# Spring开发框架

## Spring开发框架

### 传统开发问题分析

如果说使用最原始的MVC开发模式，我们会发现整体的开发，虽然代码直观，并且也是技术的原始支持，可是开发的效率实在是不高。

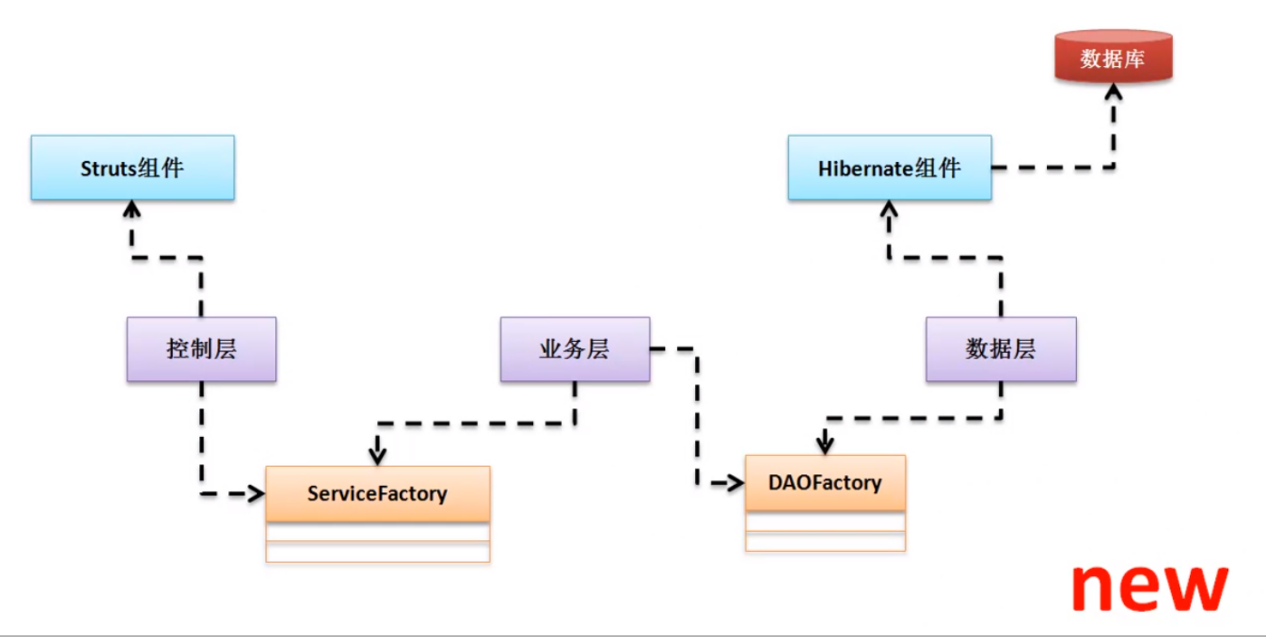
于是可以进行如下改进：

显示层改进：JQuery+JSON，简化了数据的传输，优化了DOM文档的处理，提供了丰富的选择器，并且可以使用BootStrap实现页面的风格展示。

控制层改进：Struts2.x 提供有转换器，更好的跳转结构，拦截器的提供适合数据验证，方便的文件上传处理。用户只需要关注控制层细节的编写即可，

数据层改进：Hibernate隐藏了用户具体的jdbc操作处理细节。

于是发现最终所有的层都在发生着改变，那么业务层的问题越来越明显，即使使用了Struts2.x 的联合开发，业务层也不得不面对一堆的问题：一堆的工程类（new）



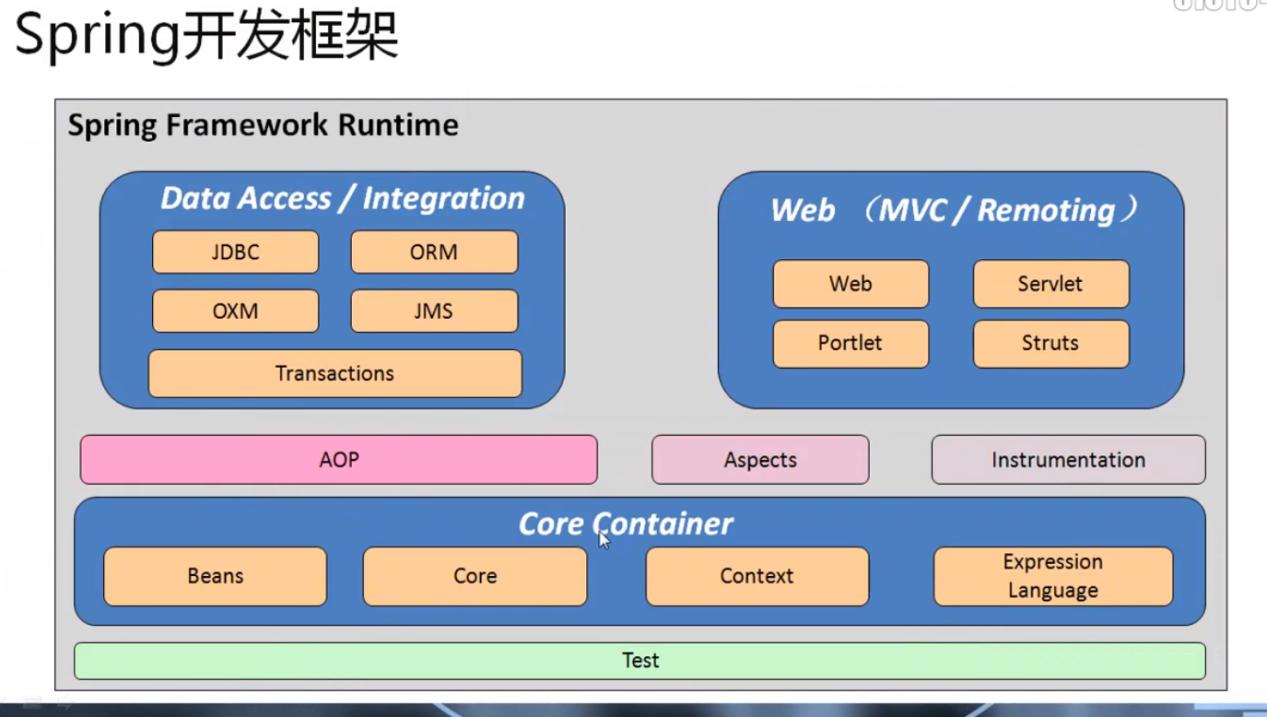
正因为如此，实际的开发过程中，如何更好的处理new (工厂类),如何处理核心业务操作，也成为了问题，为了达到某一目的总是重复编写者代码。

所以在这样的状态下，Spring的产生，几乎颠覆了传统开发中的设计思想。提出了自己的容器，并采用依赖注入和控制反转结合切面编程的设计，让用户在开发中不再关注于所有的辅助手段，也不再关注于对象的创建与回收。用户只关心要完成的核心功能。

总结Spring的核心本质=超级大工厂+超级代理设计。

### Spring整体介绍

#### Spring 的组成



在整个Spring开发中，有几个核心的组成模块：

1. 核心容器组件：提供有相关的工厂方面的支持

·beans：在Spring中所有java类的对象管理都有Spring负责，用户开发的时候不再需要处理new（业务层、数据层工厂类）;

·core：包含了Spring最底层的开发支持，例如：依赖注入与关系配置、数据访问、包括资源访问等；

·Context：上下文环境，在Spring容器中，所有对象的生命周期处理以及事物处理都将通过此模块完成；

·表达式语言：Spring允许开发者自己定义表达式操作，以增强Spring管理；

1. 切面编程组件：提供了相关代理方面的支持

·AOP：是切面编程灵魂所在，利用切面编程来解决所有辅助性操作，例如数据库连接关闭，事物提交回滚；

·Aspect：提供了切面编程中，确认切入点的语法；

·Instrumetation：是在jdk1.5之后，提供的一个新功能，主要用于检测jvm在运行中代码的动态处理;

##### 数据访问模块：

·JDBC：jdbc开发操作问题，实际上许多的开发框架都已经发现，所以在Spring里面也提供与jdbc风格类似，但是又简化的开发模板，

·ORM：负责各种ORMapping组件的连接处理（Hibernate、IBatis、JDO、MyBatis），包括自定义的jdbc连接;

·OXM：提供了一个对象与XML文件之间的相互转换；

·JMS：提供了消息服务的支持;

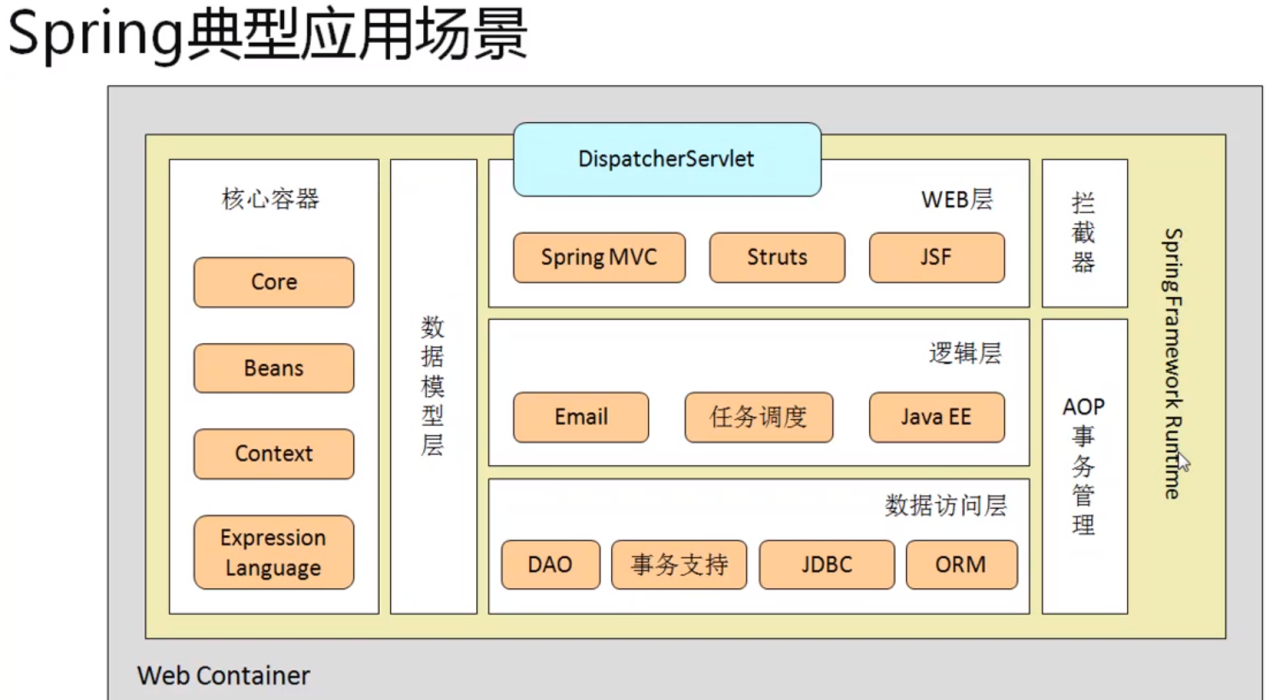
·Transactions：提供了事务的支持操作；

##### web支持模块

·Web开发支持：Spring自己提供了一套自己的MVC实现框架，可以说是现在最好的框架，那就是SpringMVC;

·Struts：方便的与各种开源框架整合，但不包括Struts1（整合性很差）;

通过以上介绍，可以发现Spring几乎是一个完美的整体，除了可以自己进行各种项目的开发支持外，还可以方便的整合与其他的开发框架。



实际的开发中，会由Spring容器来维护所有的开发组件的管理。

### 控制反转（Ioc）

按照以往的编程方式，使用者必须知道接口，必须知道子类、还要知道子类（N个）做什么用的。于是new操作也就成为了代码的最大耦合点。

为了解决开发者必须关注底层实现细节，所以控制反转出现了，****Ioc—Inversion of Control，即“控制反转”，不是什么技术，而是一种设计思想。****控制什么，反转给了谁，下面将进行详细说明。

　在平时的java应用开发中，我们要实现某一个功能或者说是完成某个业务逻辑时至少需要两个或以上的对象来协作完成，在没有使用Spring的时候，每个对象在需要使用他的合作对象时，自己均要使用像new object() 这样的语法来将合作对象创建出来，这个合作对象是由自己主动创建出来的，创建合作对象的主动权在自己手上，自己需要哪个合作对象，就主动去创建，创建合作对象的主动权和创建时机是由自己把控的，而这样就会使得对象间的耦合度高了。　　所以****控制反转IoC(Inversion of Control)是说创建对象的控制权进行转移，以前创建对象的主动权和创建时机是由自己把控的，而现在这种权力转移到第三方****，比如转移交给了IoC容器，它就是一个专门用来创建对象的工厂，你要什么对象，它就给你什么对象，有了 IoC容器，依赖关系就变了，原先的依赖关系就没了，它们都依赖IoC容器了，通过IoC容器来建立它们之间的关系。实现了松耦合。

DI(依赖注入)只是Ioc设计思想的一种实现。

Spring能带来的好处：

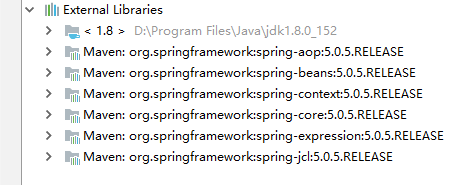
·让用户可以更加方便的取得实例化对象；

·让用户避免工厂（new）的麻烦（将new的（控制权）过程反转给了Ioc容器）；

·让用户避免了对象转型所带来的安全问题。

搭建Spring开发框架：

<**dependency**>  
 <**groupId**>org.springframework</**groupId**>  
 <**artifactId**>spring-context</**artifactId**>  
 <**version**>5.0.5.RELEASE</**version**>  
</**dependency**>



## 二 、Spring依赖注入

在Spring中最为核心的两大组件：IOC&DI、AOP，其中IOC称为控制反转，实际上是将对象的创建交给了Spring负责处理，但是如果只观察的话，严格来讲，并不能全面观察到Spring优点，还需要进一步观察DI（依赖注入，控制反转设计思想的一种实现）。

在整个Spring里面可以发现，其最大的特点是：直接设置类的完整名称，而后就可以取得类的实例化对象，按这样的思路，Spring的核心设计思想就是反射。

### 2.1 构造方法注入

任何类的对象实例化都是一定要使用类中的无参构造方法完成。反射实例化的本质就是在不显示的指定构造器的情况下，默认调用无参构造，这也是为什么，在创建对象时，如果定义了其他构造方法，一定要显示的声明无参构造的原因。不提供无参构造则Class类中的newInstance()将无法使用，此时需要找到指定参数的构造方法，利用Constructor类来进行调用。

范例：定义Dept类（无参构造）

public class Dept {

private Integer deptNo;

private String dname;

private String loc;

}

在Spring配置文件中，将dept类交给Spring（IOC）容器管理:

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

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

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

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

<bean id="dept" class="Dept"/>

</beans>

实现有参构造的注入，有三种方式：

·按参数顺序索引配置：

1. 修改Dept类

public class Dept {

private Integer deptNo;

private String dname;

private String loc;

public Dept(Integer deptNo, String dname, String loc) {

this.deptNo = deptNo;

this.dname = dname;

this.loc = loc;

}

@Override

public String toString() {

return "Dept{" +

"deptNo=" + deptNo +

", dname='" + dname + '\'' +

", loc='" + loc + '\'' +

'}';

}

1. 修改spring-config.xml配置：

<bean id="dept" class="Dept">

<constructor-arg index="0" value="10"/>

<constructor-arg index="1" value="美女部"/>

<constructor-arg index="2" value="成都"/>

</bean>

此时只要注意参数传递的时候只需要注意参数的顺序以及匹配的类型，比如部门编号只能是数字。

·按参数类型配置

修改spring-config.xml文件：

<bean id="dept" class="Dept">

<constructor-arg type="java.lang.Integer" value="10"/>

<constructor-arg type="java.lang.String" value="美女部"/>

<constructor-arg type="java.lang.String" value="成都"/>

</bean>

·按参数名称配置

修改spring-config.xml文件：

<bean id="dept" class="Dept">

<constructor-arg name="deptNo" value="10"/>

<constructor-arg name="dname" value="美女部"/>

<constructor-arg name="loc" value="成都"/>

</bean>

当然也可以bean中指定要是用的参数名称：

先修改Dept类：

public class Dept {

private Integer deptNo;

private String dname;

private String loc;

**@ConstructorProperties(value = {"dno","deptname","dloc"})**

public Dept(Integer deptNo, String dname, String loc) {

this.deptNo = deptNo;

this.dname = dname;

this.loc = loc;

}

@Override

public String toString() {

return "Dept{" +

"deptNo=" + deptNo +

", dname='" + dname + '\'' +

", loc='" + loc + '\'' +

'}';

