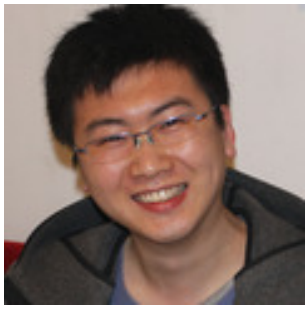


跟我学spring3(1-7)

作者: jinnianshilongnian
http://jinnianshilongnian.iteye.com

跟我学spring3

目 录

1. spring

1.1 【第二章】 IoC 之 2.3 IoC的配置使用——跟我学Spring3

发表时间: 2012-02-20 关键字: spring

2.3.1 XML配置的结构

一般配置文件结构如下：

java代码：

查看 复制到剪贴板 打印

```
<beans>
   <import resource=" resource1.xml" />
   <bean id=" bean1" class=" " ></bean>
   <bean id=" bean2" class=" " ></bean>
<bean name=" bean2" class=" " ></bean>
   <alias alias="bean3" name="bean2"/>
   <import resource=" resource2.xml" />
</beans>
```

1、<bean>标签主要用来进行Bean定义；

2、alias用于定义Bean别名的；

3、import用于导入其他配置文件的Bean定义，这是为了加载多个配置文件，当然也可以把这些配置文件构造为一个数组（new String[] {“config1.xml”, config2.xml}）传给ApplicationContext实现进行加载多个配置文件，那一个更适合由用户决定；这两种方式都是通过调用Bean Definition Reader 读取Bean定义，内部实现没有任何区别。<import>标签可以放在<beans>下的任何位置，没有顺序关系。

2.3.2 Bean的配置

Spring IoC容器目的就是管理Bean，这些Bean将根据配置文件中的Bean定义进行创建，而Bean定义在容器内部由BeanDefinition对象表示，该定义主要包含以下信息：

●全限定类名（FQN）：用于定义Bean的实现类;

●Bean行为定义：这些定义了Bean在容器中的行为；包括作用域（单例、原型创建）、是否惰性初始化及生命周期等；

●Bean创建方式定义：说明是通过构造器还是工厂方法创建Bean；

●Bean之间关系定义：即对其他bean的引用，也就是依赖关系定义，这些引用bean也可以称之为同事bean 或依赖bean，也就是依赖注入。

Bean定义只有“全限定类名”在当使用构造器或静态工厂方法进行实例化bean时是必须的，其他都是可选的定义。难道Spring只能通过配置方式来创建Bean吗？回答当然不是，某些SingletonBeanRegistry接口实现类实现也允许将那些

非BeanFactory创建的、已有的用户对象注册到容器中，这些对象必须是共享的，比如使用DefaultListableBeanFactory 的registerSingleton() 方法。不过建议采用元数据定义。

2.3.3 Bean的命名

每个Bean可以有一个或多个id（或称之为标识符或名字），在这里我们把**第一个id称为“标识符”，其余id叫做“别名”**；这些id在IoC容器中必须唯一。如何为Bean指定id呢，有以下几种方式；

一、 不指定id，只配置必须的全限定类名，由IoC容器为其生成一个标识，客户端必须通过接口“T getBean(Class<T> requiredType)”获取Bean；

java代码：

```
查看 复制到剪贴板 打印
1. <bean class=” cn.javass.spring.chapter2.helloworld.HelloImpl”/> （1）
```

测试代码片段如下：

java代码：

```
查看 复制到剪贴板 打印
1. @Test
2. public void test1() {
3. BeanFactory beanFactory =
4.   new ClassPathXmlApplicationContext("chapter2/namingbean1.xml");
5.   //根据类型获取bean
6.   HelloApi helloApi = beanFactory.getBean(HelloApi.class);
7.   helloApi.sayHello();
8. }
```

二、指定id，必须在Ioc容器中唯一；

java代码：

```
查看 复制到剪贴板 打印
1. <bean id=" bean" class=" cn.javass.spring.chapter2.helloworld.HelloImpl" /> (2)
```

测试代码片段如下：

java代码：

```
查看 复制到剪贴板 打印
1. @Test
2. public void test2() {
3. BeanFactory beanFactory =
4. new ClassPathXmlApplicationContext("chapter2/namingbean2.xml");
5. //根据id获取bean
6. HelloApi bean = beanFactory.getBean("bean", HelloApi.class);
7. bean.sayHello();
8. }
```

三、指定name，这样name就是“标识符”，必须在Ioc容器中唯一；

java代码：

```
查看 复制到剪贴板 打印
1. <bean name=" bean" class=" cn.javass.spring.chapter2.helloworld.HelloImpl" /> (3)
```

测试代码片段如下：

java代码：

```
查看 复制到剪贴板 打印
1. @Test
2. public void test3() {
3. BeanFactory beanFactory =
4. new ClassPathXmlApplicationContext("chapter2/namingbean3.xml");
5. //根据name获取bean
6. HelloApi bean = beanFactory.getBean("bean", HelloApi.class);
7. bean.sayHello();
8. }
```

四、指定id和name，id就是标识符，而name就是别名，必须在Ioc容器中唯一；

java代码：

查看 复制到剪贴板 打印

```
<bean id=" bean1" name=" alias1"
class=" cn.javass.spring.chapter2.helloworld.HelloImpl" />
<!-- 如果id和name一样，IoC容器能检测到，并消除冲突 -->
<bean id="bean3" name="bean3" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>          ( 4 )
```

测试代码片段如下：

java代码：

查看 复制到剪贴板 打印

```
@Test
public void test4() {
BeanFactory beanFactory =
new ClassPathXmlApplicationContext("chapter2/namingbean4.xml");
   //根据id获取bean
   HelloApi bean1 = beanFactory.getBean("bean1", HelloApi.class);
   bean1.sayHello();
   //根据别名获取bean
   HelloApi bean2 = beanFactory.getBean("alias1", HelloApi.class);
   bean2.sayHello();
   //根据id获取bean
   HelloApi bean3 = beanFactory.getBean("bean3", HelloApi.class);
   bean3.sayHello();
   String[] bean3Alias = beanFactory.getAliases("bean3");
   //因此别名不能和id一样，如果一样则由IoC容器负责消除冲突
   Assert.assertEquals(0, bean3Alias.length);
}
```

五、指定多个name，多个name用“，”、“；”、“ ”分割，第一个被用作标识符，其他的（alias1、alias2、alias3）是别名，所有标识符也必须在Ioc容器中唯一；

java代码：

```
查看 复制到剪贴板 打印
<bean name=" bean1;alias11,alias12;alias13 alias14"
    class=" cn.javass.spring.chapter2.helloworld.HelloImpl" />
<!-- 当指定id时，name指定的标识符全部为别名 -->
<bean id="bean2" name="alias21;alias22"
class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>          ( 5 )
```

测试代码片段如下：

java代码：

```
查看 复制到剪贴板 打印
@Test
public void test5() {
BeanFactory beanFactory =
new ClassPathXmlApplicationContext("chapter2/namingbean5.xml");
   //1根据id获取bean
   HelloApi bean1 = beanFactory.getBean("bean1", HelloApi.class);
   bean1.sayHello();
   //2根据别名获取bean
   HelloApi alias11 = beanFactory.getBean("alias11", HelloApi.class);
   alias11.sayHello();
   //3验证确实是四个别名
   String[] bean1Alias = beanFactory.getAliases("bean1");
   System.out.println("=======namingbean5.xml bean1 别名========");
   for(String alias : bean1Alias) {
      System.out.println(alias);
   }
   Assert.assertEquals(4, bean1Alias.length);
   //根据id获取bean
   HelloApi bean2 = beanFactory.getBean("bean2", HelloApi.class);
   bean2.sayHello();
   //2根据别名获取bean
   HelloApi alias21 = beanFactory.getBean("alias21", HelloApi.class);
   alias21.sayHello();
   //验证确实是两个别名
   String[] bean2Alias = beanFactory.getAliases("bean2");
   System.out.println("=======namingbean5.xml bean2 别名========");
   for(String alias : bean2Alias) {
      System.out.println(alias);
   }
   Assert.assertEquals(2, bean2Alias.length);
}
```

六、使用<alias>标签指定别名，别名也必须在IoC容器中唯一

java代码：

```
查看 复制到剪贴板 打印
1. <bean name="bean" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
2. <alias alias="alias1" name="bean"/>
3. <alias alias="alias2" name="bean"/>                    ( 6 )
```

测试代码片段如下：

java代码：

```
查看 复制到剪贴板 打印
1. @Test
2. public void test6() {
3. BeanFactory beanFactory =
4. new ClassPathXmlApplicationContext("chapter2/namingbean6.xml");
5.    //根据id获取bean
6.    HelloApi bean = beanFactory.getBean("bean", HelloApi.class);
7.   bean.sayHello();
8.    //根据别名获取bean
9.    HelloApi alias1 = beanFactory.getBean("alias1", HelloApi.class);
10.   alias1.sayHello();
11.   HelloApi alias2 = beanFactory.getBean("alias2", HelloApi.class);
12.   alias2.sayHello();
13.   String[] beanAlias = beanFactory.getAliases("bean");
14.   System.out.println("=======namingbean6.xml bean 别名========");
15.   for(String alias : beanAlias) {
16.       System.out.println(alias);
17.   }
18.   System.out.println("=======namingbean6.xml bean 别名========");
19.    Assert.assertEquals(2, beanAlias.length);
20. }
```

以上测试代码在cn.javass.spring.chapter2.NamingBeanTest.java文件中。

从定义来看，name或id如果指定它们中的一个时都作为“标识符”，那为什么还要有id和name同时存在呢？这是因为当使用基于XML的配置元数据时，在XML中id是一个真正的XML id属性，因此当其他的定义来引用这个id时就体现出id的好处了，可以利用XML解析器来验证引用的这个id是否存在，从而更早的发现是否引用了一个不存在的bean，而使用name，则可能要在真正使用bean时才能发现引用一个不存在的bean。

●**Bean命名约定：**Bean的命名遵循XML命名规范，但最好符合Java命名规范，由“字母、数字、下划线组成“，而且应该养成一个良好的命名习惯， 比如采用“驼峰式”，即第一个单词首字母开始，从第二个单词开始首字母大写开始，这样可以增加可读性。

2.3.4 实例化Bean

Spring IoC容器如何实例化Bean呢？传统应用程序可以通过new和反射方式进行实例化Bean。而Spring IoC容器则需要根据Bean定义里的配置元数据使用反射机制来创建Bean。在Spring IoC容器中根据Bean定义创建Bean主要有以下几种方式：

一、使用构造器实例化Bean：这是最简单的方式，Spring IoC容器即能使用默认空构造器也能使用有参数构造器两种方式创建Bean，如以下方式指定要创建的Bean类型：

使用空构造器进行定义，使用此种方式，class属性指定的类必须有空构造器

java代码：

查看 复制到剪贴板 打印

```
1. <bean name="bean1" class="cn.javass.spring.chapter2.HelloImpl2"/>
```

使用有参数构造器进行定义，使用此中方式，可以使用< constructor-arg >标签指定构造器参数值，其中index表示位置，value表示常量值，也可以指定引用，指定引用使用ref来引用另一个Bean定义，后边会详细介绍：

java代码：

查看 复制到剪贴板 打印

```
1. <bean name="bean2" class="cn.javass.spring.chapter2.HelloImpl2">
2. <!-- 指定构造器参数 -->
3.     <constructor-arg index="0" value="Hello Spring!"/>
4. </bean>
```

知道如何配置了，让我们做个例子的例子来实践一下吧：

（1）准备Bean class(HelloImpl2.java)，该类有一个空构造器和一个有参构造器：

java代码：

查看 复制到剪贴板 打印

```
1. package cn.javass.spring.chapter2;
2. public class HelloImpl2 implements HelloApi {
```

```
    private String message;
public HelloImpl2() {
        this.message = "Hello World!";
    }
    Public HelloImpl2(String message) {
        this.message = message;
    }
    @Override
    public void sayHello() {
        System.out.println(message);
    }
}
```

（2）在配置文件(resources/chapter2/instantiatingBean.xml)配置Bean定义，如下所示：

java代码：

```
查看 复制到剪贴板 打印
<!--使用默认构造参数-->
<bean name="bean1" class="cn.javass.spring.chapter2.HelloImpl2"/>
  <!--使用有参数构造参数-->

<bean name="bean2" class="cn.javass.spring.chapter2.HelloImpl2">
<!-- 指定构造器参数 -->
  <constructor-arg index="0" value="Hello Spring!"/>
</bean>
```

（3）配置完了，让我们写段测试代码（InstantiatingContainerTest）来看下是否工作吧：

java代码：

```
查看 复制到剪贴板 打印
@Test
public void testInstantiatingBeanByConstructor() {
    //使用构造器
  BeanFactory beanFactory =
new ClassPathXmlApplicationContext("chapter2/instantiatingBean.xml");
    HelloApi bean1 = beanFactory.getBean("bean1", HelloApi.class);
    bean1.sayHello();
    HelloApi bean2 = beanFactory.getBean("bean2", HelloApi.class);
    bean2.sayHello();
}
```

二、使用静态工厂方式实例化Bean，使用这种方式除了指定必须的class属性，还要指定factory-method属性来指定实例化Bean的方法，而且使用静态工厂方法也允许指定方法参数，spring IoC容器将调用此属性指定的方法来获取Bean，配置如下所示：

（1）先来看看静态工厂类代码吧HelloApiStaticFactory：

java代码：

```
查看 复制到剪贴板 打印
1. public class HelloApiStaticFactory {
2.    //工厂方法
3.       public static HelloApi newInstance(String message) {
4.             //返回需要的Bean实例
5.          return new HelloImpl2(message);
6. }
7. }
```

（2）静态工厂写完了，让我们在配置文件(resources/chapter2/instantiatingBean.xml)配置Bean定义：

java代码：

```
查看 复制到剪贴板 打印
1. <!-- 使用静态工厂方法 -->
2. <bean id="bean3" class="cn.javass.spring.chapter2.HelloApiStaticFactory" factory-
method="newInstance">
3.     <constructor-arg index="0" value="Hello Spring!"/>
4. </bean>
```

（3）配置完了，写段测试代码来测试一下吧，InstantiatingBeanTest：

java代码：

```
查看 复制到剪贴板 打印
1. @Test
2. public void testInstantiatingBeanByStaticFactory() {
3.       //使用静态工厂方法
4.       BeanFactory beanFactory =
5. new ClassPathXmlApplicationContext("chaper2/instantiatingBean.xml");
```

```
6.        HelloApi bean3 = beanFactory.getBean("bean3", HelloApi.class);
7.        bean3.sayHello();
8.    }
```

三、使用实例工厂方法实例化Bean，使用这种方式不能指定class属性，此时必须使用factory-bean属性来指定工厂Bean，factory-method属性指定实例化Bean的方法，而且使用实例工厂方法允许指定方法参数，方式和使用构造器方式一样，配置如下：

（1）实例工厂类代码（HelloApiInstanceFactory.java）如下：

java代码：

查看 复制到剪贴板 打印

```
1.
2.  package cn.javass.spring.chapter2;
3.  public class HelloApiInstanceFactory {
4.  public HelloApi newInstance(String message) {
5.          return new HelloImpl2(message);
6.    }
7.  }
8.
9.
```

（2）让我们在配置文件(resources/chapter2/instantiatingBean.xml)配置Bean定义：

java代码：

查看 复制到剪贴板 打印

```
1.  <!—1、定义实例工厂Bean -->
2.  <bean id="beanInstanceFactory"
3.  class="cn.javass.spring.chapter2.HelloApiInstanceFactory"/>
4.  <!—2、使用实例工厂Bean创建Bean -->
5.  <bean id="bean4"
6.  factory-bean="beanInstanceFactory"
7.      factory-method="newInstance">
8.   <constructor-arg index="0" value="Hello Spring!"></constructor-arg>
9.  </bean>
```

（3）测试代码InstantiatingBeanTest：

java代码：

查看 复制到剪贴板 打印

```
@Test
public void testInstantiatingBeanByInstanceFactory() {
//使用实例工厂方法
    BeanFactory beanFactory =
new ClassPathXmlApplicationContext("chapter2/instantiatingBean.xml");
    HelloApi bean4 = beanFactory.getBean("bean4", HelloApi.class);
    bean4.sayHello();
}
```

通过以上例子我们已经基本掌握了如何实例化Bean了，大家是否注意到？这三种方式只是配置不一样，从获取方式看完全一样，没有任何不同。这也是Spring IoC的魅力，Spring IoC帮你创建Bean，我们只管使用就可以了，是不是很简单。

2.3.5 小结

到此我们已经讲完了Spring IoC基础部分，包括IoC容器概念，如何实例化容器，Bean配置、命名及实例化，Bean获取等等。不知大家是否注意到到目前为止，我们只能通过简单的实例化Bean，没有涉及Bean之间关系。接下来一章让我们进入配置Bean之间关系章节，也就是依赖注入。

1.2 【第二章】 IoC 之 2.1 IoC基础 ——跟我学Spring3

发表时间: 2012-02-20 关键字: spring

2.1.1 IoC是什么

Ioc—Inversion of Control，即“控制反转”，不是什么技术，而是一种设计思想。在Java开发中，Ioc意味着将你设计好的对象交给容器控制，而不是传统的在你的对象内部直接控制。如何理解好Ioc呢？理解好Ioc的关键是要明确“谁控制谁，控制什么，为何是反转（有反转就应该有正转了），哪些方面反转了”，那我们来深入分析一下：

●**谁控制谁，控制什么：**传统Java SE程序设计，我们直接在对象内部通过new进行创建对象，是程序主动去创建依赖对象；而IoC是有专门一个容器来创建这些对象，即由Ioc容器来控制对象的创建；谁控制谁？当然是IoC 容器控制了对象；控制什么？那就是主要控制了外部资源获取（不只是对象包括比如文件等）。

●**为何是反转，哪些方面反转了：**有反转就有正转，传统应用程序是由我们自己在对象中主动控制去直接获取依赖对象，也就是正转；而反转则是由容器来帮忙创建及注入依赖对象；为何是反转？因为由容器帮我们查找及注入依赖对象，对象只是被动的接受依赖对象，所以是反转；哪些方面反转了？依赖对象的获取被反转了。

用图例说明一下，传统程序设计如图2-1，都是主动去创建相关对象然后再组合起来：

图2-1 传统应用程序示意图

当有了IoC/DI的容器后，在客户端类中不再主动去创建这些对象了，如图2-2所示:

图2-2有IoC/DI容器后程序结构示意图

1.1.2 IoC能做什么

IoC不是一种技术，只是一种思想，一个重要的面向对象编程的法则，它能指导我们如何设计出松耦合、更优良的程序。传统应用程序都是由我们在类内部主动创建依赖对象，从而导致类与类之间高耦合，难于测试；有了IoC容器后，把创建和查找依赖对象的控制权交给了容器，由容器进行注入组合对象，所以对象与对象之间是松散耦合，这样也方便测试，利于功能复用，更重要的是使得程序的整个体系结构变得非常灵活。

其实IoC对编程带来的最大改变不是从代码上，而是从思想上，发生了“主从换位”的变化。应用程序原本是老大，要获取什么资源都是主动出击，但是在IoC/DI思想中，应用程序就变成被动的了，被动的等待IoC容器来创建并注入它所需要的资源了。

IoC很好的体现了面向对象设计法则之一—— 好莱坞法则：“别找我们，我们找你”；即由IoC容器帮对象找相应的依赖对象并注入，而不是由对象主动去找。

2.1.3 IoC和DI

DI—Dependency Injection，即“依赖注入”：是组件之间依赖关系由容器在运行期决定，形象的说，即由容器动态的将某个依赖关系注入到组件之中。依赖注入的目的并非为软件系统带来更多功能，而是为了提升组件重用的频率，并为系统搭建一个灵活、可扩展的平台。通过依赖注入机制，我们只需要通过简单的配置，而无需任何代码就可指定目标需要的资源，完成自身的业务逻辑，而不需要关心具体的资源来自何处，由谁实现。

理解DI的关键是：“谁依赖谁，为什么需要依赖，谁注入谁，注入了什么”，那我们来深入分析一下：

●**谁依赖于谁：**当然是应用程序依赖于IoC容器；

●**为什么需要依赖：**应用程序需要IoC容器来提供对象需要的外部资源；

●**谁注入谁：**很明显是IoC容器注入应用程序某个对象，应用程序依赖的对象；

●**注入了什么：**就是注入某个对象所需要的外部资源（包括对象、资源、常量数据）。

IoC和DI由什么关系呢？其实它们是同一个概念的不同角度描述，由于控制反转概念比较含糊（可能只是理解为容器控制对象这一个层面，很难让人想到谁来维护对象关系），所以2004年大师级人物Martin Fowler又给出了一个新的名字：“依赖注入”，相对IoC 而言，**“依赖注入”明确描述了“被注入对象依赖IoC容器配置依赖对象”。**

注：如果想要更加深入的了解IoC和DI，请参考大师级人物Martin Fowler的一篇经典文章《Inversion of Control Containers and the Dependency Injection pattern》，原文地址：http://www.martinfowler.com/articles/injection.html。

转自【http://sishuok.com/forum/blogPost/list/2427.html】

1.3 【第二章】 IoC 之 2.2 IoC 容器基本原理 ——跟我学Spring3

发表时间: 2012-02-20 关键字: spring

2.2.1 IoC容器的概念

IoC容器就是具有依赖注入功能的容器，IoC容器负责实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。应用程序无需直接在代码中new相关的对象，应用程序由IoC容器进行组装。在Spring中BeanFactory是IoC容器的实际代表者。

Spring IoC容器如何知道哪些是它管理的对象呢？这就需要配置文件，Spring IoC容器通过读取配置文件中的配置元数据，通过元数据对应用中的各个对象进行实例化及装配。一般使用基于xml配置文件进行配置元数据，而且Spring与配置文件完全解耦的，可以使用其他任何可能的方式进行配置元数据，比如注解、基于java文件的、基于属性文件的配置都可以。

那Spring IoC容器管理的对象叫什么呢？

2.2.2 Bean的概念

由IoC容器管理的那些组成你应用程序的对象我们就叫它Bean， Bean就是由Spring容器初始化、装配及管理的对象，除此之外，bean就与应用程序中的其他对象没有什么区别了。那IoC怎样确定如何实例化Bean、管理Bean之间的依赖关系以及管理Bean呢？这就需要配置元数据，在Spring中由BeanDefinition代表，后边会详细介绍，配置元数据指定如何实例化Bean、如何组装Bean等。概念知道的差不多了，让我们来做个简单的例子。

2.2.3 Hello World

一、配置环境：

l **JDK安装：**安装最新的JDK，至少需要Java 1.5及以上环境；

l **开发工具：**SpringSource Tool Suite，简称STS，是个基于Eclipse的开发环境，用以构建Spring应用，其最新版开始支持Spring 3.0及OSGi开发工具，但由于其太庞大，很多功能不是我们所必需的所以我们选择Eclipse+ SpringSource Tool插件进行Spring应用开发；到eclipse官网下载最新的Eclipse，注意我们使用的是Eclipse IDE for Java EE Developers（eclipse-jee-helios-SR1）；

