;


import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.Set;


public class Hero {
	private int id;
	private String name;
	private List<String> jobs;
	private Set<String> set;
	private Map<String,String> map;
	private Properties prop;

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


	public void setName(String name) {
		this.name = name;
	}


	public void setJobs(List<String> jobs) {
		this.jobs = jobs;
	}


	public void setSet(Set<String> set) {
		this.set = set;
	}


	public void setMap(Map<String, String> map) {
		this.map = map;
	}


	public void setProp(Properties prop) {
		this.prop = prop;
	}


	@Override
	public String toString() {
		return "Hero [id=" + id + ", name=" + name + ", jobs=" + jobs
				+ ", set=" + set + ", map=" + map + ", prop=" + prop + "]";
	}
}
```

```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-3.2.xsd"
	>

	<bean id="hero" class="cn.tedu.beans.Hero">
```

```
                <property name="id" value="123"></property>
                <property name="name" value="亚瑟 "></property>
                <property name="jobs">
                        <list>
                                <value>上单</value>
                                <value>打野</value>
                                <value>辅助</value>
                                <value>中单</value>
                        </list>
                </property>
                <property name="set">
                        <set>
                                <value>aaa</value>
                                <value>bbb</value>
                                <value>ccc</value>
                                <value>aaa</value>
                        </set>
                </property>
                <property name="map">
                        <map>
                                 <entry key="addr" value="王者荣耀"></entry>
                                 <entry key="addr" value="英雄联盟"></entry>
                                 <entry key="skill" value="风火轮"></entry>
                                 <entry key="age" value="19"></entry>
                        </map>
                </property>
                <property name="prop">
                        <props>
                                <prop key="k1">v1</prop>
                                <prop key="k2">v2</prop>
                                <prop key="k3">v3</prop>
                                <prop key="k4">v4</prop>
                        </props>
                </property>
        </bean>


</beans>
```

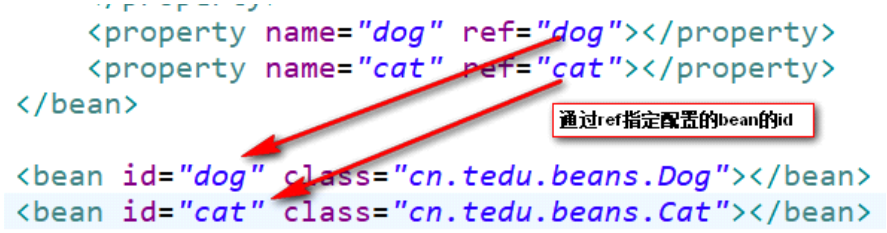
```
@Test
/**
 * SpringDI set方式属性注入 - Spring内置的可直接注入类型的注入
 */
public void test1(){
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
        Hero hero = (Hero) context.getBean("hero");
        System.out.println(hero);
}
```

b. 非Spring内置的可以直接注入类型的注入：

```
package cn.tedu.beans;


import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.Set;


public class Hero {
        private int id;
        private String name;
        private List<String> jobs;
        private Set<String> set;
        private Map<String,String> map;
        private Properties prop;
        private Dog dog;
        private Cat cat;


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


        public void setName(String name) {
                this.name = name;
        }


        public void setJobs(List<String> jobs) {
                this.jobs = jobs;
        }


        public void setSet(Set<String> set) {
                this.set = set;
```

```
                this.set = set;
        }

        public void setMap(Map<String, String> map) {
                this.map = map;
        }

        public void setProp(Properties prop) {
                this.prop = prop;
        }

        public void setDog(Dog dog) {
                this.dog = dog;
        }

        public void setCat(Cat cat) {
                this.cat = cat;
        }

        @Override
        public String toString() {
                return "Hero [id=" + id + ", name=" + name + ", jobs=" + jobs
                                + ", set=" + set + ", map=" + map + ", prop=" + prop + ", dog="
                                + dog + ", cat=" + cat + "]";
        }

}
```

```
<?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-3.2.xsd"
        >

        <bean id="hero" class="cn.tedu.beans.Hero">
                <property name="id" value="123"></property>
                <property name="name" value="亚瑟 "></property>
                <property name="jobs">
                        <list>
                                <value>上单</value>
                                <value>打野</value>
                                <value>辅助</value>
                                <value>中单</value>
                        </list>
                </property>
                <property name="set">
                        <set>
                                <value>aaa</value>
                                <value>bbb</value>
                                <value>ccc</value>
                                <value>aaa</value>
                        </set>
                </property>
                <property name="map">
                        <map>
                                 <entry key="addr" value="王者荣耀"></entry>
                                 <entry key="addr" value="英雄联盟"></entry>
                                 <entry key="skill" value="风火轮"></entry>
                                 <entry key="age" value="19"></entry>
                        </map>
                </property>
                <property name="prop">
                        <props>
                                <prop key="k1">v1</prop>
                                <prop key="k2">v2</prop>
                                <prop key="k3">v3</prop>
                                <prop key="k4">v4</prop>
                        </props>
                </property>
                <property name="dog" ref="dog"></property>
                <property name="cat" ref="cat"></property>
        </bean>

        <bean id="dog" class="cn.tedu.beans.Dog"></bean>
        <bean id="cat" class="cn.tedu.beans.Cat"></bean>

</beans>
```




```
    <property name="dog" ref="dog"></property>
    <property name="cat" ref="cat"></property>
</bean>

<bean id="dog" class="cn.tedu.beans.Dog"></bean>
<bean id="cat" class="cn.tedu.beans.Cat"></bean>
```

```
@Test
/**
 * SpringDI set方式属性注入 - 非Spring内置的可以直接注入类型的注入
 */
public void test2(){
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
        Hero hero = (Hero) context.getBean("hero");
        System.out.println(hero);
}
```

3. 基于构造方法的注入

对象属性设置的另一种方式是在对象创建的过程中通过构造方法传入并设置对象的属性。spring也可以通过这样的构造方法实现属性的注入。

```
package cn.tedu.beans;


public class Student {
        private int id;
        private String name;
        private Dog dog;

        public Student(int id, String name, Dog dog) {
                this.id = id;
                this.name = name;
                this.dog = dog;
        }


        @Override
        public String toString() {
                return "Student [id=" + id + ", name=" + name + ", dog=" + dog + "]";
        }
}
```

```
<?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-3.2.xsd"
        >

        <bean id="student" class="cn.tedu.beans.Student">
                <!--
                        index:为构造方法的第几个参数 进行配置
                        name:为构造方法的哪个名字的参数进行配置
                             **index 和 name 可以配置任何一个或同时配置 但要求一旦配置必须正确
                             **推荐优先使用index方式配置 防止没有源码造成name无法匹配到对应参数
                        type:该构造方法参数的类型
                        value:该构造方法参数的值 ,用来指定基本值
                        ref:该构造方法参数的值,用来指定引用其他bean的值
                 -->
                <constructor-arg index="0" name="id" value="999"/>
                <constructor-arg index="1" type="java.lang.String" value="张无忌"/>
                <constructor-arg name="dog" ref="dog"/>
        </bean>

        <bean id="dog" class="cn.tedu.beans.Dog"></bean>

</beans>
```

```
@Test
/**
 * SpringDI 构造方法方式属性注入
 */
public void test3(){
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
        Student student = (Student) context.getBean("student");
        System.out.println(student);
}
```

4. 自动装配

在Spring的set方式实现的注入过程中，支持自动装配机制，所谓自动装配机制，会根据要设置的javabean属性的名字 或 类型 到spring中自动寻找对应id 或 类型的<bean>进行设置，从而省去依次配置的过程，简化了配置。

为 指定<bean>开启自动装配：

```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-3.2.xsd"
        >

        <!--
                autowire设定自动装配:
                        byName:根据javabean中需要注入的属性的名字 ，在spring容器中找对应id的<bean>将该<bean>的对象复制给 当前的属性
                        byType:根据javabean中需要注入的属性的类型，在spring容器中找对应class类型的<bean>将该<bean>的对象复制给 当前的属性
                        **byType方式 根据类型进行匹配，可能匹配到多个<bean>,此时会抛出异常。而byName是通过id来寻找<bean>，id没有重复，不会有这方面的问题，所以推荐使用byName方式
         -->
        <bean id="teacher" class="cn.tedu.beans.Teacher" autowire="byName"></bean>
        <bean id="dog" class="cn.tedu.beans.Dog"></bean>
        <bean id="cat" class="cn.tedu.beans.Cat"></bean>

</beans>
```

为 全局配置自动装配：

```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-3.2.xsd"
        default-autowire="byName"
        >

        <!--
                autowire设定自动装配:
                        byName:根据javabean中需要注入的属性的名字 ，在spring容器中找对应id的<bean>将该<bean>的对象复制给 当前的属性
                        byType:根据javabean中需要注入的属性的类型，在spring容器中找对应class类型的<bean>将该<bean>的对象复制给 当前的属性
                        **byType方式 根据类型进行匹配，可能匹配到多个<bean>,此时会抛出异常。而byName是通过id来寻找<bean>，id没有重复，不会有这方面的问题，所以推荐使用byName方式
         -->
        <bean id="teacher" class="cn.tedu.beans.Teacher"></bean>
        <bean id="dog" class="cn.tedu.beans.Dog"></bean>
        <bean id="cat" class="cn.tedu.beans.Cat"></bean>

</beans>
```

```java
package cn.tedu.beans;


public class Teacher {
        private Dog dog;
        private Cat cat;
        public void setDog(Dog dog) {
                this.dog = dog;
        }
        public void setCat(Cat cat) {
                this.cat = cat;
        }

        @Override
        public String toString() {
                return "Teacher [dog=" + dog + ", cat=" + cat + "]";
        }
}
```

```java
@Test
/**
 * SpringDI 自动装配
 */
public void test4(){
      ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
      Teacher teacher = (Teacher) context.getBean("teacher");
      System.out.println(teacher);
}
```

Spring注解方式 实现 IOC 和 DI

2018年10月15日 10:45

1. 注解概念

所谓注解就是给程序看的提示信息，很多时候都用来作为轻量级配置的方式。

关于注解的知识点，参看java基础课程中java基础加强部分的内容。

2. Spring中的注解

Spring除了默认的使用xml配置文件的方式实现配置之外，也支持使用注解方式实现配置，这种方式效率更高，配置信息清晰，修改更方便，推荐使用。

引入context名称空间：

在MyEclipse中导入spring-context-3.2.xsd约束文件，要求Spring来管理。

在applicationContext.xml文件中，引入该schema文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"

    xmlns:context="http://www.springframework.org/schema/context"
    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-3.2.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.2.xsd
    ">
</beans>
```