}

接下来修改spring-config.xml文件：

<bean id="dept" class="Dept">

<constructor-arg name="dno" value="10"/>

<constructor-arg name="deptname" value="美女部"/>

<constructor-arg name="dloc" value="成都"/>

</bean>

### 2.2 setter注入

从正常的简单java类的使用来将，往往都会利用无参构造方法进行对象实例化处理，而后再利用setter设置具体的内容，而这点在Spring中可以说实现的很完美。

范例：定义要使用的类

**public class** Dept {  
 **private** Integer **deptNo**;  
 **private** String **dname**;  
 **private** String **loc**;  
  
 **public void** setDeptNo(Integer deptNo) {  
 **this**.**deptNo** = deptNo;  
 }  
  
 **public void** setDname(String dname) {  
 **this**.**dname** = dname;  
 }  
  
 **public void** setLoc(String loc) {  
 **this**.**loc** = loc;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "Dept{"** +  
 **"deptNo="** + **deptNo** +  
 **", dname='"** + **dname** + **'\''** +  
 **", loc='"** + **loc** + **'\''** +  
 **'}'**;  
 }  
}

以上程序准备了一序列的setter方法，这个时候就可以利用配置文件的方式实现setter方法的调用。

范例：配置spring-config.xml文件

*<?***xml version="1.0" encoding="UTF-8"***?>*<**beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>  
 <**bean id="str" class="java.lang.String"**>  
 *<!--利用String类的有参构造：String(String str)-->* <**constructor-arg index="0" value="成都"**/>  
 </**bean**>  
 <**bean id="dept" class="Dept"**>  
 *<!--一旦使用了property就意味着要调用类中的指定属性的setter方法-->* <**property name="deptNo" value="10"**/>  
 <**property name="dname"**>  
 <**value**>销售部</**value**>  
 </**property**>  
 <**property name="loc" ref="str"**><!--第一种方式-->  
 *<!--<ref bean="str">第二种方式</ref>-->* </**property**>  
 </**bean**>  
</**beans**>

在为每个类属性设置内容的时候实际上都可以采用两种方式完成：

1. value：表示设置具体的内容;
2. Ref：表示引用其他定义的bean对象;

如果使用了setter方法，严格来讲所有属性都可以设置,也就包含了其他引用对象，例如：一个雇员属于一个部门，则应该创建一个雇员类，而后引用这个部门类<property ref=”dept”></property>。

实践上，常用的数据类型里面，还包括有Boolean类型，如果是通过配置文件设置Boolean内容的话，则有以下几种选择方式：yes与no、true与false、on与off、0与1。

<**property name="flag" value="1"**/>

利用此类方式就可以直接在配置文件中观察出所有的对象间的关联关系。

### 2.3 集合注入

对于集合的注入操作在Spring中支持以下几种：List集合（数组）、Set集合、Map集合、Properties集合。

#### 2.3.1范例：注入List集合

创建Member类：

**public class** Member {  
 **private** List<String> **list**;  
  
 **public void** setList(List<String> list) {  
 **this**.**list** = list;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "Member{"** +  
 **"list="** + **list** +  
 **'}'**;  
 }  
}

随后在配置文件之中配置此类，并设置集合的内容。修改spring-config.xml配置文件

<bean id="list" class="pojo.Member">

<property name="list">

<list value-type="java.lang.String">

<value>成都</value>

<value>武汉</value>

<value>上海</value>

</list>

</property>

</bean>

如果不设置“value-type”的属性,也可以为集合设置内容。默认情况下，Spring会为List接口实例化对象。

#### 2.3.2 Set集合的设置

修改Member类

**private** Set<String> **set**;

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

@Override  
**public** String toString() {  
 **return "Member{"** +  
 **"list="** + **list** +  
 **", set="** + **set** +  
 **'}'**;  
}

...

在spring-config.xml中配置：

<**bean id="list" class="pojo.Member"**>  
 <**property name="list"**>  
 <**list value-type="java.lang.String"**>  
 <**value**>成都</**value**>  
 <**value**>武汉</**value**>  
 <**value**>上海</**value**>  
 </**list**>  
 </**property**>  
 <**property name="set"**>  
 <**set**>  
 <**value**>a</**value**>  
 <**value**>a</**value**>  
 <**value**>c</**value**>  
 </**set**>  
 </**property**>  
</**bean**>

默认情况下Spring为Set接口对象实例化，使用的是LinkedHashSet子类，实际上这个子类的特点相当于融合了List与Set的特点，List按顺序保存，而Set不允许重复。

#### 2.3.3 Map 集合设置

如果要设置Map的集合,那么每一个Map集合中都要求存放有一个Map.Entry的接口对象，而这表示一个完整的实体，在这个实体里面包含有key与value.

范例：修改Member类

private Set<String> set;

public void setEntry(Map<String, String> entry) {

this.entry = entry;

}

...

修改spring-config.xml文件

<**bean id="list" class="pojo.Member"**>  
 <**property name="list"**>  
 <**list value-type="java.lang.String"**>  
 <**value**>成都</**value**>  
 <**value**>武汉</**value**>  
 <**value**>上海</**value**>  
 </**list**>  
 </**property**>  
 <**property name="set"**>  
 <**set**>  
 <**value**>a</**value**>  
 <**value**>c</**value**>  
 </**set**>  
 </**property**>  
 <**property name="entry"**>  
 <**map**>  
 <**entry key="yangmi" value="杨幂"**></**entry**>  
 <**entry**>  
 <**key**>  
 <**value**>baiqian</**value**>  
 </**key**>  
 <**value**>白浅</**value**>  
 </**entry**>  
 </**map**>  
 </**property**>  
</**bean**>

在Map集合里面可以存放各种数据类型，而不仅仅只是字符串，最重要的是，如果要定义字符串可以直接定义properties保存。

#### 2.3.4 properties集合注入

修改Member类

**private** Properties **properties**;

**public void** setProperties(Properties properties) {  
 **this**.**properties** = properties;  
}

...

修改spring-config.xml文件

<**bean id="list" class="pojo.Member"**>  
 <**property name="list"**>  
 <**list value-type="java.lang.String"**>  
 <**value**>成都</**value**>  
 <**value**>武汉</**value**>  
 <**value**>上海</**value**>  
 </**list**>  
 </**property**>  
 <**property name="set"**>  
 <**set**>  
 <**value**>a</**value**>  
 <**value**>c</**value**>  
 </**set**>  
 </**property**>  
 <**property name="entry"**>  
 <**map**>  
 <**entry key="yangmi" value="杨幂"**></**entry**>  
 <**entry**>  
 <**key**>  
 <**value**>baiqian</**value**>  
 </**key**>  
 <**value**>白浅</**value**>  
 </**entry**>  
 </**map**>  
 </**property**>  
 <**property name="properties"**>  
 <**props**>  
 <**prop key="yangmi"**>杨幂</**prop**>  
 <**prop key="baiqian"**>白浅</**prop**>  
 </**props**>  
 </**property**>  
</**bean**>

以上就实现了四种常见集合的注入过程，而所有的注入过程都是通过配置文件显示的定义出来。当然，对于集合的操作，严格来说在实际开发中，也可以进行关系的配置。比如：

一个部门包含有多个雇员，这个时候就可以通过集合的方式来配置。

范例：定义emp类

**public class** Emp {  
 **private** Integer **emo**;  
  
 **private** String **ename**;  
  
 **private** Integer **deptNo**;  
  
 **public void** setEmo(Integer emo) {  
 **this**.**emo** = emo;  
 }  
  
 **public void** setEname(String ename) {  
 **this**.**ename** = ename;  
 }  
  
 **public void** setDeptNo(Integer deptNo) {  
 **this**.**deptNo** = deptNo;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "Emp{"** +  
 **"emo="** + **emo** +  
 **", ename='"** + **ename** + **'\''** +  
 **", deptNo="** + **deptNo** +  
 **'}'**;  
 }  
}

范例：修改Dept类定义

**private** List<Emp> **emps**;  
  
**public void** setEmps(List<Emp> emps) {  
 **this**.**emps** = emps;  
}

....

修改spring-config.xml文件

<**bean id="dept" class="pojo.Dept"**>  
 *<!--一旦使用了property就意味着要调用类中的指定属性的setter方法-->* <**property name="deptNo" value="10"**/>  
 <**property name="dname"**>  
 <**value**>销售部</**value**>  
 </**property**>  
 <**property name="emps"**>  
 <**list**>  
 <**ref bean="empa"**></**ref**>  
 <**ref bean="empb"**></**ref**>  
 <**ref bean="empc"**></**ref**>  
 </**list**>  
 </**property**>  
 <**property name="loc" ref="str"**>  
 *<!--<ref bean="str"></ref>-->* </**property**>  
 <**property name="flag" value="1"**/>  
</**bean**>  
<**bean id="empa" class="pojo.Emp"**>  
 <**property name="deptNo" value="10"**></**property**>  
 <**property name="emo" value="1001"**></**property**>  
 <**property name="ename" value="杨幂"**></**property**>  
</**bean**>  
<**bean id="empb" class="pojo.Emp"**>  
 <**property name="deptNo" value="10"**></**property**>  
 <**property name="emo" value="1002"**></**property**>  
 <**property name="ename" value="大幂幂"**></**property**>  
</**bean**>  
<**bean id="empc" class="pojo.Emp"**>  
 <**property name="deptNo" value="10"**></**property**>  
 <**property name="emo" value="1003"**></**property**>  
 <**property name="ename" value="白浅"**></**property**>  
</**bean**>

测试结果：

Dept{deptNo=10, dname='销售部', loc='成都', flag=true, emps=[Emp{emo=1001, ename='杨幂', deptNo=10}, Emp{emo=1002, ename='大幂幂', deptNo=10}, Emp{emo=1003, ename='白浅', deptNo=10}]}

Process finished with exit code 0

### 2.4 内部Bean

如果在一个对象中，要引用其他的类对象，，但这个对象只引用一次的话，就可以采用内部的方式来进行处理。

范例：为一个雇员配置一个部门

<**bean id="empa" class="pojo.Emp"**>  
 <**property name="deptNo" value="10"**></**property**>  
 <**property name="emo" value="1001"**></**property**>  
 <**property name="ename" value="杨幂"**></**property**>  
 <**property name="dept"** >  
 <**bean id="d" class="pojo.Dept"**>  
 <**property name="deptNo" value="20"**/>  
 <**property name="dname"**>  
 <**value**>销售部</**value**>  
 </**property**>  
 </**bean**>  
 </**property**>  
</**bean**>

### 2.5 使用p命名空间

为方便属性内容的配置，又提供了一种p命名空间的设置方式。

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

如果现在的项目中出现了p命名空间，则可以实现更为简单的POJO类的实例化设置内容操作。它会使用“p:属性=内容”形式完成，如果要进行引用，则使用“p:属性-ref=其他bean”的方式处理。

范例：假设有如下的bean关系

Dept类

**public class** Dept {  
 **private** Integer **deptNo**;  
 **private** String **dname**;  
 **private** String **loc**;  
 **private boolean flag**;  
  
 **public void** setDeptNo(Integer deptNo) {  
 **this**.**deptNo** = deptNo;  
 }  
  
 **public void** setDname(String dname) {  
 **this**.**dname** = dname;  
 }  
  
 **public void** setLoc(String loc) {  
 **this**.**loc** = loc;  
 }  
  
 **public void** setFlag(**boolean** flag) {  
 **this**.**flag** = flag;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "Dept{"** +  
 **"deptNo="** + **deptNo** +  
 **", dname='"** + **dname** + **'\''** +  
 **", loc='"** + **loc** + **'\''** +  
 **", flag="** + **flag** +  
 **'}'**;  
 }  
}

Emp类

**public class** Emp {  
 **private** Integer **emo**;  
  
 **private** String **ename**;  
  
 **private** Dept **dept**;  
  
 **public void** setEmo(Integer emo) {  
 **this**.**emo** = emo;  
 }  
  
 **public void** setEname(String ename) {  
 **this**.**ename** = ename;  
 }  
  
 **public void** setDept(Dept dept) {  
 **this**.**dept** = dept;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "Emp{"** +  
 **"emo="** + **emo** +  
 **", ename='"** + **ename** + **'\''** +  
 **", dept="** + **dept** +  
 **'}'**;  
 }  
}

随后可以使用p命名空间简化配置

范例：简化配置

*<?***xml version="1.0" encoding="UTF-8"***?>*<**beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xmlns:p="http://www.springframework.org/schema/p"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>  
  
 <**bean id="emp" class="pojo.Emp" p:emo="7369" p:ename="杨幂" p:dept-ref="dept"**/>  
 <**bean id="dept" class="pojo.Dept" p:deptNo="10" p:dname="美女部" p:loc="成都"**/>  
</**beans**>

从总体来讲会发现此时的设置方式可以简化配置的长度。

### 2.6 Bean的其他配置

默认情况下，当一个类配置到Spring之后，实际上会在容器启动的时候自动进行初始化。Spring也提供有延迟加载bean的配置，也就是说第一次使用的时候进行加载。

<**bean id="emp" class="pojo.Emp" lazy-init="true"**>  
 <**property name="ename" value="杨幂"**></**property**>  
 <**property name="emo" value="6379"**></**property**>  
</**bean**>

在好多web开发中都会提供初始化与销毁的控制操作，这样就留给了用户一些自己的空间。为了方便进行初始化与销毁的自定义实现，Spring对此也有相当丰富的支持。

范例：定义一个类

**public class** Message {  
  
 **public** Message() {  
 System.***out***.println(**"构造方法初始化..."**);  
 }  
  
 **public void** init(){  
 System.***out***.println(**"Message 自定义初始化方法init..."**);  
 }  
  
 **public void** destroy(){  
 System.***out***.println(**"Message 自定义销毁方法destroy..."**);  
 }  
}

这个类没有任何继承关系，是一个完全独立的程序类，要想其自定义的初始化与销毁起作用，那么还学要进行特殊的配置以及特殊的调用。

范例：在spring-config.xml文件中配置

<**bean id="message" class="hello.Message" init-method="init" destroy-method="destroy"**></**bean**>

默认情况下如果勒种存在有构造方法，则一定要调用构造方法，而构造方法调用之后才会自动调用自定义初始化方法。如果想要调用自定义销毁方法，则必须明确调用销毁处理方法，而这个方法被定义在ClassPathXmlApplicationContext（AbstractApplicationContext父类）类中。

范例：手工处理销毁

ClassPathXmlApplicationContext context = **new** ClassPathXmlApplicationContext(**"spring-config.xml"**);  
Message message = context.getBean(**"message"**, Message.**class**);  
context.registerShutdownHook();

这种处理机制给了用户更方便的生命周期控制方法。

### 2.7 自动装配

自动装配指的是在SPring之中可以自动的来决定依赖对象的注入操作，避免了手工操作的麻烦之处。对于自动装配的操作有几种常用的模式：byType、byName。

范例：定义出要使用的关系类

**public class** Emp {  
 **private** Integer **emo**;  
  
 **private** String **ename**;  
  
 **private** Dept **dept**;

**public void** setEmo(Integer emo) {  
 **this**.**emo** = emo;  
 }  
  
 **public void** setEname(String ename) {  
 **this**.**ename** = ename;  
 }  
  
 **public void** setDept(Dept dept) {  
 **this**.**dept** = dept;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "Emp{"** +  
 **"emo="** + **emo** +  
 **", ename='"** + **ename** + **'\''** +  
 **", dept="** + **dept** +  
 **'}'**;  
 }  
}

**public class** Dept {  
 **private** Integer **deptNo**;  
 **private** String **dname**;  
 **private** String **loc**;  
 **private boolean flag**;  
  
 **public void** setDeptNo(Integer deptNo) {  
 **this**.**deptNo** = deptNo;  
 }  
  
 **public void** setDname(String dname) {  
 **this**.**dname** = dname;  
 }  
  
 **public void** setLoc(String loc) {  
 **this**.**loc** = loc;  
 }  
  
 **public void** setFlag(**boolean** flag) {  
 **this**.**flag** = flag;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "Dept{"** +  
 **"deptNo="** + **deptNo** +  
 **", dname='"** + **dname** + **'\''** +  
 **", loc='"** + **loc** + **'\''** +  
 **", flag="** + **flag** +  
 **'}'**;  
 }  
}

修改Spring-config.xml文件，使用byType方式自动装配：

<**bean id="ds" class="pojo.Dept"**>  
 <**property name="deptNo" value="10"**></**property**>  
</**bean**>  
<**bean id="emp" class="pojo.Emp" autowire="byType"**>  
 <**property name="ename" value="杨幂"**></**property**>  
 <**property name="emo" value="6379"**></**property**>  
</**bean**>

此时id和名称属性不一致一样能注入（ds和属性名称dept不相同）。也可以进行名称匹配，此时就要求属性名称与定义的bean名称完成匹配。

范例：根据名称自动匹配（id=”dept”和属性名称dept相同）

<**bean id="dept" class="pojo.Dept"**>  
 <**property name="deptNo" value="10"**></**property**>  
</**bean**>  
<**bean id="emp" class="pojo.Emp" autowire="byName"**>  
 <**property name="ename" value="杨幂"**></**property**>  
 <**property name="emo" value="6379"**></**property**>  
</**bean**>