安装插件：启动Eclipse，选择Help->Install New Software，如图2-3所示

图2-3 安装

2、首先安装SpringSource Tool Suite插件依赖，如图2-4:

Name为：SpringSource Tool Suite Dependencies

Location为：http://dist.springsource.com/release/TOOLS/composite/e3.6

图2-4 安装

3、安装SpringSource Tool Suite插件，只需安装如图2-5所选中的就可以：

Name为：SpringSource Tool Suite

Location为：http://dist.springsource.com/release/TOOLS/update/e3.6

图2-4 安装

4、安装完毕，开始项目搭建吧。

Spring 依赖：本书使用spring-framework-3.0.5.RELEASE

spring-framework-3.0.5.RELEASE-with-docs.zip表示此压缩包带有文档的；

spring-framework-3.0.5.RELEASE-dependencies.zip表示此压缩包中是spring的依赖jar包，所以需要什么依赖从这里找就好了；

下载地址：http://www.springsource.org/download

二、开始Spring Hello World之旅

1、准备需要的jar包

核心jar包：从下载的spring-framework-3.0.5.RELEASE-with-docs.zip中dist目录查找如下jar包

org.springframework.asm-3.0.5.RELEASE.jar

org.springframework.core-3.0.5.RELEASE.jar

org.springframework.beans-3.0.5.RELEASE.jar

org.springframework.context-3.0.5.RELEASE.jar

org.springframework.expression-3.0.5.RELEASE.jar

依赖的jar包：从下载的spring-framework-3.0.5.RELEASE-dependencies.zip中查找如下依赖jar包

com.springsource.org.apache.log4j-1.2.15.jar

com.springsource.org.apache.commons.logging-1.1.1.jar

com.springsource.org.apache.commons.collections-3.2.1.jar

2、创建标准Java工程：

（1）选择“window”—> “Show View” —>“Package Explorer” ，使用包结构视图；

图2-5 包结构视图

（2）创建标准Java项目，选择“File”—>“New”—>“Other” ；然后在弹出来的对话框中选择“Java Project”创建标准Java项目；

图2-6 创建Java项目

图2-7 创建Java项目

图2-8 创建Java项目

（3）配置项目依赖库文件，右击项目选择“Properties”；然后在弹出的对话框中点击“Add JARS”在弹出的对话框中选择“lib”目录下的jar包；然后再点击“Add Library”，然后在弹出的对话框中选择“Junit”，选择“Junit4”；

图2-9 配置项目依赖库文件

图2-10 配置项目依赖库文件

图2-11 配置项目依赖库文件

（4）项目目录结构如下图所示，其中“src”用于存放java文件；“lib”用于存放jar文件；“resources”用于存放配置文件；

图2-12 项目目录结构

3、项目搭建好了，让我们来开发接口，此处我们只需实现打印“Hello World!”，所以我们定义一个“sayHello”接口，代码如下：

java代码：

查看 复制到剪贴板 打印

```
1. package cn.javass.spring.chapter2.helloworld;
2. public interface HelloApi {
3.     public void sayHello();
4. }
```

4、接口开发好了，让我们来通过实现接口来完成打印“Hello World!”功能；

java代码：

```
查看 复制到剪贴板 打印
1. package cn.javass.spring.chapter2.helloworld;
2. public class HelloImpl implements HelloApi {
3.         @Override
4.         public void sayHello() {
5.             System.out.println("Hello World!");
6.         }
7. }
```

5、接口和实现都开发好了，那如何使用Spring IoC容器来管理它们呢？这就需要配置文件，让IoC容器知道要管理哪些对象。让我们来看下配置文件chapter2/helloworld.xml（放到resources目录下）：

java代码：

```
查看 复制到剪贴板 打印
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:context="http://www.springframework.org/schema/context"
6. xsi:schemaLocation="
7. http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
8. http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.0.xsd">
9.   <!-- id 表示你这个组件的名字，class表示组件类 -->
10. <bean id="hello" class="cn.javass.spring.chapter2.helloworld.HelloImpl"></bean>
11. </beans>
```

6、现在万一具备，那如何获取IoC容器并完成我们需要的功能呢？首先应该实例化一个IoC容器，然后从容器中获取需要的对象，然后调用接口完成我们需要的功能，代码示例如下：

java代码：

```
查看 复制到剪贴板 打印
1. package cn.javass.spring.chapter2.helloworld;
2. import org.junit.Test;
3. import org.springframework.context.ApplicationContext;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5. public class HelloTest {
6.     @Test
7.     public void testHelloWorld() {
8.         //1、读取配置文件实例化一个IoC容器
9.         ApplicationContext context = new ClassPathXmlApplicationContext("helloworld.xml");
10.        //2、从容器中获取Bean，注意此处完全“面向接口编程，而不是面向实现”
11.        HelloApi helloApi = context.getBean("hello", HelloApi.class);
12.        //3、执行业务逻辑
13.        helloApi.sayHello();
14.    }
```

```
15. }
16.
```

7、自此一个完整的Spring Hello World已完成，是不是很简单，让我们深入理解下容器和Bean吧。

2.2.4 详解IoC容器

在Spring Ioc容器的代表就是org.springframework.beans包中的BeanFactory接口，BeanFactory接口提供了IoC容器最基本功能；而org.springframework.context包下的ApplicationContext接口扩展了BeanFactory，还提供了与Spring AOP集成、国际化处理、事件传播及提供不同层次的context实现 (如针对web应用的WebApplicationContext)。简单说， BeanFactory提供了IoC容器最基本功能，而 ApplicationContext 则增加了更多支持企业级功能支持。ApplicationContext完全继承BeanFactory，因而BeanFactory所具有的语义也适用于ApplicationContext。

容器实现一览：

• **XmlBeanFactory：**BeanFactory实现，提供基本的IoC容器功能，可以从classpath或文件系统等获取资源；

（1） File file = new File("fileSystemConfig.xml");

Resource resource = new FileSystemResource(file);

BeanFactory beanFactory = new XmlBeanFactory(resource);

（2）

Resource resource = new ClassPathResource("classpath.xml");

BeanFactory beanFactory = new XmlBeanFactory(resource);

• **ClassPathXmlApplicationContext：**ApplicationContext实现，从classpath获取配置文件；

BeanFactory beanFactory = new ClassPathXmlApplicationContext("classpath.xml");

• **FileSystemXmlApplicationContext：**ApplicationContext实现，从文件系统获取配置文件。

BeanFactory beanFactory = new FileSystemXmlApplicationContext("fileSystemConfig.xml");

具体代码请参考cn.javass.spring.chapter2.InstantiatingContainerTest.java。

ApplicationContext接口获取Bean方法简介：

• Object getBean(String name) 根据名称返回一个Bean，客户端需要自己进行类型转换；

• T getBean(String name, Class<T> requiredType) 根据名称和指定的类型返回一个Bean，客户端无需自己进行类型转换，如果类型转换失败，容器抛出异常；

• T getBean(Class<T> requiredType) 根据指定的类型返回一个Bean，客户端无需自己进行类型转换，如果没有或有多于一个Bean存在容器将抛出异常；

• Map<String, T> getBeansOfType(Class<T> type) 根据指定的类型返回一个键值为名字和值为Bean对象的 Map，如果没有Bean对象存在则返回空的Map。

让我们来看下IoC容器到底是如何工作。在此我们以xml配置方式来分析一下：

一、准备配置文件：就像前边Hello World配置文件一样，在配置文件中声明Bean定义也就是为Bean配置元数据。

二、由IoC容器进行解析元数据： IoC容器的Bean Reader读取并解析配置文件，根据定义生成BeanDefinition配置元数据对象，IoC容器根据BeanDefinition进行实例化、配置及组装Bean。

三、实例化IoC容器：由客户端实例化容器，获取需要的Bean。

整个过程是不是很简单，执行过程如图2-5，其实IoC容器很容易使用，主要是如何进行Bean定义。下一章我们详细介绍定义Bean。

图2-5 Spring Ioc容器

2.2.5 小结

除了测试程序的代码外，也就是程序入口，所有代码都没有出现Spring任何组件，而且所有我们写的代码没有实现框架拥有的接口，因而能非常容易的替换掉Spring，是不是非入侵。

客户端代码完全面向接口编程，无需知道实现类，可以通过修改配置文件来更换接口实现，客户端代码不需要任何修改。是不是低耦合。

如果在开发初期没有真正的实现，我们可以模拟一个实现来测试，不耦合代码，是不是很方便测试。

Bean之间几乎没有依赖关系，是不是很容易重用。

转自【http://sishuok.com/forum/blogPost/list/2428.html】

1.4 【第三章】 DI 之 3.1 DI的配置使用 ——跟我学spring3

发表时间: 2012-02-21 关键字: spring

3.1.1 依赖和依赖注入

传统应用程序设计中所说的依赖一般指“类之间的关系”，那先让我们复习一下类之间的关系：

泛化：表示类与类之间的继承关系、接口与接口之间的继承关系；

实现：表示类对接口的实现；

依赖：当类与类之间有使用关系时就属于依赖关系，不同于关联关系，依赖不具有“拥有关系”，而是一种“相识关系”，只在某个特定地方（比如某个方法体内）才有关系。

关联：表示类与类或类与接口之间的依赖关系，表现为“拥有关系”；具体到代码可以用实例变量来表示；

聚合：属于是关联的特殊情况，体现部分-整体关系，是一种弱拥有关系；整体和部分可以有不一样的生命周期；是一种弱关联；

组合：属于是关联的特殊情况，也体现了体现部分-整体关系，是一种强“拥有关系”；整体与部分有相同的生命周期，是一种强关联；

Spring IoC容器的依赖有两层含义：**Bean依赖容器**和**容器注入Bean的依赖资源**：

Bean依赖容器：也就是说Bean要依赖于容器，这里的依赖是指容器负责创建Bean并管理Bean的生命周期，正是由于由容器来控制创建Bean并注入依赖，也就是控制权被反转了，这也正是IoC名字的由来，**此处的有依赖是指Bean和容器之间的依赖关系。**

容器注入Bean的依赖资源：容器负责注入Bean的依赖资源，依赖资源可以是Bean、外部文件、常量数据等，在Java中都反映为对象，并且由容器负责组装Bean之间的依赖关系，**此处的依赖是指Bean之间的依赖关系，可以认为是传统类与类之间的“关联”、“聚合”、“组合”关系。**

为什么要应用依赖注入，应用依赖注入能给我们带来哪些好处呢？

动态替换Bean依赖对象，程序更灵活：替换Bean依赖对象，无需修改源文件：应用依赖注入后，由于可以采用配置文件方式实现，从而能随时动态的替换Bean的依赖对象，无需修改java源文件；

更好实践面向接口编程，代码更清晰：在Bean中只需指定依赖对象的接口，接口定义依赖对象完成的功能，通过容器注入依赖实现；

更好实践优先使用对象组合，而不是类继承：因为IoC容器采用注入依赖，也就是组合对象，从而更好的实践对象组合。

- 采用对象组合，Bean的功能可能由几个依赖Bean的功能组合而成，其Bean本身可能只提供少许功能或根本无任何功能，全部委托给依赖Bean，对象组合具有动态性，能更方便的替换掉依赖Bean，从而改变Bean功能；
- 而如果采用类继承，Bean没有依赖Bean，而是采用继承方式添加新功能，，而且功能是在编译时就确定了，不具有动态性，而且采用类继承导致Bean与子Bean之间高度耦合，难以复用。

增加Bean可复用性：依赖于对象组合，Bean更可复用且复用更简单；

降低Bean之间耦合：由于我们完全采用面向接口编程，在代码中没有直接引用Bean依赖实现，全部引用接口，而且不会出现显示的创建依赖对象代码，而且这些依赖是由容器来注入，很容易替换依赖实现类，从而降低Bean与依赖之间耦合；

代码结构更清晰：要应用依赖注入，代码结构要按照规约方式进行书写，从而更好的应用一些最佳实践，因此代码结构更清晰。

从以上我们可以看出，其实依赖注入只是一种装配对象的手段，设计的类结构才是基础，如果设计的类结构不支持依赖注入，Spring IoC容器也注入不了任何东西，从而从根本上说**“如何设计好类结构才是关键，依赖注入只是一种装配对象手段”。**

前边IoC一章我们已经了解了Bean依赖容器，那容器如何注入Bean的依赖资源，Spring IoC容器注入依赖资源主要有以下两种基本实现方式：

构造器注入：就是容器实例化Bean时注入那些依赖，通过在在Bean定义中指定构造器参数进行注入依赖，包括实例工厂方法参数注入依赖，但静态工厂方法参数不允许注入依赖；

setter注入：通过setter方法进行注入依赖；

方法注入：能通过配置方式替换掉Bean方法，也就是通过配置改变Bean方法 功能。

我们已经知道注入实现方式了，接下来让我们来看看具体配置吧。

3.1.2 构造器注入

使用构造器注入通过配置构造器参数实现，构造器参数就是依赖。除了构造器方式，还有静态工厂、实例工厂方法可以进行构造器注入。如图3-1所示：

图3-1 实例化

构造器注入可以根据参数索引注入、参数类型注入或Spring3支持的参数名注入，但参数名注入是有限制的，需要使用在编译程序时打开调试模式（即在编译时使用“javac –g:vars”在class文件中生成变量调试信息，默认是不包含变量调试信息的，从而能获取参数名字，否则获取不到参数名字）或在构造器上使用@ConstructorProperties（java.beans.ConstructorProperties）注解来指定参数名。

首先让我们准备测试构造器类HelloImpl3.java，该类只有一个包含两个参数的构造器：

java代码：

查看 复制到剪贴板 打印

```
1.  package cn.javass.spring.chapter3.helloworld;
2.  public class HelloImpl3 implements HelloApi {
3.      private String message;
4.  private int index;
5.  //@java.beans.ConstructorProperties({"message", "index"})
6.      public HelloImpl3(String message, int index) {
7.          this.message = message;
8.          this.index = index;
9.      }
10.     @Override
11.     public void sayHello() {
12.         System.out.println(index + ":" + message);
13.     }
14. }
```

一、根据参数索引注入，使用标签“<constructor-arg index="1" value="1"/>”来指定注入的依赖，其中“index”表示索引，从0开始，即第一个参数索引为0，“value”来指定注入的常量值，配置方式如下：

二、根据参数类型进行注入，使用标签“<constructor-arg type="java.lang.String" value="Hello World!"/>”来指定注入的依赖，其中“type”表示需要匹配的参数类型，可以是基本类型也可以是其他类型，如“int”、“java.lang.String”，“value”来指定注入的常量值，配置方式如下：

三、根据参数名进行注入，使用标签“<constructor-arg name="message" value="Hello World!"/>”来指定注入的依赖，其中“name”表示需要匹配的参数名字，“value”来指定注入的常量值，配置方式如下：

四、让我们来用具体的例子来看一下构造器注入怎么使用吧。

（1）首先准备Bean类，在此我们就使用“HelloImpl3”这个类。

（2）有了Bean类，接下来要进行Bean定义配置，我们需要配置三个Bean来完成上述三种依赖注入测试，其中Bean“byIndex”是通过索引注入依赖；Bean“byType”是根据类型进行注入依赖；Bean“byName”是根据参数名字进行注入依赖，具体配置文件（resources/chapter3/ constructorDependencyInject.xml）如下：

java代码：

查看 复制到剪贴板 打印

```
1.  <!-- 通过构造器参数索引方式依赖注入 -->
```

```
<bean id="byIndex" class="cn.javass.spring.chapter3.HelloImpl3">
<constructor-arg index="0" value="Hello World!"/>
  <constructor-arg index="1" value="1"/>
</bean>
<!-- 通过构造器参数类型方式依赖注入 -->
<bean id="byType" class="cn.javass.spring.chapter3.HelloImpl3">
  <constructor-arg type="java.lang.String" value="Hello World!"/>
  <constructor-arg type="int" value="2"/>
</bean>
<!-- 通过构造器参数名称方式依赖注入 -->
<bean id="byName" class="cn.javass.spring.chapter3.HelloImpl3">
  <constructor-arg name="message" value="Hello World!"/>
  <constructor-arg name="index" value="3"/>
</bean>
```

（3）配置完毕后，在测试之前，因为我们使用了通过构造器参数名字注入方式，请确保编译时class文件包含“变量信息”，具体查看编译时是否包含“变量调试信息”请右击项目，在弹出的对话框选择属性；然后在弹出的对话框选择“Java Compiler”条目，在“Class 文件 生成”框中选择“添加变量信息到Class文件（调试器使用）”，具体如图3-2：

图3-2 编译时打开“添加变量信息选项”

（4）接下来让我们测试一下配置是否工作，具体测试代码（cn.javass.spring.chapter3. DependencyInjectTest）如下：

java代码：

查看 复制到剪贴板 打印

```
@Test
public void testConstructorDependencyInjectTest() {
BeanFactory beanFactory =  new ClassPathXmlApplicationContext("chapter3/
constructorDependencyInject.xml");
//获取根据参数索引依赖注入的Bean
HelloApi byIndex = beanFactory.getBean("byIndex", HelloApi.class);
byIndex.sayHello();
//获取根据参数类型依赖注入的Bean
HelloApi byType = beanFactory.getBean("byType", HelloApi.class);
byType.sayHello();
//获取根据参数名字依赖注入的Bean
HelloApi byName = beanFactory.getBean("byName", HelloApi.class);
byName.sayHello();
}
```

通过以上测试我们已经会基本的构造器注入配置了，在测试通过参数名字注入时，除了可以使用以上方式，还可以通过在构造器上添加@java.beans.ConstructorProperties({"message", "index"})注解来指定参数名字，在HelloImpl3构造器上把注释掉的“ConstructorProperties”打开就可以了，这个就留给大家做练习，自己配置然后测试一下。

五、大家已经会了构造器注入，那让我们再看一下静态工厂方法注入和实例工厂注入吧，其实它们注入配置是完全一样，在此我们只示范一下静态工厂注入方式和实例工厂方式配置，测试就留给大家自己练习：

（1）静态工厂类

java代码：

查看 复制到剪贴板 打印

```
1. //静态工厂类
2. package cn.javass.spring.chapter3;
3. import cn.javass.spring.chapter2.helloworld.HelloApi;
4. public class DependencyInjectByStaticFactory {
5.     public static HelloApi newInstance(String message, int index) {
6.         return new HelloImpl3(message, index);
7.     }
8. }
```

静态工厂类Bean定义配置文件（chapter3/staticFactoryDependencyInject.xml）

java代码：

查看 复制到剪贴板 打印

```
1. <bean id="byIndex"
2. class="cn.javass.spring.chapter3.DependencyInjectByStaticFactory" factory-method="newInstance">
3. <constructor-arg index="0" value="Hello World!"/>
4. <constructor-arg index="1" value="1"/>
5. </bean>
6. <bean id="byType"
7. class="cn.javass.spring.chapter3.DependencyInjectByStaticFactory" factory-method="newInstance">
8. <constructor-arg type="java.lang.String" value="Hello World!"/>
9.     <constructor-arg type="int" value="2"/>
10. </bean>
11. <bean id="byName"
12. class="cn.javass.spring.chapter3.DependencyInjectByStaticFactory" factory-method="newInstance">
13.     <constructor-arg name="message" value="Hello World!"/>
14.     <constructor-arg name="index" value="3"/>
15. </bean>
```

（2）实例工厂类

java代码：

查看 复制到剪贴板 打印

```
1.  //实例工厂类
2.  package cn.javass.spring.chapter3;
3.  import cn.javass.spring.chapter2.helloworld.HelloApi;
4.  public class DependencyInjectByInstanceFactory {
5.      public HelloApi newInstance(String message, int index) {
6.          return new HelloImpl3(message, index);
7.      }
8.  }
```

实例工厂类Bean定义配置文件（chapter3/instanceFactoryDependencyInject.xml）

java代码：

查看 复制到剪贴板 打印

```
1.  <bean id="instanceFactory"
2.  class="cn.javass.spring.chapter3.DependencyInjectByInstanceFactory"/>
3.
4.  <bean id="byIndex"
5.  factory-bean="instanceFactory"  factory-method="newInstance">
6.      <constructor-arg index="0" value="Hello World!"/>
7.          <constructor-arg index="1" value="1"/>
8.  </bean>
9.  <bean id="byType"
10. factory-bean="instanceFactory" factory-method="newInstance">
11. <constructor-arg type="java.lang.String" value="Hello World!"/>
12. <constructor-arg type="int" value="2"/>
13. </bean>
14. <bean id="byName"
15. factory-bean="instanceFactory" factory-method="newInstance">
16. <constructor-arg name="message" value="Hello World!"/>
17. <constructor-arg name="index" value="3"/>
18. </bean>
```

（3）测试代码和构造器方式完全一样，只是配置文件不一样，大家只需把测试文件改一下就可以了。还有一点需要大家注意就是静态工厂方式和实例工厂方式根据参数名字注入的方式只支持通过在class文件中添加“变量调试信息”方式才能运行，ConstructorProperties注解方式不能工作，它只对构造器方式起作用，**不建议使用根据参数名进行构造器注入。**

3.1.3 setter注入

setter注入，是通过在通过构造器、静态工厂或实例工厂实例好Bean后，通过调用Bean类的setter方法进行注入依赖，如图3-3所示：

图3-3 setter注入方式

setter注入方式只有一种根据setter名字进行注入：

知道配置方式了，接下来先让我们来做个简单例子吧。

（1）准备测试类HelloImpl4，需要两个setter方法“setMessage”和“setIndex”：

java代码：

查看 复制到剪贴板 打印

```
1.  package cn.javass.spring.chapter3;
2.  import cn.javass.spring.chapter2.helloworld.HelloApi;
3.  public class HelloImpl4 implements HelloApi {
4.      private String message;
5.      private int index;
6.  //setter方法
7.      public void setMessage(String message) {
8.          this.message = message;
9.      }
10.     public void setIndex(int index) {
11.         this.index = index;
12.     }
13.     @Override
14.     public void sayHello() {
15.         System.out.println(index + ":" + message);
16.     }
17. }
```

（2）配置Bean定义，具体配置文件（resources/chapter3/setterDependencyInject.xml）片段如下：

java代码：

查看 复制到剪贴板 打印

```
1.  <!-- 通过setter方式进行依赖注入 -->
2.      <bean id="bean" class="cn.javass.spring.chapter3.HelloImpl4">
3.          <property name="message" value="Hello World!"/>
4.          <property name="index">
5.              <value>1</value>
6.          </property>
7.      </bean>
```