**可以将以上头信息加入MyEclipse模版，方便后续自动生成。

3. 使用类注解

使用Spring的类注解可以通过注解注册类为bean，省去了配置文件中的<bean>配置。

a. 开启包扫描

在spring的配置文件中，开启包扫描，指定spring自动扫描哪些个包下的类。

<context:component-scan base-package="cn.tedu.beans"/>

案例：

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

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

xmlns:context="http://www.springframework.org/schema/context"

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-3.2.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.2.xsd
        ">
        <!-- 开启包扫描 -->
        <context:component-scan base-package="cn.tedu.beans">
        </context:component-scan>
        <!--
            <bean id="person" class="cn.tedu.beans.Person"></bean>
            <bean id="cat" class="cn.tedu.beans.Cat"></bean>
            <bean id="dog" class="cn.tedu.beans.Dog"></bean>
          -->
    </beans>
```

b. 使用注解注册bean

这个包中的类会在spring容器启动时自动被扫描，检测是否需要自动配置为bean.

在配置的包中的类上使用@Component注解，则这个类会自动被注册为bean，使用当前类的class为<bean>的class，通常情况下使用类名首字母小写为<bean>id。

案例：

```
package cn.tedu.beans;
import org.springframework.stereotype.Component;

@Component
public class Person{
}
```

c. bean的id

可以使bean类实现BeanNameAware接口，并实现其中的setBeanName方法，spring容器会在初始化bean时，调用此方法告知当前bean的id。通过这个方式可以获取bean的id信息。

通常情况下注解注册bean使用类名首字母小写为bean的id，但是如果类名的第二个字母为大写则首字母保留原样.

```
cn.tedu.beans.Person --> <bean id="person" class="cn.tedu.beans.Person"/>
cn.tedu.beans.NBA --> <bean id="NBA" class="cn.tedu.beans.NBA"/>
```

也可以通过在@Component中配置value属性，明确的指定当前类在注册到spring时bean的id

案例：

```
package cn.tedu.beans;

import org.springframework.beans.factory.BeanNameAware;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

@Component("per")
public class Person implements BeanNameAware{
    @Override
    public void setBeanName(String name) {
        System.out.println("==="+this.getClass().getName()+"==="+name);
    }
}
```

4. 使用属性注解

使用属性注解，可以为bean配置属性的注入过程，省去了在配置文件中进行注入配置的过程，更加便捷。

a. 在配置文件中开启属性注解功能

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    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-3.2.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.2.xsd
    ">
    <!-- 开启属性注解 -->
    <context:annotation-config></context:annotation-config>
</beans>
```

b. 使用属性注解注入bean类型数据：

在bean中的属性上通过如下注解声明属性注入

@Autowired

也可以使用@Qualifier(value="dog1")注解,明确的指定,要注入哪个id的bean

代码：

```
package cn.tedu.beans;

import org.springframework.beans.factory.BeanNameAware;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

public class Person implements BeanNameAware{
    @Autowired
    private Dog dog;
    @Autowired
    private Cat cat;

    public Dog getDog() {
        return dog;
    }
    public void setDogx(Dog dog) {
        this.dog = dog;
    }
    public Cat getCat() {
        return cat;
    }
    public void setCat(Cat cat) {
        this.cat = cat;
    }
    @Override
    public String toString() {
        return "Person [dog=" + dog + ", cat=" + cat + "]";
    }

    @Override
```

```
    public void setBeanName(String name) {
        System.out.println("==============="+this.getClass().getName()+
        "===="+name);
    }

}
```

c. 属性注入bean类型数据的原理:

当spring容器解析xml时,发现开启了属性注解,则会在创建bean时,检测属性上是否存在@Autowired注解,如果发现该注解,则会通过当前属性的名称寻找是否存在该id的bean,如果存在则注入进来,如果不存在,再检查是否存在和当前属性类型相同的bean,如果由则注入进来,如果都没有则抛出异常.

**也可以使用@Resource(name="id")指定注入给定id的bean，但是这种方式不建议大家使用。

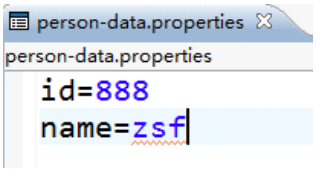
d. spring内置支持注入类型的注解方式的注入 - 非集合类型

spring中可以通过注解方式 注册bean，并可以通过@Autowired实现属性的自动注入，但注入的都是自定义的bean类型，如果类中包含例如 int long String等spring内置可注入的类型时，又该如何注入呢？ 可以使用@Value注解来实现注入。

```
package cn.tedu.beans;

import org.springframework.beans.factory.BeanNameAware;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component("per")
public class Person implements BeanNameAware{
    @Value("999")
    private int id;

    @Value("zs")
    private String name;

    @Autowired
```

```
    private Dog dog;

    @Autowired
    private Cat cat;

    public Dog getDog() {
        return dog;
    }
    public void setDogx(Dog dog) {
        this.dog = dog;
    }
    public Cat getCat() {
        return cat;
    }
    public void setCat(Cat cat) {
        this.cat = cat;
    }

    @Override
    public String toString() {
        return "Person [id=" + id + ", name=" + name + ", dog=" + dog
                + ", cat=" + cat + "]";
    }
    @Override
    public void setBeanName(String name) {
        System.out.println("==="+this.getClass().getName()+"==="+name);
    }

}
```

这种方式可以实现spring内置类型的注入，但是这种方式将注入的值写死在了代码中，后续如果希望改变注入的初始值，必须来修改源代码，此时可以将这些值配置到一个properties配置文件中，再在spring中进行引入。



applicationContext.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    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-3.2.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.2..
    ">
    <!-- 开启属性注解 -->
    <context:annotation-config></context:annotation-config>
    <context:component-scan base-package="cn.tedu.beans"></context:co
    <context:property-placeholder
        location="classpath:/person-data.properties"/>
</beans>
```

Person.java

```java
package cn.tedu.beans;

import org.springframework.beans.factory.BeanNa

@Component("per")
public class Person implements BeanNameAware{
    @Value("${id}")
    private int id;

    @Value("${name}")
    private String name;

    @Autowired
    private Dog dog;

    @Autowired
    private Cat cat;
```

e. spring内置支持注入类型的注解方式的注入 - 集合类型

需要将集合类型的数据配置到spring配置文件中，再通过@Value引入

配置过程：

将spring-util-3.2.xsd交给MyEclipse管理

在当前spring容器的配置文件中导入util名称空间

再通过适当的util标签注册数据

案例：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:util="http://www.springframework.org/schema/util"
    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-3.2.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.2.xsd
    http://www.springframework.org/schema/util
```

```
        http://www.springframework.org/schema/util/spring-util-3.2.xsd
        ">
        <!-- 开启属性注解 -->
        <context:annotation-config></context:annotation-config>
        <context:component-scan base-package="cn.tedu.beans">
        </context:component-scan>
        <context:property-placeholder location="classpath:/person-
        data.properties"/>

        <util:list id="l1">
            <value>北京</value>
            <value>上海</value>
            <value>广州</value>
            <value>深证</value>
        </util:list>

        <util:set id="s1">
            <value>法师</value>
            <value>射手</value>
            <value>打野</value>
            <value>战士</value>
            <value>打野</value>
            <value>坦克</value>
            <value>打野</value>
        </util:set>

        <util:map id="m1">
            <entry key="k1" value="v1"></entry>
            <entry key="k2" value="v2"></entry>
            <entry key="k3" value="v3"></entry>
            <entry key="k1" value="v4"></entry>
        </util:map>
    </beans>
```

再在类的属性中通过@Value注入赋值

```
    package cn.tedu.beans;
```

```
import java.util.List;
import java.util.Map;
import java.util.Set;

import org.springframework.beans.factory.BeanNameAware;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component("per")
public class Person implements BeanNameAware{
    @Value("${id}")
    private int id;

    @Value("${name}")
    private String name;

    @Value("#{@l1}")
    private List<String> addr;

    @Value("#{@s1}")
    private Set<String> jobs;

    @Value("#{@m1}")
    private Map<String,String> map;

    @Autowired
    private Dog dog;

    @Autowired
    private Cat cat;

    public Dog getDog() {
        return dog;
    }
```

```
    public void setDogx(Dog dog) {
        this.dog = dog;
    }
    public Cat getCat() {
        return cat;
    }
    public void setCat(Cat cat) {
        this.cat = cat;
    }

    public void setBeanName(String name) {
        System.out.println("==============="+this.getClass().getName()+
        "===="+name);
    }
    @Override
    public String toString() {
        return "Person [id=" + id + ", name=" + name + ", addr=" + addr
                + ", jobs=" + jobs + ", map=" + map + ", dog=" + dog + ",
                cat="
                + cat + "]";
    }
}
```

5. 其他注解

a. @Scope(value="prototype")

配置修饰的类的bean是单例还是多例，如果不配置默认为单例

案例：

```
package cn.tedu.beans;

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component
@Scope("prototype")
public class Dog {

}
```

b. @Lazy

配置修饰的类的bean采用懒加载机制

案例：

```
package cn.tedu.beans;

import org.springframework.context.annotation.Lazy;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component
@Lazy
public class Dog {
    public Dog() {
        System.out.println("Dog...被创建出来了...");
    }
}
```

c. @PostConstruct

在bean对应的类中 修饰某个方法 将该方法声明为初始化方法，对象创建之后立即执行。

d. @PreDestroy

在bean对应的类中 修饰某个方法 将该方法声明为销毁的方法，对象销毁之前调用的方法。

案例：

```
package cn.tedu.beans;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

import org.springframework.stereotype.Component;

@Component
public class Dog {
    public Dog() {
        System.out.println("Dog...被创建出来了...");
    }

    @PostConstruct
    public void init(){
        System.out.println("Dog的初始化方法。。。");
    }

    @PreDestroy
    public void destory(){
        System.out.println("Dog的销毁方法。。。");
    }
}
```

e. @Controller @Service @Repository @Component

这四个注解的功能是完全相同的，都是用来修饰类，将类声明为Spring管理的bean的。

其中@Component一般认为是通用的注解

而@Controller用在软件分层中的控制层，一般用在web层

而@Service用在软件分层中的业务访问层，一般用在service层

而@Repository用在软件分层中的数据访问层，一般用在dao层

利用Spring IOC DI 实现软件分层解耦

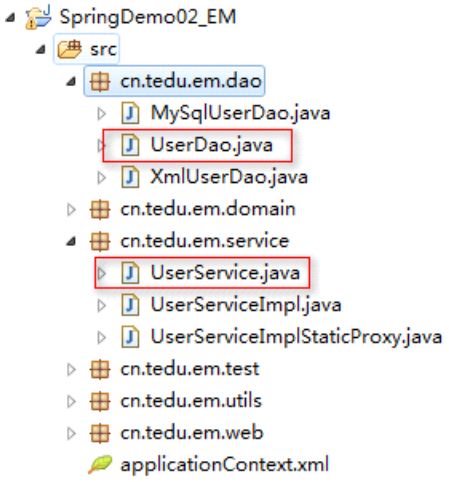
2018年10月15日 17:19

1. 软件分层思想回顾

在软件领域有MVC软件设计思想，指导着软件开发过程。在javaee开发领域，javaee的经典三层架构MVC设计思想的经典应用。而在软件设计思想中，追求的是"高内聚 低耦合"的目标，利用Spring的IOC 和 DI 可以非常方便的实现这个需求。