如果一个类需要有很多用用关系，按照此种方式配置，会节约不少的“property”属性定义。但问题是同类型很多，同名很多，此时这个时候可以进行主选与候选的配置。

范例：配置主选（ **primary="true"**）

<**bean id="dept" class="pojo.Dept" primary="true"**>  
 <**property name="deptNo" value="10"**></**property**>  
</**bean**>  
<**bean id="depta" class="pojo.Dept"**>  
 <**property name="deptNo" value="20"**></**property**>  
</**bean**>  
<**bean id="emp" class="pojo.Emp" autowire="byType"**>  
 <**property name="ename" value="杨幂"**></**property**>  
 <**property name="emo" value="6379"**></**property**>  
</**bean**>

范例：取消主选（autowire-candidate="false"）

<**bean id="dept" class="pojo.Dept"**>  
 <**property name="deptNo" value="10"**></**property**>  
</**bean**>  
<**bean id="depta" class="pojo.Dept" autowire-candidate="false"**>  
 <**property name="deptNo" value="20"**></**property**>  
</**bean**>  
<**bean id="emp" class="pojo.Emp" autowire="byType"**>  
 <**property name="ename" value="杨幂"**></**property**>  
 <**property name="emo" value="6379"**></**property**>  
</**bean**>

**autowire-candidate="false" 表示从自动装配的候选列表中移除候选。**

### 2.8 基于Annotation的注入（核心）

在spring-config.xml文件中追加context命名空间，支持annotation配置。

*<?***xml version="1.0" encoding="UTF-8"***?>*<**beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xmlns:p="http://www.springframework.org/schema/p"  
 xmlns:context="http://www.springframework.org/schema/context"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  
 http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.3.xsd"**>  
  
   
</**beans**>

除了添加以上命名空间外，还需要在配置文件中启用annotation，并且配置annotation的扫描包路径。

*<!--表示启用annotation配置-->* <**context:annotation-config**/>  
 *<!--定义扫描包，表示在此包中定义的所有程序类支持annotation配置-->* <**context:component-scan base-package="hello"**></**context:component-scan**>  
 *<!--也可以逐一定义扫描包-->  
<!--  
 <context:component-scan base-package="hello,pojo"></context:component-scan>  
-->*

虽然那个配置了annotation扫描包配置，但是想要正确的进行扫描处理，还需要使用三个annotation（这三个名字不一样，但本质一样）。

·如果是定义的Dao子类，建议使用@Component；

·如果是定义的是Service子类建议使用@Service；

·如果是定义的是Action类建议使用@Repository;

对象的引用有两种方式：

方式一：

**private** IDeptDao **deptDao**;  
  
*//此时表示要在整个Spring中，自动查找名称为deptDaoImpl的子类对象实例进行引用传递*@Resource(name=**"deptDaoImpl"**)  
**public void** setDeptDao(IDeptDao deptDao) {  
 **this**.**deptDao** = deptDao;  
}

方式二：

*//表示自动根据类型引用*@Resource  
**private** IDeptDao **deptDao**;

那么此时会自动找到类型为IDeptDao子类定义的对象进行引用。

## 资源访问

在Spring 中提供了一组新的资源处理类：org.springframework.core.io.Resource,此接口提供了一组与具体的输入来源无关的信息读取处理。同时也定义有一序列信息读取方法。

|  |  |  |  |
| --- | --- | --- | --- |
| No. | 方法名称 | 类型 | 描述 |
| 1 | public InputStream getInputStream() throws IOException; | 普通 | 取得输入流对象，InputStreamResource接口继承 |
| 2 | public boolean exists(); | 普通 | 资源是否存在 |
| 3 | public long contentLength() throws IOException; | 普通 | 资源大小 |
| 4 | public File getFile() throws IOException; | 普通 | 取得读取的资源文件 |
| 5 | public long lastModified() throws IOException; | 普通 | 取得最后一次修改日期 |

在以后的编写过程中，根本就不会看见Resource类，Resource类本身是个接口，如果想使用接口类，则需要一些特定的子类：内存资源（ByteArrayResource）、文件资源（FileSystemResource）、CLASSPATH资源（ClassPathResource）、网络资源（UrlResource）、Web容器资源（ServletContextResource）。

#### 2.1 读取不同资源

范例：内存资源

public class ByteArrayResourceDemo {

public static void main(String[] args) throws Exception {

// 设置了内存的输入资源

Resource resource = new ByteArrayResource("Hello World".getBytes());

if (resource.exists()) { // 现在资源存在

// 如果要想进行资源的读取，肯定要通过Resource父接口InputStreamResource完成

Scanner scan = new Scanner(resource.getInputStream());

scan.useDelimiter("\n") ;

while (scan.hasNext()) {

System.out.println(scan.next());

}

scan.close();

}

}

}

范例：文件读取

File file = **new** File(**"E:"**+File.***separator***+**"msg.txt"**);  
Resource resource = **new** FileSystemResource(file);  
**if**(resource.exists()){  
 **try** {  
 Scanner scanner = **new** Scanner(resource.getInputStream());  
 **while**(scanner.hasNext()){  
 System.***out***.println(scanner.next());  
 }  
 scanner.close();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
}

有了Resource接口设计之后对于资源的读取操作就相当于有了一组规范。

范例：读取CLASSPATH下的资源

Resource resource = **new** ClassPathResource(**"spring-config.xml"**);  
**if**(resource.exists()){  
 **try** {  
 Scanner scanner = **new** Scanner(resource.getInputStream());  
 **while**(scanner.hasNext()){  
 System.***out***.println(scanner.next());  
 }  
 scanner.close();  
 } **catch** (IOException e) {  
  
 }

上面的方法只适合读取CLASSPATH路径下的内容，如果有包则需要加上包名称，

### 2.2 ResourceLoader接口

在spring中可以使用ResourceLoader接口来进行Resource对象的取得。此接口定义如下：

*/\*\*  
 \* 根据指定的字符串对象来进行资源的读取  
 \** ***@param location*** *资源定位信息  
 \** ***@return*** *Resource接口实例  
 \*/***public** Resource getResource(java.lang.String location);  
  
**public**.ClassLoade getClassLoader();

可以发现ResourceLoader接口中的getResource()方法，可以以字符串的形式来定位资源，但是这个字符串是有格式要求的：

·文件资源读取：“file:路径”;

·网路资源读取：“http://路径”；

·CLASSPATH读取：“classpath:路径”;

想要验证ResourceLoader接口类也比较麻烦，要使用其子类“DefaultResourceLoader”来获取实例化对象。

范例：读取资源

ResourceLoader loader = **new** DefaultResourceLoader();  
*// Resource resource = loader.getResource("file:E:" + File.separator + "msg.txt");  
// Resource resource = loader.getResource("http://www.baidu.com");* Resource resource = loader.getResource(**"classpath:spring-config.xml"**);  
 **if** (resource.exists()) {  
 **try** {  
 Scanner scanner = **new** Scanner(resource.getInputStream());  
 **while** (scanner.hasNext()) {  
 System.***out***.println(scanner.next());  
 }  
 scanner.close();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }

### 2.3 注入Resource

在实际开发中，用户可能根本不会关注Resource、ResourceLoader，尤其是ResourceLoader类除了用于标记一个字符串外真没有用处，所以在Spring的开发过程中，可以直接通过字符串以配置文件的方式进行资源路径的注入。

范例：定义一个专门负责资源注入与读取的类

**public class** MyResource {  
 **private** Resource **resource**;  
  
 *//注入的处理操作使用此方法完成* **public void** setResource(Resource resource) {  
 **this**.**resource** = resource;  
 }  
  
 **public void** print() **throws** Exception {  
 **if** (**this**.**resource**.exists()) {  
 *// 如果要想进行资源的读取，肯定要通过Resource父接口InputStreamResource完成* Scanner scanner = **new** Scanner(**this**.**resource**.getInputStream());  
 scanner.useDelimiter(**"\n"**);  
 **while** (scanner.hasNext()) {  
 System.***out***.println(scanner.next());  
 }  
 scanner.close();  
 }  
 }  
}

范例：在spring-config.xml文件中进行配置

·设置文件资源：

<bean id="myres" class="util.MyResource">

<property name="resource" value="file:E:\\msg.txt"></property>

</bean>

·读取CLASSPATH资源：

<bean id="myres" class="util.MyResource">

<property name="resource" value="classpath:spring-config.xml"></property>

</bean>

在以后的开发中如果进行一些配置文件的读取，那么就可以使用Resource接口及Spring的注入转换完成。

### 2.4 路径通配符

如果现在要读取的资源，可能很多的文件夹都存在，那么怎么进行这些内容的读取呢？Spring继承了Ant编译工具中提出的通配符过年概念，支持以下三种通配符：

·“？”：匹配任意的一位字符，如果现在使用了“spring-config?.xml”，那么可以表示“spring-config1.xml”或者是“spring-config2.xml”文件。

·“\*”：匹配任意的多位字符，现在假设cn.xxx下存在有spring-config,cn.xxx.demo下存在spring-config,cn.xxx.util下存在有spring-config,那么访问的时候只需要写上“cn/xxx/\*/spring-config.xml”,那么就表示所有的“cn.xxx”单层级下的内容都可以找到。

·“\*\*”：匹配任意级别的多位字符，如果编写为“cn/xxx/\*\*/spring-config.xml”,那么就表示在“cn.xxx”包中的所有子包里面包含有spring-config.xml文件的内容都进行访问。

如果要进行通配符资源路径的读取，那么需要使用一个特殊的类，这个类需要满足于正则匹配要求：

“org.springframework.core.io.support.ResourcePatternResolver”,这是一个接口，需要使用其子类“PathMatchingResourcePatternResolver”，取得接口实例化对象，取得之后就可以使用“ResourcePatternResolver”接口中定义的资源取得方法：

·取得一组资源：Resource[] getResources(java.lang.String locationPattern)，因为存在有通配符的问题，所以此时返回的是资源数组。

范例：读取classpath资源

ResourcePatternResolver loader = **new** PathMatchingResourcePatternResolver();  
*// Resource resource = loader.getResource("file:E:" + File.separator + "msg.txt");  
// Resource resource = loader.getResource("http://www.baidu.com");  
 //读取单个资源  
 //Resource resource = loader.getResource("classpath:spring-config.xml");  
 //读取一组资源* Resource[] resources = loader.getResources(**"classpath:spring/\*\*/spring-config.xml"**);  
 **for**(Resource resource : resources){  
 **if** (resource.exists()) {  
 **try** {  
 Scanner scanner = **new** Scanner(resource.getInputStream());  
 **while** (scanner.hasNext()) {  
 System.***out***.println(scanner.next());  
 }  
 scanner.close();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 }

### 2.5 注入多个Resource路径（数组）

范例：定义一个可以注入数组的类

**public class** MyResourceArray {  
 **private** Resource[] **resource**;  
  
 *//注入的处理操作使用此方法完成* **public void** setResource(Resource[] resource) {  
 **this**.**resource** = resource;  
 }  
  
 **public void** print() **throws** Exception {  
 **for** (**int** x = 0; x < **this**.**resource**.**length**; x++) {  
 **if** (**this**.**resource**[x].exists()) {  
 *// 如果要想进行资源的读取，肯定要通过Resource父接口InputStreamResource完成* Scanner scanner = **new** Scanner(**this**.**resource**[x].getInputStream());  
 scanner.useDelimiter(**"\n"**);  
 **while** (scanner.hasNext()) {  
 System.***out***.println(scanner.next());  
 }  
 scanner.close();  
 }  
 }  
 }  
}

范例：传入路径

<**bean id="rarr" class="util.MyResourceArray"**>  
 <**property name="resource"**>  
 <**array**>  
 <**value**>classpath:META-INF/license.txt</**value**>  
 <**value**>classpath:META-INF/notice.txt</**value**>  
 </**array**>  
 </**property**>  
</**bean**>

### 三、任务调度（定时任务）

如果只是纯粹的使用java开发，能够想到的调用只能Timer、TimerTask类，但实际使用中这两个类很难控制，例如很多开发里面要求每天12点进行，就必须设置好开始点，而后还要自己计算出间隔，再然后还需要排列不同任务的出现时间。

实际开发中药开发定时任务，有两个选择：

·quartz：企业级的定时调度组件，需要单独下载开发包；

·Spring Task:轻量级组件，配置简单，可以使用annotation配置;

以上两种组件可以有两种调度方式：

·按照指定的时间间隔进行调度，这点和Timer、TimerTask是一样的。

·可以在指定的时间到达之后自动触发。

#### 3.1 Quartz实现定时调度

添加jar包：

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.0.5.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>5.0.5.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-tx</artifactId>

<version>5.0.5.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-support</artifactId>

<version>5.0.5.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

<version>5.0.5.RELEASE</version>

</dependency>

<dependency>

<groupId>org.quartz-scheduler</groupId>

<artifactId>quartz</artifactId>

<version>2.3.0</version>

</dependency>

<dependency>

<groupId>org.quartz-scheduler</groupId>

<artifactId>quartz-jobs</artifactId>

<version>2.2.1</version>

</dependency>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.11</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-simple</artifactId>

<version>1.7.25</version>

</dependency>

如果想使用此类模式完成定时调度，那么实际上有两种使用方式：

·要求明确的去继承一个父类；

·利用配置实现，不再强制要求去继承父类。

对于Quartz操作已经在Spring中进行了继承配置。

3.1.1 使用继承父类的方式实现定时调度

所有的任务类要求继承“org.springframework.scheduling.quartz.QuartzJobBean”

1. 定义任务的执行类

public class MyTask extends QuartzJobBean {

@Override

protected void executeInternal(JobExecutionContext jobExecutionContext) throws JobExecutionException {

System.out.println("当前日期时间：" );

}

1. 此时任务类是一个完全独立的任务类，而且Spring也并不知道其存在，那么就需要在spring-config.xml文件里面配置此任务类的信息。

<bean id="myTask" class="org.springframework.scheduling.quartz.JobDetailFactoryBean">

<!--此为任务执行之后任务处理执行类，默认调用executeInternal()方法-->

<property name="jobClass" value="task.MyTask"></property>

<property name="jobDataAsMap">

<map>

<!-- 容器启动之后立即触发 -->

<entry key="timeout" value="0"></entry>

</map>

</property>

</bean>

1. 设置任务的触发作业，对于触发作业有两类：

·设置间隔触发，例如xxx时间之后重复执行；

<!-- 此处配置的是一个间隔触发，既然是间隔触发就必须设置要触发时的任务 -->

<bean id="simpleTrigger" class="org.springframework.scheduling.quartz.SimpleTriggerFactoryBean">

<!--需要执行的任务-->

<property name="jobDetail" ref="myTask"></property>

<!--立即执行-->

<property name="startDelay" value="0"></property>

<!--每两秒执行一次-->

<property name="repeatInterval" value="2000"></property>

</bean>

·利用cron设置触发时间;

<!-- 设置一个定时触发 -->

<bean id="cronTrigger" class="org.springframework.scheduling.quartz.CronTriggerFactoryBean">

<property name="jobDetail" ref="myTask"></property>

<!--每分钟执行一次-->

<property name="cronExpression" value="0 \* \* \* \* ?"></property>

</bean>

1. 定义调度工厂，调度工厂就相当于在容器启动的时候可以实现任务的触发操作，所以此时必须设置好与之匹配的触发器。

·设置间隔触发器：

<!--调度器-->

<bean id="scheduler" class="org.springframework.scheduling.quartz.SchedulerFactoryBean">

<property name="triggers">

<list>

<ref bean="simpleTrigger"/>

</list>

</property>

</bean>

·设置定时触发器：

<bean id="cronTrigger" class="org.springframework.scheduling.quartz.CronTriggerFactoryBean" lazy-init="false">