（3）该写测试进行测试一下是否满足能工作了，其实测试代码一点没变，变的是配置：

java代码：

查看 复制到剪贴板 打印

```
@Test
public void testSetterDependencyInject() {
    BeanFactory beanFactory =
new ClassPathXmlApplicationContext("chapter3/setterDependencyInject.xml");
   HelloApi bean = beanFactory.getBean("bean", HelloApi.class);
    bean.sayHello();
}
```

知道如何配置了，但Spring如何知道setter方法？如何将值注入进去的呢？其实方法名是要遵守约定的，setter注入的方法名要遵循“JavaBean getter/setter 方法命名约定”：

JavaBean：是本质就是一个POJO类，但具有一下限制：

该类必须要**有公共的无参构造器**，如public HelloImpl4() {}；

属性为private访问级别，不建议public，如private String message;

属性必要时通过一组setter（修改器）和getter（访问器）方法来访问；

setter方法，以“set” 开头，后跟首字母大写的属性名，如“setMesssage”,简单属性一般只有一个方法参数，方法返回值通常为“void”；

getter方法，一般属性以“get”开头，对于boolean类型一般以“is”开头，后跟首字母大写的属性名，如“getMesssage”，“isOk”；

还有一些其他特殊情况，**比如属性有连续两个大写字母开头，如“URL”,则setter/getter方法为：“setURL”和“getURL”**，其他一些特殊情况请参看“Java Bean”命名规范。

3.1.4 注入常量

注入常量是依赖注入中最简单的。配置方式如下所示：

java代码：

```
查看 复制到剪贴板 打印
1. <property name="message" value="Hello World!"/>
2. 或
3. <property name="index"><value>1</value></property><span class="Apple-style-span" style="font-size: 14px; white-space: normal; background-color: #ffffff;"> </span>
```

以上两种方式都可以，从配置来看第一种更简洁。注意此处“value”中指定的全是字符串，由Spring容器将此字符串转换成属性所需要的类型，如果转换出错，将抛出相应的异常。

Spring容器目前能对各种基本类型把配置的String参数转换为需要的类型。

注：Spring类型转换系统对于boolean类型进行了容错处理，除了可以使用“true/false”标准的Java值进行注入，还能使用“yes/no”、“on/off”、“1/0”来代表“真/假”，所以大家在学习或工作中遇到这种类似问题不要觉得是人家配置错了，而是Spring容错做的非常好。

java代码：

```
查看 复制到剪贴板 打印
1. 测试类
2. public class BooleanTestBean {
3.     private boolean success;
4.     public void setSuccess(boolean success) {
5.         this.success = success;
6.     }
7.     public boolean isSuccess() {
8.         return success;
9.     }
10. }
11. 配置文件（chapter3/booleanInject.xml）片段：
12. <!-- boolean参数值可以用on/off -->
13. <bean id="bean2" class="cn.javass.spring.chapter3.bean.BooleanTestBean">
14.     <property name="success" value="on"/>
15. </bean>
16. <!-- boolean参数值可以用yes/no -->
17. <bean id="bean3" class="cn.javass.spring.chapter3.bean.BooleanTestBean">
18.     <property name="success" value="yes"/>
19. </bean>
20. <!-- boolean参数值可以用1/0 -->
21. <bean id="bean4" class="cn.javass.spring.chapter3.bean.BooleanTestBean">
22.     <property name="success" value="1"/>
23. </bean>
```

3.1.5 注入Bean ID

用于注入Bean的ID，ID是一个常量不是引用，且类似于注入常量，但提供错误验证功能，配置方式如下所示：

java代码：

```
查看 复制到剪贴板 打印
1. <property name="id"><idref bean="bean1"/></property>
```

java代码：

```
查看 复制到剪贴板 打印
1. <property name="id"><idref local="bean2"/></property>
```

两种方式都可以，上述配置本质上在运行时等于如下方式

java代码：

```
查看 复制到剪贴板 打印
1. <bean id="bean1" class="....."/>
2. <bean id="idrefBean1" class=".....">
3. <property name="id" value ="bean1"/>
4. </bean>
```

第一种方式可以在容器初始化时校验被引用的Bean是否存在，如果不存在将抛出异常，而第二种方式只有在Bean实际使用时才能发现传入的Bean的ID是否正确，可能发生不可预料的错误。因此如果想注入Bean的ID，推荐使用第一种方式。

接下来学习一下如何使用吧：

首先定义测试Bean：

java代码：

```
查看 复制到剪贴板 打印
1. package cn.javass.spring.chapter3.bean
2. public class IdRefTestBean {
3.     private String id;
4.     public String getId() {
5.         return id;
6.     }
7.     public void setId(String id) {
```

```
        this.id = id;
    }
}
```

其次定义配置文件（chapter3/idRefInject.xml）：

java代码：

```
查看 复制到剪贴板 打印
<bean id="bean1" class="java.lang.String">
<constructor-arg index="0" value="test"/>
</bean>
<bean id="bean2" class="java.lang.String">
    <constructor-arg index="0" value="test"/>
</bean>
```

java代码：

```
查看 复制到剪贴板 打印
<bean id="idrefBean1" class="cn.javass.spring.chapter3.bean.IdRefTestBean">
        <property name="id"><idref bean="bean1"/></property>
</bean>
<bean id="idrefBean2" class="cn.javass.spring.chapter3.bean.IdRefTestBean">
    <property name="id"><idref local="bean2"/></property>
</bean>
```

从配置中可以看出，注入的Bean的ID是一个java.lang.String类型，即字符串类型，因此注入的同样是常量，只是具有校验功能。

<idref bean="....."/>将在容器初始化时校验注入的ID对于的Bean是否存在，如果不存在将抛出异常。

<idref local="....."/>将在XML解析时校验注入的ID对于的Bean在当前配置文件中是否存在，如果不存在将抛出异常，它不同于<idref bean="....."/>是校验发生在XML解析式而非容器初始化时，且只检查当前配置文件中是否存在相应的Bean。

3.1.6 注入集合、数组和字典

Spring不仅能注入简单类型数据，还能注入集合（Collection、无序集合Set、有序集合List）类型、数组(Array)类型、字典(Map)类型数据、Properties类型数据，接下来就让我们一个个看看如何注入这些数据类型的数据。

一、注入集合类型：包括Collection类型、Set类型、List类型数据：

（1）List类型：需要使用<list>标签来配置注入，其具体配置如下：

让我们来写个测试来练习一下吧：

准备测试类：

java代码：

```
查看 复制到剪贴板 打印
1.  package cn.javass.spring.chapter3.bean;
2.  import java.util.List;
3.  public class ListTestBean {
4.      private List<String> values;
5.      public List<String> getValues() {
6.          return values;
7.      }
8.      public void setValues(List<String> values) {
9.          this.values = values;
10.     }
11. }
```

进行Bean定义，在配置文件（resources/chapter3/listInject.xml）中配置list注入：

java代码：

```
查看 复制到剪贴板 打印
1.  <bean id="listBean" class="cn.javass.spring.chapter3.bean.ListTestBean">
2.      <property name="values">
3.          <list>
4.              <value>1</value>
5.              <value>2</value>
6.              <value>3</value>
7.          </list>
8.      </property>
9.  </bean>
```

测试代码：

java代码：

```
查看 复制到剪贴板 打印
1. @Test
2. public void testListInject() {
3.    BeanFactory beanFactory =
4. new ClassPathXmlApplicationContext("chapter3/listInject.xml");
5. ListTestBean listBean = beanFactory.getBean("listBean", ListTestBean.class);
6. System.out.println(listBean.getValues().size());
7. Assert.assertEquals(3, listBean.getValues().size());
8. }
```

（2）Set类型：需要使用<set>标签来配置注入，其配置参数及含义和<lsit>标签完全一样，在此就不阐述了：

准备测试类：

java代码：

```
查看 复制到剪贴板 打印
1. package cn.javass.spring.chapter3.bean;
2. import java.util.Collection;
3. public class CollectionTestBean {
4.    private Collection<String> values;
5.    public void setValues(Collection<String> values) {
6.       this.values = values;
7.    }
8.    public Collection<String> getValues() {
9.       return values;
10.   }
11. }
```

进行Bean定义，在配置文件（resources/chapter3/listInject.xml）中配置list注入：

java代码：

```
查看 复制到剪贴板 打印
1. <bean id="setBean" class="cn.javass.spring.chapter3.bean.SetTestBean">
2. <property name="values">
3. <set>
4. <value>1</value>
5. <value>2</value>
6. <value>3</value>
7. </set>
8. </property>
9. </bean>
```

具体测试代码就不写了，和listBean测试代码完全一样。

（2）Collection类型：因为Collection类型是Set和List类型的基类型，所以使用<set>或<list>标签都可以进行注入，配置方式完全和以上配置方式一样，只是将测试类属性改成“Collection”类型，如果配置有问题，可参考cn.javass.spring.chapter3.DependencyInjectTest测试类中的testCollectionInject测试方法中的代码。

二、**注入数组类型**：需要使用<array>标签来配置注入，其中标签属性“value-type”和“merge”和<list>标签含义完全一样，具体配置如下：

如果练习时遇到配置问题，可以参考“cn.javass.spring.chapter3.DependencyInjectTest”测试类中的testArrayInject测试方法中的代码。

三、注入字典（Map）类型：字典类型是包含键值对数据的数据结构，需要使用<map>标签来配置注入，其属性“key-type”和“value-type”分别指定“键”和“值”的数据类型，其含义和<list>标签的“value-type”含义一样，在此就不罗嗦了，并使用<key>子标签来指定键数据，<value>子标签来指定键对应的值数据，具体配置如下：

如果练习时遇到配置问题，可以参考“cn.javass.spring.chapter3.DependencyInjectTest”测试类中的testMapInject测试方法中的代码。

四、**Properties注入**：Spring能注入java.util.Properties类型数据，需要使用<props>标签来配置注入，键和值类型必须是String，不能变，子标签<prop key=”键” >值</prop>来指定键值对，具体配置如下：

如果练习时遇到配置问题，可以参考cn.javass.spring.chapter3.DependencyInjectTest测试类中的testPropertiesInject测试方法中的代码。

到此我们已经把简单类型及集合类型介绍完了，大家可能会问怎么没见注入“Bean之间关系”的例子呢？接下来就让我们来讲解配置Bean之间依赖关系，也就是注入依赖Bean。

3.1.7 引用其它Bean

上边章节已经介绍了注入常量、集合等基本数据类型和集合数据类型，本小节将介绍注入依赖Bean及注入内部Bean。

引用其他Bean的步骤与注入常量的步骤一样，可以通过构造器注入及setter注入引用其他Bean，只是引用其他Bean的注入配置稍微变化了一下：可以将“<constructor-arg index="0" value="Hello World!"/>”和“<property name="message" value="Hello World!"/>”中的value属性替换成bean属性，其中bean属性指定配置文件中的其

他Bean的id或别名。另一种是把<value>标签替换为<.ref bean=”beanName”>，bean属性也是指定配置文件中的其他Bean的id或别名。那让我们看一下具体配置吧：

一、构造器注入方式：

（1）通过” <constructor-arg>”标签的ref属性来引用其他Bean，这是最简化的配置：

（2）通过” <constructor-arg>”标签的子<ref>标签来引用其他Bean，使用bean属性来指定引用的Bean：

二、setter注入方式：

（1）通过” <property>”标签的ref属性来引用其他Bean，这是最简化的配置：

（2）通过” <property>”标签的子<ref>标签来引用其他Bean，使用bean属性来指定引用的Bean：

三、接下来让我们用个具体例子来讲解一下具体使用吧：

（1）首先让我们定义测试引用Bean的类，在此我们可以使用原有的HelloApi实现，然后再定义一个装饰器来引用其他Bean，具体装饰类如下：

java代码：

查看 复制到剪贴板 打印

```
1.  package cn.javass.spring.chapter3.bean;
2.  import cn.javass.spring.chapter2.helloworld.HelloApi;
3.  public class HelloApiDecorator implements HelloApi {
4.  private HelloApi helloApi;
5.  //空参构造器
6.     public HelloApiDecorator() {
7.  }
8.  //有参构造器
9.     public HelloApiDecorator(HelloApi helloApi) {
10.        this.helloApi = helloApi;
11. }
12. public void setHelloApi(HelloApi helloApi) {
13.        this.helloApi = helloApi;
14.    }
15.    @Override
16.    public void sayHello() {
17.        System.out.println("==========装饰一下==========");
18.        helloApi.sayHello();
19.        System.out.println("==========装饰一下==========");
20.    }
```

```
21. }
```

（2）定义好了测试引用Bean接下来该在配置文件(resources/chapter3/beanInject.xml)进行配置Bean定义了，在此将演示通过构造器及setter方法方式注入依赖Bean：

java代码：

查看 复制到剪贴板 打印

```
1.  <!-- 定义依赖Bean -->
2.  <bean id="helloApi" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
3.  <!-- 通过构造器注入 -->
4.  <bean id="bean1" class="cn.javass.spring.chapter3.bean.HelloApiDecorator">
5.  <constructor-arg index="0" ref="helloApi"/>
6.  </bean>
7.  <!-- 通过构造器注入 -->
8.  <bean id="bean2" class="cn.javass.spring.chapter3.bean.HelloApiDecorator">
9.     <property name="helloApi"><ref bean=" helloApi"/></property>
10. </bean>
```

（3）测试一下吧，测试代码(cn.javass.spring.chapter3.DependencyInjectTest)片段如下：

java代码：

查看 复制到剪贴板 打印

```
1.  @Test
2.  public void testBeanInject() {
3.     BeanFactory beanFactory =
4.  new ClassPathXmlApplicationContext("chapter3/beanInject.xml");
5.     //通过构造器方式注入
6.     HelloApi bean1 = beanFactory.getBean("bean1", HelloApi.class);
7.     bean1.sayHello();
8.     //通过setter方式注入
9.     HelloApi bean2 = beanFactory.getBean("bean2", HelloApi.class);
10.    bean2.sayHello();
11. }
```

四、其他引用方式：除了最基本配置方式以外，Spring还提供了另外两种更高级的配置方式，<ref local=”” />和<ref parent=”” />：

（1）<ref local=”” />配置方式：用于引用通过<bean id=” beanName” >方式中通过id属性指定的Bean，它能利用XML解析器的验证功能在读取配置文件时来验证引用的Bean是否存在。因此如果在当前配置文件中有相互引用的Bean可以采用<ref local>方式从而如果配置错误能在开发调试时就发现错误。

如果引用一个在当前配置文件中不存在的Bean将抛出如下异常：

org.springframework.beans.factory.xml.XmlBeanDefinitionStoreException: Line21 inXML document from class path resource [chapter3/beanInject2.xml] is invalid; nested exception is org.xml.sax.SAXParseException: cvc-id.1: There is no ID/IDREF binding for IDREF 'helloApi'.

<ref local>具体配置方式如下：

（2）<ref parent=”“ />配置方式：用于引用父容器中的Bean，不会引用当前容器中的Bean，当然父容器中的Bean和当前容器的Bean是可以重名的，获取顺序是先查找当前容器中的Bean，如果找不到再从父容器找。具体配置方式如下：

接下来让我们用个例子演示一下<ref local>和<ref parent>的配置过程：

首先还是准备测试类，在此我们就使用以前写好的HelloApiDecorator和HelloImpl4类；其次进行Bean定义，其中当前容器bean1引用本地的”helloApi“，而”bean2”将引用父容器的”helloApi“，配置如下：

java代码：

查看 复制到剪贴板 打印

```
<!-- sources/chapter3/parentBeanInject.xml表示父容器配置-->
<!--注意此处可能子容器也定义一个该Bean-->
<bean id="helloApi" class="cn.javass.spring.chapter3.HelloImpl4">
<property name="index" value="1"/>
<property name="message" value="Hello Parent!"/>
</bean>
```

java代码：

查看 复制到剪贴板 打印

```
<!-- sources/chapter3/localBeanInject.xml表示当前容器配置-->
<!-- 注意父容器中也定义了id 为 helloApi的Bean -->
<bean id="helloApi" class="cn.javass.spring.chapter3.HelloImpl4">
<property name="index" value="1"/>
   <property name="message" value="Hello Local!"/>
</bean>
<!-- 通过local注入 -->
<bean id="bean1" class="cn.javass.spring.chapter3.bean.HelloApiDecorator">
<constructor-arg index="0"><ref local="helloApi"/></constructor-arg>
</bean>
<!-- 通过parent注入 -->
<bean id="bean2" class="cn.javass.spring.chapter3.bean.HelloApiDecorator">
<property name="helloApi"><ref parent="helloApi"/></property>
</bean>
```

（3）写测试类测试一下吧，具体代码片段如下：

java代码：

查看 复制到剪贴板 打印

```
@Test
public void testLocalAndparentBeanInject() {
//初始化父容器
ApplicationContext parentBeanContext =
new ClassPathXmlApplicationContext("chapter3/parentBeanInject.xml");
//初始化当前容器
ApplicationContext beanContext = new ClassPathXmlApplicationContext(
new String[] {"chapter3/localBeanInject.xml"}, parentBeanContext);
    HelloApi bean1 = beanContext.getBean("bean1", HelloApi.class);
    bean1.sayHello();//该Bean引用local bean
HelloApi bean2 = beanContext.getBean("bean2", HelloApi.class);
bean2.sayHello();//该Bean引用parent bean
}
```

“bean1”将输出“Hello Local!”表示引用当前容器的Bean，”bean2”将输出“Hello Paren!”，表示引用父容器的Bean，如配置有问题请参考cn.javass.spring.chapter3.DependencyInjectTest中的testLocalAndparentBeanInject测试方法。

3.1.8 内部Bean定义

内部Bean就是在<property>或<constructor-arg>内通过<bean>标签定义的Bean，该Bean不管是否指定id或name，该Bean都会有唯一的匿名标识符，而且不能指定别名，该内部Bean对其他外部Bean不可见，具体配置如下：

（1）让我们写个例子测试一下吧，具体配置文件如下：

java代码：

查看 复制到剪贴板 打印

```
<bean id="bean" class="cn.javass.spring.chapter3.bean.HelloApiDecorator">
<property name="helloApi">
<bean id="helloApi" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
</property>
</bean>
```

（2）测试代码（cn.javass.spring.chapter3.DependencyInjectTest.testInnerBeanInject）：

java代码：

查看 复制到剪贴板 打印

```
@Test
public void testInnerBeanInject() {
ApplicationContext context =
new ClassPathXmlApplicationContext("chapter3/innerBeanInject.xml");
HelloApi bean = context.getBean("bean", HelloApi.class);
bean.sayHello();
}
```

3.1.9 处理null值

Spring通过<value>标签或value属性注入常量值，所有注入的数据都是字符串，那如何注入null值呢？通过“null”值吗？当然不是因为如果注入“null”则认为是字符串。Spring通过<null/>标签注入null值。即可以采用如下配置方式：

3.1.10 对象图导航注入支持

所谓对象图导航是指类似a.b.c这种点缀访问形式的访问或修改值。Spring支持对象图导航方式依赖注入。对象图导航依赖注入有一个限制就是比如a.b.c对象导航图注入中a和b必须为非null值才能注入c，否则将抛出空指针异常。

Spring不仅支持对象的导航，还支持数组、列表、字典、Properties数据类型的导航，对Set数据类型无法支持，因为无法导航。

数组和列表数据类型可以用array[0]、list[1]导航，注意“[]”里的必须是数字，因为是按照索引进行导航，对于数组类型注意不要数组越界错误。

字典Map数据类型可以使用map[1]、map[str]进行导航，其中“[]”里的是基本类型，无法放置引用类型。

让我们来练习一下吧。首先准备测试类，在此我们需要三个测试类，以便实现对象图导航功能演示：

NavigationC类用于打印测试代码，从而观察配置是否正确；具体类如下所示：

java代码：

查看 复制到剪贴板 打印

```
package cn.javass.spring.chapter3.bean;
public class NavigationC {
    public void sayNavigation() {
        System.out.println("===navigation c");
    }
```

```
6. }
```

NavigationB类，包含对象和列表、Properties、数组字典数据类型导航，而且这些复合数据类型保存的条目都是对象，正好练习一下如何往复合数据类型中注入对象依赖。具体类如下所示：

java代码：

查看 复制到剪贴板 打印

```
1.  package cn.javass.spring.chapter3.bean;
2.  import java.util.List;
3.  import java.util.Map;
4.  import java.util.Properties;
5.  public class NavigationB {
6.      private NavigationC navigationC;
7.      private List<NavigationC> list;
8.      private Properties properties;
9.      private NavigationC[] array = new NavigationC[1];
10.     private Map<String, NavigationC> map;
11.    //由于setter和getter方法占用太多空间，故省略，大家自己实现吧
12. }
```

NavigationA类是我们的前端类，通过对它的导航进行注入值，具体代码如下：

java代码：

查看 复制到剪贴板 打印

```
1.  package cn.javass.spring.chapter3.bean;
2.  public class NavigationA {
3.      private NavigationB navigationB;
4.      public void setNavigationB(NavigationB navigationB) {
5.          this.navigationB = navigationB;
6.      }
7.      public NavigationB getNavigationB() {
8.          return navigationB;
9.      }
10. }
```

接下来该进行Bean定义配置（resources/chapter3/navigationBeanInject.xml）了，首先让我们配置一下需要被导航的数据，NavigationC和NavigationB类，其中配置NavigationB时注意要确保比如array字段不为空值，这就需要或者在代码中赋值如“NavigationC[] array = new NavigationC[1];”，或者通过配置文件注入如“<list></list>”注入一个不包含条目的列表。具体配置如下：

java代码：

```
查看 复制到剪贴板 打印
<bean id="c" class="cn.javass.spring.chapter3.bean.NavigationC"/>
<bean id="b" class="cn.javass.spring.chapter3.bean.NavigationB">
<property name="list"><list></list></property>
   <property name="map"><map></map></property>
   <property name="properties"><props></props></property>
</bean>
```

配置完需要被导航的Bean定义了，该来配置NavigationA导航Bean了，在此需要注意，由于“navigationB”属性为空值，在此需要首先注入“navigationB”值；还有对于数组导航不能越界否则报错；具体配置如下：

java代码：

```
查看 复制到剪贴板 打印
<bean id="a" class="cn.javass.spring.chapter3.bean.NavigationA">
<!-- 首先注入navigatiionB 确保它非空 -->
<property name="navigationB" ref="b"/>
<!-- 对象图导航注入 -->
<property name="navigationB.navigationC" ref="c"/>
<!-- 注入列表数据类型数据 -->
<property name="navigationB.list[0]" ref="c"/>
<!-- 注入map类型数据 -->
<property name="navigationB.map[key]" ref="c"/>
<!-- 注入properties类型数据 -->
<property name="navigationB.properties[0]" ref="c"/>
<!-- 注入properties类型数据 -->
<property name="navigationB.properties[1]" ref="c"/>
<!-- 注入数组类型数据 ，注意不要越界-->
<property name="navigationB.array[0]" ref="c"/>
</bean>
```