2. Spring IOC DI 改造EasyMall

在层与层之间设计接口，面向接口编程：



不再直接创建对象，而是通过Spring注入：

```
@Controller
public class RegistServlet {

    @Autowired
    private UserService userService;

    public void regist(){
        User user = new User(1,"zs","贼帅
        userService.registUser(user);
    }
}
@Service
public class UserServiceImpl implements UserService {

    @Autowired
    private UserDao userDao;

    public void upToVIP(User user){
```

如果存在多个实现类，则通过指定名称声明<bean>的id，实现使用指定实现类的bean:

```java
import cn.tedu.em.domain.User;
public interface UserDao {
    public void addUser(User user);
    public void updateUser(User user);
    public void deleteUser(int id);
    public User queryUser(int id);
}
```

```java
@Repository
public class XmlUserDao implements UserDao{
```

```java
@Repository("userDao")
public class MySqlUserDao implements UserDao{
```

```java
@Service
public class UserServiceImpl implements UserService {

    @Autowired
    private UserDao userDao;
```

**在Spring中，如果自动装配一个接口，默认会先通过属性名找对应id的bean，如果找不到就会去寻找是否存在该接口的实现类的bean，如果存在且只存在一个，则会将该bean注入，如果不存在或存在多个，则抛出异常。

SpringAOP基础 - 代理设计模式

2018年10月15日 16:46

1. 改造过后的EasyMall的问题

改造过后的EasyMall成功解决了耦合的问题，但是在很多地方仍然存在非该层应该实现的功能，造成了 无法“高内聚”的现象，同时存在大量存在重复代码，开发效率低下。

```
@Service
public class UserServiceImpl implements UserService {
    @Autowired
    private UserDao userDao;

    @Override
    public void registUser(User user) {
        try {
            System.out.println("校验权限。。。");
            System.out.println("开启事务。。。");
            System.out.println("记录日志。。。");
            userDao.addUser(user);
            System.out.println("提交事务。。。");
        } catch (Exception e) {
            System.out.println("回滚事务");
            e.printStackTrace();
        }
    }

    @Override
    public void upToVIP(User user) {
        try {
            System.out.println("校验权限。。。");
            System.out.println("开启事务。。。");
            System.out.println("记录日志。。。");
            userDao.updateUser(user);
            System.out.println("提交事务。。。");
        } catch (Exception e) {
            System.out.println("回滚事务");
            e.printStackTrace();
        }

    }

    @Override
    public void removeUser(User user) {
        try {
```

```
                System.out.println("校验权限。。。");
                System.out.println("开启事务。。。");
                System.out.println("记录日志。。。");
                userDao.deleteUser(user.getId());
                System.out.println("提交事务。。。");
            } catch (Exception e) {
                System.out.println("回滚事务");
                e.printStackTrace();
            }
        }

    }
```

此时可以通过代理设计模式，将这部分代码提取到代理者中，简化层中的代码。

2. 静态代理模式

```
package cn.tedu.staticproxy;
public interface SJSkill {
    public void 吃();
    public void 唱歌();
}

package cn.tedu.staticproxy;
public class FBB implements SJSkill{
    public void 吃(){
        System.out.println("fbb吃饭。。。");
    }
    public void 唱歌(){
        System.out.println("fbb唱歌。。。");
    }
}

package cn.tedu.staticproxy;
public class JJRStaticProxy implements SJSkill{

    private FBB fbb = new FBB();

    @Override
    public void 吃() {
        System.out.println("权限认证：你谁啊？？？？");
        fbb.吃();
        System.out.println("记录日志：等我，我记一下来访记录");
    }
```

```
    @Override
    public void 唱歌() {
        System.out.println("权限认证：你谁啊？？？？");
        fbb.唱歌();
        System.out.println("记录日志：等我，我记一下来访记录");
    }

}
```

```
package cn.tedu.staticproxy;
import org.junit.Test;
public class StaticProxyTest {
    @Test
    public void test01(){
        JJRStaticProxy jjr = new JJRStaticProxy();
        jjr.吃();
        jjr.唱歌();
    }
}
```

静态代理设计模式特点：

优点：

结构清晰 易于理解

缺点：

如果被代理者有多个方法，则代理者也需要开发多个方法，其中往往存在大量重复代码，仍然存在代码重复。

静态代理设计模式解决了软件分层过程中 额外的功能代码侵入模块的问题，将额外的功能代码提取到了代理者中进行，但是静态代理实现的代理者中存在大量重复的代码，并没有解决代码重复问题。所以在真正开发中--包括spring的底层，基本不会使用静态代理。

3. 动态代理 - jdk内置的动态代理

在jdk中提供了动态代理实现的工具类，直接使用该工具类就可以创建出代理者，并且可以通过内置的回调函数指定代理在工作时的执行逻辑，从而实现基于jdk原生api的动态代理机制。

java.lang.reflect

类 Proxy

java.lang.Object

static Object	**newProxyInstance**(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)

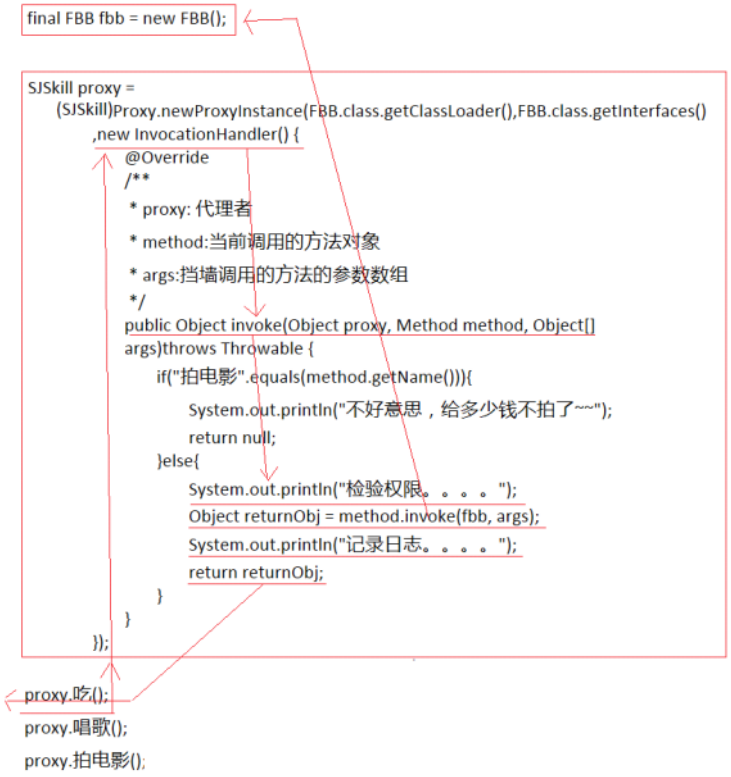
	InvocationHandler h) 返回一个指定接口的代理类实例，该接口可以将方法调用指派到指定的调用处理程序。

案例：

```
package cn.tedu.javaproxy;

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

import org.junit.Test;

public class JavaProxyTest {
    @Test
    public void test01(){

        //被代理者
        final FBB fbb = new FBB();

        //java动态代理方式 生成fbb的代理者
        /**
         * classLoader:用来生成代理者类的类加载器，通常可以传入被代理者类的类加载器
         * interfaces: 要求生成的代理者实现的接口们，通常就是实现和被代理者相同的接口，保证具有和被代理者相同的方法
         * invocationHandler: 用来设定回调函数的回调接口，使用者需要写一个类实现此接口，从而实现其中的invoke方法，
         * 在其中编写代码处理代理者调用方法时的回调过程，通常在这里调用真正对象身上的方法，并且在方法之前或之后做额外操作。
         */
        SJSkill proxy = (SJSkill)
        Proxy.newProxyInstance(FBB.class.getClassLoader(),FBB.class.getInterfaces()
                ,new InvocationHandler() {
                    @Override
                    /**
                     * proxy: 代理者
                     * method:当前调用的方法对象
                     * args:挡墙调用的方法的参数数组
                     */
                    public Object invoke(Object proxy, Method method, Object[] args)
                    throws Throwable {
```

```
                    if("拍电影".equals(method.getName())){
                        System.out.println("不好意思，给多少钱不拍了~~");
                        return null;
                    }else{
                        System.out.println("检验权限。。。。");
                        Object returnObj = method.invoke(fbb, args);
                        System.out.println("记录日志。。。。");
                        return returnObj;
                    }
                }
            });
        //从此之后，不允许直接调用被代理者身上的方法，而是要通过代理者来调用
        //fbb.吃();
        //fbb.唱歌();
        proxy.吃();
        proxy.唱歌();
        proxy.拍电影();
    }
}
```

java动态代理的原理图：



java动态代理的特点：

优点：

不需要像静态代理一样被代理方法都要实现一遍，而只需要在回调函数中进行处理就可以了，重复代码只需编写一次。

缺点：

java的动态代理是通过代理者实现和被代理者相同的接口来保证两者具有相同的方法的，如果被代理者想要被代理的方法不属于任何接口，则生成的代理者自然无法具有这个方法，也就无法实现对该方法的代理。