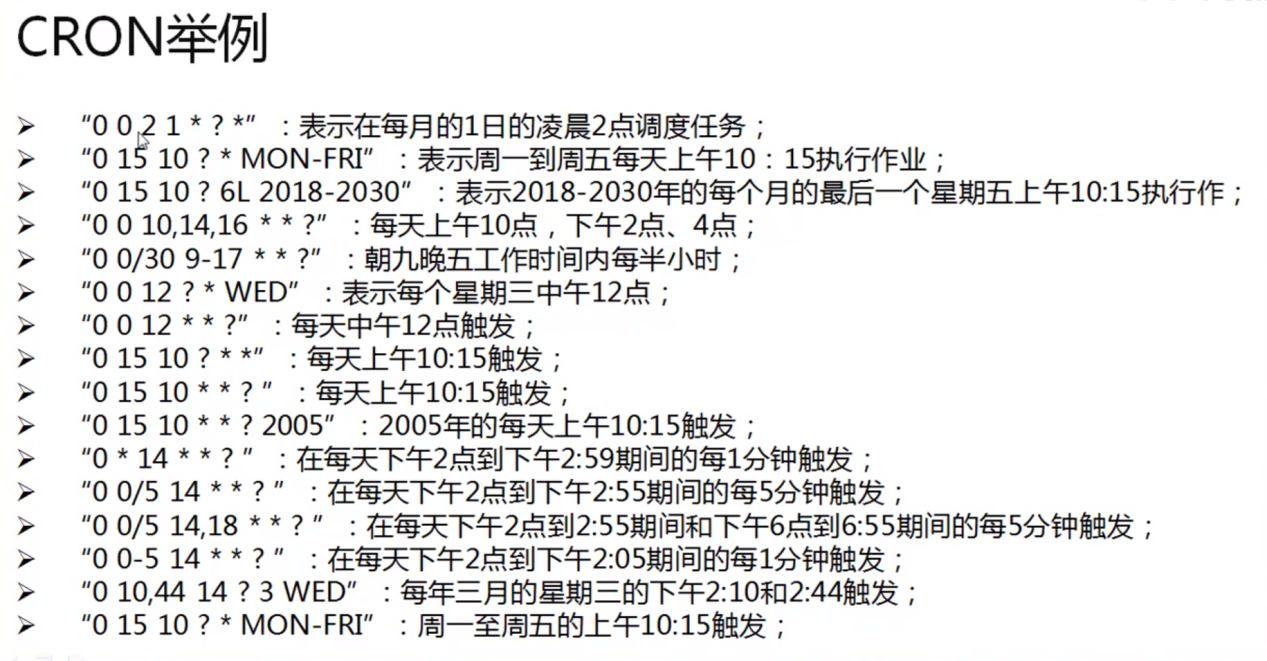
<property name="jobDetail" ref="myTask"/>

<property name="cronExpression" value="0 \* \* \* \* ?"/><!--每分钟执行一次-->

</bean>







只要是强制性的继承就会存在单继承的局限性，所以在Quartz实现方式里面针对此类的形式又给出了不需要强制性继承处理的模式，此时需要配置实现了。

3.1.2 不继承父类实现定时调度

可参考Spring官方给出的文档:

spring-framework-4.3.16.RELEASE/docs/spring-framework-reference/html/scheduling.html

1. 定义一个任务处理类：

public class ExampleBusinessObject {

// properties and collaborators

public void doIt() {

// do the actual work

}

}

<bean id="exampleBusinessObject" class="examples.ExampleBusinessObject"/>

1. 那么所有的配置需要在spring-config.xml文件中负责完成。

定义job:

<bean id="jobDetail" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">

<property name="targetObject" ref="exampleBusinessObject"/>

<property name="targetMethod" value="doIt"/>

<property name="concurrent" value="false"/>

</bean>

其中concurrent属性标识的解释为：对于相同的JobDetail，当指定多个Trigger时, 很可能第一个job完成之前，第二个job就开始了。指定concurrent设为false，多个job不会并发运行，第二个job将不会在第一个job完成之前开始。

定义触发器：

<bean id="cronTrigger" class="org.springframework.scheduling.quartz.CronTriggerFactoryBean">

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

<!-- run every morning at 6 AM -->

<property name="cronExpression" value="0 \* \* \* \* ?"/>

</bean>

定义调度程序：

<bean class="org.springframework.scheduling.quartz.SchedulerFactoryBean">

<property name="triggers">

<list>

<ref bean="cronTrigger"/>

<ref bean="cronTrigger"/>

</list>

</property>

</bean>

### 3.2 Spring Task 任务调度

从Spring3.0开始在Spring 中提供了“Spring Task”工具配置，并且不需要调入其他任何第三方组件包。

#### 3.2.1 基于配置文件定义

要进行任务调度，那么一定要有一个任务的执行类，定义任务执行类。

范例：定义一个任意类，并 将其交给Spring管理

public class MyTask {

public void runJob() {

System.out.println("\*\*\* 【当前日期时间】"

+ new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS")

.format(new java.util.Date()));

}

}

<bean id="myTask" class="task.MyTask"/>

如果想让任务起作用，那么必须在是spring-config.xml文件中引入task的命名空间，而后进行配置。

*<?***xml version="1.0" encoding="UTF-8"***?>*<**beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:task="http://www.springframework.org/schema/task"  
 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-4.3.xsd  
 http://www.springframework.org/schema/task http://www.springframework.org/schema/task/spring-task-4.3.xsd"**>  
</**beans**>

然后在配置文件中定义作业(间隔触发)：

<**task:scheduled-tasks**>  
 <**task:scheduled ref="myTask" method="runJob" fixed-delay="5000"**/>*<!--间隔5秒-->  
 <!--此处可以定义多个任务-->*</**task:scheduled-tasks**>

也可以配置corn触发：

<**task:scheduled-tasks**>  
 <**task:scheduled ref="myTask" method="runJob" cron="0 \* \* \* \* ?"**/>*<!--间隔1分钟-->  
 <!--此处可以定义多个任务-->*</**task:scheduled-tasks**>

#### 3.2.2 基于annotation的配置

范例：修改spring配置文件

*<?***xml version="1.0" encoding="UTF-8"***?>*<**beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xmlns:p="http://www.springframework.org/schema/p"  
 xmlns:task="http://www.springframework.org/schema/task"  
 xmlns:context="http://www.springframework.org/schema/context"  
 xsi:schemaLocation="http://www.springframework.org/schema/task   
 http://www.springframework.org/schema/task/spring-task-4.3.xsd  
 http://www.springframework.org/schema/beans   
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd  
 http://www.springframework.org/schema/context   
 http://www.springframework.org/schema/context/spring-context-4.3.xsd"**>  
 *<!-- 表示现在启用Annotation配置，配置的只能够是任务 -->* <**task:annotation-driven**/>  
 <**context:annotation-config**/>  
   
 <**context:component-scan base-package="task"**/>  
</**beans**>

随后修改程序代码：

·配置间隔触发

@Component

public class MyTask {

//间隔2秒执行

@Scheduled(fixedDelay = 2000)

public void runJob() {

System.out.println("\*\*\* 【我爱杨幂！】"

+ new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS")

.format(new java.util.Date()));

}

}

·配置corn触发

@Component

public class MyTask {

//每分钟执行一次

@Scheduled(cron = "0 \* \* \* \* ?")

public void runJob() {

System.out.println("\*\*\* 【我爱杨幂！】"

+ new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS")

.format(new java.util.Date()));

}

}

#### 3.2.3 任务调度池

所谓的任务就是属于一个线程，如果定义了若干个任务，那么这些线程每次只能执行一个。

范例：在已有项目中新增加一个任务

@Component

public class ExampleBusinessObject {

@Scheduled(fixedDelay = 2000)

public void doIt() {

System.out.println("我爱杨幂！");

}

}

此时，如果有个任务，需要执行很长时间，这个过程中其他任务就需要等待，因为默认只允许一个任务线程执行。所以在这种情况下，可以配置一个线程池，利用线程池可以同时开启多个任务线程。

范例：在spring=config.xml中配置线程池

<!-- 表示现在启用Annotation配置，配置的只能够是任务 -->

<task:annotation-driven scheduler="myScheduler" />

<!-- 告诉容器可以启动20个任务的线程 -->

<task:scheduler id="myScheduler" pool-size="20"/>

<!--<task:scheduled-tasks scheduler="myScheduler">

<task:scheduled ref="beanA" method="methodA" fixed-delay="5000"/>

</task:scheduled-tasks>-->

此时就表示多个任务间的延时不会相互影响，

## 四、Spring表达式语言

SpEL(Spring Expression Language,Spring 表达式语言)可以实现更加丰富的字符串操作，可以实现字符串定义方法调用、字符串实例化对象，但是代码编写非常的复杂。

### 表达式的编程入门

表达式执行类：“org.springframework.expression.Expression”;

|- 通过此类可以最终得到结果

|- 利用的是SpEL的标准表达式处理子类：org.springframework.expression.spel.standard.SpelExpressionParser;

表达式解析器：org.springframework.expression.spel.standard.SpelExpressionParser；

|- 主要是负责为给定的表达式进行内容的解析操作处理。

表达式的操作属性：org.springframework.expression.EvaluationContext

|- 例如说表达式所需要的变量。

也就是说在表达式的字符串里面允许设置一些替代的内容，供用户动态设置使用。

public class SpELDemo03 {

public static void main(String[] args) {

// 现在是将一个整个的语句直接定义成了字符串，其中对于字符串截取的开始索引与结束索引使用替代变量

String str = "(\"Hello \" + \"World !!!\").substring(#start,#end)" ;

// 1、定义一个专属的表达式的解析工具

ExpressionParser parser = new SpelExpressionParser() ; // 定义一个Spring表达式解析器

// 2、定义一个表达式的处理类

Expression exp = parser.parseExpression(str) ; // 从字符串里面解析出内容

// 3、进行最终表达式的计算中进行替代变量内容的设置

EvaluationContext context = new StandardEvaluationContext() ;

context.setVariable("start", 6);

context.setVariable("end", 9);

// 4、通过表达式进行结果的计算

System.out.println(exp.getValue(context));

}

}

public class SpELDemo02 {

public static void main(String[] args) {

String str = "(\"Hello\"+\"World\").substring(6,8)";

//定义一个专属的表达式解析工具

ExpressionParser expressionParser = new SpelExpressionParser();

//定义一个表达式处理类

Expression expression = expressionParser.parseExpression(str);

//进行最终的表达式计算

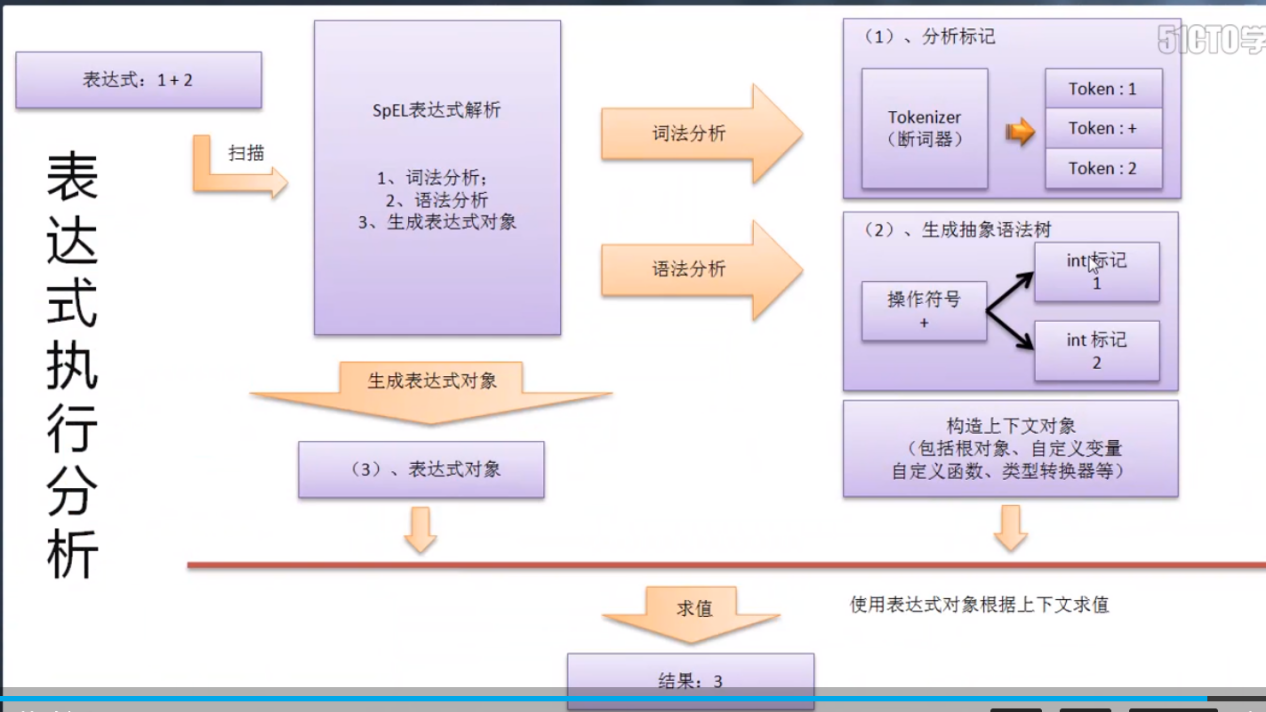
EvaluationContext context = new StandardEvaluationContext();

System.out.println(expression.getValue(expression));

}

}

### 表达式原理



### 自定义分隔符

在JSP中发现所有的表达式语言里面，都存在有一个范围标记，使用”${}”完成，在SpEL中如果用户有需要，也可以自己实现分隔符定义。

分隔符是有两个组成部分的，以JSP中的“EL”为例，分为“${”、“}”两个部分。如果要实现自定义，可以采用“org.springframework.expression.ExpressionParser”实际上定义有另外一个解析表达式的方法，

·使用特定的表达式解析处理：public Expression parseExpression(String var1, ParserContext var2) throws ParseException;

org.springframework.expression.ParserContext就是用于边界匹配的，其有如下方法：

·表示是否使用此模板：public boolean isTemplate();

·边界开始符号：public String getExpressionPrefix();

·边界结束符号：public String getExpressionSuffix();

范例：使用自定义边界

**public class** SpELDemo02 {  
 **public static void** main(String[] args) {  
 String str = **"#[1+2]"**;  
 *//定义一个专属的表达式解析工具* ExpressionParser expressionParser = **new** SpelExpressionParser();  
 *//定义一个表达式处理类* Expression expression = expressionParser.parseExpression(str, **new** ParserContext() {  
 @Override  
 **public boolean** isTemplate() {  
 **return true**;  
 }  
  
 @Override  
 **public** String getExpressionPrefix() {  
 **return "#["**;  
 }  
  
 @Override  
 **public** String getExpressionSuffix() {  
 **return "]"**;  
 }  
 });  
 *//进行最终的表达式计算* EvaluationContext context = **new** StandardEvaluationContext();  
 System.***out***.println(expression.getValue(expression));  
 }  
}

由于此时已经明确的定义出了要使用的边界，在解析的时候会自动剔除掉边界符号，然后在进行正常的表达式解析处理。

### 基本表达式

#### 字面表达式

|  |  |  |  |
| --- | --- | --- | --- |
| No. | 类型 | 表达式 | 执行结果 |
| 1 | 字符串 | "'杨幂'+'我爱你'";  "\"杨幂\"+\"我爱你\"" | 杨幂我爱你 |
| 2 | 整型 | "1"; | 1 |
| 3 | 长整型 | “1E10” | E表示10的几次方，如果是1E20表示，1x1020 |
| 4 | 布尔型 | “true” | 布尔型的内容 |
| 5 | Null类型 | “null” | null |

#### 算术表达式

|  |  |  |  |
| --- | --- | --- | --- |
| No. | 运算类型 | 表达式 | 执行结果 |
| 1 | 加 | “1+1” | 2 |
| “’Hello’+’ world’” | Hello World |
| 2 | 四则运算 | "1+2\*3-4/5" | 现在是整数，则小数点被抹去，结果是7 |
| "1+2.0\*3-4.0/5" | 现在是小数数，则int变为小数，结果是6.2 |
| 3 | 求模 | "10%3" | 1 |
| "10 MOD 3" | 1 |
| 4 | 幂运算 | "2^3" | 2\*2\*2=8 |
| 5 | 除法 | “10 DIV 2” "10 div 2" | 5 |

#### 关系表达式

|  |  |  |  |
| --- | --- | --- | --- |
| No. | 运算符 | 表达式 | 执行结果 |
| 1 | 等于 | “10 == 10” | true |
| “10 EQ 10” |
| 2 | 不等于 | "10 != 10" | false |
| "10 NE 10" |
| 3 | 区间范围 | 3 between {1,5}  3 BETWEEN {1,5} | true |

在进行关系判断的时候，可以使用“==”（EQ）、“！=”（NE）、“>”（GT）、“<”（LT）、“>=”（GE）、“<=”(LE)、区间范围（between）。

#### 逻辑表达式

逻辑表达式就是与（and/AND）、或（or/OR）、非（！、not/NOT）。只要有关系运算那么就一定要有逻辑运算与之匹配。

#### 三目运算

|  |  |  |  |
| --- | --- | --- | --- |
| No. | 类型 | 表达式 | 执行结果 |
| 1 | 三目运算 | "10 >2 ? 'Hello' : 'World'" | Hello |
| 2 | Null运算 | "null == null ? 'hello' : 'world'" | hello |
| 3 | null?:false | false |
| 4 | 布尔运算 | true ? :false | true |

|  |  |  |  |
| --- | --- | --- | --- |
| No. | 类型 | 表达式 | 执行结果 |
| 1 | 正则 | "'123' matches '\\d{3}'"; | true |