配置完毕，具体测试代码在cn.javass.spring.chapter3. DependencyInjectTest，让我们看下测试代码吧：

java代码：

```
查看 复制到剪贴板 打印
//对象图导航
@Test
public void testNavigationBeanInject() {
ApplicationContext context =
new ClassPathXmlApplicationContext("chapter3/navigationBeanInject.xml");
NavigationA navigationA = context.getBean("a", NavigationA.class);
navigationA.getNavigationB().getNavigationC().sayNavigation();
navigationA.getNavigationB().getList().get(0).sayNavigation();
navigationA.getNavigationB().getMap().get("key").sayNavigation();
```

```
navigationA.getNavigationB().getArray()[0].sayNavigation();
((NavigationC)navigationA.getNavigationB().getProperties().get("1"))
.sayNavigation();
}
```

测试完毕，应该输出5个“===navigation c”，是不是很简单，注意这种方式是不推荐使用的，了解一下就够了，最好使用3.1.5一节使用的配置方式。

3.1.11配置简写

让我们来总结一下依赖注入配置及简写形式，其实我们已经在以上部分穿插着进行简化配置了：

一、构造器注入：

1）常量值

简写：<constructor-arg index="0" value="常量"/>

全写：<constructor-arg index="0"><value>常量</value></constructor-arg>

2）引用

简写：<constructor-arg index="0" ref="引用"/>

全写：<constructor-arg index="0"><ref bean="引用"/></constructor-arg>

二、setter注入：

1）常量值

简写：<property name="message" value="常量"/>

全写：<property name="message"><value>常量</value></ property>

2）引用

简写：<property name="message" ref="引用"/>

全写：<property name="message"><ref bean="引用"/></ property>

3）数组：<array>没有简写形式

4）列表：<list>没有简写形式

5）集合：<set>没有简写形式

6）字典

简写：<map>

<entry key="键常量" value="值常量"/>

<entry key-ref="键引用" value-ref="值引用"/>

</map>

全写：<map>

<entry><key><value>键常量</value></key><value>值常量</value></entry>

<entry><key><ref bean="键引用"/></key><ref bean="值引用"/></entry>

</map>

7）Properties：没有简写形式

三、使用p命名空间简化setter注入：

使用p命名空间来简化setter注入，具体使用如下：

java代码：

查看 复制到剪贴板 打印

```
<?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-3.0.xsd">
<bean id="bean1" class="java.lang.String">
      <constructor-arg index="0" value="test"/>
   </bean>
<bean id="idrefBean1" class="cn.javass.spring.chapter3.bean.IdRefTestBean"
p:id="value"/>
<bean id="idrefBean2" class="cn.javass.spring.chapter3.bean.IdRefTestBean"
p:id-ref="bean1"/>
</beans>
```

- **xmlns:p="http://www.springframework.org/schema/p" ：首先指定p命名空间；**
- **<bean id="....." class="....." p:id="value"/>** ：常量setter注入方式，其等价于<property name="id" value="value"/>；

。

- **<bean id="....." class="....." p:id-ref="bean1"/>** ：引用setter注入方式，其等价于<property name="id" ref="bean1"/>。

1.5 【第三章】 DI 之 3.2 循环依赖 ——跟我学spring3

发表时间: 2012-02-21 关键字: spring

3.2.1 什么是循环依赖

循环依赖就是循环引用，就是两个或多个Bean相互之间的持有对方，比如CircleA引用CircleB，CircleB引用CircleC，CircleC引用CircleA，则它们最终反映为一个环。此处不是循环调用，循环调用是方法之间的环调用。如图3-5所示：

图3-5 循环引用

循环调用是无法解决的，除非有终结条件，否则就是死循环，最终导致内存溢出错误。

Spring容器循环依赖包括构造器循环依赖和setter循环依赖，那Spring容器如何解决循环依赖呢？首先让我们来定义循环引用类：

java代码：

查看 复制到剪贴板 打印

```
1.  package cn.javass.spring.chapter3.bean;
2.  public class CircleA {
3.      private CircleB circleB;
4.      public CircleA() {
5.      }
6.      public CircleA(CircleB circleB) {
7.          this.circleB = circleB;
8.      }
9.  public void setCircleB(CircleB circleB)
10. {
11.         this.circleB = circleB;
12.     }
13. public void a() {
14.    circleB.b();
15. }
16. }
```

java代码：

查看 复制到剪贴板 打印

```
1.  package cn.javass.spring.chapter3.bean;
2.  public class CircleB {
3.      private CircleC circleC;
4.      public CircleB() {
5.      }
6.      public CircleB(CircleC circleC) {
7.          this.circleC = circleC;
8.      }
9.  public void setCircleC(CircleC circleC)
10. {
```

```
        this.circleC = circleC;
    }
    public void b() {
        circleC.c();
    }
}
```

java代码：

查看 复制到剪贴板 打印

```
package cn.javass.spring.chapter3.bean;
public class CircleC {
    private CircleC circleC;
    public CircleB() {
    }
    public CircleB(CircleC circleC) {
        this.circleC = circleC;
    }
public void setCircleC(CircleC circleC)
{
        this.circleC = circleC;
    }
    public void b() {
        circleC.c();
    }
}
```

3.2.2 Spring如何解决循环依赖

一、构造器循环依赖：表示通过构造器注入构成的循环依赖，此依赖是无法解决的，只能抛出BeanCurrentlyInCreationException异常表示循环依赖。

如在创建CircleA类时，构造器需要CircleB类，那将去创建CircleB，在创建CircleB类时又发现需要CircleC类，则又去创建CircleC，最终在创建CircleC时发现又需要CircleA；从而形成一个环，没办法创建。

Spring容器将每一个正在创建的Bean 标识符放在一个“当前创建Bean池”中，Bean标识符在创建过程中将一直保持在这个池中，因此如果在创建Bean过程中发现自己已经在“当前创建Bean池”里时将抛出BeanCurrentlyInCreationException异常表示循环依赖；而对于创建完毕的Bean将从“当前创建Bean池”中清除掉。

1）首先让我们看一下配置文件（chapter3/circleInjectByConstructor.xml）：

java代码：

查看 复制到剪贴板 打印

```

<bean id="circleA" class="cn.javass.spring.chapter3.bean.CircleA">
<constructor-arg index="0" ref="circleB"/>
</bean>
<bean id="circleB" class="cn.javass.spring.chapter3.bean.CircleB">
```

```
<constructor-arg index="0" ref="circleC"/>
</bean>
<bean id="circleC" class="cn.javass.spring.chapter3.bean.CircleC">
<constructor-arg index="0" ref="circleA"/>
</bean>
```

2）写段测试代码（cn.javass.spring.chapter3.CircleTest）测试一下吧：

java代码：

```
@Test(expected = BeanCurrentlyInCreationException.class)
public void testCircleByConstructor() throws Throwable {
try {
    new ClassPathXmlApplicationContext("chapter3/circleInjectByConstructor.xml");
  }
  catch (Exception e) {
    //因为要在创建circle3时抛出；
    Throwable e1 = e.getCause().getCause().getCause();
    throw e1;
  }
}
```

让我们分析一下吧：

1、Spring容器创建“circleA” Bean，首先去“当前创建Bean池”查找是否当前Bean正在创建，如果没发现，则继续准备其需要的构造器参数“circleB”，并将“circleA” 标识符放到“当前创建Bean池”；

2、Spring容器创建“circleB” Bean，首先去“当前创建Bean池”查找是否当前Bean正在创建，如果没发现，则继续准备其需要的构造器参数“circleC”，并将“circleB” 标识符放到“当前创建Bean池”；

3、Spring容器创建“circleC” Bean，首先去“当前创建Bean池”查找是否当前Bean正在创建，如果没发现，则继续准备其需要的构造器参数“circleA”，并将“circleC” 标识符放到“当前创建Bean池”；

4、到此为止Spring容器要去创建“circleA”Bean，发现该Bean 标识符在“当前创建Bean池”中，因为表示循环依赖，抛出BeanCurrentlyInCreationException。

二、setter循环依赖：表示通过setter注入方式构成的循环依赖。

对于setter注入造成的依赖是通过Spring容器提前暴露刚完成构造器注入但未完成其他步骤（如setter注入）的Bean来完成的，而且只能解决单例作用域的Bean循环依赖。

如下代码所示，通过提前暴露一个单例工厂方法，从而使其他Bean能引用到该Bean。

java代码：

```
查看 复制到剪贴板 打印
addSingletonFactory(beanName, new ObjectFactory() {
    public Object getObject() throws BeansException {
        return getEarlyBeanReference(beanName, mbd, bean);
    }
});
```

具体步骤如下：

1、Spring容器创建单例“circleA” Bean，首先根据无参构造器创建Bean，并暴露一个“ObjectFactory”用于返回一个提前暴露一个创建中的Bean，并将“circleA” 标识符放到“当前创建Bean池”；然后进行setter注入“circleB”；

2、Spring容器创建单例“circleB” Bean，首先根据无参构造器创建Bean，并暴露一个“ObjectFactory”用于返回一个提前暴露一个创建中的Bean，并将“circleB” 标识符放到“当前创建Bean池”，然后进行setter注入“circleC”；

3、Spring容器创建单例“circleC” Bean，首先根据无参构造器创建Bean，并暴露一个“ObjectFactory”用于返回一个提前暴露一个创建中的Bean，并将“circleC” 标识符放到“当前创建Bean池”，然后进行setter注入“circleA”；进行注入“circleA”时由于提前暴露了“ObjectFactory”工厂从而使用它返回提前暴露一个创建中的Bean；

4、最后在依赖注入“circleB”和“circleA”，完成setter注入。

对于“prototype”作用域Bean，Spring容器无法完成依赖注入，因为“prototype”作用域的Bean，Spring容器不进行缓存，因此无法提前暴露一个创建中的Bean。

java代码：

```
查看 复制到剪贴板 打印
<!-- 定义Bean配置文件，注意scope都是“prototype” -->
<bean id="circleA" class="cn.javass.spring.chapter3.bean.CircleA" scope="prototype">
      <property name="circleB" ref="circleB"/>
  </bean>
  <bean id="circleB" class="cn.javass.spring.chapter3.bean.CircleB" scope="prototype">
      <property name="circleC" ref="circleC"/>
  </bean>
  <bean id="circleC" class="cn.javass.spring.chapter3.bean.CircleC" scope="prototype">
      <property name="circleA" ref="circleA"/>
  </bean>
```

java代码：

查看 复制到剪贴板 打印

```
1.  //测试代码cn.javass.spring.chapter3.CircleTest
2.  @Test(expected = BeanCurrentlyInCreationException.class)
3.  public void testCircleBySetterAndPrototype () throws Throwable {
4.      try {
5.          ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext(
6.  "chapter3/circleInjectBySetterAndPrototype.xml");
7.          System.out.println(ctx.getBean("circleA"));
8.      }
9.      catch (Exception e) {
10.         Throwable e1 = e.getCause().getCause().getCause();
11.         throw e1;
12.     }
13. }
```

对于“singleton”作用域Bean，可以通过“setAllowCircularReferences(false);”来禁用循环引用：

java代码：

查看 复制到剪贴板 打印

```
1.  @Test(expected = BeanCurrentlyInCreationException.class)
2.  public void testCircleBySetterAndSingleton2() throws Throwable {
3.      try {
4.          ClassPathXmlApplicationContext ctx =
5.  new ClassPathXmlApplicationContext();
6.          ctx.setConfigLocation("chapter3/circleInjectBySetterAndSingleton.xml");
7.          ctx.refresh();
8.      }
9.      catch (Exception e) {
10.         Throwable e1 = e.getCause().getCause().getCause();
11.         throw e1;
12.     }
13. }
```

1.6 【第三章】 DI 之 3.1 DI的配置使用 ——跟我学spring3

发表时间: 2012-02-21 关键字: spring

3.1.1 依赖和依赖注入

传统应用程序设计中所说的依赖一般指“类之间的关系”，那先让我们复习一下类之间的关系：

泛化：表示类与类之间的继承关系、接口与接口之间的继承关系；

实现：表示类对接口的实现；

依赖：当类与类之间有使用关系时就属于依赖关系，不同于关联关系，依赖不具有“拥有关系”，而是一种“相识关系”，只在某个特定地方（比如某个方法体内）才有关系。

关联：表示类与类或类与接口之间的依赖关系，表现为“拥有关系”；具体到代码可以用实例变量来表示；

聚合：属于是关联的特殊情况，体现部分-整体关系，是一种弱拥有关系；整体和部分可以有不一样的生命周期；是一种弱关联；

组合：属于是关联的特殊情况，也体现了体现部分-整体关系，是一种强“拥有关系”；整体与部分有相同的生命周期，是一种强关联；

Spring IoC容器的依赖有两层含义：**Bean依赖容器**和**容器注入Bean的依赖资源**：

Bean依赖容器：也就是说Bean要依赖于容器，这里的依赖是指容器负责创建Bean并管理Bean的生命周期，正是由于由容器来控制创建Bean并注入依赖，也就是控制权被反转了，这也正是IoC名字的由来，**此处的有依赖是指Bean和容器之间的依赖关系。**

容器注入Bean的依赖资源：容器负责注入Bean的依赖资源，依赖资源可以是Bean、外部文件、常量数据等，在Java中都反映为对象，并且由容器负责组装Bean之间的依赖关系，**此处的依赖是指Bean之间的依赖关系，可以认为是传统类与类之间的“关联”、“聚合”、“组合”关系。**

为什么要应用依赖注入，应用依赖注入能给我们带来哪些好处呢？

动态替换Bean依赖对象，程序更灵活：替换Bean依赖对象，无需修改源文件：应用依赖注入后，由于可以采用配置文件方式实现，从而能随时动态的替换Bean的依赖对象，无需修改java源文件；

更好实践面向接口编程，代码更清晰：在Bean中只需指定依赖对象的接口，接口定义依赖对象完成的功能，通过容器注入依赖实现；

更好实践优先使用对象组合，而不是类继承：因为IoC容器采用注入依赖，也就是组合对象，从而更好的实践对象组合。

- 采用对象组合，Bean的功能可能由几个依赖Bean的功能组合而成，其Bean本身可能只提供少许功能或根本无任何功能，全部委托给依赖Bean，对象组合具有动态性，能更方便的替换掉依赖Bean，从而改变Bean功能；
- 而如果采用类继承，Bean没有依赖Bean，而是采用继承方式添加新功能，，而且功能是在编译时就确定了，不具有动态性，而且采用类继承导致Bean与子Bean之间高度耦合，难以复用。

增加Bean可复用性：依赖于对象组合，Bean更可复用且复用更简单；

降低Bean之间耦合：由于我们完全采用面向接口编程，在代码中没有直接引用Bean依赖实现，全部引用接口，而且不会出现显示的创建依赖对象代码，而且这些依赖是由容器来注入，很容易替换依赖实现类，从而降低Bean与依赖之间耦合；

代码结构更清晰：要应用依赖注入，代码结构要按照规约方式进行书写，从而更好的应用一些最佳实践，因此代码结构更清晰。

从以上我们可以看出，其实依赖注入只是一种装配对象的手段，设计的类结构才是基础，如果设计的类结构不支持依赖注入，Spring IoC容器也注入不了任何东西，从而从根本上说**“如何设计好类结构才是关键，依赖注入只是一种装配对象手段”。**

前边IoC一章我们已经了解了Bean依赖容器，那容器如何注入Bean的依赖资源，Spring IoC容器注入依赖资源主要有以下两种基本实现方式：

构造器注入：就是容器实例化Bean时注入那些依赖，通过在在Bean定义中指定构造器参数进行注入依赖，包括实例工厂方法参数注入依赖，但静态工厂方法参数不允许注入依赖；

setter注入：通过setter方法进行注入依赖；

方法注入：能通过配置方式替换掉Bean方法，也就是通过配置改变Bean方法 功能。

我们已经知道注入实现方式了，接下来让我们来看看具体配置吧。

3.1.2 构造器注入

使用构造器注入通过配置构造器参数实现，构造器参数就是依赖。除了构造器方式，还有静态工厂、实例工厂方法可以进行构造器注入。如图3-1所示：

图3-1 实例化

构造器注入可以根据参数索引注入、参数类型注入或Spring3支持的参数名注入，但参数名注入是有限制的，需要使用在编译程序时打开调试模式（即在编译时使用“javac –g:vars”在class文件中生成变量调试信息，默认是不包含变量调试信息的，从而能获取参数名字，否则获取不到参数名字）或在构造器上使用@ConstructorProperties（java.beans.ConstructorProperties）注解来指定参数名。

首先让我们准备测试构造器类HelloImpl3.java，该类只有一个包含两个参数的构造器：

java代码：

查看 复制到剪贴板 打印

```
1.  package cn.javass.spring.chapter3.helloworld;
2.  public class HelloImpl3 implements HelloApi {
3.      private String message;
4.  private int index;
5.  //@java.beans.ConstructorProperties({"message", "index"})
6.      public HelloImpl3(String message, int index) {
7.          this.message = message;
8.          this.index = index;
9.      }
10.     @Override
11.     public void sayHello() {
12.         System.out.println(index + ":" + message);
13.     }
14. }
```

一、根据参数索引注入，使用标签“<constructor-arg index="1" value="1"/>”来指定注入的依赖，其中“index”表示索引，从0开始，即第一个参数索引为0，“value”来指定注入的常量值，配置方式如下：

二、根据参数类型进行注入，使用标签“<constructor-arg type="java.lang.String" value="Hello World!"/>”来指定注入的依赖，其中“type”表示需要匹配的参数类型，可以是基本类型也可以是其他类型，如“int”、“java.lang.String”，“value”来指定注入的常量值，配置方式如下：

三、根据参数名进行注入，使用标签“<constructor-arg name="message" value="Hello World!"/>”来指定注入的依赖，其中“name”表示需要匹配的参数名字，“value”来指定注入的常量值，配置方式如下：

四、让我们来用具体的例子来看一下构造器注入怎么使用吧。

（1）首先准备Bean类，在此我们就使用“HelloImpl3”这个类。

（2）有了Bean类，接下来要进行Bean定义配置，我们需要配置三个Bean来完成上述三种依赖注入测试，其中Bean“ byIndex”是通过索引注入依赖；Bean“ byType”是根据类型进行注入依赖；Bean“ byName”是根据参数名字进行注入依赖，具体配置文件（resources/chapter3/ constructorDependencyInject.xml）如下：

java代码：

查看 复制到剪贴板 打印

```
<!-- 通过构造器参数索引方式依赖注入 -->
<bean id="byIndex" class="cn.javass.spring.chapter3.HelloImpl3">
<constructor-arg index="0" value="Hello World!"/>
   <constructor-arg index="1" value="1"/>
</bean>
<!-- 通过构造器参数类型方式依赖注入 -->
<bean id="byType" class="cn.javass.spring.chapter3.HelloImpl3">
  <constructor-arg type="java.lang.String" value="Hello World!"/>
  <constructor-arg type="int" value="2"/>
</bean>
<!-- 通过构造器参数名称方式依赖注入 -->
<bean id="byName" class="cn.javass.spring.chapter3.HelloImpl3">
  <constructor-arg name="message" value="Hello World!"/>
  <constructor-arg name="index" value="3"/>
</bean>
```

（3）配置完毕后，在测试之前，因为我们使用了通过构造器参数名字注入方式，请确保编译时class文件包含“变量信息”，具体查看编译时是否包含“变量调试信息”请右击项目，在弹出的对话框选择属性；然后在弹出的对话框选择“Java Compiler”条目，在“Class 文件 生成”框中选择“添加变量信息到Class文件（调试器使用）”，具体如图3-2：

图3-2 编译时打开“添加变量信息选项”

（4）接下来让我们测试一下配置是否工作，具体测试代码（cn.javass.spring.chapter3. DependencyInjectTest）如下：

java代码：

查看 复制到剪贴板 打印

```
@Test
public void testConstructorDependencyInjectTest() {
BeanFactory beanFactory =  new ClassPathXmlApplicationContext("chapter3/
constructorDependencyInject.xml");
//获取根据参数索引依赖注入的Bean
HelloApi byIndex = beanFactory.getBean("byIndex", HelloApi.class);
byIndex.sayHello();
//获取根据参数类型依赖注入的Bean
HelloApi byType = beanFactory.getBean("byType", HelloApi.class);
byType.sayHello();
//获取根据参数名字依赖注入的Bean
HelloApi byName = beanFactory.getBean("byName", HelloApi.class);
byName.sayHello();
}
```

通过以上测试我们已经会基本的构造器注入配置了，在测试通过参数名字注入时，除了可以使用以上方式，还可以通过在构造器上添加@java.beans.ConstructorProperties({"message", "index"})注解来指定参数名字，在

HelloImpl3构造器上把注释掉的“ConstructorProperties”打开就可以了，这个就留给大家做练习，自己配置然后测试一下。

五、大家已经会了构造器注入，那让我们再看一下静态工厂方法注入和实例工厂注入吧，其实它们注入配置是完全一样，在此我们只示范一下静态工厂注入方式和实例工厂方式配置，测试就留给大家自己练习：

（1）静态工厂类

java代码：

查看 复制到剪贴板 打印

```
1. //静态工厂类
2. package cn.javass.spring.chapter3;
3. import cn.javass.spring.chapter2.helloworld.HelloApi;
4. public class DependencyInjectByStaticFactory {
5.     public static HelloApi newInstance(String message, int index) {
6.         return new HelloImpl3(message, index);
7.     }
8. }
```

静态工厂类Bean定义配置文件（chapter3/staticFactoryDependencyInject.xml）

java代码：

查看 复制到剪贴板 打印

```
1. <bean id="byIndex"
2. class="cn.javass.spring.chapter3.DependencyInjectByStaticFactory" factory-method="newInstance">
3. <constructor-arg index="0" value="Hello World!"/>
4. <constructor-arg index="1" value="1"/>
5. </bean>
6. <bean id="byType"
7. class="cn.javass.spring.chapter3.DependencyInjectByStaticFactory" factory-method="newInstance">
8. <constructor-arg type="java.lang.String" value="Hello World!"/>
9.     <constructor-arg type="int" value="2"/>
10. </bean>
11. <bean id="byName"
12. class="cn.javass.spring.chapter3.DependencyInjectByStaticFactory" factory-method="newInstance">
13.     <constructor-arg name="message" value="Hello World!"/>
14.     <constructor-arg name="index" value="3"/>
15. </bean>
```

（2）实例工厂类

java代码：

```
查看 复制到剪贴板 打印
1. //实例工厂类
2. package cn.javass.spring.chapter3;
3. import cn.javass.spring.chapter2.helloworld.HelloApi;
4. public class DependencyInjectByInstanceFactory {
5.     public HelloApi newInstance(String message, int index) {
6.         return new HelloImpl3(message, index);
7.     }
8. }
```