所以java的动态代理机制是基于接口进行的，受制于要代理的方法是否有接口的支持。

4. 动态代理 - 第三方包cglib实现的动态代理

CGLIB是第三方提供的动态代理的实现工具，不管有没有接口都可以实现动态代理。

CGLIB实现动态代理的原理是 生成的动态代理是被代理者的子类，所以代理者具有和父类即被代理者 相同的方法，从而实现代理。

a. 导入CGLIB相关包

之前导入的spring包中就包含了CGLIB

spring-core-3.2.3.RELEASE.jar

b. 开发CGLIB程序

案例：

```
package cn.tedu.cglibproxy;

import java.lang.reflect.Method;

import org.junit.Test;
import org.springframework.cglib.proxy.Enhancer;
import org.springframework.cglib.proxy.MethodInterceptor;
import org.springframework.cglib.proxy.MethodProxy;

public class CglibProxyTest {
    @Test
    public void test01(){
        final FBB fbb = new FBB();

        //增强器
        Enhancer enhancer = new Enhancer();

        //设定接口 -- 此方法要求生成的动态代理额外实现指定接口们 ，单cglib动态代理不是靠接口实现的，所以可以不设置
        enhancer.setInterfaces(fbb.getClass().getInterfaces());

        //设定父类 -- 此处要传入被代理者的类，cglib是通过集成被代理者的类来持有和被代理者相同的方法的，此方法必须设置
        enhancer.setSuperclass(fbb.getClass());
```

```
        //设定回调函数 -- 为增强器设定回调函数，之后通过增强器生成的代理对象调用任何方法都会走到此回调函数中，实现调用真正被代理对象的方法的效果
        enhancer.setCallback(new MethodInterceptor() {
            @Override
            public Object intercept(Object proxy, Method method, Object[] args,
                    MethodProxy methodProxy) throws Throwable {
                if("拍电影".equals(method.getName())){
                    System.out.println("对不起，不拍了~~~");
                    return null;
                }else{
                    System.out.println("检查权限。。。");
                    Object returnObj = method.invoke(fbb, args);
                    System.out.println("记录日志。。。");
                    return returnObj;
                }
            }
        });

        //生成代理对象
        FBB proxy = (FBB) enhancer.create();
        proxy.吃();
        proxy.唱歌();
        proxy.拍电影();
    }
}
```

CGLIB动态代理原理图：

CGLIB动态代理的特点：

优点：无论是否有接口都可以实现动态代理，使用场景基本不受限

缺点：第三方提供的动态代理机制，不是原生的，需要导入第三方开发包才可以使用。

5. 使用代理改造EasyMall

使用代理改造EasyMall，将功能代码提取到代理者中，实现“高内聚”的效果。

```
package cn.tedu.em.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import cn.tedu.em.dao.UserDao;
import cn.tedu.em.domain.User;

@Service
public class UserServiceImpl implements UserService {

    @Autowired
    private UserDao userDao;

    public void upToVIP(User user){
        userDao.updateUser(user);
    }

    public void removeUser(User user){
        userDao.deleteUser(5);
    }

    public void registUser(User user){
        userDao.addUser(user);
    }
}
```

```
package cn.tedu.em.service;

import java.lang.reflect.Method;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.cglib.proxy.Enhancer;
import org.springframework.cglib.proxy.MethodInterceptor;
import org.springframework.cglib.proxy.MethodProxy;
import org.springframework.stereotype.Service;

@Service
public class UserServiceImplCglibProxy {
```

```
    @Autowired
    @Qualifier("userServiceImpl")
    private UserService userService;

    public UserServiceImplCglibProxy() {

    }

    public UserService getCglibProxy() {
        Enhancer enhancer = new Enhancer();
        enhancer.setInterfaces(userService.getClass().getInterfaces());
        enhancer.setSuperclass(userService.getClass());
        enhancer.setCallback(new MethodInterceptor() {
            @Override
            public Object intercept(Object proxy, Method method, Object[] args,
                    MethodProxy mproxy) throws Throwable {
                try {
                    System.out.println("校验权限。。。");
                    System.out.println("开启事务。。。");
                    System.out.println("记录日志。。。");

                    Object returnObj = method.invoke(userService, args);

                    System.out.println("提交事务。。。");
                    return returnObj;
                } catch (Exception e) {
                    System.out.println("回滚事务");
                    e.printStackTrace();
                    throw new RuntimeException(e);
                }
            }
        });
        return (UserService) enhancer.create();
    }
}
```

```
package cn.tedu.em.web;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;

import cn.tedu.em.domain.User;
import cn.tedu.em.service.UserServiceImplCglibProxy;

@Controller
public class RegistServlet {
```

```
    @Autowired
    //private UserService userService;
    //private UserServiceImplJavaProxy proxy;
    private UserServiceImplCglibProxy proxy;
    public void regist(){
        User user = new User(1,"zs","贼帅的男子","piaoqian@tedu.cn");
        //proxy.getJavaProxy().upToVIP(user);
        proxy.getCglibProxy().upToVIP(user);
    }
}
```

SpringAOP详解

2018年10月16日 15:02

1. Spring aop中的基本概念

- *连接点（Joinpoint）*：在程序执行过程中某个特定的点，比如某方法调用的时候或者处理异常的时候。在Spring AOP中，一个连接点*总是*表示一个方法的执行。

  通俗讲：

  层于层之间调用的过程中，目标层中可供调用的方法，就称之为连接点。

- *切入点（Pointcut）*：匹配连接点的断言。通知和一个切入点表达式关联，并在满足这个切入点的连接点上运行（例如，当执行某个特定名称的方法时）。切入点表达式如何和连接点匹配是AOP的核心：Spring缺省使用AspectJ切入点语法。

  通俗讲：

  在连接点的基础上 增加上切入规则 选择出需要进行增强的切入点 这些基于切入规则选出来的连接点 就称之为切入点。

- *切面（Aspect）*：一个关注点的模块化，这个关注点可能会横切多个对象。事务管理是J2EE应用中一个关于横切关注点的很好的例子。在Spring AOP中，切面可以使用基于模式）或者基于@Aspect注解的方式来实现。

  通俗讲：

  狭义上就是 当spring拦截下切入点后 将这些切入点 交给 处理类 进行功能的增强，这个处理类就称之为切面。

  广义上来讲 将spring底层的代理 切入点 和 处理类 加在一起 实现的 对层与层之间调用过程进行增强的机制 称之为切面。

- *通知（Advice）*：在切面的某个特定的连接点上执行的动作。其中包括了“around”、“before”和“after”等不同类型的通知（通知的类型将在后面部分进行讨论）。许多AOP框架（包括Spring）都是以*拦截器*做通知模型，并维护一个以连接点为中心的拦截器链。

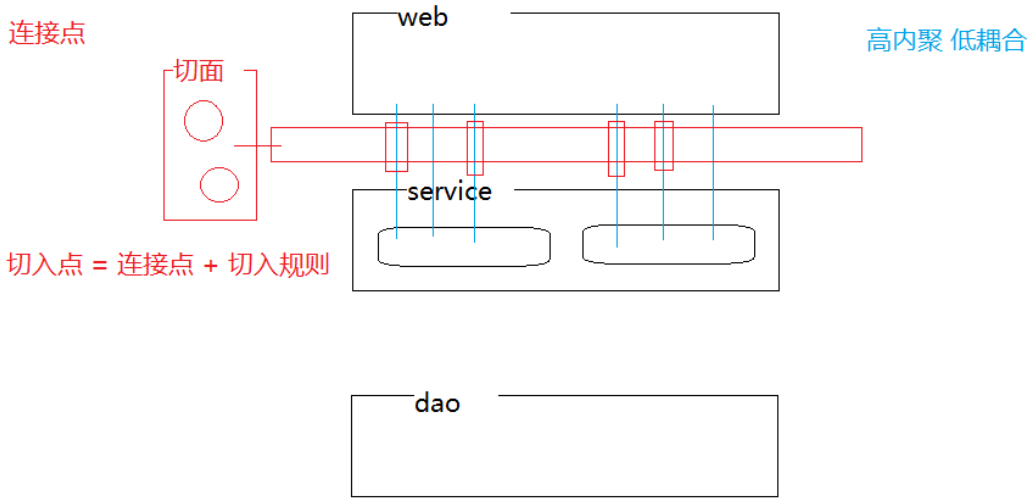
  通俗讲：

  在spring底层的代理拦截下切入点后，将切入点交给切面类，切面类中就要有处理这些切入点的方法，这些方法就称之为通知（也叫增强 增强方法）。针对于切入点执行的过程，通知还分为不同的类型，分别关注切入点在执行过程中的不同的时机。

- *目标对象（Target Object）*：被一个或者多个切面所通知的对象。也被称做*被通知（advised）*对象。 既然Spring AOP是通过运行时代理实现的，这个对象永远是一个*被代理（proxied）*对象。

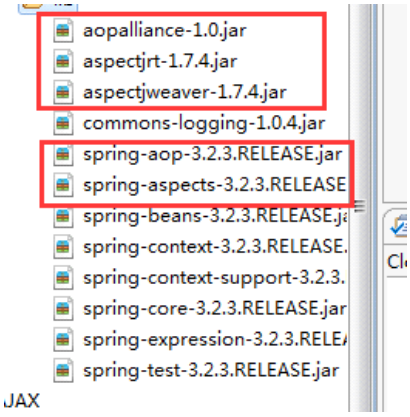
  通俗讲：

  就是真正希望被访问到的对象。spring底层的动态代理对他进行了代理，具体能不能真的访问到目标对象，或在目标对象真正执行之前和之后是否做一些额外的操作，取决于切面。



2. spring的aop入门案例

a. 导入aop相关开发包



b. 创建一个切面类

```
@Component
public class FirstAspect {
    public void before(){
    }
}
```

c. 定义通知

```java
@Component
public class FirstAspect {
    public void before(){
        System.out.println("before...");
    }
}
```

d. 定义一个连接点

```java
public class AOPTest {
    @Test
    public void test01(){
        ApplicationContext context = new ClassPath
        UserService userService = (UserService) co
        userService.addUser();
    }
}
```

一个连接点

```java
public interface UserService {
    public void addUser();
    public void updateUser();
    public void deleteUser();
    public void queryUser();
}
```

```java
@Service("userService")
public class UserServiceImpl implements UserService {

    @Override
    public void addUser() {
        System.out.println("增加用户。。");
    }

    @Override
    public void updateUser() {
        System.out.println("修改用户。。");
    }

    @Override
    public void deleteUser() {
        System.out.println("删除用户。。");
    }
```

e. 配置切入点

在MyEclipse中导入aop的schema约束文件，以便于在配置文件中可以提示标签。

在其中配置切入点：

```xml
<aop:config>
    <aop:pointcut
        expression="within(cn.tedu.service.UserServiceImpl)"
        id="pc01"/>
</aop:config>
```

f. 定义切面

```xml
<aop:config>
    <!-- 配置切入点 -->
    <aop:pointcut
        expression="within(cn.tedu.service.UserServiceImpl)"
        id="pc01"/>

    <!-- 配置切面 -->
    <aop:aspect ref="firstAspect">
        <aop:before method="before" pointcut-ref="pc01"/>
    </aop:aspect>
</aop:config>
```