#### 正则表达式

### Class类型表达式

Class类型的表达式主要是实现放射的操作机制处理的。

|  |  |  |  |
| --- | --- | --- | --- |
| No. | 类型 | 表达式 | 执行结果 |
| 1 | Class | "T(String)" | class java.lang.String |
| 2 | 静态属性 | "T(Integer).MAX\_VALUE" | 2147483647 |
| 3 | 静态方法 | "T(Integer).parseInt('123')" | 123 |
| 4 | 实例化对象 | "new java.util.Date()" | Tue May 01 17:40:10 CST 2018 |
| 5 | Instanceof | "'hello' instanceof T(String)" | true |

范例：描述一个Class<String> 类的对象

String str = **"T(String)"**;  
*//定义一个专属的表达式解析工具*ExpressionParser expressionParser = **new** SpelExpressionParser();  
*//定义一个表达式处理类*Expression expression = expressionParser.parseExpression(str);  
*//进行最终的表达式计算*EvaluationContext context = **new** StandardEvaluationContext();  
Object o = expression.getValue(expression, Object.**class**);  
System.***out***.println(o);

范例：取得Integer对象中定义的MAX\_VALUE属性内容

*//定义一个专属的表达式解析工具*ExpressionParser expressionParser = **new** SpelExpressionParser();  
*//定义一个表达式处理类*Expression expression = expressionParser.parseExpression(str);  
*//进行最终的表达式计算*EvaluationContext context = **new** StandardEvaluationContext();  
Object o = expression.getValue(expression, Object.**class**);  
System.***out***.println(o);

范例：调用Integer.parseInt()方法

String str = **"T(Integer).parseInt('123')"**;  
*//定义一个专属的表达式解析工具*ExpressionParser expressionParser = **new** SpelExpressionParser();  
*//定义一个表达式处理类*Expression expression = expressionParser.parseExpression(str);  
*//进行最终的表达式计算*EvaluationContext context = **new** StandardEvaluationContext();  
Object o = expression.getValue(expression, Object.**class**);  
System.***out***.println(o);

### 变量操作

在表达式操作处理过程中，可以利用变量来进行内容的动态设置，而这一切主要依靠EvaluationContext类来完成。

范例：观察变量定义

// 1、定义一个专属的表达式的解析工具

ExpressionParser parser = new SpelExpressionParser() ; // 定义一个Spring表达式解析器

// 2、定义一个表达式的处理类

Expression exp = parser.parseExpression(str) ; // 从字符串里面解析出内容

// 3、进行最终表达式的计算中进行替代变量内容的设置

EvaluationContext context = new StandardEvaluationContext() ;

// 4、通过表达式进行结果的计算

Object num = exp.getValue(context,Object.class) ;

System.out.println(num);

当设置完变量之后，如果没有定义变量的具体内容，则最终的结果将返回null.

在操作的时候如果使用了EvaluationContext，实际上里面会留有一个根变量的概念，其实例化的子类StandardEvaluationContext中提供有一个方法：

**public** StandardEvaluationContext(Object **rootObject**) {  
 **this**.rootObject = **new** TypedValue(rootObject);  
}

范例：观察一下根变量

String str = **"#root"** ;  
*// 1、定义一个专属的表达式的解析工具*ExpressionParser parser = **new** SpelExpressionParser() ; *// 定义一个Spring表达式解析器  
// 2、定义一个表达式的处理类*Expression exp = parser.parseExpression(str) ; *// 从字符串里面解析出内容  
// 3、进行最终表达式的计算中进行替代变量内容的设置*EvaluationContext context = **new** StandardEvaluationContext(**"a"**) ;  
*// 4、通过表达式进行结果的计算*Object num = exp.getValue(context,Object.**class**) ;  
System.***out***.println(num);

默认的根变量名称为“root”,如果要输出根变量的数据，就使用“#root”引用。

范例：利用根变量的内容进行三目运算

String str = "#root=='mldn' ? '好好学习' : '你完蛋了。'" ; // 表示的是直接使用root根变量

// 1、定义一个专属的表达式的解析工具

ExpressionParser parser = new SpelExpressionParser() ; // 定义一个Spring表达式解析器

// 2、定义一个表达式的处理类

Expression exp = parser.parseExpression(str) ; // 从字符串里面解析出内容

// 3、进行最终表达式的计算中进行替代变量内容的设置，要设置根变量的数据

EvaluationContext context = new StandardEvaluationContext("mldn") ;

// 4、通过表达式进行结果的计算

Object num = exp.getValue(context,Object.class) ;

System.out.println(num);

既然根变量可以进行内容的设置，那么就可以进一步扩展

范例：方法引用

·方法引用，在于为一个方法设置一个新的名字，虽然jdk1.8之后，增加有方法引用的概念，但早在Spring里面就已经为用户通过表达式得到方法引用的技术实现。

*//设置好要引用的方法*Method method = Integer.**class**.getMethod(**"parseInt"**, String.**class**);  
String str = **"#myInt('123')"**; *// 表示的是直接使用root根变量*ExpressionParser parser = **new** SpelExpressionParser(); *// 定义一个Spring表达式解析器*Expression exp = parser.parseExpression(str); *// 从字符串里面解析出内容  
// 现在直接为根变量进行内容的设置，但是最终执行表达式的时候没有使用到根变量，而是使用到了指定类的属性*EvaluationContext context = **new** StandardEvaluationContext();  
context.setVariable(**"myInt"**, method); *//实现方法引用*Object num = exp.getValue(context, Object.**class**);  
System.***out***.println(num);

在表达式变量设置的时候，除了可以具体设置数值内容外，也可以实现方法的引用设置。

除了对方法实现引用之外，还可以取得类中的属性进行控制。（实际开发中，属性是不可能被外部所看见的，所有的属性进行访问的时候调用的都是getter方法）。

范例：实现属性的调用

String str = **"time"**; *// 表示的是直接使用root根变量*ExpressionParser parser = **new** SpelExpressionParser(); *// 定义一个Spring表达式解析器*Expression exp = parser.parseExpression(str); *// 从字符串里面解析出内容  
// 现在直接为根变量进行内容的设置，但是最终执行表达式的时候没有使用到根变量，而是使用到了指定类的属性*EvaluationContext context = **new** StandardEvaluationContext(**new** Date());  
Object num = exp.getValue(context, Object.**class**);  
System.***out***.println(num);

调用属性，本质是是调用getter方法，属性的首字母要求大写，那么理论上，调用getter方法，key可以写成“Time”。

String str = **"#root.Time"**; *// 表示的是直接使用root根变量*ExpressionParser parser = **new** SpelExpressionParser(); *// 定义一个Spring表达式解析器*Expression exp = parser.parseExpression(str); *// 从字符串里面解析出内容  
// 现在直接为根变量进行内容的设置，但是最终执行表达式的时候没有使用到根变量，而是使用到了指定类的属性*EvaluationContext context = **new** StandardEvaluationContext(**new** Date());  
Object num = exp.getValue(context, Object.**class**);  
System.***out***.println(num);

默认情况下进行属性调用的时候，需要有一个对象，而且这个对象，传递给了“root”变量，那么如果不传递呢？

范例：观察错误代码

EvaluationContext context = new StandardEvaluationContext();

如果此时代码没有设置root的内容。一旦调用，则会提示没有发现Time的成员，因为传递的根变量对象内容是null。为了解决此类问题，在Spring中，引入了Google的Groovry表达式的安全运算符来避免空运算。

范例：使用安全运算符

String str = **"#root?.Time"**; *// 表示的是直接使用root根变量*ExpressionParser parser = **new** SpelExpressionParser(); *// 定义一个Spring表达式解析器*Expression exp = parser.parseExpression(str); *// 从字符串里面解析出内容  
// 现在直接为根变量进行内容的设置，但是最终执行表达式的时候没有使用到根变量，而是使用到了指定类的属性*EvaluationContext context = **new** StandardEvaluationContext();  
Object num = exp.getValue(context, Object.**class**);  
System.***out***.println(num);

此时这种方式的代码将更好的避免Null带来的问题。

在SpEL也可以轻松的进行配置文件的内容的使用。

范例定义一个Message类，并交给Spring管理

范例：在程序中引入配置变量

如果要想引入外部程序的bean的内容，那么就必须定义一个Bean解析工厂类，而这个类的处理不在EvaluationContext定义，只能够使用其子类StandardEvaluationContext完成。

ApplicationContext ctx = **new** ClassPathXmlApplicationContext(**"spring-config.xml"**);  
*// 表示引用配置文件中的bean的方法*String str = **"@msg.getInfo()"**;  
*//定义解析器*ExpressionParser expressionParser = **new** SpelExpressionParser();  
*//定义表达式处理类*Expression expression = expressionParser.parseExpression(str);  
*//进行替代内容设置*StandardEvaluationContext evaluationContext = **new** StandardEvaluationContext();  
*// 引用容器中的内容*evaluationContext.setBeanResolver(**new** BeanFactoryResolver(ctx));  
*//输出最终计算结果*System.***out***.println(expression.getValue(evaluationContext));

### 集合表达式

范例：处理List集合

List集合对于程序而言就是一组数组。表达式在处理程序的过程中使用的是Collections来实现集合内容的定义，此时创建的是不可修改内容的的List集合class java.util.Collections$UnmodifiableRandomAccessList

问:请解释List与数组的区别？

·List是一个动态对象数组，内容可变，数组是一个静态的集合，数组长度不可改变；

·List可以通过Collections创建不可改变的集合；

·如果确定数组长度的情况下，不建议使用LIst集合。

范例：索引访问集合

String str = **"{10,20,30}[1]"**;  
*//定义解析器*ExpressionParser expressionParser = **new** SpelExpressionParser();  
*//定义表达式处理类*Expression expression = expressionParser.parseExpression(str);  
*//进行替代内容设置*EvaluationContext evaluationContext = **new** StandardEvaluationContext();  
  
*//输出最终计算结果*System.***out***.println(expression.getValue(evaluationContext));

如果要设置可变集合，则必须采用变量的方式完成处理。

范例：设置可变的List集合

List<Integer> list = **new** ArrayList<>();  
list.add(10);  
list.add(20);  
list.add(30);  
String str = **"#allData[1]"**;  
*//定义解析器*ExpressionParser expressionParser = **new** SpelExpressionParser();  
*//定义表达式处理类*Expression expression = expressionParser.parseExpression(str);  
*//进行替代内容设置*EvaluationContext evaluationContext = **new** StandardEvaluationContext();  
evaluationContext.setVariable(**"allData"**, list);  
*//输出最终计算结果*System.***out***.println(expression.getValue(evaluationContext));

既然可以设置List集合，那么也就可以Set。

范例：设置Map集合

Map<String,String> map = **new** HashMap<>();  
map.put(**"ym"**,**"杨幂"**);  
map.put(**"bq"**,**"白浅"**);  
String str = **"#allData['bq']"**;  
*//定义解析器*ExpressionParser expressionParser = **new** SpelExpressionParser();  
*//定义表达式处理类*Expression expression = expressionParser.parseExpression(str);  
*//进行替代内容设置*EvaluationContext evaluationContext = **new** StandardEvaluationContext();  
evaluationContext.setVariable(**"allData"**, map);  
*//输出最终计算结果*System.***out***.println(expression.getValue(evaluationContext));

范例：修改List集合

List<Integer> list = **new** ArrayList<>();  
list.add(10);  
list.add(20);  
list.add(30);  
String str = **"#allData[1]=100"**;  
*//定义解析器*ExpressionParser expressionParser = **new** SpelExpressionParser();  
*//定义表达式处理类*Expression expression = expressionParser.parseExpression(str);  
*//进行替代内容设置*EvaluationContext evaluationContext = **new** StandardEvaluationContext();  
evaluationContext.setVariable(**"allData"**, list);  
*//输出最终计算结果*System.***out***.println(expression.getValue(evaluationContext));

范例：修改Map集合

Map<String,String> map = **new** HashMap<>();  
map.put(**"ym"**,**"杨幂"**);  
map.put(**"bq"**,**"白浅"**);  
String str = **"#allData['bq'] = '白浅，我爱你！'"**;  
*//定义解析器*ExpressionParser expressionParser = **new** SpelExpressionParser();  
*//定义表达式处理类*Expression expression = expressionParser.parseExpression(str);  
*//进行替代内容设置*EvaluationContext evaluationContext = **new** StandardEvaluationContext();  
evaluationContext.setVariable(**"allData"**, map);  
*//输出最终计算结果*System.***out***.println(expression.getValue(evaluationContext));

以上两种修改，并不能发挥出SpEL的特点，它的特点是可以实现自身的迭代修改。

范例：迭代修改List集合

List<Integer> all = **new** ArrayList<Integer>();  
all.add(10);  
all.add(20);  
String str = **"#allData.![1000 + #this]"**; *// 设置操作表达式，定义了一个变量*ExpressionParser parser = **new** SpelExpressionParser(); *// 定义一个Spring表达式解析器*Expression exp = parser.parseExpression(str); *// 从字符串里面解析出内容*EvaluationContext context = **new** StandardEvaluationContext();  
context.setVariable(**"allData"**, all); *// 设置可变集合*Object obj = exp.getValue(context, Object.**class**); *// 返回的是修改后的数据*System.***out***.println(obj);  
System.***out***.println(all);

在集合修改的时候，不是在原来的基础上变更的，而是创建一个集合，并将变更后的内容存到新集合中。

范例：将Map集合中的key和value组合成新的List集合

Map<String, String> all = **new** HashMap<>();  
all.put(**"ym"**, **"杨幂"**);  
all.put(**"bq"**, **"白浅"**);  
String s = **"#allData.![#this.key+':'+#this.value]"**;  
ExpressionParser parser = **new** SpelExpressionParser(); *// 定义一个Spring表达式解析器*Expression exp = parser.parseExpression(s); *// 从字符串里面解析出内容*EvaluationContext context = **new** StandardEvaluationContext();  
context.setVariable(**"allData"**, all); *// 设置可变集合*Object obj = exp.getValue(context, Object.**class**);  
System.***out***.println(obj);  
System.***out***.println(all);

范例：过滤数据

Map<String, String> all = **new** HashMap<>();  
all.put(**"ym"**, **"杨幂"**);  
all.put(**"bq"**, **"白浅"**);  
String s = **"#allData.?[#this.key.contains('bq')]"**;  
ExpressionParser parser = **new** SpelExpressionParser(); *// 定义一个Spring表达式解析器*Expression exp = parser.parseExpression(s); *// 从字符串里面解析出内容*EvaluationContext context = **new** StandardEvaluationContext();  
context.setVariable(**"allData"**, all); *// 设置可变集合*Object obj = exp.getValue(context, Object.**class**);  
System.***out***.println(obj);  
System.***out***.println(all);

### 实际使用：配置文件的操作

1. 修改spring-config.xml文件

<**bean id ="str" class="java.lang.String"**>  
 <**constructor-arg value="大幂幂，杨幂"**></**constructor-arg**>  
</**bean**>  
  
  
<**bean id="msg" class="task.Message"**>  
 <**property name="info" value="#{str.substring(0,3)+',我爱你！'}"**></**property**>  
</**bean**>

1. 测试程序

ApplicationContext ctx = **new** ClassPathXmlApplicationContext(**"spring-config.xml"**);  
Message msg = ctx.getBean(**"msg"**, Message.**class**);  
System.***out***.println(msg.getInfo());

由于Spring本身是支持Annotation的，可以是用注解代码的引用控制。

1. 配置文件中添加Annotation支持

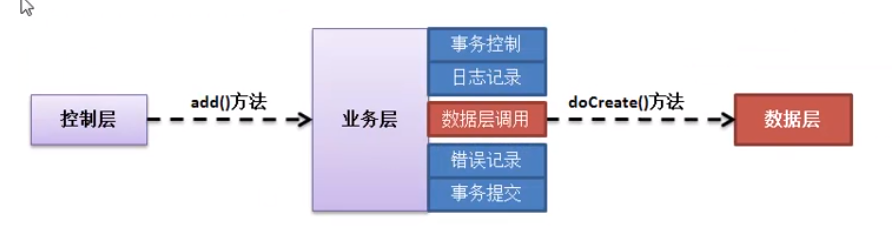
*<?***xml version="1.0" encoding="UTF-8"***?>*<**beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xmlns:context="http://www.springframework.org/schema/context"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.3.xsd  
 http://www.springframework.org/schema/context   
 http://www.springframework.org/schema/context/spring-context-4.3.xsd"**>  
 <**context:annotation-config**/>  
 <**context:component-scan base-package="task"**/>  
  
 <**bean id="str" class="java.lang.String"**>  
 <**constructor-arg value="大幂幂，杨幂"**/>  
 </**bean**>  
</**beans**>

1. 修改Message配置文件

@Component  
**public class** Message {  
 @Value(**"#{str.substring(4)+'，我爱你！'}"**)  
 **private** String **info**;  
  