实例工厂类Bean定义配置文件（chapter3/instanceFactoryDependencyInject.xml）

java代码：

```
查看 复制到剪贴板 打印
1. <bean id="instanceFactory"
2. class="cn.javass.spring.chapter3.DependencyInjectByInstanceFactory"/>
3.
4. <bean id="byIndex"
5. factory-bean="instanceFactory"  factory-method="newInstance">
6.     <constructor-arg index="0" value="Hello World!"/>
7.       <constructor-arg index="1" value="1"/>
8. </bean>
9. <bean id="byType"
10. factory-bean="instanceFactory" factory-method="newInstance">
11. <constructor-arg type="java.lang.String" value="Hello World!"/>
12. <constructor-arg type="int" value="2"/>
13. </bean>
14. <bean id="byName"
15. factory-bean="instanceFactory" factory-method="newInstance">
16. <constructor-arg name="message" value="Hello World!"/>
17. <constructor-arg name="index" value="3"/>
18. </bean>
```

（3）测试代码和构造器方式完全一样，只是配置文件不一样，大家只需把测试文件改一下就可以了。还有一点需要大家注意就是静态工厂方式和实例工厂方式根据参数名字注入的方式只支持通过在class文件中添加“变量调试信息”方式才能运行，ConstructorProperties注解方式不能工作，它只对构造器方式起作用，**不建议使用根据参数名进行构造器注入。**

3.1.3 setter注入

setter注入，是通过在通过构造器、静态工厂或实例工厂实例好Bean后，通过调用Bean类的setter方法进行注入依赖，如图3-3所示：

图3-3 setter注入方式

setter注入方式只有一种根据setter名字进行注入：

知道配置方式了，接下来先让我们来做个简单例子吧。

（1）准备测试类HelloImpl4，需要两个setter方法“setMessage”和“setIndex”：

java代码：

查看 复制到剪贴板 打印

```
1.  package cn.javass.spring.chapter3;
2.  import cn.javass.spring.chapter2.helloworld.HelloApi;
3.  public class HelloImpl4 implements HelloApi {
4.      private String message;
5.      private int index;
6.  //setter方法
7.      public void setMessage(String message) {
8.          this.message = message;
9.      }
10.     public void setIndex(int index) {
11.         this.index = index;
12.     }
13.     @Override
14.     public void sayHello() {
15.         System.out.println(index + ":" + message);
16.     }
17. }
```

（2）配置Bean定义，具体配置文件（resources/chapter3/setterDependencyInject.xml）片段如下：

java代码：

查看 复制到剪贴板 打印

```
1.  <!-- 通过setter方式进行依赖注入 -->
2.      <bean id="bean" class="cn.javass.spring.chapter3.HelloImpl4">
3.          <property name="message" value="Hello World!"/>
4.          <property name="index">
5.              <value>1</value>
```

```
        </property>
    </bean>
```

（3）该写测试进行测试一下是否满足能工作了，其实测试代码一点没变，变的是配置：

java代码：

```
@Test
public void testSetterDependencyInject() {
    BeanFactory beanFactory =
new ClassPathXmlApplicationContext("chapter3/setterDependencyInject.xml");
   HelloApi bean = beanFactory.getBean("bean", HelloApi.class);
    bean.sayHello();
}
```

知道如何配置了，但Spring如何知道setter方法？如何将值注入进去的呢？其实方法名是要遵守约定的，setter注入的方法名要遵循“JavaBean getter/setter 方法命名约定”：

JavaBean：是本质就是一个POJO类，但具有一下限制：

该类必须要**有公共的无参构造器**，如public HelloImpl4() {}；

属性为private访问级别，不建议public，如private String message;

属性必要时通过一组setter（修改器）和getter（访问器）方法来访问；

setter方法，以“set” 开头，后跟首字母大写的属性名，如“setMesssage”,简单属性一般只有一个方法参数，方法返回值通常为“void”；

getter方法，一般属性以“get”开头，对于boolean类型一般以“is”开头，后跟首字母大写的属性名，如“getMesssage”，“isOk”；

还有一些其他特殊情况，**比如属性有连续两个大写字母开头，如“URL”,则setter/getter方法为：“setURL”和“getURL”**，其他一些特殊情况请参看“Java Bean”命名规范。

3.1.4 注入常量

注入常量是依赖注入中最简单的。配置方式如下所示：

java代码：

查看 复制到剪贴板 打印

```
<property name="message" value="Hello World!"/>
或
<property name="index"><value>1</value></property><span class="Apple-style-span" style="font-size: 14px; white-space: normal; background-color: #ffffff;"> </span>
```

以上两种方式都可以，从配置来看第一种更简洁。注意此处“value”中指定的全是字符串，由Spring容器将此字符串转换成属性所需要的类型，如果转换出错，将抛出相应的异常。

Spring容器目前能对各种基本类型把配置的String参数转换为需要的类型。

注：Spring类型转换系统对于boolean类型进行了容错处理，除了可以使用“true/false”标准的Java值进行注入，还能使用“yes/no”、“on/off”、“1/0”来代表“真/假”，所以大家在学习或工作中遇到这种类似问题不要觉得是人家配置错了，而是Spring容错做的非常好。

java代码：

查看 复制到剪贴板 打印

```
测试类
public class BooleanTestBean {
    private boolean success;
    public void setSuccess(boolean success) {
        this.success = success;
    }
    public boolean isSuccess() {
        return success;
    }
}
配置文件（chapter3/booleanInject.xml）片段：
<!-- boolean参数值可以用on/off -->
<bean id="bean2" class="cn.javass.spring.chapter3.bean.BooleanTestBean">
    <property name="success" value="on"/>
</bean>
<!-- boolean参数值可以用yes/no -->
<bean id="bean3" class="cn.javass.spring.chapter3.bean.BooleanTestBean">
    <property name="success" value="yes"/>
</bean>
<!-- boolean参数值可以用1/0 -->
<bean id="bean4" class="cn.javass.spring.chapter3.bean.BooleanTestBean">
    <property name="success" value="1"/>
</bean>
```

3.1.5 注入Bean ID

用于注入Bean的ID，ID是一个常量不是引用，且类似于注入常量，但提供错误验证功能，配置方式如下所示：

java代码：

```
查看 复制到剪贴板 打印
1. <property name="id"><idref bean="bean1"/></property>
```

java代码：

```
查看 复制到剪贴板 打印
1. <property name="id"><idref local="bean2"/></property>
```

两种方式都可以，上述配置本质上在运行时等于如下方式

java代码：

```
查看 复制到剪贴板 打印
1. <bean id="bean1" class="....."/>
2. <bean id="idrefBean1" class=".....">
3. <property name="id" value ="bean1"/>
4. </bean>
```

第一种方式可以在容器初始化时校验被引用的Bean是否存在，如果不存在将抛出异常，而第二种方式只有在Bean实际使用时才能发现传入的Bean的ID是否正确，可能发生不可预料的错误。因此如果想注入Bean的ID，推荐使用第一种方式。

接下来学习一下如何使用吧：

首先定义测试Bean：

java代码：

```
查看 复制到剪贴板 打印
1. package cn.javass.spring.chapter3.bean
```

```
2.  public class IdRefTestBean {
3.      private String id;
4.      public String getId() {
5.          return id;
6.      }
7.      public void setId(String id) {
8.          this.id = id;
9.      }
10. }
```

其次定义配置文件（chapter3/idRefInject.xml）：

java代码：

```
查看  复制到剪贴板  打印
1.  <bean id="bean1" class="java.lang.String">
2.  <constructor-arg index="0" value="test"/>
3.  </bean>
4.  <bean id="bean2" class="java.lang.String">
5.      <constructor-arg index="0" value="test"/>
6.  </bean>
```

java代码：

```
查看  复制到剪贴板  打印
1.  <bean id="idrefBean1" class="cn.javass.spring.chapter3.bean.IdRefTestBean">
2.      <property name="id"><idref bean="bean1"/></property>
3.  </bean>
4.  <bean id="idrefBean2" class="cn.javass.spring.chapter3.bean.IdRefTestBean">
5.      <property name="id"><idref local="bean2"/></property>
6.  </bean>
```

从配置中可以看出，注入的Bean的ID是一个java.lang.String类型，即字符串类型，因此注入的同样是常量，只是具有校验功能。

<idref bean="....."/>将在容器初始化时校验注入的ID对于的Bean是否存在，如果不存在将抛出异常。

<idref local="....."/>将在XML解析时校验注入的ID对于的Bean在当前配置文件中是否存在，如果不存在将抛出异常，它不同于<idref bean="....."/>是校验发生在XML解析式而非容器初始化时，且只检查当前配置文件中是否存在相应的Bean。

3.1.6 注入集合、数组和字典

Spring不仅能注入简单类型数据，还能注入集合（Collection、无序集合Set、有序集合List）类型、数组(Array)类型、字典(Map)类型数据、Properties类型数据，接下来就让我们一个个看看如何注入这些数据类型的数据。

一、注入集合类型：包括Collection类型、Set类型、List类型数据：

（1）List类型：需要使用<list>标签来配置注入，其具体配置如下：

让我们来写个测试来练习一下吧：

准备测试类：

java代码：

查看 复制到剪贴板 打印

```
1.  package cn.javass.spring.chapter3.bean;
2.  import java.util.List;
3.  public class ListTestBean {
4.      private List<String> values;
5.      public List<String> getValues() {
6.          return values;
7.      }
8.      public void setValues(List<String> values) {
9.          this.values = values;
10.     }
11. }
```

进行Bean定义，在配置文件（resources/chapter3/listInject.xml）中配置list注入：

java代码：

查看 复制到剪贴板 打印

```
1.  <bean id="listBean" class="cn.javass.spring.chapter3.bean.ListTestBean">
2.      <property name="values">
3.          <list>
4.              <value>1</value>
5.              <value>2</value>
6.              <value>3</value>
7.          </list>
8.      </property>
9.  </bean>
```

测试代码：

java代码：

```
查看 复制到剪贴板 打印
@Test
public void testListInject() {
  BeanFactory beanFactory =
new ClassPathXmlApplicationContext("chapter3/listInject.xml");
ListTestBean listBean = beanFactory.getBean("listBean", ListTestBean.class);
System.out.println(listBean.getValues().size());
Assert.assertEquals(3, listBean.getValues().size());
}
```

（2）Set类型：需要使用<set>标签来配置注入，其配置参数及含义和<lsit>标签完全一样，在此就不阐述了：

准备测试类：

java代码：

```
查看 复制到剪贴板 打印
package cn.javass.spring.chapter3.bean;
import java.util.Collection;
public class CollectionTestBean {
   private Collection<String> values;
   public void setValues(Collection<String> values) {
      this.values = values;
   }
   public Collection<String> getValues() {
      return values;
   }
}
```

进行Bean定义，在配置文件（resources/chapter3/listInject.xml）中配置list注入：

java代码：

```
查看 复制到剪贴板 打印
<bean id="setBean" class="cn.javass.spring.chapter3.bean.SetTestBean">
<property name="values">
<set>
<value>1</value>
```

```
<value>2</value>
<value>3</value>
</set>
</property>
</bean>
```

具体测试代码就不写了，和listBean测试代码完全一样。

（2）Collection类型：因为Collection类型是Set和List类型的基类型，所以使用<set>或<list>标签都可以进行注入，配置方式完全和以上配置方式一样，只是将测试类属性改成“Collection”类型，如果配置有问题，可参考cn.javass.spring.chapter3.DependencyInjectTest测试类中的testCollectionInject测试方法中的代码。

二、**注入数组类型**：需要使用<array>标签来配置注入，其中标签属性“value-type”和“merge”和<list>标签含义完全一样，具体配置如下：

如果练习时遇到配置问题，可以参考“cn.javass.spring.chapter3.DependencyInjectTest”测试类中的testArrayInject测试方法中的代码。

三、注入字典（Map）类型：字典类型是包含键值对数据的数据结构，需要使用<map>标签来配置注入，其属性“key-type”和“value-type”分别指定“键”和“值”的数据类型，其含义和<list>标签的“value-type”含义一样，在此就不罗嗦了，并使用<key>子标签来指定键数据，<value>子标签来指定键对应的值数据，具体配置如下：

如果练习时遇到配置问题，可以参考“cn.javass.spring.chapter3.DependencyInjectTest”测试类中的testMapInject测试方法中的代码。

四、**Properties注入**：Spring能注入java.util.Properties类型数据，需要使用<props>标签来配置注入，键和值类型必须是String，不能变，子标签<prop key=”键” >值</prop>来指定键值对，具体配置如下：

如果练习时遇到配置问题，可以参考cn.javass.spring.chapter3.DependencyInjectTest测试类中的testPropertiesInject测试方法中的代码。

到此我们已经把简单类型及集合类型介绍完了，大家可能会问怎么没见注入“Bean之间关系”的例子呢？接下来就让我们来讲解配置Bean之间依赖关系，也就是注入依赖Bean。

3.1.7 引用其它Bean

上边章节已经介绍了注入常量、集合等基本数据类型和集合数据类型，本小节将介绍注入依赖Bean及注入内部Bean。

引用其他Bean的步骤与注入常量的步骤一样，可以通过构造器注入及setter注入引用其他Bean，只是引用其他Bean的注入配置稍微变化了一下：可以将“<constructor-arg index="0" value="Hello World!"/>”和“<property name="message" value="Hello World!"/>”中的value属性替换成bean属性，其中bean属性指定配置文件中的其他Bean的id或别名。另一种是把<value>标签替换为<.ref bean=” beanName” >，bean属性也是指定配置文件中的其他Bean的id或别名。那让我们看一下具体配置吧：

一、构造器注入方式：

（1）通过” <constructor-arg>”标签的ref属性来引用其他Bean，这是最简化的配置：

（2）通过” <constructor-arg>”标签的子<ref>标签来引用其他Bean，使用bean属性来指定引用的Bean：

二、setter注入方式：

（1）通过” <property>”标签的ref属性来引用其他Bean，这是最简化的配置：

（2）通过” <property>”标签的子<ref>标签来引用其他Bean，使用bean属性来指定引用的Bean：

三、接下来让我们用个具体例子来讲解一下具体使用吧：

（1）首先让我们定义测试引用Bean的类，在此我们可以使用原有的HelloApi实现，然后再定义一个装饰器来引用其他Bean，具体装饰类如下：

java代码：

查看 复制到剪贴板 打印

```
1.  package cn.javass.spring.chapter3.bean;
2.  import cn.javass.spring.chapter2.helloworld.HelloApi;
3.  public class HelloApiDecorator implements HelloApi {
4.  private HelloApi helloApi;
5.  //空参构造器
6.      public HelloApiDecorator() {
7.  }
8.  //有参构造器
9.      public HelloApiDecorator(HelloApi helloApi) {
10.         this.helloApi = helloApi;
```

```
}
public void setHelloApi(HelloApi helloApi) {
    this.helloApi = helloApi;
  }
  @Override
  public void sayHello() {
    System.out.println("==========装饰一下==========");
    helloApi.sayHello();
    System.out.println("==========装饰一下==========");
  }
}
```

（2）定义好了测试引用Bean接下来该在配置文件(resources/chapter3/beanInject.xml)进行配置Bean定义了，在此将演示通过构造器及setter方法方式注入依赖Bean：

java代码：

查看 复制到剪贴板 打印

```
<!-- 定义依赖Bean -->
<bean id="helloApi" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
<!-- 通过构造器注入 -->
<bean id="bean1" class="cn.javass.spring.chapter3.bean.HelloApiDecorator">
<constructor-arg index="0" ref="helloApi"/>
</bean>
<!-- 通过构造器注入 -->
<bean id="bean2" class="cn.javass.spring.chapter3.bean.HelloApiDecorator">
  <property name="helloApi"><ref bean=" helloApi"/></property>
</bean>
```

（3）测试一下吧，测试代码(cn.javass.spring.chapter3.DependencyInjectTest)片段如下：

java代码：

查看 复制到剪贴板 打印

```
@Test
public void testBeanInject() {
  BeanFactory beanFactory =
new ClassPathXmlApplicationContext("chapter3/beanInject.xml");
  //通过构造器方式注入
  HelloApi bean1 = beanFactory.getBean("bean1", HelloApi.class);
  bean1.sayHello();
  //通过setter方式注入
  HelloApi bean2 = beanFactory.getBean("bean2", HelloApi.class);
  bean2.sayHello();
}
```

四、其他引用方式：除了最基本配置方式以外，Spring还提供了另外两种更高级的配置方式，<ref local=”” />和<ref parent=”” />：

（1）<ref local=”” />配置方式：用于引用通过<bean id=”beanName”>方式中通过id属性指定的Bean，它能利用XML解析器的验证功能在读取配置文件时来验证引用的Bean是否存在。因此如果在当前配置文件中有相互引用的Bean可以采用<ref local>方式从而如果配置错误能在开发调试时就发现错误。

如果引用一个在当前配置文件中不存在的Bean将抛出如下异常：

org.springframework.beans.factory.xml.XmlBeanDefinitionStoreException: Line21 inXML document from class path resource [chapter3/beanInject2.xml] is invalid; nested exception is org.xml.sax.SAXParseException: cvc-id.1: There is no ID/IDREF binding for IDREF 'helloApi'.

<ref local>具体配置方式如下：

（2）<ref parent=”” />配置方式：用于引用父容器中的Bean，不会引用当前容器中的Bean，当然父容器中的Bean和当前容器的Bean是可以重名的，获取顺序是直接到父容器找。具体配置方式如下：

接下来让我们用个例子演示一下<ref local>和<ref parent>的配置过程：

首先还是准备测试类，在此我们就使用以前写好的HelloApiDecorator和HelloImpl4类；其次进行Bean定义，其中当前容器bean1引用本地的”helloApi”，而”bean2”将引用父容器的”helloApi”，配置如下：

java代码：

查看 复制到剪贴板 打印

```
<!-- sources/chapter3/parentBeanInject.xml表示父容器配置-->
<!--注意此处可能子容器也定义一个该Bean-->
<bean id="helloApi" class="cn.javass.spring.chapter3.HelloImpl4">
<property name="index" value="1"/>
<property name="message" value="Hello Parent!"/>
</bean>
```

java代码：

查看 复制到剪贴板 打印

```
<!-- sources/chapter3/localBeanInject.xml表示当前容器配置-->
<!-- 注意父容器中也定义了id 为 helloApi的Bean -->
<bean id="helloApi" class="cn.javass.spring.chapter3.HelloImpl4">
<property name="index" value="1"/>
```

```
    <property name="message" value="Hello Local!"/>
</bean>
<!-- 通过local注入 -->
<bean id="bean1" class="cn.javass.spring.chapter3.bean.HelloApiDecorator">
<constructor-arg index="0"><ref local="helloApi"/></constructor-arg>
</bean>
<!-- 通过parent注入 -->
<bean id="bean2" class="cn.javass.spring.chapter3.bean.HelloApiDecorator">
<property name="helloApi"><ref parent="helloApi"/></property>
</bean>
```

（3）写测试类测试一下吧，具体代码片段如下：

java代码：

查看 复制到剪贴板 打印

```
@Test
public void testLocalAndparentBeanInject() {
//初始化父容器
ApplicationContext parentBeanContext =
new ClassPathXmlApplicationContext("chapter3/parentBeanInject.xml");
//初始化当前容器
ApplicationContext beanContext = new ClassPathXmlApplicationContext(
new String[] {"chapter3/localBeanInject.xml"}, parentBeanContext);
    HelloApi bean1 = beanContext.getBean("bean1", HelloApi.class);
    bean1.sayHello();//该Bean引用local bean
HelloApi bean2 = beanContext.getBean("bean2", HelloApi.class);
bean2.sayHello();//该Bean引用parent bean
}
```

“bean1”将输出“Hello Local!”表示引用当前容器的Bean，“ bean2”将输出“Hello Paren!” ，表示引用父容器的Bean，如配置有问题请参考cn.javass.spring.chapter3.DependencyInjectTest中的testLocalAndparentBeanInject测试方法。

3.1.8 内部Bean定义

内部Bean就是在<property>或<constructor-arg>内通过<bean>标签定义的Bean，该Bean不管是否指定id或name，该Bean都会有唯一的匿名标识符，而且不能指定别名，该内部Bean对其他外部Bean不可见，具体配置如下：

（1）让我们写个例子测试一下吧，具体配置文件如下：

java代码：

```
查看 复制到剪贴板 打印
1. <bean id="bean" class="cn.javass.spring.chapter3.bean.HelloApiDecorator">
2. <property name="helloApi">
3. <bean id="helloApi" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
4. </property>
5. </bean>
```

（2）测试代码（cn.javass.spring.chapter3.DependencyInjectTest.testInnerBeanInject）：

java代码：

```
查看 复制到剪贴板 打印
1. @Test
2. public void testInnerBeanInject() {
3. ApplicationContext context =
4. new ClassPathXmlApplicationContext("chapter3/innerBeanInject.xml");
5. HelloApi bean = context.getBean("bean", HelloApi.class);
6. bean.sayHello();
7. }
```

3.1.9 处理null值

Spring通过<value>标签或value属性注入常量值，所有注入的数据都是字符串，那如何注入null值呢？通过“null”值吗？当然不是因为如果注入“null”则认为是字符串。Spring通过<null/>标签注入null值。即可以采用如下配置方式：

3.1.10 对象图导航注入支持

所谓对象图导航是指类似a.b.c这种点缀访问形式的访问或修改值。Spring支持对象图导航方式依赖注入。对象图导航依赖注入有一个限制就是比如a.b.c对象导航图注入中a和b必须为非null值才能注入c，否则将抛出空指针异常。

Spring不仅支持对象的导航，还支持数组、列表、字典、Properties数据类型的导航，对Set数据类型无法支持，因为无法导航。

数组和列表数据类型可以用array[0]、list[1]导航，注意“[]”里的必须是数字，因为是按照索引进行导航，对于数组类型注意不要数组越界错误。

字典Map数据类型可以使用map[1]、map[str]进行导航，其中“[]”里的是基本类型，无法放置引用类型。

让我们来练习一下吧。首先准备测试类，在此我们需要三个测试类，以便实现对象图导航功能演示：

NavigationC类用于打印测试代码，从而观察配置是否正确；具体类如下所示：

java代码：

查看 复制到剪贴板 打印

```
1. package cn.javass.spring.chapter3.bean;
2. public class NavigationC {
3.     public void sayNavigation() {
4.         System.out.println("===navigation c");
5.     }
6. }
```

NavigationB类，包含对象和列表、Properties、数组字典数据类型导航，而且这些复合数据类型保存的条目都是对象，正好练习一下如何往复合数据类型中注入对象依赖。具体类如下所示：

java代码：

查看 复制到剪贴板 打印

```
1. package cn.javass.spring.chapter3.bean;
2. import java.util.List;
3. import java.util.Map;
4. import java.util.Properties;
5. public class NavigationB {
6.     private NavigationC navigationC;
7.     private List<NavigationC> list;
8.     private Properties properties;
9.     private NavigationC[] array = new NavigationC[1];
10.    private Map<String, NavigationC> map;
11.   //由于setter和getter方法占用太多空间，故省略，大家自己实现吧
12. }
```