切面类

切面类中的通知方法

该通知方法绑定到那个切入点

g. 执行方法，发现切面确实起作用

```
2018-10-16 16:29:2
信息: Loading XML be
2018-10-16 16:29:2
信息: Pre-instantiat
before...
增加用户。。
```

3. 切入点表达式

a. within表达式

通过类名进行匹配 粗粒度的切入点表达式

within(包名.类名)

则这个类中的所有的方法都会别表达式识别，成为切入点。

```xml
<aop:pointcut
    expression="within(cn.tedu.service.UserServiceImpl)"
    id="pc01"/>
```

在within表达式中可以使用*号匹配符，匹配指定包下所有的类，注意，只匹配当前包，不包括当前包的子孙包。

```xml
<aop:config>
    <!-- 配置切入点 -->
    <aop:pointcut
        expression="within(cn.tedu.service.*)"
        id="pc01"/>

    <!-- 配置切面 -->
    <aop:aspect ref="firstAspect">
        <aop:before method="before" pointcut-ref="pc01"/>
    </aop:aspect>
</aop:config>
```

此处的*代表切出该包下的所有的类

在within表达式中也可以用*号匹配符，匹配包

```xml
<aop:config>
    <!-- 配置切入点 -->
    <aop:pointcut
        expression="within(cn.tedu.service.*.*)"
        id="pc01"/>

    <!-- 配置切面 -->
    <aop:aspect ref="firstAspect">
        <aop:before method="before" pointcut-ref="pc01"/>
    </aop:aspect>
</aop:config>
```

此处第一个*代表一层子目录
第二个*代表这个子目录下的所有的类

在within表达式中也可以用..*号匹配符，匹配指定包下及其子孙包下的所有的类

```
<aop:config>
    <!-- 配置切入点 -->
    <aop:pointcut
        expression="within(cn.tedu.service..*)"
        id="pc01"/>

    <!-- 配置切面 -->
    <aop:aspect ref="firstAspect">
        <aop:before method="before" pointcut-ref="pc01"/>
    </aop:aspect>
</aop:config>
```

此处的..*代表匹配指定包下及其子孙包下所有的类

b. execution()表达式

语法:execution(返回值类型 包名.类名.方法名(参数类型,参数类型…))

例子1：

```
<aop:pointcut expression=
"execution(void cn.tedu.service.UserServiceImpl.addUser(java.lang.String))"
id="pc1"/>
```

该切入点规则表示，切出指定包下指定类下指定名称指定参数指定返回值的方法。

例子2：

```
<aop:pointcut expression="execution(* cn.tedu.service.*.query())" id="pc1"/>
```

该切入点规则表示，切出指定包下所有的类中的query方法，要求无参，但返回值类型不限。

例子3：

```
<aop:pointcut expression="execution(* cn.tedu.service..*.query())"
id="pc1"/>
```

该切入点规则表示，切出指定包及其子孙包下所有的类中的query方法，要求无参，但返回值类型不限。

例子4：

```
<aop:pointcut expression="execution(* cn.tedu.service..
*.query(int,java.lang.String))" id="pc1"/>
```

该切入点规则表示，切出指定包及其子孙包下所有的类中的query方法，要求参数为int java.langString类型，但返回值类型不限。

例子5：

```
<aop:pointcut expression="execution(* cn.tedu.service..*.query(..))"
id="pc1"/>
```

该切入点规则表示，切出指定包及其子孙包下所有的类中的query方法，参数数量及类型不限，返回值类型不限。

例子6：

```
<aop:pointcut expression="execution(* cn.tedu.service..*.*(..))" id="pc1"/>
```

该切入点规则表示，切出指定包及其子孙包下所有的类中的任意方法，参数数量及类型不限，返回值类型不限。这种写法等价于within表达式的功能。

4. SpringAOP的原理

Spring会在用户获取对象时，生成目标对象的代理对象，之后根据切入点规则，匹配用户连接点，得到切入点，当切入点被调用时，不会直接去找目标对象，而是通过代理对象拦截之后交由切面类中的指定的通知执行来进行增强。

Spring自动为目标对象生成代理对象，默认情况下，如果目标对象实现过接口，则采用java的动态代理机制，如果目标对象没有实现过接口，则采用cglib动态代理。开发者可以可以在

spring中进行配置，要求无论目标对象是否实现过接口，都强制使用cglib动态代理。

```
<aop:config proxy-target-class="true">
    <!-- 配置切入点 -->
    <aop:pointcut
        expression="execution(* cn.tedu.service..*.*(..))"
        id="pc01"/>

    <!-- 配置切面 -->
    <aop:aspect ref="firstAspect">
        <aop:before method="before" pointcut-ref="pc01"/>
    </aop:aspect>
</aop:config>
```

此选项要求无论目标对象是否实现过接口都采用cglib动态代理生成代理者

5. Spring的五大通知类型

a. 前置通知

在目标方法执行之前执行执行的通知

前置通知方法，可以没有参数，也可以额外接收一个JoinPoint，Spring会自动将该对象传入，代表当前的连接点，通过该对象可以获取目标对象 和 目标方法相关的信息。

注意，如果接收JoinPoint，必须保证其为方法的第一个参数，否则报错。

配置方法：

```
<aop:before method="before" pointcut-ref="pc01"/>
```

```
@Component
public class FirstAspect {
    public void before(){
        System.out.println("before...");
    }
}
```

```
@Component
public class FirstAspect {
    public void before(JoinPoint jp){
        Class clz = jp.getTarget().getClass();
        Signature signature = jp.getSignature();
        String name = signature.getName();
        System.out.println("before...["+clz+"]...["+name+"]...");
    }
```

可以选择额外的传入一个JoinPoint连接点对象，要注意必须用方法的第一个参数接收

通过JoinPoint对象获取更多信息

b. 环绕通知

在目标方法执行之前和之后都可以执行额外代码的通知。

在环绕通知中必须显式的调用目标方法，目标方法才会执行，这个显式调用时通过ProceedingJoinPoint来实现的，可以在环绕通知中接收一个此类型的形参，spring容器会自动将该对象传入，注意这个参数必须处在环绕通知的第一个形参位置。

**要注意，只有环绕通知可以接收ProceedingJoinPoint，而其他通知只能接收JoinPoint。

环绕通知需要返回返回值，否则真正调用者将拿不到返回值，只能得到一个null。

环绕通知有

控制目标方法是否执行、有控制是否返回值、甚至改变返回值

的能力

环绕通知虽然有这样的能力，但一定要慎用，不是技术上不可行，而是要小心不要破坏

了软件分层的“高内聚 低耦合”的目标。

```
<aop:around method="around" pointcut-ref="pc01"/>
```

```
public Object around(ProceedingJoinPoint jp) throws Throwable{
    System.out.println("around before...");
    Object obj = jp.proceed();//--显式的调用目标方法
    System.out.println("around after...");
    return obj;
}
```

c. 后置通知

在目标方法执行之后执行的通知。

在后置通知中也可以选择性的接收一个JoinPoint来获取连接点的额外信息，但是这个参数必须处在参数列表的第一个。

```
<aop:after-returning method="afterReturn" pointcut-ref="pc01" />
```

```
public void afterReturn(JoinPoint jp){
    Class clz = jp.getTarget().getClass();
    Signature signature = jp.getSignature();
    String name = signature.getName();
    System.out.println("afterReturn...["+clz+"]...["+name+"]...");
}
```

在后置通知中，还可以通过配置获取返回值

```
<aop:after-returning method="afterReturn"
        pointcut-ref="pc01" returning="msg"/>
```

```
public void afterReturn(JoinPoint jp,Object msg){
    Class clz = jp.getTarget().getClass();
    Signature signature = jp.getSignature();
    String name = signature.getName();
    System.out.println("afterReturn...["+clz+"]...[
}
```

一定要保证JoinPoint处在参数列表的第一位，否则抛异常

```
public void afterReturn(Object msg,JoinPoint jp){
    Class clz = jp.getTarget().getClass();
    Signature signature = jp.getSignature();
    String name = signature.getName();
    System.out.println("afterReturn...["+clz+"]...
}
```

一定要保证JoinPoint处在参数列表的第一个位置否则抛出异常

```
cn\tedu\service\UserServiceImpl.class]: BeanPostProcessor b
Instantiation of bean failed; nested exception is org.springfr
ArgumentException: error at ::0 formal unbound in pointcut
```

d. 异常通知

在目标方法抛出异常时执行的通知

配置方法：

```
<aop:after-throwing method="afterThrow" pointcut-ref="pc01"/>
```

```
public void afterThrow(){
    System.out.println("afterThrow...");
}
```

可以配置传入JoinPoint获取目标对象和目标方法相关信息，但必须处在参数列表第一位

另外，还可以配置参数，让异常通知可以接收到目标方法抛出的异常对象

```
<aop:after-throwing method="afterThrow"
        pointcut-ref="pc01" throwing="e"/>

public void afterThrow(JoinPoint jp,Throwable e){
    Class clz = jp.getTarget().getClass();
    String name = jp.getSignature().getName();
    System.out.println("afterThrow..["+clz+"]..["+name+"].
}
```

e. 最终通知

是在目标方法执行之后执行的通知。和后置通知不同之处在于，后置通知是在方法正常返回后执行的通知，如果方法没有正常返-例如抛出异常，则后置通知不会执行。而最终通知无论如何都会在目标方法调用过后执行，即使目标方法没有正常的执行完成。另外，后置通知可以通过配置得到返回值，而最终通知无法得到。

配置方式：

```
<aop:after method="after" pointcut-ref="pc01" />

public void after(){
    System.out.println("after...");
}
```