 **public void** setInfo(String info) {  
 **this**.**info** = info;  
 }  
  
 **public** String getInfo() {  
 **return info**;  
 }  
}

## 五、AOP编程

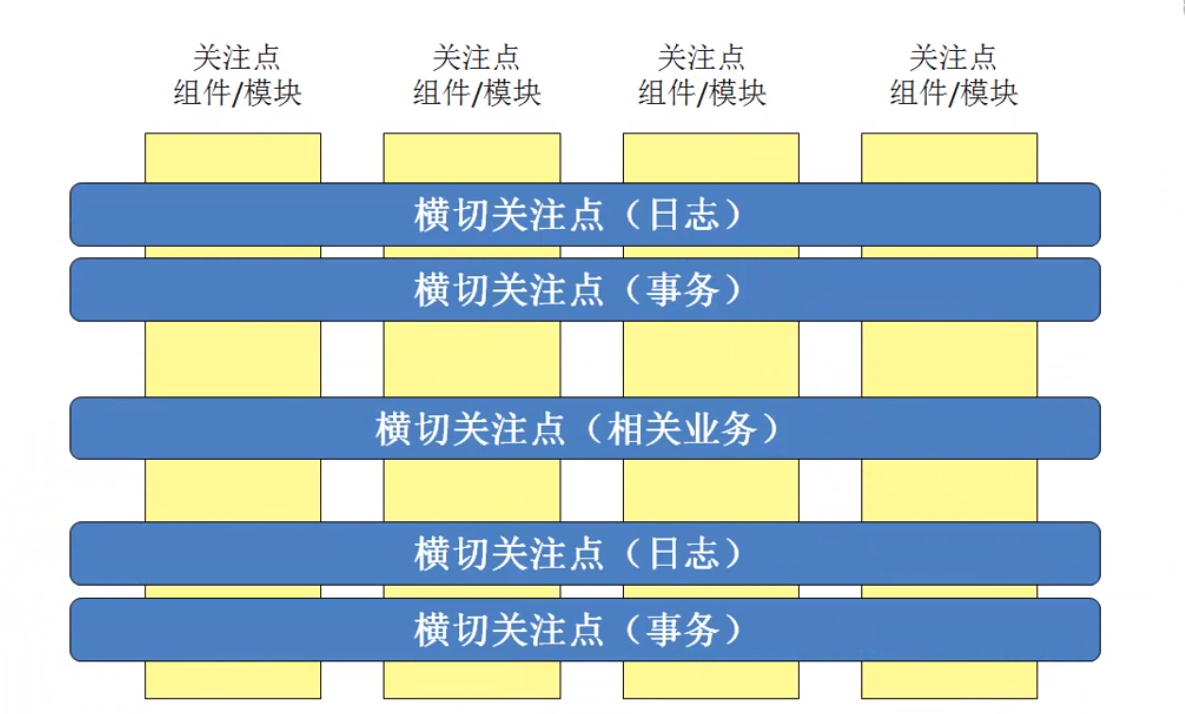
5.1 AOP 编程简介



关注点：你无论怎么处理，最终可以处理业务层的操作都是要归结于方法上的，而方法无论怎么变化，名称也是有依据的，例如：addXxx();

横切面：所有的关注点如果要想发挥出自己的作用，那么必须有相应的处理代码；

织入：所有的关于切面的配置以及切入点的配置，都可以可以直接在文件中完成；



其实切面的本质就是编写代码进行程序的相应的辅助功能的实现，为此在AOP编程里面，提供有如下的几类通知：

·前置通知（Before Advice）：在业务操作之前控制；

·后置通知（After Advice）：在具体的业务执行之后完成配置；

|-后置返回通知：（After Returning Advice）：在业务层完成处理之后返回内容的时候控制;

|-后置异常通知（After Throwing Advice）：在业务层调用产生异常之后的控制处理；

|-后置最终通知（After Advice ）:不管是否出现异常，都要执行此类功能;

·环绕通知（Round Finally Advice）：可以在之前和之后一起负责处理；

#### 5.2 AOP的初步实现

1、spring-config.xml配置文件修改

添加AOP命名空间，注解支持：

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

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

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

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

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

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

xsi:schemaLocation="

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-4.1.xsd

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-4.1.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-4.1.xsd">

<context:annotation-config />

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

</beans>

1. 编写独立的控制类，并将其交给SPring管理

public class ServiceProxy {

public void beforeMethod() {

System.out.println("####### 【ServiceProxy】public void beforeMethod(){}");

}

public void afterMethod() {

System.out.println("####### 【ServiceProxy】public void afterMethod(){}");

}

}

<!-- 如果要想让切面控制有效，那么必须将此类的对象交由Spring进行管理 -->

<bean id="serviceProxy" class="cn.mldn.aop.ServiceProxy"/>

1. 配置切入点

<aop:config>

<aop:pointcut expression="execution(\* cn.mldn.service..\*.\*(..))" id="pointCut"/>

</aop:config>

切入点表达式配置如下：“execution(\* cn.mldn.service..\*.\*(..))”,其配置的语法结构：

·“execution(修饰符匹配? 返回值类型匹配 操作类型匹配？ 名称匹配（参数匹配） 抛出异常匹配)”：主要是用于定义切入点的语法;

·在本次操作中，没有编写修饰符，如果要编写public、private，可以出现0次或1次；

·“\*”表示返回值，即支持所有的返回值类型；

·“cn.mldn.service..”包名称匹配，如果是“..”表示任意的子包下；

·“\*.\*”描述类方法，“\*.\*”描述的是这个包下所有的类与所有的方法；

·“（..）”描述的是参数，如果是“..”表示的是任意参数;

1. 设置切面：

<aop:config>

<aop:pointcut expression="execution(\* cn.mldn.service..\*.\*(..))" id="pointCut"/>

<aop:aspect ref="serviceProxy"> <!-- 进行切面的配置，如果是切面控制应该有一个控制的类 -->

<!-- 不同的切面处理操作可以使用同一个切入点，或者每一个操作来定义属于自己的切入点 -->

<aop:before method="beforeMethod" pointcut-ref="pointCut"/>

<aop:after method="afterMethod" pointcut="execution(\* cn.mldn.service..\*.\*(..))"/>

</aop:aspect>

</aop:config>

现在可以发现，没有硬编码，动态通过配置文件设置切入点以及切入处理类。



### 5.3 AOP深入操作

1、范例：在前置通知中定义参数接收

public void beforeMethod(Object obj) {

System.out.println("####### 【ServiceProxy】public void beforeMethod(){}，参数内容：" + obj);

}

在配置文件中，需要明确的定义出要接收的参数

范例：修改spring-config.xml配置文件

<aop:config>

<aop:pointcut expression="execution(\* cn.mldn.service..\*.\*(..)) **and args(id)**" id="pointCut"/>

<aop:aspect ref="serviceProxy"> <!-- 进行切面的配置，如果是切面控制应该有一个控制的类 -->

<!-- 不同的切面处理操作可以使用同一个切入点，或者每一个操作来定义属于自己的切入点 -->

<aop:before method="beforeMethod" pointcut-ref="pointCut" **arg-names="id"**/>

<aop:after method="afterMethod" pointcut="execution(\* cn.mldn.service..\*.\*(..))"/>

</aop:aspect>

</aop:config>

1. 实现返回值处理

对于方法的返回值，可以进行指定的模式处理。

定义新的控制方法：

public void returnMethod(Object val) { // 处理返回值

System.out.println("####### 【ServiceProxy】public void returnMethod(){}，返回值：" + val);

}

修改配置文件：

<aop:config>

<aop:pointcut expression="execution(\* cn.mldn.service..\*.\*(..)) and args(id)" id="pointCut"/>

<aop:aspect ref="serviceProxy"> <!-- 进行切面的配置，如果是切面控制应该有一个控制的类 -->

<!-- 不同的切面处理操作可以使用同一个切入点，或者每一个操作来定义属于自己的切入点 -->

<aop:before method="beforeMethod" pointcut-ref="pointCut" arg-names="id"/>

<aop:after method="afterMethod" pointcut="execution(\* cn.mldn.service..\*.\*(..))"/>

<aop:after-returning method="returnMethod" pointcut="execution(\* cn.mldn.service..\*.\*(..))" returning="v" arg-names="v"/>

</aop:aspect>

</aop:config>

1. 异常处理

@Override

public boolean delete(String mid) throws Exception {

// System.out.println("\*\*\*\*\*\*\*\*\* 【MemberService】数据删除处理。");

throw new Exception("我就是要抛异常，你怎么着？") ;

// return false;

}

修改ServiceProxy类：

public void throwMethod(Exception e) { // 对异常进行处理

System.out.println("####### 【ServiceProxy】public void throwMethod(){}，异常信息：" + e);

}

修改切入点中切面配置：

<aop:after-throwing method="throwMethod" pointcut="execution(\* cn.xxx.service..\*.\*(..))" throwing="e" arg-names="e"/>

以上几种通知处理操作都是固化的。

1. 环绕通知

包含了以上的所有通知效果，使用环绕通知，不在需要定义三个处理方法，一个方法就可以搞定。

定义环绕通知处理：

public Object aroundMethod(ProceedingJoinPoint point) throws Throwable { // 调用具体的执行方法

System.out.println("@@@@@ 【环绕通知】aroundMethod() - before，参数：" + Arrays.toString(point.getArgs())); // 取得所有传递过来的参数

// 此时可以针对于参数接收后处理后再传递的操作

Object obj = point.proceed(new Object [] {"abc"}) ; // 自己来处理内容

System.out.println("@@@@@ 【环绕通知】aroundMethod() - after，返回结果：" + obj);

return true ; // 不按照结果返回数据

}

修改配置文件：

<aop:around method="aroundMethod" pointcut="execution(\* cn.mldn.service..\*.\*(..))"/>

#### 5.4 基于annotation配置

修改spring-config.xml配置文件

*<?***xml version="1.0" encoding="UTF-8"***?>*<**beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xmlns:context="http://www.springframework.org/schema/context"  
 xmlns:aop="http://www.springframework.org/schema/aop"  
 xsi:schemaLocation="  
 http://www.springframework.org/schema/aop  
 http://www.springframework.org/schema/aop/spring-aop-4.1.xsd  
 http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-4.1.xsd  
 http://www.springframework.org/schema/context  
 http://www.springframework.org/schema/context/spring-context-4.1.xsd"**>  
 <**context:annotation-config**/>  
 <**context:component-scan base-package="task"**/>  
  ***<!--表示当前的开发支持注解-->* <aop:aspectj-autoproxy/>**  
</**beans**>

修改ServiceProxy文件：

@Aspect

@Component

public class ServiceProxy {

@Before(value="execution(\* cn.mldn.service..\*.\*(..)) and args(id)",argNames="id")

public void beforeMethod(Object obj) {

System.out.println("####### 【ServiceProxy】public void beforeMethod(){}，参数内容：" + obj);

}

@After(value="execution(\* cn.mldn.service..\*.\*(..))")

public void afterMethod() {

System.out.println("####### 【ServiceProxy】public void afterMethod(){}");

}

@AfterReturning(value="execution(\* cn.mldn.service..\*.\*(..))",returning="v",argNames="v")

public void returnMethod(Object val) { // 处理返回值

System.out.println("####### 【ServiceProxy】public void returnMethod(){}，返回值：" + val);

}

@AfterThrowing(value="execution(\* cn.mldn.service..\*.\*(..))",throwing="e",argNames="e")

public void throwMethod(Exception e) { // 对异常进行处理

System.out.println("####### 【ServiceProxy】public void throwMethod(){}，异常信息：" + e);

}

@Around(value="execution(\* cn.mldn.service..\*.\*(..))")

public Object aroundMethod(ProceedingJoinPoint point) throws Throwable { // 调用具体的执行方法

System.out.println("@@@@@ 【环绕通知】aroundMethod() - before，参数：" + Arrays.toString(point.getArgs())); // 取得所有传递过来的参数

// 此时可以针对于参数接收后处理后再传递的操作

Object obj = point.proceed(new Object [] {"abc"}) ; // 自己来处理内容

System.out.println("@@@@@ 【环绕通知】aroundMethod() - after，返回结果：" + obj);

return true ; // 不按照结果返回数据

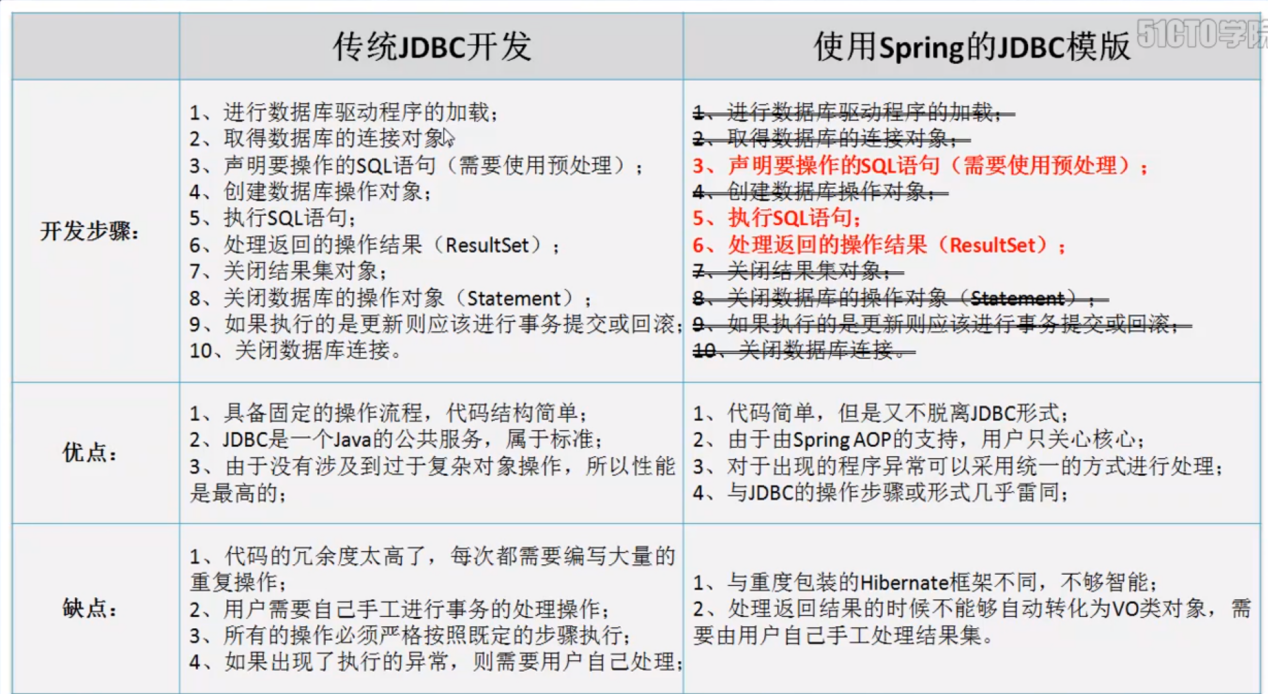
}

}

### JDBC操作模板

#### 6.1 JdbcTemplate简介

在Spring里面有ORM设计思想，这其中最大的体现是JdbcTemplate组件，这提出的是一个数据层的操作解决方法。但是它又与hibernate不同，hibernate最大特点是重度包装，而且一旦处理不当，就会有性能问题，Mybatis(Ibatis)属于中度包装，但是有些时候，进行jdbc操作的话，并不需要一个包装组件，毕竟包装组件需要而外的配置，所以只是希望传统的jdbc操作有更加便利的操作方式。Jdbc绝对是性能最高的，没有任何的时间，空间复杂度。



综合来讲，JdbcTemplate具备了jdbc的高性能，同时又最大程度的保证了jdbc的原始性，唯一的缺点是JdbcTemplate没有设计成像Hibernate或Mybatis那样的智能组件，所以很大一部分代码需要用户来控制。

#### 6.2 连接数据库

如果想在Spring里面进行数据库的连接操作，则必须使用一个类org.springframework.jdbc.datasource.DriverManagerDataSource，这个类中定义有如下方法：

|  |  |  |  |
| --- | --- | --- | --- |
| No. | 方法名称 | 类型 | 描述 |
| 1 | public DriverManagerDataSource() | 构造 | 无参构造方法 |
| 2 | public DriverManagerDataSource(String url, String username,String password) | 构造 | 设置数据库的连接路径，用户名，密码 |
| 3 | public void setDriverClassName(String driverClassName) | 普通 | 设置要使用的驱动程序 |
| 4 | public void setUrl(String url) | 普通 | 设置连接路径 |
| 5 | public void setUsername(String username) | 普通 | 设置用户名 |
| 6 | public void setPassword(String password) | 普通 | 设置密码 |

修改配置文件

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

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.1.xsd">

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="org.gjt.mm.mysql.Driver"/>

<property name="url" value="jdbc:mysql://localhost:3306/mldn"/>

<property name="username" value="root"/>

<property name="password" value="mysqladmin"/>

</bean>

</beans>

取得连接对象：

ApplicationContext ctx = new ClassPathXmlApplicationContext(

"applicationContext.xml");

DriverManagerDataSource dataSource = ctx.getBean("dataSource",

DriverManagerDataSource.class);

System.out.println(dataSource);