NavigationA类是我们的前端类，通过对它的导航进行注入值，具体代码如下：

java代码：

查看 复制到剪贴板 打印

```
1. package cn.javass.spring.chapter3.bean;
2. public class NavigationA {
3.     private NavigationB navigationB;
4.     public void setNavigationB(NavigationB navigationB) {
5.         this.navigationB = navigationB;
6.     }
```

```
    public NavigationB getNavigationB() {
        return navigationB;
    }
}
```

接下来该进行Bean定义配置（resources/chapter3/navigationBeanInject.xml）了，首先让我们配置一下需要被导航的数据，NavigationC和NavigationB类，其中配置NavigationB时注意要确保比如array字段不为空值，这就需要或者在代码中赋值如“NavigationC[] array = new NavigationC[1];”，或者通过配置文件注入如“<list></list>”注入一个不包含条目的列表。具体配置如下：

java代码：

```
查看 复制到剪贴板 打印
<bean id="c" class="cn.javass.spring.chapter3.bean.NavigationC"/>
<bean id="b" class="cn.javass.spring.chapter3.bean.NavigationB">
<property name="list"><list></list></property>
    <property name="map"><map></map></property>
    <property name="properties"><props></props></property>
</bean>
```

配置完需要被导航的Bean定义了，该来配置NavigationA导航Bean了，在此需要注意，由于“navigationB”属性为空值，在此需要首先注入“navigationB”值；还有对于数组导航不能越界否则报错；具体配置如下：

java代码：

```
查看 复制到剪贴板 打印
<bean id="a" class="cn.javass.spring.chapter3.bean.NavigationA">
<!-- 首先注入navigatiionB 确保它非空 -->
<property name="navigationB" ref="b"/>
<!-- 对象图导航注入 -->
<property name="navigationB.navigationC" ref="c"/>
<!-- 注入列表数据类型数据 -->
<property name="navigationB.list[0]" ref="c"/>
<!-- 注入map类型数据 -->
<property name="navigationB.map[key]" ref="c"/>
<!-- 注入properties类型数据 -->
<property name="navigationB.properties[0]" ref="c"/>
<!-- 注入properties类型数据 -->
<property name="navigationB.properties[1]" ref="c"/>
<!-- 注入数组类型数据 ，注意不要越界-->
<property name="navigationB.array[0]" ref="c"/>
</bean>
```

配置完毕，具体测试代码在cn.javass.spring.chapter3. DependencyInjectTest，让我们看下测试代码吧：

java代码：

查看 复制到剪贴板 打印

```
1.  //对象图导航
2.  @Test
3.  public void testNavigationBeanInject() {
4.  ApplicationContext context =
5.  new ClassPathXmlApplicationContext("chapter3/navigationBeanInject.xml");
6.  NavigationA navigationA = context.getBean("a", NavigationA.class);
7.  navigationA.getNavigationB().getNavigationC().sayNavigation();
8.  navigationA.getNavigationB().getList().get(0).sayNavigation();
9.  navigationA.getNavigationB().getMap().get("key").sayNavigation();
10. navigationA.getNavigationB().getArray()[0].sayNavigation();
11. ((NavigationC)navigationA.getNavigationB().getProperties().get("1"))
12. .sayNavigation();
13. }
14.
```

测试完毕，应该输出5个“===navigation c”，是不是很简单，注意这种方式是不推荐使用的，了解一下就够了，最好使用3.1.5一节使用的配置方式。

3.1.11配置简写

让我们来总结一下依赖注入配置及简写形式，其实我们已经在以上部分穿插着进行简化配置了：

一、构造器注入：

1）常量值

简写：<constructor-arg index="0" value="常量"/>

全写：<constructor-arg index="0"><value>常量</value></constructor-arg>

2）引用

简写：<constructor-arg index="0" ref="引用"/>

全写：<constructor-arg index="0"><ref bean="引用"/></constructor-arg>

二、setter注入：

1）常量值

简写：<property name="message" value="常量"/>

全写：<property name="message"><value>常量</value></ property>

2）引用

简写：<property name="message" ref="引用"/>

全写：<property name="message"><ref bean="引用"/></ property>

3）数组：<array>没有简写形式

4）列表：<list>没有简写形式

5）集合：<set>没有简写形式

6）字典

简写：<map>

<entry key="键常量" value="值常量"/>

<entry key-ref="键引用" value-ref="值引用"/>

</map>

全写：<map>

<entry><key><value>键常量</value></key><value>值常量</value></entry>

<entry><key><ref bean="键引用"/></key><ref bean="值引用"/></entry>

</map>

7）Properties：没有简写形式

三、使用p命名空间简化setter注入：

使用p命名空间来简化setter注入，具体使用如下：

java代码：

查看 复制到剪贴板 打印

```
<?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-3.0.xsd">
<bean id="bean1" class="java.lang.String">
       <constructor-arg index="0" value="test"/>
   </bean>
<bean id="idrefBean1" class="cn.javass.spring.chapter3.bean.IdRefTestBean"
p:id="value"/>
<bean id="idrefBean2" class="cn.javass.spring.chapter3.bean.IdRefTestBean"
```

```
14.  p:id-ref="bean1"/>
15.  </beans>
```

- **xmlns:p="http://www.springframework.org/schema/p" ：首先指定p命名空间；**
- **<bean id="……" class="……" p:id="value"/> ：**常量setter注入方式，其等价于<property name="id" value="value"/>；

。

- **<bean id="……" class="……" p:id-ref="bean1"/> ：**引用setter注入方式，其等价于<property name="id" ref="bean1"/>。

1.7 【第三章】 DI 之 3.2 循环依赖 ——跟我学spring3

发表时间: 2012-02-21 关键字: spring, IOC, 企业应用

3.2.1 什么是循环依赖

循环依赖就是循环引用，就是两个或多个Bean相互之间的持有对方，比如CircleA引用CircleB，CircleB引用CircleC，CircleC引用CircleA，则它们最终反映为一个环。此处不是循环调用，循环调用是方法之间的环调用。如图3-5所示：

图3-5 循环引用

循环调用是无法解决的，除非有终结条件，否则就是死循环，最终导致内存溢出错误。

Spring容器循环依赖包括构造器循环依赖和setter循环依赖，那Spring容器如何解决循环依赖呢？首先让我们来定义循环引用类：

java代码：

查看 复制到剪贴板 打印

```
1.  package cn.javass.spring.chapter3.bean;
2.  public class CircleA {
3.      private CircleB circleB;
4.      public CircleA() {
5.      }
6.      public CircleA(CircleB circleB) {
7.          this.circleB = circleB;
8.      }
9.  public void setCircleB(CircleB circleB)
10. {
11.         this.circleB = circleB;
12.     }
13. public void a() {
14.   circleB.b();
15. }
16. }
```

java代码：

查看 复制到剪贴板 打印

```
1.  package cn.javass.spring.chapter3.bean;
2.  public class CircleB {
3.      private CircleC circleC;
4.      public CircleB() {
5.      }
6.      public CircleB(CircleC circleC) {
```

```
        this.circleC = circleC;
    }
public void setCircleC(CircleC circleC)
{
        this.circleC = circleC;
    }
    public void b() {
        circleC.c();
    }
}
```

java代码：

```
查看 复制到剪贴板 打印
package cn.javass.spring.chapter3.bean;
public class CircleC {
    private CircleA circleA;
    public CircleC() {
    }
    public CircleC(CircleA circleA) {
        this.circleA = circleA;
    }
public void setCircleA(CircleA circleA)
{
        this.circleA = circleA;
    }
    public void c() {
        circleA.a();
    }
}
```

3.2.2 Spring如何解决循环依赖

一、构造器循环依赖：表示通过构造器注入构成的循环依赖，此依赖是无法解决的，只能抛出BeanCurrentlyInCreationException异常表示循环依赖。

如在创建CircleA类时，构造器需要CircleB类，那将去创建CircleB，在创建CircleB类时又发现需要CircleC类，则又去创建CircleC，最终在创建CircleC时发现又需要CircleA；从而形成一个环，没办法创建。

Spring容器将每一个正在创建的Bean 标识符放在一个“当前创建Bean池”中，Bean标识符在创建过程中将一直保持在这个池中，因此如果在创建Bean过程中发现自己已经在“当前创建Bean池”里时将抛出BeanCurrentlyInCreationException异常表示循环依赖；而对于创建完毕的Bean将从“当前创建Bean池”中清除掉。

1）首先让我们看一下配置文件（chapter3/circleInjectByConstructor.xml）：

java代码：

```
查看 复制到剪贴板 打印
1.
2. <bean id="circleA" class="cn.javass.spring.chapter3.bean.CircleA">
3. <constructor-arg index="0" ref="circleB"/>
4. </bean>
5. <bean id="circleB" class="cn.javass.spring.chapter3.bean.CircleB">
6. <constructor-arg index="0" ref="circleC"/>
7. </bean>
8. <bean id="circleC" class="cn.javass.spring.chapter3.bean.CircleC">
9. <constructor-arg index="0" ref="circleA"/>
10. </bean>
11.
```

2）写段测试代码（cn.javass.spring.chapter3.CircleTest）测试一下吧：

java代码：

```
查看 复制到剪贴板 打印
1. @Test(expected = BeanCurrentlyInCreationException.class)
2. public void testCircleByConstructor() throws Throwable {
3. try {
4. new ClassPathXmlApplicationContext("chapter3/circleInjectByConstructor.xml");
5. }
6. catch (Exception e) {
7. //因为要在创建circle3时抛出；
8. Throwable e1 = e.getCause().getCause().getCause();
9. throw e1;
10. }
11. }
```

让我们分析一下吧：

1、Spring容器创建“circleA” Bean，首先去“当前创建Bean池”查找是否当前Bean正在创建，如果没发现，则继续准备其需要的构造器参数“circleB”，并将“circleA” 标识符放到“当前创建Bean池”；

2、Spring容器创建“circleB” Bean，首先去“当前创建Bean池”查找是否当前Bean正在创建，如果没发现，则继续准备其需要的构造器参数“circleC”，并将“circleB” 标识符放到“当前创建Bean池”；

3、Spring容器创建“circleC” Bean，首先去“当前创建Bean池”查找是否当前Bean正在创建，如果没发现，则继续准备其需要的构造器参数“circleA”，并将“circleC” 标识符放到“当前创建Bean池”；

4、到此为止Spring容器要去创建“circleA”Bean，发现该Bean 标识符在“当前创建Bean池”中，因为表示循环依赖，抛出BeanCurrentlyInCreationException。

二、setter循环依赖：表示通过setter注入方式构成的循环依赖。

对于setter注入造成的依赖是通过Spring容器提前暴露刚完成构造器注入但未完成其他步骤（如setter注入）的Bean来完成的，而且只能解决单例作用域的Bean循环依赖。

如下代码所示，通过提前暴露一个单例工厂方法，从而使其他Bean能引用到该Bean。

java代码：

查看 复制到剪贴板 打印

```
addSingletonFactory(beanName, new ObjectFactory() {
    public Object getObject() throws BeansException {
        return getEarlyBeanReference(beanName, mbd, bean);
    }
});
```

具体步骤如下：

1、Spring容器创建单例“circleA” Bean，首先根据无参构造器创建Bean，并暴露一个“ObjectFactory”用于返回一个提前暴露一个创建中的Bean，并将“circleA” 标识符放到“当前创建Bean池”；然后进行setter注入“circleB”；

2、Spring容器创建单例“circleB” Bean，首先根据无参构造器创建Bean，并暴露一个“ObjectFactory”用于返回一个提前暴露一个创建中的Bean，并将“circleB” 标识符放到“当前创建Bean池”，然后进行setter注入“circleC”；

3、Spring容器创建单例“circleC” Bean，首先根据无参构造器创建Bean，并暴露一个“ObjectFactory”用于返回一个提前暴露一个创建中的Bean，并将“circleC” 标识符放到“当前创建Bean池”，然后进行setter注入“circleA”；进行注入“circleA”时由于提前暴露了“ObjectFactory”工厂从而使用它返回提前暴露一个创建中的Bean；

4、最后在依赖注入“circleB”和“circleA”，完成setter注入。

对于“prototype”作用域Bean，Spring容器无法完成依赖注入，因为“prototype”作用域的Bean，Spring容器不进行缓存，因此无法提前暴露一个创建中的Bean。

java代码：

查看 复制到剪贴板 打印

```
<!-- 定义Bean配置文件，注意scope都是“prototype” -->
<bean id="circleA" class="cn.javass.spring.chapter3.bean.CircleA" scope="prototype">
    <property name="circleB" ref="circleB"/>
</bean>
<bean id="circleB" class="cn.javass.spring.chapter3.bean.CircleB" scope="prototype">
    <property name="circleC" ref="circleC"/>
```

```
    </bean>
    <bean id="circleC" class="cn.javass.spring.chapter3.bean.CircleC" scope="prototype">
        <property name="circleA" ref="circleA"/>
    </bean>
```

java代码：

```
查看 复制到剪贴板 打印
//测试代码cn.javass.spring.chapter3.CircleTest
@Test(expected = BeanCurrentlyInCreationException.class)
public void testCircleBySetterAndPrototype () throws Throwable {
    try {
        ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext(
"chapter3/circleInjectBySetterAndPrototype.xml");
        System.out.println(ctx.getBean("circleA"));
    }
    catch (Exception e) {
        Throwable e1 = e.getCause().getCause().getCause();
        throw e1;
    }
}
```

对于“singleton”作用域Bean，可以通过“setAllowCircularReferences(false);”来禁用循环引用：

java代码：

```
查看 复制到剪贴板 打印
@Test(expected = BeanCurrentlyInCreationException.class)
public void testCircleBySetterAndSingleton2() throws Throwable {
    try {
        ClassPathXmlApplicationContext ctx =
new ClassPathXmlApplicationContext();
        ctx.setConfigLocation("chapter3/circleInjectBySetterAndSingleton.xml");
        ctx.refresh();
    }
    catch (Exception e) {
        Throwable e1 = e.getCause().getCause().getCause();
        throw e1;
    }
}
```

补充：出现循环依赖是设计上的问题，一定要避免！

请参考《敏捷软件开发：原则、模式与实践》中的“无环依赖”原则

包之间的依赖结构必须是一个直接的无环图形（DAG）。也就是说，在依赖结构中不允许出现环（循环依赖）。

1.8 【第三章】 DI 之 3.3 更多DI的知识 ——跟我学spring3

发表时间: 2012-02-21

3.3.1 延迟初始化Bean

延迟初始化也叫做惰性初始化，指不提前初始化Bean，而是只有在真正使用时才创建及初始化Bean。

配置方式很简单只需在<bean>标签上指定 “lazy-init” 属性值为“true”即可延迟初始化Bean。

Spring容器会在创建容器时提前初始化“singleton”作用域的Bean，“singleton”就是单例的意思即整个容器每个Bean只有一个实例，后边会详细介绍。Spring容器预先初始化Bean通常能帮助我们提前发现配置错误，所以如果没有什么情况建议开启，除非有某个Bean可能需要加载很大资源，而且很可能在整个应用程序生命周期中很可能使用不到，可以设置为延迟初始化。

延迟初始化的Bean通常会在第一次使用时被初始化；或者在被非延迟初始化Bean作为依赖对象注入时在会随着初始化该Bean时被初始化，因为在这时使用了延迟初始化Bean。

容器管理初始化Bean消除了编程实现延迟初始化，完全由容器控制，只需在需要延迟初始化的Bean定义上配置即可，比编程方式更简单，而且是无侵入代码的。

具体配置如下：

java代码：

查看 复制到剪贴板 打印

```
1. <bean id="helloApi"
2. class="cn.javass.spring.chapter2.helloworld.HelloImpl"
3. lazy-init="true"/>
```

3.3.2 使用depends-on

depends-on是指指定Bean初始化及销毁时的顺序，使用depends-on属性指定的Bean要先初始化完毕后才初始化当前Bean，由于只有“singleton”Bean能被Spring管理销毁，所以当指定的Bean都是“singleton”时，使用depends-on属性指定的Bean要在指定的Bean之后销毁。

配置方式如下：

java代码：

查看 复制到剪贴板 打印

```
1. <bean id="helloApi" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
2. <bean id="decorator"
3.    class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
4.    depends-on="helloApi">
```

```
    <property name="helloApi"><ref bean="helloApi"/></property>
</bean>
```

“decorator”指定了“depends-on”属性为“helloApi”，所以在“decorator”Bean初始化之前要先初始化“helloApi”，而在销毁“helloApi”之前先要销毁“decorator”，大家注意一下销毁顺序，与文档上的不符。

“depends-on”属性可以指定多个Bean，若指定多个Bean可以用“;”、“,”、空格分割。

那“depends-on”有什么好处呢？主要是给出明确的初始化及销毁顺序，比如要初始化“decorator”时要确保“helloApi”Bean的资源准备好了，否则使用“decorator”时会看不到准备的资源；而在销毁时要先在“decorator”Bean的把对“helloApi”资源的引用释放掉才能销毁“helloApi”，否则可能销毁“helloApi”时而“decorator”还保持着资源访问，造成资源不能释放或释放错误。

让我们看个例子吧，在平常开发中我们可能需要访问文件系统，而文件打开、关闭是必须配对的，不能打开后不关闭，从而造成其他程序不能访问该文件。让我们来看具体配置吧：

1）准备测试类：

ResourceBean从配置文件中配置文件位置，然后定义初始化方法init中打开指定的文件，然后获取文件流；最后定义销毁方法destroy用于在应用程序关闭时调用该方法关闭掉文件流。

DependentBean中会注入ResourceBean，并从ResourceBean中获取文件流写入内容；定义初始化方法init用来定义一些初始化操作并向文件中输出文件头信息；最后定义销毁方法用于在关闭应用程序时想文件中输出文件尾信息。

具体代码如下：

java代码：

查看 复制到剪贴板 打印

```
package cn.javass.spring.chapter3.bean;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
public class ResourceBean {
    private FileOutputStream fos;
    private File file;
```

```
    //初始化方法
    public void init() {
        System.out.println("ResourceBean:========初始化");
        //加载资源,在此只是演示
        System.out.println("ResourceBean:========加载资源，执行一些预操作");
        try {
            this.fos = new FileOutputStream(file);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
    //销毁资源方法
    public void destroy() {
        System.out.println("ResourceBean:========销毁");
        //释放资源
        System.out.println("ResourceBean:========释放资源，执行一些清理操作");
        try {
            fos.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public FileOutputStream getFos() {
        return fos;
    }
    public void setFile(File file) {
        this.file = file;
    }
}
```

java代码：

查看 复制到剪贴板 打印

```
package cn.javass.spring.chapter3.bean;
import java.io.IOException;
public class DependentBean {
    ResourceBean resourceBean;
    public void write(String ss) throws IOException {
        System.out.println("DependentBean:========写资源");
        resourceBean.getFos().write(ss.getBytes());
    }
    //初始化方法
    public void init() throws IOException {
        System.out.println("DependentBean:========初始化");
resourceBean.getFos().write("DependentBean:========初始化=====".getBytes());
    }
    //销毁方法
    public void destroy() throws IOException {
        System.out.println("DependentBean:========销毁");
        //在销毁之前需要往文件中写销毁内容
        resourceBean.getFos().write("DependentBean:========销毁=====".getBytes());
    }

    public void setResourceBean(ResourceBean resourceBean) {
        this.resourceBean = resourceBean;
    }
}
```

2）类定义好了，让我们来进行Bean定义吧，具体配置文件如下：

java代码：

查看 复制到剪贴板 打印

```
<bean id="resourceBean"
    class="cn.javass.spring.chapter3.bean.ResourceBean"
    init-method="init" destroy-method="destroy">
    <property name="file" value="D:/test.txt"/>
</bean>
<bean id="dependentBean"
    class="cn.javass.spring.chapter3.bean.DependentBean"
    init-method="init" destroy-method="destroy" depends-on="resourceBean">
    <property name="resourceBean" ref="resourceBean"/>
</bean>
```

<property name="file" value="D:/test.txt"/>配置：Spring容器能自动把字符串转换为java.io.File。

init-method="init" ：指定初始化方法，在构造器注入和setter注入完毕后执行。

destroy-method="destroy"：指定销毁方法，只有“singleton”作用域能销毁，“prototype”作用域的一定不能，其他作用域不一定能；后边再介绍。

在此配置中，resourceBean初始化在dependentBean之前被初始化，resourceBean销毁会在dependentBean销毁之后执行。

3）配置完毕，测试一下吧：

java代码：

查看 复制到剪贴板 打印

```
package cn.javass.spring.chapter3;
import java.io.IOException;
import org.junit.Test;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import cn.javass.spring.chapter3.bean.DependentBean;
public class MoreDependencyInjectTest {
    @Test
    public void testDependOn() throws IOException {
        ClassPathXmlApplicationContext context =
new ClassPathXmlApplicationContext("chapter3/depends-on.xml");
        //一点要注册销毁回调，否则我们定义的销毁方法不执行
        context.registerShutdownHook();
        DependentBean dependentBean =
```

```
14.  context.getBean("dependentBean", DependentBean.class);
15.          dependentBean.write("aaa");
16.      }
17.  }
```

测试跟其他测试完全一样，只是在此我们一定要注册销毁方法回调，否则销毁方法不会执行。

如果配置没问题会有如下输出：

java代码：

```
查看  复制到剪贴板  打印
1.  ResourceBean:========初始化
2.  ResourceBean:========加载资源，执行一些预操作
3.  DependentBean:==========初始化
4.  DependentBean:==========写资源
5.  DependentBean:==========销毁
6.  ResourceBean:========销毁
7.  ResourceBean:========释放资源，执行一些清理操作
8.
```

3.3.3 自动装配

自动装配就是指由Spring来自动地注入依赖对象，无需人工参与。

目前Spring3.0支持“no”、“byName”、“byType”、“constructor”四种自动装配，默认是“no”指不支持自动装配的，其中Spring3.0已不推荐使用之前版本的“autodetect”自动装配，推荐使用Java 5+支持的（@Autowired）注解方式代替；如果想支持“autodetect”自动装配，请将schema改为“spring-beans-2.5.xsd”或去掉。

自动装配的好处是减少构造器注入和setter注入配置，减少配置文件的长度。自动装配通过配置<bean>标签的“autowire”属性来改变自动装配方式。接下来让我们挨着看下配置的含义。

一、default：表示使用默认的自动装配，默认的自动装配需要在<beans>标签中使用default-autowire属性指定，其支持“no”、“byName”、“byType”、“constructor”四种自动装配，如果需要覆盖默认自动装配，请继续往下看；

二、no：意思是不支持自动装配，必须明确指定依赖。

三、byName：通过设置Bean定义属性autowire="byName"，意思是根据名字进行自动装配，只能用于setter注入。比如我们有方法“setHelloApi”，则“byName”方式Spring容器将查找名字为helloApi的Bean并注入，如果找不到指定的Bean，将什么也不注入。