最终通知也可以额外接收一个JoinPoint参数，来获取目标对象和目标方法相关信息，但一定要保证必须是第一个参数。

```
public void after(JoinPoint jp){
    Class clz = jp.getTarget().getClass();
    String name = jp.getSignature().getName();
    System.out.println("after..["+clz+"]..["+name+"]...");
}
```

f. 五种通知执行的顺序

i. 在目标方法没有抛出异常的情况下

前置通知

环绕通知的调用目标方法之前的代码

目标方法

环绕通知的调用目标方法之后的代码

后置通知

最终通知

ii. 在目标方法抛出异常的情况下：

前置通知

环绕通知的调用目标方法之前的代码

目标方法 抛出异常 异常通知

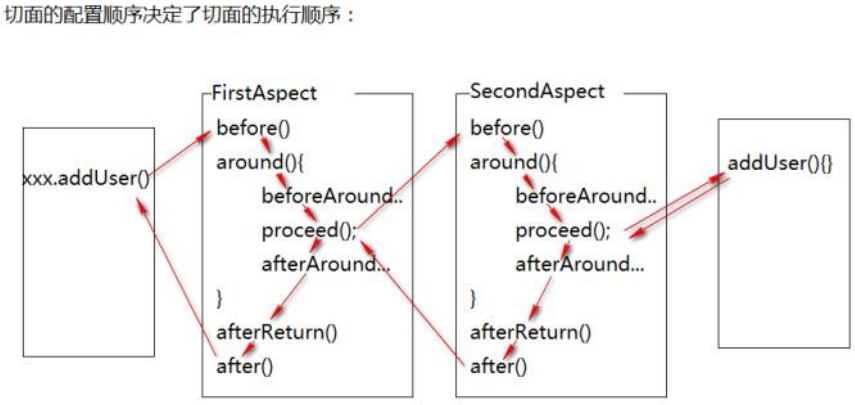
最终通知

iii. 如果存在多个切面：

多切面执行时，采用了责任链设计模式。

切面的配置顺序决定了切面的执行顺序，多个切面执行的过程，类似于方法调用的过程，在环绕通知的proceed()执行时，去执行下一个切面或如果没有下一个切面执

行目标方法，从而达成了如下的执行过程：



如果目标方法抛出异常：

g. 五种通知的常见使用场景

前置通知	记录日志(方法将被调用)
环绕通知	控制事务 权限控制
后置通知	记录日志(方法已经成功调用)
异常通知	异常处理 控制事务
最终通知	记录日志(方法已经调用，但不一定成功)

6. AOP的注解方式实现

spring也支持注解方式实现AOP，相对于配置文件方式，注解配置更加的轻量级，配置、修改更加方便。

a. 开启AOP的注解配置方式

```
<!-- aop注解配置 -->
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>
```

b. 将制定的类标志位一个切面

```
@Component
@Aspect
public class FirstAspect {
```

c. 配置通知 制定切入点规则

前置通知 @Before

环绕通知 @Around

后置通知 @AfterReturning

异常通知 @AfterThrowing

最终通知 @After

```
@Before("execution(* cn.tedu.service.*.*(..))")
public void before(){
    System.out.println("1before...");
}
```

**通过注解的配置 等价于 配置文件的配置

```
<aop:pointcut
    expression="execution(* cn.tedu.service.UserServiceImpl.addUs
    id="pc01"/>
<aop:aspect ref="firstAspect">
    <aop:before method="before" pointcut-ref="pc01"/>
</aop:aspect>

@Component
@Aspect
public class FirstAspect {
    @Before("excution(* cn.tedu.service.*.*(..))")
    public void before(){
        System.out.println("1before...");
    }
```

d. 如果一个切面中多个通知 重复使用同一个切入点表达式，则可以将该切入点表达式单独定义，后续引用，注意，在当前切面中通过注解定义的切入点只在当前切面中起作用，其他切面看不到。

```
@Pointcut("execution(* cn.tedu.service.*.*(..))")
public void mx(){

}

@Before("mx()")
public void before(){
    System.out.println("1before...");
}

@Around("mx()")
public Object around(ProceedingJoinPoint jp) throws Throwable{
    System.out.println("1around before...");
    Object obj = jp.proceed();//--显式的调用目标方法
    System.out.println("1around after...");
```

声明一个切入点，最关键的是切入点表达式
方法叫什么无所谓，只是用作后续引用

e. 在后置通知的注解中，也可以额外配置一个returning属性，来指定一个参数名接受目标方法执行后的返回值

```
@AfterReturning(value="mx()",returning="msg")
public void afterReturn(String msg){
    System.out.println("1afterReturn..."+msg);
}
```

f. 在异常通知的注解中，也可以额外配置一个throwing属性，来指定一个参数名接受目标方法抛出的异常对象

```
@AfterThrowing(value = "mx()",throwing="e")
public void afterThrow(Throwable e){
    System.out.println("1afterThrow.."+e.getMessage());
}
```

SpringAOP案例

2018年10月17日 15:05

1. 异常信息收集

在业务方法执行时，如果有异常抛出，则根据异常信息记录日志

```
@Component
@Aspect
public class ExceptionAspect {
	private FileWriter writer = null;

	{
		try {
			writer = new FileWriter("err.log");
		} catch (IOException e) {
			e.printStackTrace();
		}
	}

	@AfterThrowing(value="execution(public * *(..))",throwing="e")
	public void afterThrowing(JoinPoint ji ,Throwable e) throws
	Exception{
		Class<?> clz = ji.getTarget().getClass();
		String mName = ji.getSignature().getName();
		String msg = e.getMessage();
		String err = "--["+clz+"]--["+mName+"]--["+msg+"]--";
		writer.write(err);
		writer.flush();
	}

}
```

2. 统计业务方法执行的时间

统计所有业务方法执行时的耗时

```
package cn.tedu.aop;

import org.aspectj.lang.ProceedingJoinPoint;
```

```
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

@Component
@Aspect
public class UseTimeAspect {

    @Around("execution(* add*(..))")
    public void around(ProceedingJoinPoint jp) throws Throwable{
        //--获取目标类
        Class<?> clz = jp.getTarget().getClass();
        //--获取目标方法
        String mName = jp.getSignature().getName();
        //--开始时间
        long begin = System.currentTimeMillis();
        //--执行目标方法
        jp.proceed();
        //--获取结束时间
        long end = System.currentTimeMillis();
        //--计算耗时
        long use = end - begin;

        System.out.println("--["+clz+"]["+mName+"]方法共用时
        ["+use+"]毫秒--");

    }
}
```

3. 实现事务控制

通过AOP实现事务控制，通过注解来标识方法是否需要事务

a. 开发事务注解

```
package cn.tedu.anno;

import java.lang.annotation.ElementType;
```

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Trans {
}
```

b. 在业务层使用注解

```
package cn.tedu.service;

import org.springframework.stereotype.Service;

import cn.tedu.anno.Trans;

@Service("userService")
public class UserServiceImpl implements UserService {

    @Trans
    @Override
    public String addUser(String name) {
        int i = 1/0;
        System.out.println("增加用户。。");
        return "zs";
    }

    @Trans
    @Override
    public void updateUser() {
        System.out.println("修改用户。。");
    }

    @Trans
    @Override
    public void deleteUser() {
        System.out.println("删除用户。。");
    }
```

```
    @Override
    public void query() {
        System.out.println("查询用户。。");
    }

}
```

c. 开发切面，实现环绕通知 和 异常通知 根据目标方法是否有@Trans决定要不要增强事务控制

```
package cn.tedu.aop;

import java.lang.reflect.Method;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.Signature;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.aspectj.lang.reflect.MethodSignature;
import org.springframework.stereotype.Component;

import cn.tedu.anno.Trans;
import cn.tedu.utils.TransactionManager;

@Component
@Aspect
public class TransAspect {

    @Pointcut("execution(* cn.tedu.service.*.*(..))")
    public void pc(){
    }

    @Around("pc()")
    public void around(ProceedingJoinPoint jp) throws Throwable{
        //--获取当前的目标方法
        MethodSignature signature = (MethodSignature)
        jp.getSignature();
```

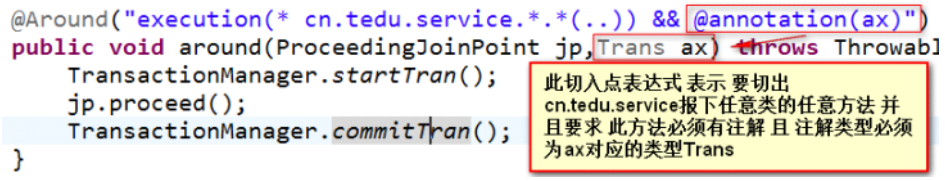
```
        //--!!!!此方法 如果底层使用的是jdk的动态动态代理 得到的将是接口上的方法
        Method method = signature.getMethod();
        //--获取实现类上的method
        Class<? extends Object> clz = jp.getTarget().getClass();
        Method realMethod = clz.getMethod(method.getName(), method.getParameterTypes());
        //--判断目标方法上是否有@Trans
        if(realMethod.isAnnotationPresent(Trans.class)){
            //--有@Trans，走事务
            TransactionManager.startTran();
            jp.proceed();
            TransactionManager.commitTran();
        }else{
            //--没有@Trans，不走事务
            jp.proceed();
        }

    }

    @AfterThrowing("pc()")
    public void throwing(JoinPoint jp) throws Exception{
        //--获取当前的目标方法
        MethodSignature signature = (MethodSignature) jp.getSignature();
        //--!!!!此方法 如果底层使用的是jdk的动态动态代理 得到的将是接口上的方法
        Method method = signature.getMethod();
        //--获取实现类上的method
        Class<? extends Object> clz = jp.getTarget().getClass();
        Method realMethod = clz.getMethod(method.getName(), method.getParameterTypes());
        //--判断目标方法上是否有@Trans
        if(realMethod.isAnnotationPresent(Trans.class)){
            //--有@Trans，走事务
```

```
                TransactionManager.rollbackTran();
            }else{
                //--没有@Trans，不走事务
            }
        }
    }
```

d. 进行测试

```
package cn.tedu.test;

import org.junit.Test;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationConte
xt;

import cn.tedu.service.UserService;
public class AOPTest {
    @Test
    public void test01(){
        ApplicationContext context = new
        ClassPathXmlApplicationContext("applicationContext.xml");
        UserService userService = (UserService)
        context.getBean("userService");
        String result = userService.addUser("zsf");
        userService.updateUser();
        userService.deleteUser();
        userService.query();
    }
}
```

e. 改进:利用spring提供的切入点表达式来直接获取注解




```
package cn.tedu.aop;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
```

```
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

import cn.tedu.anno.Trans;
import cn.tedu.utils.TransactionManager;

@Component
@Aspect
public class TransAspect2 {

    @Around("execution(* cn.tedu.service.*.*(..)) &&
    @annotation(ax)")
    public void around(ProceedingJoinPoint jp,Trans ax) throws
    Throwable{
        TransactionManager.startTran();
        jp.proceed();
        TransactionManager.commitTran();
    }

    @AfterThrowing("execution(* cn.tedu.service.*.*(..)) &&
    @annotation(ax)")
    public void throwing(JoinPoint jp,Trans ax) throws Exception{
        TransactionManager.rollbackTran();
    }
}
```

4. 通过aop进行权限控制

通过aop来实现权限控制，通过自定义注解声明业务方法是否需要权限控制，通过权限注解上的属性声明需要什么样的权限

a. 开发权限注解