#### 6.3 数据库的CRUD操作

在Spring中org.springframework.jdbc.core.JdbcTemplate类，主要是负责数据的处理。

|  |  |  |  |
| --- | --- | --- | --- |
| No | 方法名称 | 类型 | 描述 |
| 1 | public JdbcTemplate(DataSource dataSource) | 构造 | 接收DataSource的数据库连接源 |
| 2 | public void setDataSource(DataSource dataSource) | 普通 | 设置要使用的DataSource数据源 |
| 3 | public int[] batchUpdate(String sql, List<Object[]> batchArgs) throws DataAccessException | 普通 | 数据的批量更新处理 |
| 4 | public int update(final String sql) throws DataAccessException | 普通 | 执行更新处理，insert、update、delete |
| 5 | public <T> List<T> query(String sql, Object[] args, RowMapper<T> rowMapper) throws DataAccessException | 普通 | 进行数据查询，同时实现Result与VO转换 |
| 6 | public List<Map<String, Object>> queryForList(String sql, Object... args) throws DataAccessException | 普通 | 查询数据 |
| 7 | public <T> List<T> queryForList(String sql, Object[] args, Class<T> elementType) throws DataAccessException | 普通 | 数据查询，同时设置返回结果中的泛型 |
| 8 | public Map<String, Object> queryForMap(String sql, Object... args) throws DataAccessException | 普通 | 数据查询，结果以Map的形式返回，没有查询到数据时，**会报空指针** |
| 9 | public <T> T queryForObject(String sql, Class<T> requiredType, Object... args) throws DataAccessException | 普通 | 查询出单个对象 |

记录一个以前没用过的更新处理：

ApplicationContext ctx = new ClassPathXmlApplicationContext(

"applicationContext.xml");

DriverManagerDataSource dataSource = ctx.getBean("dataSource",

DriverManagerDataSource.class);

JdbcTemplate jt = new JdbcTemplate(dataSource); // 取得JDBC模版对象

String sql = "INSERT INTO member(mid,name,age,sex,birthday,salary,note) VALUES (?,?,?,?,?,?,?)";

int len = jt.update(sql, new **PreparedStatementSetter**(){

@Override

public void setValues(PreparedStatement pstmt) throws SQLException {

pstmt.setString(1, "mldn-你好");

pstmt.setString(2, "李四");

pstmt.setInt(3, 12);

pstmt.setString(4, "男");

pstmt.setDate(5, new java.sql.Date(new java.util.Date().getTime()));

pstmt.setDouble(6, 1111.11);

pstmt.setString(7, "很麻烦");

}}); // 返回影响的数据行

System.out.println(len);

dataSource.getConnection().close(); // 关闭数据库

而后最关键的是在RowMapper接口上，这个接口要求用户自己处理：

范例：数据查询

·查询方法：public <T> List<T> query(String sql, Object[] args, RowMapper<T> rowMapper) throws DataAccessException

实现：

ApplicationContext ctx = new ClassPathXmlApplicationContext(

"applicationContext.xml");

DriverManagerDataSource dataSource = ctx.getBean("dataSource",

DriverManagerDataSource.class);

JdbcTemplate jt = new JdbcTemplate(dataSource); // 取得JDBC模版对象

String sql = "SELECT mid,name,age,sex,birthday,salary,note FROM member";

List<Member> all = jt.query(sql, new RowMapper<Member>() {

@Override

public Member mapRow(ResultSet rs, int rowNum) throws SQLException {

System.out.println("当前处理行：" + rowNum);

Member vo = new Member() ;

vo.setMid(rs.getString(1));

vo.setName(rs.getString(2));

vo.setAge(rs.getInt(3));

vo.setSex(rs.getString(4));

vo.setBirthday(rs.getDate(5));

vo.setSalary(rs.getDouble(6));

vo.setNote(rs.getString(7));

return vo;

}}) ;

System.out.println(all);

dataSource.getConnection().close(); // 关闭数据库

以上newRowMapper可以写成一个内部类。

范例：只查询单个对象

ApplicationContext ctx = new ClassPathXmlApplicationContext(

"applicationContext.xml");

DriverManagerDataSource dataSource = ctx.getBean("dataSource",

DriverManagerDataSource.class);

JdbcTemplate jt = new JdbcTemplate(dataSource); // 取得JDBC模版对象

String sql = "SELECT mid,name,age,sex,birthday,salary,note FROM member WHERE mid=?";

Member member = jt.queryForObject(sql, new RowMapper<Member>() {

@Override

public Member mapRow(ResultSet rs, int rowNum) throws SQLException {

System.out.println("当前处理行：" + rowNum);

Member vo = new Member() ;

vo.setMid(rs.getString(1));

vo.setName(rs.getString(2));

vo.setAge(rs.getInt(3));

vo.setSex(rs.getString(4));

vo.setBirthday(rs.getDate(5));

vo.setSalary(rs.getDouble(6));

vo.setNote(rs.getString(7));

return vo;

}},"mldn-hello") ;

System.out.println(member);

dataSource.getConnection().close(); // 关闭数据库

范例：查询某一列数据

ApplicationContext ctx = new ClassPathXmlApplicationContext(

"applicationContext.xml");

DriverManagerDataSource dataSource = ctx.getBean("dataSource",

DriverManagerDataSource.class);

JdbcTemplate jt = new JdbcTemplate(dataSource); // 取得JDBC模版对象

String sql = "SELECT mid FROM member";

List<String> all = jt.queryForList(sql, String.class) ;

System.out.println(all);

dataSource.getConnection().close(); // 关闭数据库

#### 6.4 使用c3p0数据库连接池

6.4.1 考虑到项目维护，往往会在项目里面建立一个database.properties文件：

#驱动程序

db.driver=org.gjt.mm.mysql.Driver

#数据库地址

db.url=jdbc:mysql://localhost:3306/mldn?useSSL=false

#数据库用户名

db.user=root

#密码

db.password=root

#连接池允许最大连接数

pool.max=100

#连接池最小连接数

pool.min=20

#初始化连接数

pool.init=10

#连接等待时间

pool.idle=100

6.4.2 创建以上配置文件后，它还是一个独立文件，需要在配置文件中引用。

<**context:property-placeholder location="classpath:database.properties"**/>（需要引用context命名空间）；

6.4.3 配置c3p0:

<**bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"**>  
 <**property name="driverClass" value="${db.driver}"**/>  
 <**property name="jdbcUrl" value="${db.url}"**/>  
 <**property name="user" value="${db.user}"**/>  
 <**property name="password" value="${db.password}"**/>  
 <**property name="maxPoolSize" value="${pool.max}"**/>  
 <**property name="minPoolSize" value="${pool.min}"**/>  
 <**property name="initialPoolSize" value="${pool.init}"**/>  
 <**property name="maxIdleTime" value="${pool.idle}"**/>  
</**bean**>

6.4.4 JdbcTemplate配置

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

#### 6.5 JdbcDaoSupport支持

考虑到各个ORM组件开发问题，在Spring针对于不同的ORM工具都会提供有一个XxxDaoSupport支持类，这个类可以简化一些组件中出现重复代码的问题。JDBCDaoSupport类是DaoSupport子类，实际上就是给Dao实现子类使用的，要求Dao实现子类，既要实现DAO接口，又要继承一个抽象的父类。

|  |  |  |  |
| --- | --- | --- | --- |
| No | 方法名称 | 类型 | 描述 |
| 1 | public final JdbcTemplate getJdbcTemplate() | 普通 | 取得JdbcTemplate模板对象 |
| 2 | public final void setDataSource(DataSource dataSource) | 普通 | 设置要使用的数据源 |
| 3 | public final void setJdbcTemplate(@Nullable JdbcTemplate jdbcTemplate) | 普通 | 设置JdbcTemplate对象 |

范例：定义一IMemberDao接口

public interface IMemberDAO {

public boolean doCreate(Member vo) throws Exception ;

}

@Component

public class MemberDAOImpl implements IMemberDAO {

@Resource

private JdbcTemplate jdbcTemplate;

@Override

public boolean doCreate(Member vo) throws Exception {

this.jdbcTemplate.update(sql);

return false;

}

}

如果按照以上的做法，最终往往需要用户自己来控制数据库的关闭操作，所以建议使用JDBCSupportDao实现支持，这个类可以在操作完成之后可以自动关闭连接。

修改MemberDAOImpl接口

public class MemberDAOImpl extends JdbcDaoSupport implements IMemberDAO {

@Autowired

public MemberDAOImpl(JdbcTemplate jdbcTemplate) { // 注入JdbcTemplate

super.setJdbcTemplate(jdbcTemplate); // 由DaoSupport管理JdbcTemplate

}

@Override

public boolean doCreate(Member vo) throws Exception {

super.getJdbcTemplate().update("") ; // 执行完成后由JdbcDaoSupport负责数据库关闭

return false;

}

}

## 七、Spring与hibernate整合

对于SSh整合操作步奏：

·Spring与Hibernate整合：主要包括配置信息的位置变化，以及事务处理的引入。

·Spring与Struts2.x整合：主要进行前端开发框架的整合。

Spring +Struts+Hibernatemaven整合依赖包：

<properties>

<!-- 统一源码的编码方式 -->

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<!-- 统一各个框架版本 -->

<struts.version>2.5.16</struts.version>

<spring.version>5.0.5.RELEASE</spring.version>

<hibernate.version>5.2.17.Final</hibernate.version>

</properties>

<dependencies>

<!-- Junit依赖 -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

<!-- Spring 核心依赖 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

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

</dependency>

<!-- Spring web依赖 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

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

</dependency>

<!-- Spring整合ORM框架依赖 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-orm</artifactId>

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

</dependency>

<!-- Struts2 核心依赖 -->

<dependency>

<groupId>org.apache.struts</groupId>

<artifactId>struts2-core</artifactId>

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

</dependency>

<!-- Struts2和Spring整合依赖 -->

<dependency>

<groupId>org.apache.struts</groupId>

<artifactId>struts2-spring-plugin</artifactId>

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

</dependency>

<!-- Hibernate 核心依赖 -->

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

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

</dependency>

<!-- MySQL 依赖 -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.42</version>

</dependency>

<!-- C3P0 依赖 -->

<dependency>

<groupId>com.mchange</groupId>

<artifactId>c3p0</artifactId>

<version>0.9.5</version>

</dependency>

<!-- AspectJ依赖 -->

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjweaver</artifactId>

<version>1.9.0</version>

</dependency>

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-core</artifactId>

<version>2.10.0</version>

</dependency>

<!-- SLF4J依赖 -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.7.25</version>

</dependency>

</dependencies>

另外，需要加类路径下的配置文件拷贝到编译路径下：

<build>

<resources>

<resource>

<directory>src/main/java</directory>

<includes>

<include>\*\*/\*.xml</include>

</includes>

<filtering>true</filtering>

</resource>

<resource>

<directory>src/main/resources</directory>

<includes>

<include>\*\*/\*.xml</include>

<include>\*\*/\*.properties</include>

</includes>

</resource>

</resources>

</build>

### 7.1 后台业务整合（Spring+Hibernate）

7.1.1 考虑到开发中需要显示日志信息，所以首先将log4j.properties文件拷贝。

7.4.2 使用maven为项目添加SSH开发包，使用idea添加Spring框架.

7.4.3 由于IDEA 不会自动添加Spring配置信息，需要将以下内容拷贝到Web.xml文件：

<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

**<context-param>** 等价于application.setAttribute()

<param-name>contextConfigLocation</param-name>

<param-value>classpath:spring-config.xml</param-value>

</context-param>

这样容器启动的时候，就可以启动Spring。

7.4.4 使用idea添加Hibernate支持，数据库连接池交由Spring管理，但是与Hibernate有关的配置文件（\*.hbm.xml、Annotation类）还是在Hibernate.cfg.xml文件中管理。将Hibernate中SessionFactory交给Spring管理。添加数据库配置信息。

7.4.5 修改Hibernate.cfg.xml配置文件

不再需要配置数据库的连接配置

<hibernate-configuration>

<!-- 配置连接工厂 -->

<session-factory>

<!-- 配置数据库的操作方言 -->

<property name="dialect">

org.hibernate.dialect.MySQLDialect

</property>

<!-- 配置显示执行SQL语句 -->

<property name="show\_sql">true</property>

<!-- 格式化显示的SQL语句 -->

<property name="format\_sql">true</property>

</session-factory>

</hibernate-configuration>

以后如果建立POJO类的时候一定会在此文件中追加一些资源配置，或者在配置缓存的时候，也可以直接通过修改此文件得到。

7.4.6 创建或拷贝database.properties文件。

db.driver=org.gjt.mm.mysql.Driver

db.url=jdbc:mysql://localhost:3306/mldn?useSSL=false

db.user=root

db.password=root

pool.max=100

pool.min=20

pool.init=10

pool.idle=100

随后在spring-config文件中引入此资源配置信息(需要context命名空间，同时也把其他的tx、p、aop等引入)：

* 在spring-config.xml文件中追加命名空间

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

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

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

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

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

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

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

xsi:schemaLocation="

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-4.3.xsd

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-4.3.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-4.3.xsd

http://www.springframework.org/schema/tx

[http://www.springframework.org/schema/tx/spring-tx.xsd">](http://www.springframework.org/schema/tx/spring-tx.xsd\">)

</beans>

* 配置数据库连接，本次使用的是c3p0组件：

<!-- 配置数据库连接池 -->

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">

<property name="driverClass" value="${db.driver}" /> <!-- 驱动程序 -->

<property name="jdbcUrl" value="${db.url}" /><!-- 连接地址 -->

<property name="user" value="${db.user}" /> <!-- 用户名 -->

<property name="password" value="${db.password}" /> <!-- 密码 -->

<property name="maxPoolSize" value="${pool.max}" /> <!-- 最大的可用连接数 -->

<property name="minPoolSize" value="${pool.min}" /> <!-- 最小维持的连接数 -->

<property name="initialPoolSize" value="${pool.init}" /> <!-- 初始化连接数 -->

<property name="maxIdleTime" value="${pool.idle}" /> <!-- 最大等待连接 -->

</bean>

7.4.7 启用Annotation支持，配置资源文件读取

<!-- 配置Annotation的支持操作 -->

<context:annotation-config />

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

<context:property-placeholder location="classpath:database.properties" />

7.4.8 将hibernate中SessionFactory交给Spring管理。

<!-- 配置Hibernate中的SessionFactory程序类，此时将通过Spring负责管理 -->

<bean id="sessionFactory"

class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">

<!-- 配置hibernate.cfg.xml文件的引用路径，通过CLASSPATH引用 -->

<property name="configLocation" value="classpath:hibernate.cfg.xml"/>

<!-- SessionFactory的执行需要dataSource支持，将之前的C3P0设置的数据源配置进来 -->

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

</bean>

Hibernate要进行操作，使用的一定是Session，与jdbcTemple一样，在Spring里面提供有一个hibernateTemplate程序类，封装了hibernate处理操作。

<!-- 程序中的HibernateTemplate可以负责Hibernate方便的处理操作，但是需要与HibernateDaoSupport整合 -->

<bean id="hibernateTemplate" class="org.springframework.orm.hibernate4.HibernateTemplate">