例如如下Bean定义配置：

java代码：

查看 复制到剪贴板 打印

```
<bean id="helloApi" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
<bean id="bean" class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
    autowire="byName"/>
```

测试代码如下：

java代码：

查看 复制到剪贴板 打印

```
package cn.javass.spring.chapter3;
import java.io.IOException;
import org.junit.Test;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import cn.javass.spring.chapter2.helloworld.HelloApi;
public class AutowireBeanTest {
   @Test
    public void testAutowireByName() throws IOException {
ClassPathXmlApplicationContext context =
new ClassPathXmlApplicationContext("chapter3/autowire-byName.xml");
        HelloApi helloApi = context.getBean("bean", HelloApi.class);
        helloApi.sayHello();
    }
}

```

是不是不要配置<property>了，如果一个bean有很多setter注入，通过“byName”方式是不是能减少很多<property>配置。此处注意了，**在根据名字注入时，将把当前Bean自己排除在外：**比如“hello”Bean类定义了“setHello”方法，则hello是不能注入到“setHello”的。

四、“byType”：通过设置Bean定义属性autowire="byType"，意思是指根据类型注入，用于setter注入，比如如果指定自动装配方式为“byType”，而“setHelloApi”方法需要注入HelloApi类型数据，则Spring容器将查找

HelloApi类型数据，如果找到一个则注入该Bean，如果找不到将什么也不注入，如果找到多个Bean将优先注入<bean>标签“primary”属性为true的Bean，否则抛出异常来表明有个多个Bean发现但不知道使用哪个。让我们用例子来讲解一下这几种情况吧。

1）根据类型只找到一个Bean，此处注意了，**在根据类型注入时，将把当前Bean自己排除在外，**即如下配置中helloApi和bean都是HelloApi接口的实现，而“bean”通过类型进行注入“HelloApi”类型数据时自己是排除在外的，配置如下（具体测试请参考AutowireBeanTest.testAutowireByType1方法）：

java代码：

查看 复制到剪贴板 打印

```
<bean class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
<bean id="bean" class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
    autowire="byType"/>
```

2）根据类型找到多个Bean时，对于集合类型（如List、Set）将注入所有匹配的候选者，而对于其他类型遇到这种情况可能需要使用“autowire-candidate”属性为false来让指定的Bean放弃作为自动装配的候选者，或使用“primary”属性为true来指定某个Bean为首选Bean：

2.1）通过设置Bean定义的“autowire-candidate”属性为false来把指定Bean后自动装配候选者中移除：

java代码：

查看 复制到剪贴板 打印

```
<bean class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
<!-- 从自动装配候选者中去除 -->
<bean class="cn.javass.spring.chapter2.helloworld.HelloImpl"
autowire-candidate="false"/>
<bean id="bean1" class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
    autowire="byType"/>

```

2.2）通过设置Bean定义的“primary”属性为false来把指定自动装配时候选者中首选Bean：

java代码：

查看 复制到剪贴板 打印

```
<bean class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
```

```
<!-- 自动装配候选者中的首选Bean-->
<bean class="cn.javass.spring.chapter2.helloworld.HelloImpl" primary="true"/>
<bean id="bean" class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
    autowire="byType"/>
```

具体测试请参考AutowireBeanTest类的testAutowireByType***方法。

五、“constructor”：通过设置Bean定义属性autowire="constructor"，功能和“byType”功能一样，根据类型注入构造器参数，只是用于构造器注入方式，直接看例子吧：

java代码：

查看 复制到剪贴板 打印

```
<bean class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
<!-- 自动装配候选者中的首选Bean-->
<bean class="cn.javass.spring.chapter2.helloworld.HelloImpl" primary="true"/>
<bean id="bean"
    class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
    autowire="constructor"/>
```

测试代码如下：

java代码：

查看 复制到剪贴板 打印

```
@Test
public void testAutowireByConstructor() throws IOException {
ClassPathXmlApplicationContext context =
 new ClassPathXmlApplicationContext("chapter3/autowire-byConstructor.xml");
HelloApi helloApi = context.getBean("bean", HelloApi.class);
helloApi.sayHello();
}
```

六、autodetect：自动检测是使用“constructor”还是“byType”自动装配方式，已不推荐使用。如果Bean有空构造器那么将采用“byType”自动装配方式，否则使用“constructor”自动装配方式。此处要把3.0的xsd替换为2.5的xsd，否则会报错。

java代码：

查看 复制到剪贴板 打印

```
<?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-2.5.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.xsd">
<bean class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
 <!-- 自动装配候选者中的首选Bean-->
 <bean class="cn.javass.spring.chapter2.helloworld.HelloImpl" primary="true"/>
 <bean id="bean"
      class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
      autowire="autodetect"/>
</beans>

```

可以采用在“<beans>”标签中通过“default-autowire”属性指定全局的自动装配方式，即如果default-autowire=“byName”，将对所有Bean进行根据名字进行自动装配。

不是所有类型都能自动装配：

- 不能自动装配的数据类型：Object、基本数据类型（Date、CharSequence、Number、URI、URL、Class、int）等；
- 通过“<beans>”标签default-autowire-candidates属性指定的匹配模式，不匹配的将不能作为自动装配的候选者，例如指定“*Service，*Dao”，将只把匹配这些模式的Bean作为候选者，而不匹配的不会作为候选者；
- 通过将“<bean>”标签的autowire-candidate属性可被设为false，从而该Bean将不会作为依赖注入的候选者。

数组、集合、字典类型的根据类型自动装配和普通类型的自动装配是有区别的：

- **数组类型、集合（Set、Collection、List）接口类型**：将根据泛型获取匹配的所有候选者并注入到数组或集合中，如“List<HelloApi> list”将选择所有的HelloApi类型Bean并注入到list中，而对于集合的具体类型将只选择一个候选者，“如 ArrayList<HelloApi> list”将选择一个类型为ArrayList的Bean注入，而不是选择所有的HelloApi类型Bean进行注入；
- **字典（Map）接口类型**：同样根据泛型信息注入，键必须为String类型的Bean名字，值根据泛型信息获取，如“Map<String, HelloApi> map” 将选择所有的HelloApi类型Bean并注入到map中，而对于具体字典类型如“HashMap<String, HelloApi> map”将只选择类型为HashMap的Bean注入，而不是选择所有的HelloApi类型Bean进行注入。

自动装配我们已经介绍完了，自动装配能带给我们什么好处呢？首先，自动装配确实减少了配置文件的量；其次，“byType”自动装配能在相应的Bean更改了字段类型时自动更新，即修改Bean类不需要修改配置，确实简单了。

自动装配也是有缺点的，最重要的缺点就是没有了配置，在查找注入错误时非常麻烦，还有比如基本类型没法完成自动装配，所以可能经常发生一些莫名其妙的错误，在此我推荐大家不要使用该方式，最好是指定明确的注入方式，或

者采用最新的Java5+注解注入方式。所以大家在使用自动装配时应该考虑自己负责项目的复杂度来进行衡量是否选择自动装配方式。

自动装配注入方式能和配置注入方式一同工作吗？当然可以，大家只需记住**配置注入的数据会覆盖自动装配注入的数据。**

大家是否注意到对于采用自动装配方式时如果没找到合适的的Bean时什么也不做，这样在程序中总会莫名其妙的发生一些空指针异常，而且是在程序运行期间才能发现，有没有办法能在提前发现这些错误呢？接下来就让我来看下依赖检查吧。

3.3.4 依赖检查

上一节介绍的自动装配，很可能发生没有匹配的Bean进行自动装配，如果此种情况发生，只有在程序运行过程中发生了空指针异常才能发现错误，如果能提前发现该多好啊，这就是依赖检查的作用。

依赖检查：用于检查Bean定义的属性都注入数据了，不管是自动装配的还是配置方式注入的都能检查，如果没有注入数据将报错，从而提前发现注入错误，只检查具有setter方法的属性。

Spring3+也不推荐配置方式依赖检查了，建议采用Java5+ @Required注解方式，测试时请将XML schema降低为2.5版本的，和自动装配中“autodetect”配置方式的xsd一样。

java代码：

查看 复制到剪贴板 打印

```
<?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-2.5.xsd
</beans>
```

依赖检查有none、simple、object、all四种方式，接下来让我们详细介绍一下：

一、none：默认方式，表示不检查；

二、objects：检查除基本类型外的依赖对象，配置方式为：dependency-check="objects"，此处我们为HelloApiDecorator添加一个String类型属性“message”，来测试如果有简单数据类型的属性为null，也不报错；

java代码：

```
查看 复制到剪贴板 打印
<bean id="helloApi" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
<!-- 注意我们没有注入helloApi，所以测试时会报错 -->
<bean id="bean"
    class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
    dependency-check="objects">
<property name="message" value="Haha"/>
</bean>
```

注意由于我们没有注入bean需要的依赖“helloApi”，所以应该抛出异常UnsatisfiedDependencyException，表示没有发现满足的依赖：

java代码：

```
查看 复制到剪贴板 打印
package cn.javass.spring.chapter3;
import java.io.IOException;
import org.junit.Test;
import org.springframework.beans.factory.UnsatisfiedDependencyException;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class DependencyCheckTest {
@Test(expected = UnsatisfiedDependencyException.class)
public void testDependencyCheckByObject() throws IOException {
//将抛出异常
    new ClassPathXmlApplicationContext("chapter3/dependency-check-object.xml");
    }
}
```

三、simple：对基本类型进行依赖检查，包括数组类型，其他依赖不报错；配置方式为：dependency-check="simple"，以下配置中没有注入message属性，所以会抛出异常：

java代码：

```
查看 复制到剪贴板 打印
<bean id="helloApi" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
<!-- 注意我们没有注入message属性，所以测试时会报错 -->
<bean id="bean"
class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
    dependency-check="simple">
  <property name="helloApi" ref="helloApi"/>
</bean>
```

四、all：对所以类型进行依赖检查，配置方式为：dependency-check="all"，如下配置方式中如果两个属性其中一个没配置将报错。

java代码：

查看 复制到剪贴板 打印

```
<bean id="helloApi" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
<bean id="bean"
class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
    dependency-check="all">
  <property name="helloApi" ref="helloApi"/>
<property name="message" value="Haha"/>
</bean>
```

依赖检查也可以通过“<beans>”标签中default-dependency-check属性来指定全局依赖检查配置。

3.3.5 方法注入

所谓方法注入其实就是通过配置方式覆盖或拦截指定的方法，通常通过代理模式实现。Spring提供两种方法注入：查找方法注入和方法替换注入。

因为Spring是通过CGLIB动态代理方式实现方法注入，也就是通过动态修改类的字节码来实现的，本质就是生成需方法注入的类的子类方式实现。

在进行测试之前，我们需要确保将“com.springsource.cn.sf.cglib-2.2.0.jar”放到lib里并添加到“Java Build Path”中的Libararies中。否则报错，异常中包含**“nested exception is java.lang.NoClassDefFoundError: cn/sf/cglib/proxy/CallbackFilter”**。

传统方式和Spring容器管理方式唯一不同的是不需要我们手动生成子类，而是通过配置方式来实现；其中如果要替换createPrinter()方法的返回值就使用查找方法注入；如果想完全替换sayHello()方法体就使用方法替换注入。 接下来让我们看看具体实现吧。

一、查找方法注入：又称为Lookup方法注入，用于注入方法返回结果，也就是说能通过配置方式替换方法返回结果。使用<lookup-method name="方法名" bean="bean名字"/>配置；其中name属性指定方法名，bean属性指定方法需返回的Bean。

方法定义格式：访问级别必须是public或protected，保证能被子类重载，可以是抽象方法，必须有返回值，必须是无参数方法，查找方法的类和被重载的方法必须为非final：

<public|protected> [abstract] <return-type> theMethodName(no-arguments);

因为“singleton”Bean在容器中只有一个实例，而“prototype”Bean是每次获取容器都返回一个全新的实例，所以如果“singleton”Bean在使用“prototype” Bean情况时，那么“prototype”Bean由于是“singleton”Bean的一个字段属性，所以获取的这个“prototype”Bean就和它所在的“singleton”Bean具有同样的生命周期，所以不是我们所期待的结果。因此查找方法注入就是用于解决这个问题。

1） 首先定义我们需要的类，Printer类是一个有状态的类，counter字段记录访问次数：

java代码：

查看 复制到剪贴板 打印

```
1. package cn.javass.spring.chapter3.bean;
2. public class Printer {
3.     private int counter = 0;
4.     public void print(String type) {
5.         System.out.println(type + " printer: " + counter++);
6.     }
7. }
```

HelloImpl5类用于打印欢迎信息，其中包括setter注入和方法注入，此处特别需要注意的是该类是抽象的，充分说明了需要容器对其进行子类化处理，还定义了一个抽象方法createPrototypePrinter用于创建“prototype”Bean，createSingletonPrinter方法用于创建“singleton”Bean，此处注意方法会被Spring拦截，不会执行方法体代码：

java代码：

查看 复制到剪贴板 打印

```
1. package cn.javass.spring.chapter3;
2. import cn.javass.spring.chapter2.helloworld.HelloApi;
3. import cn.javass.spring.chapter3.bean.Printer;
4. public abstract class HelloImpl5 implements HelloApi {
5.     private Printer printer;
6.     public void sayHello() {
7.         printer.print("setter");
8.         createPrototypePrinter().print("prototype");
9.         createSingletonPrinter().print("singleton");
10.    }
11.    public abstract Printer createPrototypePrinter();
12.    public Printer createSingletonPrinter() {
13.        System.out.println("该方法不会被执行，如果输出就错了");
14.        return new Printer();
15.    }
16.    public void setPrinter(Printer printer) {
17.        this.printer = printer;
18.    }
19. }
```

2） 开始配置了，配置文件在（resources/chapter3/lookupMethodInject.xml），其中“prototypePrinter”是“prototype”Printer，“singletonPrinter”是“singleton”Printer，“helloApi1”是“singleton”Bean，而“helloApi2”注入了“prototype”Bean：

java代码：

```
查看 复制到剪贴板 打印
<bean id="prototypePrinter"
class="cn.javass.spring.chapter3.bean.Printer" scope="prototype"/>
<bean id="singletonPrinter"
class="cn.javass.spring.chapter3.bean.Printer" scope="singleton"/>
<bean id="helloApi1" class="cn.javass.spring.chapter3.HelloImpl5" scope="singleton">
<property name="printer" ref="prototypePrinter"/>
<lookup-method name="createPrototypePrinter" bean="prototypePrinter"/>
<lookup-method name="createSingletonPrinter" bean="singletonPrinter"/>
</bean>
<bean id="helloApi2" class="cn.javass.spring.chapter3.HelloImpl5" scope="prototype">
<property name="printer" ref="prototypePrinter"/>
<lookup-method name="createPrototypePrinter" bean="prototypePrinter"/>
<lookup-method name="createSingletonPrinter" bean="singletonPrinter"/>
</bean>
```

3）测试代码如下：

java代码：

```
查看 复制到剪贴板 打印
package cn.javass.spring.chapter3;
import org.junit.Test;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import cn.javass.spring.chapter2.helloworld.HelloApi;
public class MethodInjectTest {
@Test
public void testLookup() {
ClassPathXmlApplicationContext context =
new ClassPathXmlApplicationContext("chapter3/lookupMethodInject.xml");
        System.out.println("=======singleton sayHello======");
        HelloApi helloApi1 = context.getBean("helloApi1", HelloApi.class);
        helloApi1.sayHello();
        helloApi1 = context.getBean("helloApi1", HelloApi.class);
        helloApi1.sayHello();
        System.out.println("=======prototype sayHello======");
        HelloApi helloApi2 = context.getBean("helloApi2", HelloApi.class);
        helloApi2.sayHello();
        helloApi2 = context.getBean("helloApi2", HelloApi.class);
        helloApi2.sayHello();
}}
```

其中“helloApi1”测试中，其输出结果如下：

java代码：

```
查看 复制到剪贴板 打印
1. =======singleton sayHello======
2. setter printer: 0
3. prototype printer: 0
4. singleton printer: 0
5. setter printer: 1
6. prototype printer: 0
7. singleton printer: 1
```

首先“helloApi1”是“singleton”，通过setter注入的“printer”是“prototypePrinter”，所以它应该输出“setter printer:0”和“setter printer:1”；而“createPrototypePrinter”方法注入了“prototypePrinter”，所以应该输出两次“prototype printer:0”；而“createSingletonPrinter”注入了“singletonPrinter”，所以应该输出“singleton printer:0”和“singleton printer:1”。

而“helloApi2”测试中，其输出结果如下：

java代码：

```
查看 复制到剪贴板 打印
1.
2. =======prototype sayHello======
3. setter printer: 0
4. prototype printer: 0
5. singleton printer: 2
6. setter printer: 0
7. prototype printer: 0
8. singleton printer: 3
9.
```

首先“helloApi2”是“prototype”，通过setter注入的“printer”是“prototypePrinter”，所以它应该输出两次“setter printer:0”；而“createPrototypePrinter”方法注入了“prototypePrinter”，所以应该输出两次“prototype printer:0”；而“createSingletonPrinter”注入了“singletonPrinter”，所以应该输出“singleton printer:2”和“singleton printer:3”。

大家是否注意到“createSingletonPrinter”方法应该输出“该方法不会被执行，如果输出就错了”，而实际是没输出的，这说明Spring拦截了该方法并使用注入的Bean替换了返回结果。

方法注入主要用于处理“singleton”作用域的Bean需要其他作用域的Bean时，采用Spring查找方法注入方式无需修改任何代码即能获取需要的其他作用域的Bean。

二、替换方法注入：也叫“MethodReplacer”注入，和查找注入方法不一样的是，他主要用来替换方法体。通过首先定义一个MethodReplacer接口实现，然后如下配置来实现：

java代码：

```
查看 复制到剪贴板 打印
<replaced-method name="方法名" replacer="MethodReplacer实现">
<arg-type>参数类型</arg-type>
</replaced-method>”
```

1）首先定义MethodReplacer实现，完全替换掉被替换方法的方法体及返回值，其中reimplement方法重定义方法 功能，参数obj为被替换方法的对象，method为被替换方法，args为方法参数；最需要注意的是不能再 通过 “method.invoke(obj, new String[]{"hehe"});” 反射形式再去调用原来方法，这样会产生循环调用；如果返回值类型为Void，请在实现中返回null：

java代码：

```
查看 复制到剪贴板 打印
package cn.javass.spring.chapter3.bean;
import java.lang.reflect.Method;
import org.springframework.beans.factory.support.MethodReplacer;
public class PrinterReplacer implements MethodReplacer {
    @Override
    public Object reimplement(Object obj, Method method, Object[] args)  throws Throwable {
        System.out.println("Print Replacer");
        //注意此处不能再通过反射调用了,否则会产生循环调用，知道内存溢出
        //method.invoke(obj, new String[]{"hehe"});
        return null;
    }
}
```

2）配置如下，首先定义MethodReplacer实现，使用< replaced-method >标签来指定要进行替换方法，属性name指定替换的方法名字，replacer指定该方法的重新实现者，子标签< arg-type >用来指定原来方法参数的类型，必须指定否则找不到原方法：

java代码：

```
查看 复制到剪贴板 打印
<bean id="replacer" class="cn.javass.spring.chapter3.bean.PrinterReplacer"/>
<bean id="printer" class="cn.javass.spring.chapter3.bean.Printer">
<replaced-method name="print" replacer="replacer">
        <arg-type>java.lang.String</arg-type>
    </replaced-method>
</bean>
```

```
7.
```

3）测试代码将输出“Print Replacer ”，说明方法体确实被替换了：

java代码：

查看 复制到剪贴板 打印

```
1. @Test
2. public void testMethodReplacer() {
3.     ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("chapter3/
   methodReplacerInject.xml");
4.     Printer printer = context.getBean("printer", Printer.class);
5.     printer.print("我将被替换");
6. }
```

1.9 【第三章】 DI 之 3.4 Bean的作用域 ——跟我学spring3

发表时间: 2012-02-21 关键字: spring

3.4 Bean的作用域

什么是作用域呢？即“scope”，在面向对象程序设计中一般指对象或变量之间的可见范围。而在Spring容器中是指其创建的Bean对象相对于其他Bean对象的请求可见范围。

Spring提供“singleton”和“prototype”两种基本作用域，另外提供“request”、“session”、“global session”三种web作用域；Spring还允许用户定制自己的作用域。

3.4.1 基本的作用域

一、singleton：指“singleton”作用域的Bean只会在每个Spring IoC容器中存在一个实例，而且其完整生命周期完全由Spring容器管理。对于所有获取该Bean的操作Spring容器将只返回同一个Bean。

GoF单例设计模式指“**保证一个类仅有一个实例，并提供一个访问它的全局访问点**”，介绍了两种实现：通过在类上定义静态属性保持该实例和通过注册表方式。

1）通过在类上定义静态属性保持该实例：一般指一个Java虚拟机 ClassLoader装载的类只有一个实例，一般通过类静态属性保持该实例，这样就造成需要单例的类都需要按照单例设计模式进行编码；Spring没采用这种方式，因为该方式属于侵入式设计；代码样例如下：

java代码：

查看 复制到剪贴板 打印

```
1.  package cn.javass.spring.chapter3.bean;
2.  public class Singleton {
3.      //1.私有化构造器
4.      private Singleton() {}
5.      //2.单例缓存者，惰性初始化，第一次使用时初始化
6.      private static class InstanceHolder {
7.          private static final Singleton INSTANCE = new Singleton();
8.      }
9.      //3.提供全局访问点
10.     public static Singleton getInstance() {
11.         return InstanceHolder.INSTANCE;
12.     }
13.     //4.提供一个计数器来验证一个ClassLoader一个实例
14.     private int counter=0;
15. }
```

以上定义个了个单例类，首先要私有化类构造器；其次使用InstanceHolder静态内部类持有单例对象，这样可以得到惰性初始化好处；最后提供全局访问点getInstance，使得需要该单例实例的对象能获取到；我们在此还提供了一个counter计数器来验证一个ClassLoader一个实例。具体一个ClassLoader有一个单例实例测试请参考代码“cn.javass.spring.chapter3. SingletonTest”中的“testSingleton”测试方法，里边详细演示了一个ClassLoader有一个单例实例。