```
package cn.tedu.anno;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
```

```
import java.lang.annotation.Target;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Priv {
    String value();
}
```

b. 应用权限注解

```
package cn.tedu.service;

import org.springframework.stereotype.Service;

import cn.tedu.anno.Priv;

@Service("userService")
public class UserServiceImpl implements UserService {

    @Override
    @Priv("add")
    public String addUser(String name) {
        System.out.println("增加用户。。");
        return "zs";
    }

    @Override
    @Priv("update")
    public void updateUser() {
        System.out.println("修改用户。。");
    }

    @Override
    @Priv("delete")
    public void deleteUser() {
        System.out.println("删除用户。。");
    }

    @Override
    public void query() {
```

```
            System.out.println("查询用户。。");
        }

    }
```

c. 开发切面，实现环绕通知，检查用户是否具有权限，有权限则执行目标方法，没权限提示不执行目标方法

```
package cn.tedu.aop;

import java.util.List;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

import cn.tedu.anno.Priv;
import cn.tedu.test.AOPTest;

@Component
@Aspect
public class PrivAspect {

    @Around("execution(* cn.tedu.service.*.*(..)) &&
    @annotation(anno)")
    public void around(ProceedingJoinPoint jp,Priv anno) throws
    Throwable{
        //检查权限
        //--方法要啥权限
        String needPriv = anno.value();
        //--用户权限是啥
        List<String> userPrivList = AOPTest.userPrivList;
        //判断是否匹配
        if(userPrivList.contains(needPriv)){
            //--有权限就执行目标
            System.out.println("恭喜您有权限。。");
            jp.proceed();
```

```
                }else{
                    //--没有权限就不执行目标方法
                    System.out.println("不好意思，没有权限，禁止访
                    问。。");
                }
            }

        }
```

d. 测试代码

```
        package cn.tedu.test;

        import java.util.ArrayList;
        import java.util.List;

        import org.junit.Test;
        import org.springframework.context.ApplicationContext;
        import
        org.springframework.context.support.ClassPathXmlApplicationConte
        xt;

        import cn.tedu.service.UserService;
        public class AOPTest {
            public static List<String> userPrivList = new ArrayList();
            @Test
            public void test01(){
                userPrivList.add("add");
                //userPrivList.add("update");
                userPrivList.add("delete");


                ApplicationContext context = new
                ClassPathXmlApplicationContext("applicationContext.xml");
                UserService userService = (UserService)
                context.getBean("userService");
                String result = userService.addUser("zsf");
                userService.updateUser();
                userService.deleteUser();
                userService.query();
```

```
        }
    }
```

e. 思考题

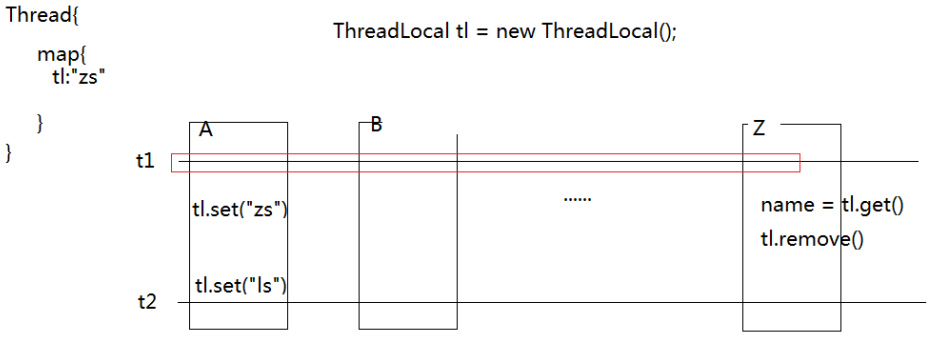
此处我们使用类中静态的List作为用户权限列表，但是在真正的web开发中，同时会有很多用户并发访问，如果使用这种方案，所有用户使用同一个权限列表，就乱掉了，怎样让每个人都有一个自己的权限列表呢？

提示 -- ThreadLocal机制

f. 通过ThreadLocal实现每个用户各自具有一个权限列表

ThreadLocal是一种传递数据的技术，线程由程序的上游下游执行，此时利用线程对象中的map来存储数据，从而实现由程序的上游下下游传递数据。

利用ThreadLocal可以实现隐式的参数传递，另外由于每个线程都有各自的Map，从而可以各自持有各自的数据，不会存在多线程并发安全问题，所以也可以作为多线程并发安全问题解决方案之一。



利用ThreadLocal可以实现权限控制案例中每个用户(对应各自的线程)持有各自独立权限了列表。

```
public static ThreadLocal<List<String>> tl = new ThreadLocal<List<String>>()
@Test
public void test01(){
    List<String> userPrivList = new ArrayList<String>();
    userPrivList.add("add");
    userPrivList.add("update")
    userPrivList.add("delete")
    tl.set(userPrivList);

    ApplicationContext context = new ClassPathXmlApplicationContext("applica
    UserService userService = (UserService) context.getBean("userService");
    String result = userService.addUser("zsf");
    userService.updateUser();
    userService.deleteUser();
    userService.query();
}
```

此行代码会在当前线程的内部的map中存储键值对供后续使用，其中键是当前的ThreadLocal对象，值是传入的

```
public class PrivAspect {

    @Around("execution(* cn.tedu.service.*.*(..)) && @annotation(anno)")
    public void around(ProceedingJoinPoint jp,Priv anno) throws Throwable{
        //检查权限
        //--方法要啥权限
        String needPriv = anno.value();
        //--用户权限是啥
        List<String> userPrivList = AOPTest.tl.get();
        //判断是否匹配
        if(userPrivList.contains(needPriv)){
            //--有权限就执行目标
            System.out.println("恭喜您有权限。。");
            jp.proceed();
        }else{
            //--没有权限就不执行目标方法
            System.out.println("不好意思，没有权限，禁止访问。。");
        }

        //--从本地线程中去除信息
        //AOPTest.tl.remove();
    }
```

此行代码从当前线程内部以tl为键获取对应的值，将之前存入的当前用户的权限列表返回

```
public class AOPTest {
    public static ThreadLocal<List<String>> tl = new ThreadLocal<L
    @Test
    public void test01(){
        List<String> userPrivList = new ArrayList<String>();
        userPrivList.add("add");
        //userPrivList.add("update");
        userPrivList.add("delete");
        tl.set(userPrivList);
        ApplicationContext context = new ClassPathXmlApplicationCo
        UserService userService = (UserService) context.getBean("u
        String result = userService.addUser("zsf");
        userService.updateUser();
        userService.deleteUser();
        userService.query();
    }
}
```

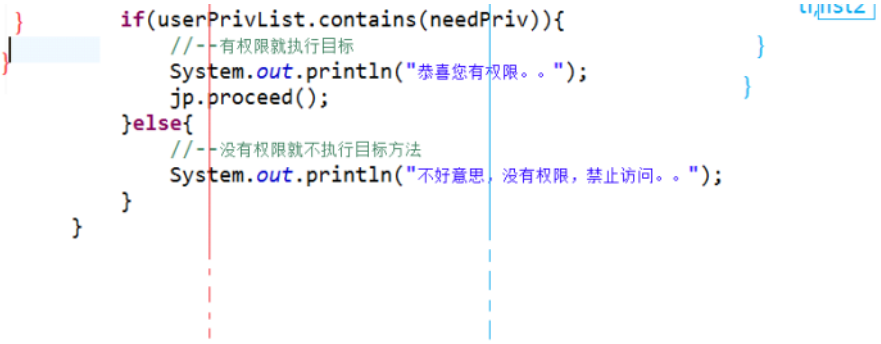
t{ map{ tl,list1 } }

t{ map{ tl,list2 } }

```
@Around("execution(* cn.tedu.service.*.*(..)) && @annotation(anno)")
public void around(ProceedingJoinPoint jp,Priv anno) throws Throwable{
    //检查权限
    //--方法要啥权限
    String needPriv = anno.value();
    //--用户权限是啥
    List<String> userPrivList = AOPTest.tl.get();
    //判断是否匹配
    if(userPrivList.contains(needPriv)){
        //--有权限就执行目标
        System.out.println("恭喜您有权限。。");
        jp.proceed();
```

t{ map{ tl,list1 } }

t{ map{ tl,list2 } }


if(userPrivList.contains(needPriv)){
//--有权限就执行目标
System.out.println("恭喜您有权限。。");
jp.proceed();
}else{
//--没有权限就不执行目标方法
System.out.println("不好意思，没有权限，禁止访问。。");
}

Spring整合JDBC

2018年10月19日 10:15

1. 回顾JDBC

a. java操作关系型数据的API。导入相关数据库的驱动包后可以通过JDBC提供的接口来操作数据库。

b. 实现JDBC的六个步骤

注册数据库驱动

获取数据库连接

获取传输器对象

传输sql执行获取结果集对象

遍历结果集获取信息

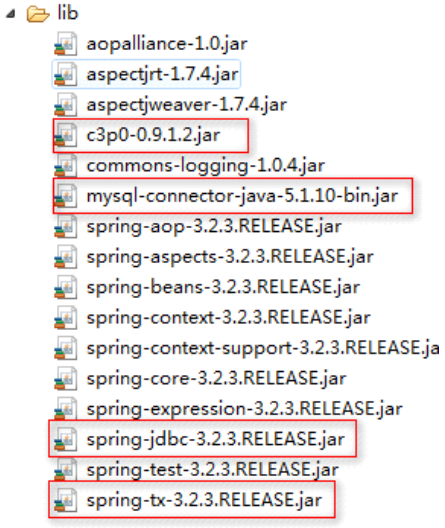
关闭资源

c. 数据库连接池(数据源)

C3P0连接池

2. Spring整合JDBC

a. 导入相关开发包



b. 将数据源交于Spring管理

jdbc.properties

```
driverClass=com.mysql.jdbc.Driver
jdbcUrl=jdbc:mysql://localhost:3306/spring
user=root
password=root
```

```
<!-- 引入配置文件 -->
<context:property-placeholder location="classpath:/jdbc.properties"/>
<!-- 配置数据源 -->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <property name="driverClass" value="${driverClass}"></property>
    <property name="jdbcUrl" value="${jdbcUrl}"></property>
    <property name="user" value="${user}"></property>
    <property name="password" value="${password}"></property>
</bean>
```

c. 通过spring获取数据源，获取连接，操作数据库