<!-- 提供有sessionFactory的引用关系 -->

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

</bean>

7.4.9 如果要使用使句酷，必须配置事务。对于事务控制，主要是采用transaction完成，所有的声明式事务，需要tx命名空间。

* 配置hibernate事务控制

<!-- 配置Hibernate的事务控制，同时可以实现数据库的关闭处理 -->

<bean id="transactionManager"

class="org.springframework.orm.hibernate4.HibernateTransactionManager">

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

</bean>

但是最为关键的事务操作控制并不是在这里进行的，而是需要进行单独引用；

<!-- 表示对于事务采用Annotation的配置方式进行 -->

<tx:annotation-driven transaction-manager="transactionManager" />

<!-- 定义通知，配置事务出现的环境，与业务层的方法有关 -->

<tx:advice id="txAdvice" transaction-manager="transactionManager">

<!-- 定义一切与服务层有关的控制方法名称，只要是使用了特定的名称那么就会自动处理事务 -->

<tx:attributes>

<!-- 所有的更新操作都必须启动一个独立的事务控制 -->

<tx:method name="insert\*" propagation="REQUIRED" />

<tx:method name="update\*" propagation="REQUIRED" />

<tx:method name="delete\*" propagation="REQUIRED" />

<tx:method name="add\*" propagation="REQUIRED" />

<tx:method name="edit\*" propagation="REQUIRED" />

<tx:method name="change\*" propagation="REQUIRED" />

<tx:method name="remove\*" propagation="REQUIRED" />

<tx:method name="rm\*" propagation="REQUIRED" />

<tx:method name="login\*" propagation="REQUIRED" />

<!-- 所有的读取操作都采用只读的方式进行处理 -->

<tx:method name="get\*" propagation="REQUIRED" read-only="true" />

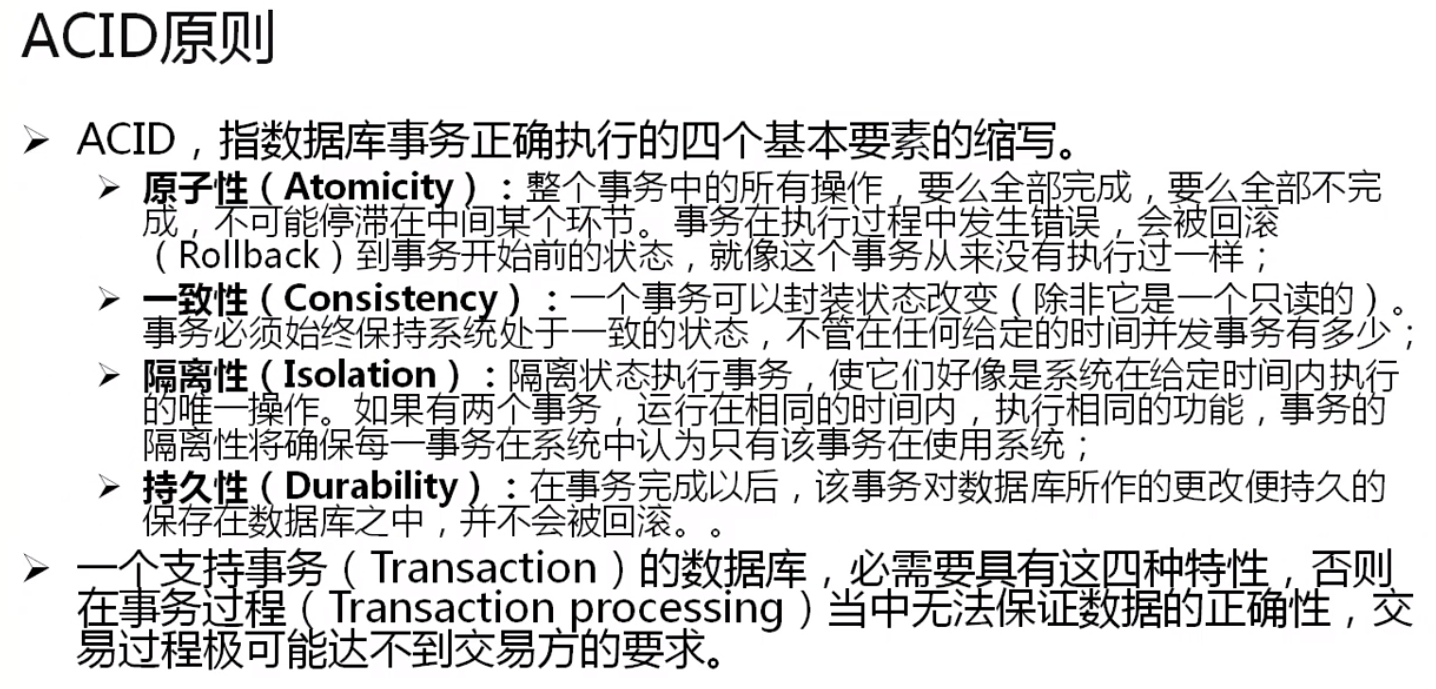
<tx:method name="load\*" propagation="REQUIRED" read-only="true" />

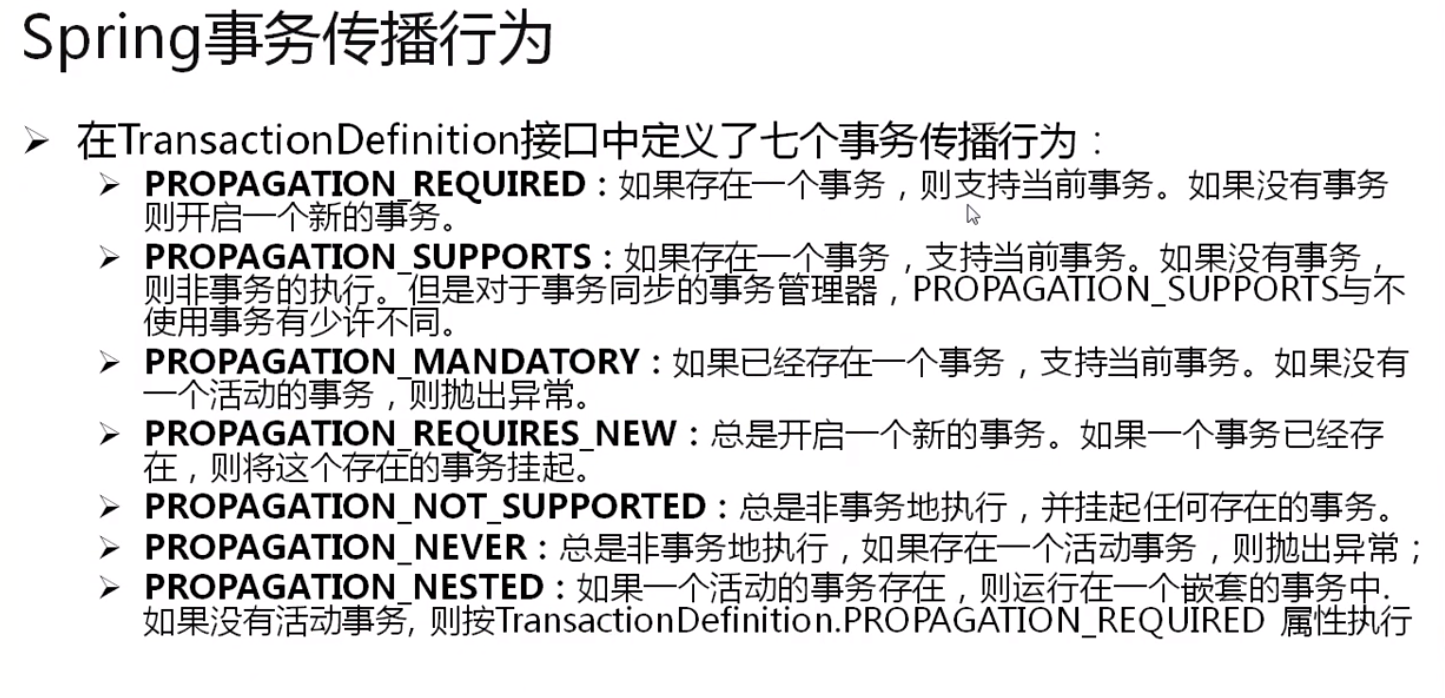
<tx:method name="list\*" propagation="REQUIRED" read-only="true" />

<tx:method name="\*" propagation="REQUIRED" read-only="true" />

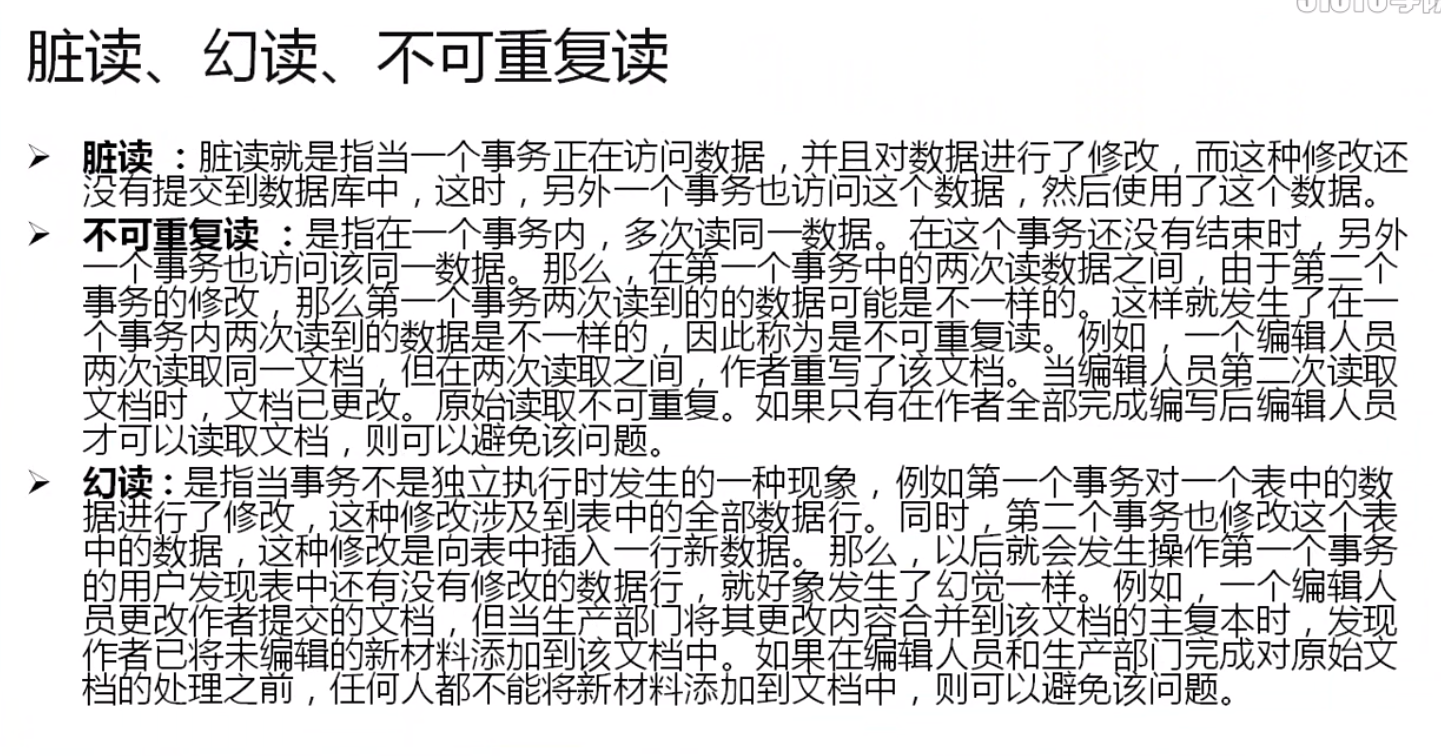
</tx:attributes>

</tx:advice>





除了传播属性，spring中还有隔离属性，先看以下概念：



* 利用AOP设置事务的切入点：

<!-- 定义事务的处理切入点 -->

<aop:config expose-proxy="true">

<!-- 在业务层中进行切入控制 -->

<aop:pointcut expression="execution(\* cn.mldn..service.\*.\*(..))" id="pointcut"/>

<!-- 所有的通知控制交由“事务的通知处理完成” -->

<aop:advisor advice-ref="txAdvice" pointcut-ref="pointcut"/>

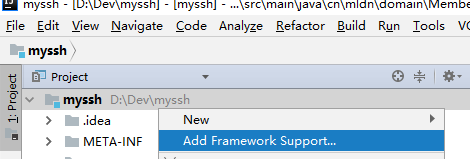
</aop:config>

7.4.10 借助idea生成Member表与POJO转换的类文件

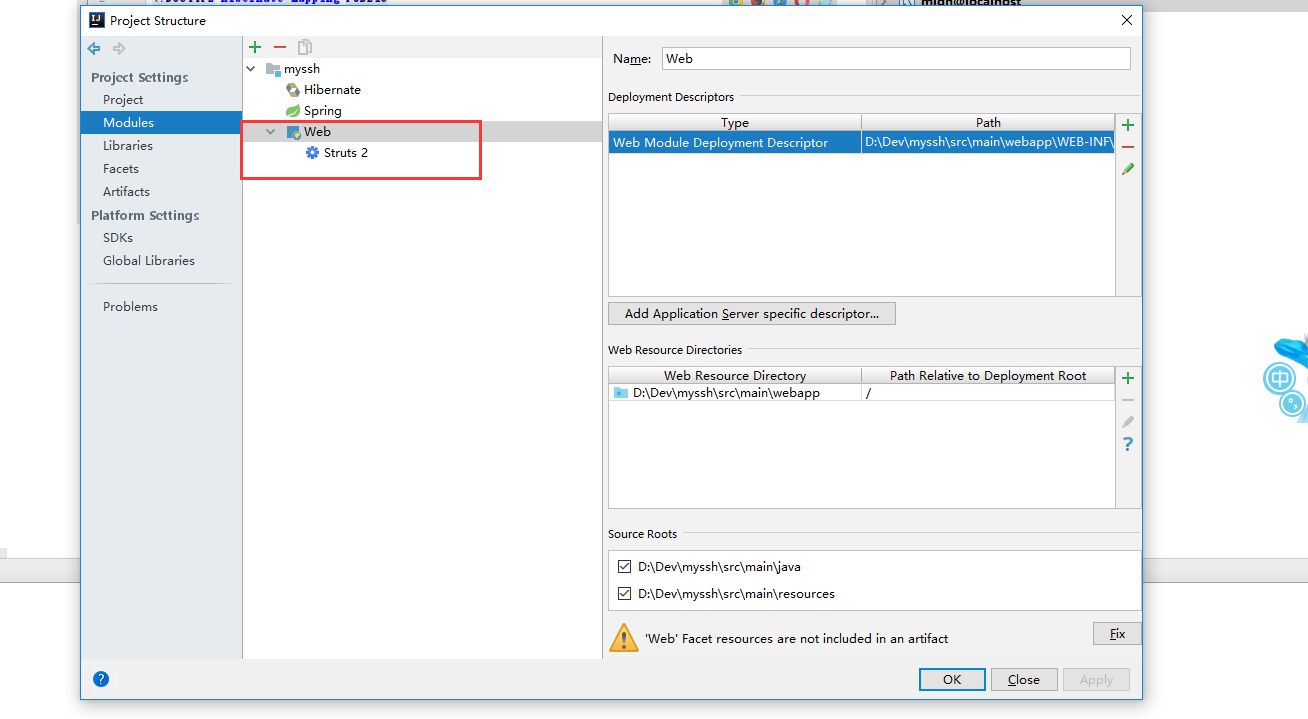
此时表示Spring与hibernate整合完成。

## 八、Spring与Struts整合

8.1 使用idea添加Struts框架支持，在此过程中发现在以下选项中找不到WebApplication。



先把项目结构里面的Web去掉，这样就可以找到了



在Web.xml文件中添加Strut配置：

<filter>

<filter-name>struts2</filter-name>

<filter-class>org.apache.struts2.dispatcher.filter.StrutsPrepareAndExecuteFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>struts2</filter-name>

<url-pattern>\*.action</url-pattern>

</filter-mapping>

创建struts.properties文件：

struts.i18n.encoding=UTF-8

# struts.custom.i18n.resources=Messages,Pages

#整合

struts.objectFactory=spring

### SSH整合开发

在Dao层开发时，千万不要直接使用SessionFactory的openSession创建session，因为所有的session将通过Spring管理。所以只能够取得当前的session。

## HibernateDaoSupport支持类

org.springframework.orm.hibernate5.support.HibernateDaoSupport是一个抽象类，也就是说Dao子类需要继承此类。

|  |  |  |  |
| --- | --- | --- | --- |
| No | 方法名称 | 类型 | 描述 |
| 1 | public HibernateDaoSupport() | 构造 | 无参构造 |
| 2 | protected final Session currentSession() throws DataAccessResourceFailureException | 普通 | 取得当前的Session对象 |
| 3 | public final void setHibernateTemplate( HibernateTemplate hibernateTemplate) | 普通 | 设置hibernateTemplate组件 |
| 4 | public final HibernateTemplate getHibernateTemplate() | 普通 | 取得hibernateTemplate组件 |

必须要求咋spring配置文件中配置hibernateTemplate开发的Bean。

<bean id="hibernateTemplate" class="org.springframework.orm.hibernate4.HibernateTemplate">

<!-- 提供有sessionFactory的引用关系 -->

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

</bean>

那么这个时候就可以在Dao的子类里面，进行控制。

@Component  
**public class** MemberDAOImpl **implements** IMemberDAO {  
  
 */\*\*  
 \* Spring 负责管理Session的打开与关闭  
 \*/* @Resource  
 **private** SessionFactory **sessionFactory**;  
  
 @Override  
 **public boolean** doCreate(Member vo) **throws** Exception {  
 **return this**.**sessionFactory**.getCurrentSession().save(vo) != **null**;  
 }  
  
 @Override  
 **public** Member findById(**int** id) **throws** Exception {  
 **return this**.**sessionFactory**.getCurrentSession().get(Member.**class**, id);  
 }  
}

修改以上的代码，用hibernateDaoSupport配置：

@Component  
**public class** MemberDAOImpl **extends** HibernateDaoSupport **implements** IMemberDAO {  
  
 @Autowired  
 **public** MemberDAOImpl(HibernateTemplate hibernateTemplate) {  
 **super**.setHibernateTemplate(hibernateTemplate);  
 }  
  
 @Override  
 **public boolean** doCreate(Member vo) **throws** Exception {  
 **return super**.currentSession().save(vo) != **null**;  
 }  
  
 @Override  
 **public** Member findById(**int** id) **throws** Exception {  
 **return super**.currentSession().get(Member.**class**, id);  
 }  
}