```
@Test
public void test02() throws SQLException{
    DataSource source = (DataSource) context.getBean("dataSource");
    Connection conn = source.getConnection();
    PreparedStatement ps = conn.prepareStatement("select * from users");
    ResultSet rs = ps.executeQuery();
    while(rs.next()){
        int id = rs.getInt("id");
        String name = rs.getString("name");
        int age = rs.getInt("age");
        User user = new User(id,name,age);
        System.out.println(user);
    }
}
```

3. Spring中的JDBC模板类

使用模板类,能够极大的简化原有JDBC的编程过程,让数据库操作变得简单.

a. 在Spring中配置JDBC模板类

```
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource"></property>
</bean>
```

b. 使用JDBC模板类实现增删改查

```
@Test
public void test02() throws SQLException{
    DataSource source = (DataSource) context.getBean("dataSource");
    Connection conn = source.getConnection();
    PreparedStatement ps = conn.prepareStatement("select * from users");
    ResultSet rs = ps.executeQuery();
    while(rs.next()){
        int id = rs.getInt("id");
        String name = rs.getString("name");
        int age = rs.getInt("age");
        User user = new User(id,name,age);
        System.out.println(user);
    }
}

@Test
public void test03() throws SQLException{
    JdbcTemplate jdbcTemplate = (JdbcTemplate) context.getBean("jdbcTemplate");
    List<Map<String,Object>> list = jdbcTemplate.queryForList("select * from users");
    System.out.println(list);
}
@Test
public void test04() throws SQLException{
    JdbcTemplate jdbcTemplate = (JdbcTemplate) context.getBean("jdbcTemplate");
    SqlRowSet srs = jdbcTemplate.queryForRowSet("select * from users where id = ?",2);
    srs.next();
    System.out.println(srs.getString("name"));
}
@Test
public void test05() throws SQLException{
```

```
        JdbcTemplate jdbcTemplate = (JdbcTemplate) context.getBean("jdbcTemplate");
        int n = jdbcTemplate.update("insert into users values (null,?,?)", "ddd",23);
        System.out.println("修改成功，影响到的行数为："+n);
    }
    @Test
    public void test06() throws SQLException{
        JdbcTemplate jdbcTemplate = (JdbcTemplate) context.getBean("jdbcTemplate");
        int n = jdbcTemplate.update("update users set age = ? where id = ?", 33,4);
        System.out.println("修改成功，影响到的行数为："+n);
    }
    @Test
    public void test07() throws SQLException{
        JdbcTemplate jdbcTemplate = (JdbcTemplate) context.getBean("jdbcTemplate");
        int n = jdbcTemplate.update("delete from users where id = ?", 4);
        System.out.println("修改成功，影响到的行数为："+n);
    }
```

c. 使用RowMapper封装bean

RowMapper接口定义了对象到列的映射关系，可以帮助我们在查询时自动封装bean。

```
    public class UserRowMapper implements RowMapper<User> {

        @Override
        public User mapRow(ResultSet rs, int i) throws SQLException {
            User user = new User();
            user.setId(rs.getInt("id"));
            user.setName(rs.getString("name"));
            user.setAge(rs.getInt("age"));
            return user;
        }

    }

    /**
     * 使用RowMap封装bean
     */
    @Test
    public void test08(){
        JdbcTemplate jdbcTemplate = (JdbcTemplate) context.getBean("jdbcTemplate");
        List<User> list = jdbcTemplate.query("select * from users",new UserRowMapper());
        System.out.println(list);
    }
    /**
     * 使用RowMap封装bean
     */
    @Test
    public void test09(){
        JdbcTemplate jdbcTemplate = (JdbcTemplate) context.getBean("jdbcTemplate");
        User user = jdbcTemplate.queryForObject("select * from users where id = ?",new
        UserRowMapper(),2);
        System.out.println(user);
```

```
}
```

d. 使用BeanPropertyRowMapper自动进行映射

BeanPropertyRowMapper内部可以使用指定类进行反射(内省)来获知类内部的属性信息，自动映射到表的列。使用它一定要注意，类的属性名要和对应表的列名必须对应的上，否则属性无法自动映射。BeanPropertyRowMapper底层通过反射(内省)来实现，相对于之前自己写的RowMapper效率比较低。

```
/**
 * 通过BeanPropertyRowMapper实现自动封装bean
 */
@Test
public void test10(){
    JdbcTemplate jdbcTemplate = (JdbcTemplate) context.getBean("jdbcTemplate");
    List<User> list = jdbcTemplate.query("select * from users", new
    BeanPropertyRowMapper(User.class));
    System.out.println(list);
}
/**
 * 通过BeanPropertyRowMapper实现自动封装bean
 */
@Test
public void test11(){
    JdbcTemplate jdbcTemplate = (JdbcTemplate) context.getBean("jdbcTemplate");
    User user = jdbcTemplate.queryForObject("select * from users where id = ?", new
    BeanPropertyRowMapper(User.class),2);
    System.out.println(user);
}
```

4. 声明式事务处理

Spring中提供了内置的事务处理机制，称之为声明式事务处理。

a. 创建项目，模拟MVC三层架构

略

b. 在配置文件中导入相关约束

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    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-3.2.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.2.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-3.2.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-3.2.xsd
    ">
```

c. 配置事务管理器

```
<!-- 配置事务管理器 -->
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
    <property name="dataSource" ref="dataSource"></property>
</bean>
```

d. 配置事务切面

```
<!-- 配置事务切面 -->
<aop:config>
    <aop:pointcut expression="execution(* cn.tedu.spring.service.*.*(..))"
        id="pc"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="pc"/>
</aop:config>
```

e. 配置事务通知

```
<!-- 配置事务通知 -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="add*" propagation="REQUIRED"/>
        <tx:method name="delete*" propagation="REQUIRED"/>
        <tx:method name="update*" propagation="REQUIRED"/>
    </tx:attributes>
</tx:advice>
```

f. 配置关系图

```
<!-- 配置数据源 -->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <property name="driverClass" value="${driverClass}"></property>
    <property name="jdbcUrl" value="${jdbcUrl}"></property>
    <property name="user" value="${user}"></property>
    <property name="password" value="${password}"></property>
</bean>

<!-- 配置事务管理器 -->
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
    <property name="dataSource" ref="dataSource"></property>
</bean>

<!-- 配置事务通知 -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="add*" propagation="REQUIRED"/>
        <tx:method name="delete*" propagation="REQUIRED"/>
        <tx:method name="update*" propagation="REQUIRED"/>
    </tx:attributes>
</tx:advice>

<!-- 配置事务切面 -->
<aop:config>
    <aop:pointcut expression="execution(* cn.tedu.spring.service.*.*(..))"
        id="pc"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="pc"/>
</aop:config>
```

g. 事务管理策略

异常的种类：

java.lang.Throwable
|-Exception
|-RuntimeException
|-其他Exception
|-Error

spring内置的事务策略，只在底层抛出的异常是运行时异常时，才会回滚，其他异常不回滚，留给用户手动处理。

也可以在配置中指定在原有规则的基础上，哪些异常额外的回滚或不回滚：

```
<!-- 配置事务通知 -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="add*" propagation="REQUIRED"
            rollback-for="java.sql.SQLException"
            no-rollback-for="java.lang.ArithmeticException"/>
        <tx:method name="delete*" propagation="REQUIRED"/>
        <tx:method name="update*" propagation="REQUIRED"/>
    </tx:attributes>
</tx:advice>
```

配置成如下形式，可以实现任意异常都自动回滚：

```
<!-- 配置事务通知 -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="add*" propagation="REQUIRED"
            rollback-for="java.lang.Throwable" />
        <tx:method name="delete*" propagation="REQUIRED"/>
        <tx:method name="update*" propagation="REQUIRED"/>
    </tx:attributes>
</tx:advice>
```

h. 注意：如果在一个业务逻辑中，需要有多部不同表的操作，此时应该在service层中完成对不同表的操作，以此保证多部操作处在同一个事务中，切勿将业务代码在web层中调用不同Service来实现，虽然正常情况下也可以完成功能，但一旦出问题，和可能只有部分操作被回滚，数据出问题。

出现这种问题，不是事务管理机制的问题，而是开发者将业务逻辑错误的在web层中进行了实现，所以切记，web层只负责和用户交互，不进行任何业务逻辑的处理，任何业务逻辑都在service层中进行。

5. 声明式事务处理 - 注解方式

a. 开启事务注解配置

```
<!-- 开启spring的注解方式配置事务 -->
<tx:annotation-driven/>
```

b. 在方法上通过注解开启事务

即可以标注在接口上，也可以标注在实现类上，理论上应该表在接口上，实现面向接口编程，但实际开发中为了方便也有人写在实现类上。

```
@Transactional
public void addUser(User user) throws SQLException;

@Override
@Transactional
public void addUser(User user) throws SQLException {
    userDao.addUser(user);
    int i = 1/0;
    //throw new SQLException();
}
```

也可以在类上使用此接口，此时类中所有方法都会有事务

```
@Transactional
public class UserServiceImpl implements UserService {
```

当在类上开启了事务后，可以此类的方法中使用如下方式，控制某个方法没有事务

```
@Override
@Transactional(propagation=Propagation.NOT_SUPPORTED)
public void addUser(User user) throws SQLException {
    userDao.addUser(user);
    int i = 1/0;
    //throw new SQLException();
}
```

通过注解控制事务时，和配置文件方式控制事务相同的是，默认只有运行时异常会回滚，非运行异常不回滚，此时可以通过如下注解选项额外配置 哪些异常需要回滚，哪些不需要。

```java
@Transactional(rollbackFor=Throwable.class,noRollbackFor=SQLException.class)
public void addUser(User user) throws SQLException {
    userDao.addUser(user);
    //int i = 1/0;
    throw new SQLException();
}
```

6. 扩展案例

**扩展案例：缓存的使用 - 通过AOP实现为查询操作增加缓存机制

```
@Component
@Aspect
public class CatchAspect {
    /**
     * 使用Map实现的缓存
     *    只会增加不会减少
     *    没有超时时间，不管过了多久都使用历史缓存
     *    没有存储优化
     */
    private Map<Integer,User> map = new HashMap<Integer,User>();

    @Around("execution(* cn.tedu.spring.service.*.queryUser(..))")
    public User around(ProceedingJoinPoint jp) throws Throwable{
        int i = (Integer) jp.getArgs()[0];
        if(map.containsKey(i)){
            System.out.println("使用缓存查询。。。");
            //有缓存
            return map.get(i);
        }else{
            System.out.println("使用数据库查询。。。");
            //没缓存
            User user = (User) jp.proceed();
            map.put(user.getId(), user);
            return user;
        }
    }
}
```