1） **通过注册表方式：** 首先将需要单例的实例通过唯一键注册到注册表，然后通过键来获取单例，让我们直接看实现吧，注意本注册表实现了Spring接口“SingletonBeanRegistry”，该接口定义了操作共享的单例对象，Spring容器实现将实现此接口；所以共享单例对象通过“registerSingleton”方法注册，通过“getSingleton”方法获取，消除了编程方式单例，注意在实现中不考虑并发：

java代码：

查看 复制到剪贴板 打印

```
1.  package cn.javass.spring.chapter3;
2.  import java.util.HashMap;
3.  import java.util.Map;
4.  import org.springframework.beans.factory.config.SingletonBeanRegistry;
5.  public class SingletonBeanRegister implements SingletonBeanRegistry {
6.     //单例Bean缓存池，此处不考虑并发
7.     private final Map<String, Object> BEANS = new HashMap<String, Object>();
8.     public boolean containsSingleton(String beanName) {
9.        return BEANS.containsKey(beanName);
10.    }
11.    public Object getSingleton(String beanName) {
12.       return BEANS.get(beanName);
13.    }
14.    @Override
15.    public int getSingletonCount() {
16.       return BEANS.size();
17.    }
18.    @Override
19.    public String[] getSingletonNames() {
20.       return BEANS.keySet().toArray(new String[0]);
21.    }
22.    @Override
23.    public void registerSingleton(String beanName, Object bean) {
24.       if(BEANS.containsKey(beanName)) {
25.          throw new RuntimeException("[" + beanName + "] 已存在");
26.       }
27.       BEANS.put(beanName, bean);
28. }
29. }
30.
```

Spring是注册表单例设计模式的实现，消除了编程式单例，而且对代码是非入侵式。

接下来让我们看看在Spring中如何配置单例Bean吧，在Spring容器中如果没指定作用域默认就是“singleton”，配置方式通过scope属性配置，具体配置如下：

java代码：

```
查看 复制到剪贴板 打印
1. <bean class="cn.javass.spring.chapter3.bean.Printer" scope="singleton"/>
```

Spring管理单例对象在Spring容器中存储如图3-5所示，Spring不仅会缓存单例对象，Bean定义也是会缓存的，对于惰性初始化的对象是在首次使用时根据Bean定义创建并存放于单例缓存池。

图3-5 单例处理

二、prototype：即原型，指每次向Spring容器请求获取Bean都返回一个全新的Bean，相对于“singleton”来说就是不缓存Bean，每次都是一个根据Bean定义创建的全新Bean。

GoF原型设计模式，指用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

Spring中的原型和GoF中介绍的原型含义是不一样的：

GoF通过用原型实例指定创建对象的种类，而Spring容器用Bean定义指定创建对象的种类；

GoF通过拷贝这些原型创建新的对象，而Spring容器根据Bean定义创建新对象。

其相同地方都是根据某些东西创建新东西，而且GoF原型必须显示实现克隆操作，属于侵入式，而Spring容器只需配置即可，属于非侵入式。

接下来让我们看看Spring如何实现原型呢？

1）首先让我们来定义Bean“原型”：Bean定义，所有对象将根据Bean定义创建；在此我们只是简单示例一下，不会涉及依赖注入等复杂实现：BeanDefinition类定义属性“class”表示原型类，“id”表示唯一标识，“scope”表示作用域，具体如下：

java代码：

查看 复制到剪贴板 打印

```
1.  package cn.javass.spring.chapter3;
2.  public class BeanDefinition {
3.      //单例
4.      public static final int SCOPE_SINGLETON = 0;
5.      //原型
6.      public static final int SCOPE_PROTOTYPE = 1;
7.      //唯一标识
8.      private String id;
9.      //class全限定名
10.     private String clazz;
11.     //作用域
12. private int scope = SCOPE_SINGLETON;
13.     //鉴于篇幅，省略setter和getter方法；
14. }
```

2）接下来让我们看看Bean定义注册表，类似于单例注册表：

java代码：

查看 复制到剪贴板 打印

```
1.  package cn.javass.spring.chapter3;
2.  import java.util.HashMap;
3.  import java.util.Map;
4.  public class BeanDifinitionRegister {
5.      //bean定义缓存，此处不考虑并发问题
6.  private final Map<String, BeanDefinition> DEFINITIONS =
7.   new HashMap<String, BeanDefinition>();
8.      public void registerBeanDefinition(String beanName, BeanDefinition bd) {
9.          //1.本实现不允许覆盖Bean定义
10.         if(DEFINITIONS.containsKey(bd.getId())) {
11.             throw new RuntimeException("已存在Bean定义，此实现不允许覆盖");
12.         }
13.         //2.将Bean定义放入Bean定义缓存池
14.         DEFINITIONS.put(bd.getId(), bd);
15.     }
16.     public BeanDefinition getBeanDefinition(String beanName) {
17.         return DEFINITIONS.get(beanName);
18.     }
19. public boolean containsBeanDefinition(String beanName) {
20.  return DEFINITIONS.containsKey(beanName);
21.     }
22. }
```

3）接下来应该来定义BeanFactory了：

java代码：

查看 复制到剪贴板 打印

```
package cn.javass.spring.chapter3;
import org.springframework.beans.factory.config.SingletonBeanRegistry;
public class DefaultBeanFactory {
    //Bean定义注册表
    private BeanDifinitionRegister DEFINITIONS = new BeanDifinitionRegister();

    //单例注册表
    private final SingletonBeanRegistry SINGLETONS = new SingletonBeanRegister();

    public Object getBean(String beanName) {
        //1.验证Bean定义是否存在
        if(!DEFINITIONS.containsBeanDefinition(beanName)) {
            throw new RuntimeException("不存在[" + beanName + "]Bean定义");
        }
        //2.获取Bean定义
        BeanDefinition bd = DEFINITIONS.getBeanDefinition(beanName);
        //3.是否该Bean定义是单例作用域
        if(bd.getScope() == BeanDefinition.SCOPE_SINGLETON) {
            //3.1 如果单例注册表包含Bean，则直接返回该Bean
            if(SINGLETONS.containsSingleton(beanName)) {
                return SINGLETONS.getSingleton(beanName);
            }
            //3.2单例注册表不包含该Bean，
            //则创建并注册到单例注册表，从而缓存
            SINGLETONS.registerSingleton(beanName, createBean(bd));
            return SINGLETONS.getSingleton(beanName);
        }
        //4.如果是原型Bean定义,则直接返回根据Bean定义创建的新Bean，
//每次都是新的，无缓存
        if(bd.getScope() == BeanDefinition.SCOPE_PROTOTYPE) {
            return createBean(bd);
        }
        //5.其他情况错误的Bean定义
        throw new RuntimeException("错误的Bean定义");
    }

```

java代码：

查看 复制到剪贴板 打印

```
public void registerBeanDefinition(BeanDefinition bd) {
    DEFINITIONS.registerBeanDefinition(bd.getId(), bd);
}

private Object createBean(BeanDefinition bd) {
    //根据Bean定义创建Bean
    try {
        Class clazz = Class.forName(bd.getClazz());
        //通过反射使用无参数构造器创建Bean
        return clazz.getConstructor().newInstance();
    } catch (ClassNotFoundException e) {
```

```
            throw new RuntimeException("没有找到Bean[" + bd.getId() + "]类");
        } catch (Exception e) {
            throw new RuntimeException("创建Bean[" + bd.getId() + "]失败");
        }
    }
```

其中方法getBean用于获取根据beanName对于的Bean定义创建的对象，有单例和原型两类Bean；registerBeanDefinition方法用于注册Bean定义，私有方法createBean用于根据Bean定义中的类型信息创建Bean。

3）测试一下吧，在此我们只测试原型作用域Bean，对于每次从Bean工厂中获取的Bean都是一个全新的对象，代码片段（BeanFatoryTest）如下：

java代码：

```
查看 复制到剪贴板 打印
@Test
public void testPrototype () throws Exception {
//1.创建Bean工厂
DefaultBeanFactory bf = new DefaultBeanFactory();
//2.创建原型 Bean定义
BeanDefinition bd = new BeanDefinition();
bd.setId("bean");
bd.setScope(BeanDefinition.SCOPE_PROTOTYPE);
bd.setClazz(HelloImpl2.class.getName());
bf.registerBeanDefinition(bd);
//对于原型Bean每次应该返回一个全新的Bean
System.out.println(bf.getBean("bean") != bf.getBean("bean"));
}
```

最后让我们看看如何在Spring中进行配置吧，只需指定<bean>标签属性“scope”属性为“prototype”即可：

java代码：

```
查看 复制到剪贴板 打印
<bean class="cn.javass.spring.chapter3.bean.Printer" scope="prototype"/>
```

Spring管理原型对象在Spring容器中存储如图3-6所示，Spring不会缓存原型对象，而是根据Bean定义每次请求返回一个全新的Bean：

图3-6 原型处理

单例和原型作用域我们已经讲完，接下来让我们学习一些在Web应用中有哪些作用域：

3.4.2 Web应用中的作用域

在Web应用中，我们可能需要将数据存储到request、session、global session。因此Spring提供了三种Web作用域：request、session、globalSession。

一、request作用域：表示每个请求需要容器创建一个全新Bean。比如提交表单的数据必须是对每次请求新建一个Bean来保持这些表单数据，请求结束释放这些数据。

二、session作用域：表示每个会话需要容器创建一个全新Bean。比如对于每个用户一般会有一个会话，该用户的用户信息需要存储到会话中，此时可以将该Bean配置为web作用域。

三、globalSession：类似于session作用域，只是其用于portlet环境的web应用。如果在非portlet环境将视为session作用域。

配置方式和基本的作用域相同，只是必须要有web环境支持，并配置相应的容器监听器或拦截器从而能应用这些作用域，我们会在集成web时讲解具体使用，大家只需要知道有这些作用域就可以了。

3.4.4 自定义作用域

在日常程序开发中，几乎用不到自定义作用域，除非又必要才进行自定义作用域。

首先让我们看下Scope接口吧：

java代码：

查看 复制到剪贴板 打印

```
1. package org.springframework.beans.factory.config;
2. import org.springframework.beans.factory.ObjectFactory;
3. public interface Scope {
```

```
    Object get(String name, ObjectFactory<?> objectFactory);
    Object remove(String name);
    void registerDestructionCallback(String name, Runnable callback);
    Object resolveContextualObject(String key);
    String getConversationId();
}
```

1）**Object get(String name, ObjectFactory<?> objectFactory)**：用于从作用域中获取Bean，其中参数objectFactory是当在当前作用域没找到合适Bean时使用它创建一个新的Bean；

2）**void registerDestructionCallback(String name, Runnable callback)**：用于注册销毁回调，如果想要销毁相应的对象则由Spring容器注册相应的销毁回调，而由自定义作用域选择是不是要销毁相应的对象；

3）**Object resolveContextualObject(String key)**：用于解析相应的上下文数据，比如request作用域将返回request中的属性。

4）**String getConversationId()**：作用域的会话标识，比如session作用域将是sessionId。

java代码：

查看 复制到剪贴板 打印

```
package cn.javass.spring.chapter3;
import java.util.HashMap;
import java.util.Map;
import org.springframework.beans.factory.ObjectFactory;
import org.springframework.beans.factory.config.Scope;
public class ThreadScope implements Scope {
private final ThreadLocal<Map<String, Object>> THREAD_SCOPE =
new ThreadLocal<Map<String, Object>>() {
    protected Map<String, Object> initialValue() {
        //用于存放线程相关Bean
        return new HashMap<String, Object>();
    }
};
```

让我们来实现个简单的thread作用域，该作用域内创建的对象将绑定到ThreadLocal内。

java代码：

查看 复制到剪贴板 打印

```
@Override
public Object get(String name, ObjectFactory<?> objectFactory) {
    //如果当前线程已经绑定了相应Bean，直接返回
    if(THREAD_SCOPE.get().containsKey(name)) {
        return THREAD_SCOPE.get().get(name);
    }
    //使用objectFactory创建Bean并绑定到当前线程上
    THREAD_SCOPE.get().put(name, objectFactory.getObject());
    return THREAD_SCOPE.get().get(name);
}
@Override
public String getConversationId() {
    return null;
}
@Override
public void registerDestructionCallback(String name, Runnable callback) {
    //此处不实现就代表类似proytotype，容器返回给用户后就不管了
}
@Override
public Object remove(String name) {
    return THREAD_SCOPE.get().remove(name);
}
@Override
public Object resolveContextualObject(String key) {
    return null;
}
}
```

Scope已经实现了，让我们将其注册到Spring容器，使其发挥作用：

java代码：

查看 复制到剪贴板 打印

```
<bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
    <property name="scopes">
      <map><entry>
            <!-- 指定scope关键字 --><key><value>thread</value></key>
            <!-- scope实现 -->      <bean class="cn.javass.spring.chapter3.ThreadScope"/>
      </entry></map>
    </property>
  </bean>
```

通过CustomScopeConfigurer的scopes属性注册自定义作用域实现，在此需要指定使用作用域的关键字“thread”，并指定自定义作用域实现。来让我们来定义一个“thread”作用域的Bean，配置（chapter3/threadScope.xml）如下：

java代码：

查看 复制到剪贴板 打印

```
1. <bean id="helloApi"
2.    class="cn.javass.spring.chapter2.helloworld.HelloImpl"
3.    scope="thread"/>
```

最后测试（cn.javass.spring.chapter3.ThreadScopeTest）一下吧，首先在一个线程中测试，在同一线程中获取的Bean应该是一样的；再让我们开启两个线程，然后应该这两个线程创建的Bean是不一样：

自定义作用域实现其实是非常简单的，其实复杂的是如果需要销毁Bean，自定义作用域如何正确的销毁Bean。

原创内容 转载请注明出处【http://sishuok.com/forum/blogPost/list/2454.html】

1.10 »Spring 之AOP AspectJ切入点语法详解（最全了，不需要再去其他地找了）

发表时间: 2012-02-21 关键字: spring, aop

6.5 AspectJ切入点语法详解

6.5.1 Spring AOP支持的AspectJ切入点指示符

切入点指示符用来指示切入点表达式目的，，在Spring AOP中目前只有执行方法这一个连接点，Spring AOP支持的AspectJ切入点指示符如下：

execution：用于匹配方法执行的连接点；

within：用于匹配指定类型内的方法执行；

this：用于匹配当前AOP代理对象类型的执行方法；注意是AOP代理对象的类型匹配，这样就可能包括引入接口也类型匹配；

target：用于匹配当前目标对象类型的执行方法；注意是目标对象的类型匹配，这样就不包括引入接口也类型匹配；

args：用于匹配当前执行的方法传入的参数为指定类型的执行方法；

@within：用于匹配所以持有指定注解类型内的方法；

@target：用于匹配当前目标对象类型的执行方法，其中目标对象持有指定的注解；

@args：用于匹配当前执行的方法传入的参数持有指定注解的执行；

@annotation：用于匹配当前执行方法持有指定注解的方法；

bean：Spring AOP扩展的，AspectJ没有对于指示符，用于匹配特定名称的Bean对象的执行方法；

reference pointcut：表示引用其他命名切入点，只有@ApectJ风格支持，Schema风格不支持。

AspectJ切入点支持的切入点指示符还有： call、get、set、preinitialization、staticinitialization、initialization、handler、adviceexecution、withincode、cflow、cflowbelow、if、@this、@withincode；但Spring AOP目前不支持这些指示符，使用这些指示符将抛出IllegalArgumentException异常。这些指示符Spring AOP可能会在以后进行扩展。

6.5.1 命名及匿名切入点

命名切入点可以被其他切入点引用，而匿名切入点是不可以的。

只有@AspectJ支持命名切入点，而Schema风格不支持命名切入点。

如下所示，@AspectJ使用如下方式引用命名切入点：

6.5.2 ；类型匹配语法

首先让我们来了解下AspectJ类型匹配的通配符：

*：匹配任何数量字符；

..：匹配任何数量字符的重复，如在类型模式中匹配任何数量子包；而在方法参数模式中匹配任何数量参数。

+：匹配指定类型的子类型；仅能作为后缀放在类型模式后边。

java代码：

查看复制到剪贴板打印

```
1. java.lang.String    匹配String类型；
2. java.*.String       匹配java包下的任何“一级子包”下的String类型；
3. 如匹配java.lang.String，但不匹配java.lang.ss.String
4. java..*             匹配java包及任何子包下的任何类型;
5.                     如匹配java.lang.String、java.lang.annotation.Annotation
6. java.lang.*ing      匹配任何java.lang包下的以ing结尾的类型；
7. java.lang.Number+   匹配java.lang包下的任何Number的自类型；
8.                     如匹配java.lang.Integer，也匹配java.math.BigInteger
```

接下来再看一下具体的匹配表达式类型吧：

匹配类型：使用如下方式匹配

java代码：

查看复制到剪贴板打印

1. 注解？ 类的全限定名字

- **注解：**可选，类型上持有的注解，如@Deprecated；
- **类的全限定名：**必填，可以是任何类全限定名。

匹配方法执行：使用如下方式匹配：

java代码：

查看复制到剪贴板打印

1. 注解？ 修饰符? 返回值类型 类型声明?方法名(参数列表) 异常列表？

- **注解：**可选，方法上持有的注解，如@Deprecated；
- **修饰符：**可选，如public、protected；
- **返回值类型：**必填，可以是任何类型模式；“*”表示所有类型；
- **类型声明：**可选，可以是任何类型模式；
- **方法名：**必填，可以使用“*”进行模式匹配；
- **参数列表：**“()”表示方法没有任何参数；“(..)”表示匹配接受任意个参数的方法，“(..,java.lang.String)”表示匹配接受java.lang.String类型的参数结束，且其前边可以接受有任意个参数的方法；“(java.lang.String,..)” 表示匹配接受java.lang.String类型的参数开始，且其后边可以接受任意个参数的方法；“(*,java.lang.String)” 表示匹配接受java.lang.String类型的参数结束，且其前边接受有一个任意类型参数的方法；
- **异常列表：**可选，以“throws 异常全限定名列表”声明，异常全限定名列表如有多个以“，”分割，如throws java.lang.IllegalArgumentException, java.lang.ArrayIndexOutOfBoundsException。

匹配Bean名称：可以使用Bean的id或name进行匹配，并且可使用通配符“*” ；

6.5.3 组合切入点表达式

AspectJ使用 且（&&）、或（||）、非（！）来组合切入点表达式。

在Schema风格下，由于在XML中使用“&&”需要使用转义字符“&”来代替之，所以很不方便，因此Spring ASP 提供了and、or、not来代替&&、||、！。

6.5.3 切入点使用示例

一、execution：使用“execution(方法表达式)”匹配方法执行；

模式	描述
public * *(..)	任何公共方法的执行
* cn.javass..IPointcutService.*()	cn.javass包及所有子包下IPointcutService接口中的任何无参方法
* cn.javass..*.*(..)	cn.javass包及所有子包下任何类的任何方法
* cn.javass..IPointcutService.*(*)	cn.javass包及所有子包下IPointcutService接口的任何只有一个参数方法
* (!cn.javass..IPointcutService+).*(..)	非“cn.javass包及所有子包下IPointcutService接口及子类型”的任何方法
* cn.javass..IPointcutService+.*()	cn.javass包及所有子包下IPointcutService接口及子类型的的任何无参方法
* cn.javass..IPointcut*.test*(java.util.Date)	cn.javass包及所有子包下IPointcut前缀类型的的以test开头的只有一个参数类型为java.util.Date的方法，注意该匹配是根据方法签名的参数类型进行匹配的，而不是根据执行时传入的参数类型决定的 如定义方法：public void test(Object obj);即使执行时传入java.util.Date，也不会匹配的；
* cn.javass..IPointcut*.test*(..) throws IllegalArgumentException, ArrayIndexOutOfBoundsException	cn.javass包及所有子包下IPointcut前缀类型的的任何方法，且抛出IllegalArgumentException和ArrayIndexOutOfBoundsException异常

* (cn.javass..IPointcutService+ && java.io.Serializable+).*(..)	任何实现了cn.javass包及所有子包下IPointcutService接口和java.io.Serializable接口的类型的任何方法
@java.lang.Deprecated * *(..)	任何持有@java.lang.Deprecated注解的方法
@java.lang.Deprecated @cn.javass..Secure * *(..)	任何持有@java.lang.Deprecated和@cn.javass..Secure注解的方法
@(java.lang.Deprecated || cn.javass..Secure) * *(..)	任何持有@java.lang.Deprecated或@ cn.javass..Secure注解的方法
(@cn.javass..Secure *) *(..)	任何返回值类型持有@cn.javass..Secure的方法
* (@cn.javass..Secure *).*(..)	任何定义方法的类型持有@cn.javass..Secure的方法
* *(@cn.javass..Secure (*) , @cn.javass..Secure (*))	任何签名带有两个参数的方法，且这个两个参数都被@Secure标记了， 如public void test(@Secure String str1, @Secure String str1);
* *((@ cn.javass..Secure *))或 * *(@ cn.javass..Secure *)	任何带有一个参数的方法，且该参数类型持有@cn.javass..Secure； 如public void test(Model model);且Model类上持有@Secure注解
* *(@cn.javass..Secure (@cn.javass..Secure *) , @ cn.javass..Secure (@cn.javass..Secure *))	任何带有两个参数的方法，且这两个参数都被@cn.javass..Secure标记了；且这两个参数的类型上都持有@ cn.javass..Secure；

* *(java.util.Map<cn.javass..Model, cn.javass..Model>, ..)	任何带有一个java.util.Map参数的方法，且该参数类型是以< cn.javass..Model, cn.javass..Model >为泛型参数；注意只匹配第一个参数为java.util.Map,不包括子类型；如public void test(HashMap<Model, Model> map, String str);将不匹配，必须使用“* *(java.util.HashMap<cn.javass..Model,cn.javass..Model>, ..)”进行匹配；而public void test(Map map, int i);也将不匹配，因为泛型参数不匹配
* *(java.util.Collection<@cn.javass..Secure *>)	任何带有一个参数（类型为java.util.Collection）的方法，且该参数类型是有一个泛型参数，该泛型参数类型上持有@cn.javass..Secure注解；如public void test(Collection<Model> collection);Model类型上持有@cn.javass..Secure
* *(java.util.Set<? extends HashMap>)	任何带有一个参数的方法，且传入的参数类型是有一个泛型参数，该泛型参数类型继承与HashMap；**Spring AOP目前测试不能正常工作**
* *(java.util.List<? super HashMap>)	任何带有一个参数的方法，且传入的参数类型是有一个泛型参数，该泛型参数类型是HashMap的基类型；如public voi test(Map map)；**Spring AOP目前测试不能正常工作**
* *(*<@cn.javass..Secure *>)	任何带有一个参数的方法，且该参数类型是有一个泛型参数，该泛型参数类型上持有@cn.javass..Secure注解；**Spring AOP目前测试不能正常工作**

二、within：使用“within(类型表达式)”匹配指定类型内的方法执行；

模式	描述
within(cn.javass..*)	cn.javass包及子包下的任何方法执行
within(cn.javass..IPointcutService+)	cn.javass包或所有子包下IPointcutService类型及子类型的任何方法
within(@cn.javass..Secure *)	持有cn.javass..Secure注解的任何类型的任何方法 必须是在目标对象上声明这个注解，在接口上声明的对它不起作用

三、this：使用“this(类型全限定名)”匹配当前AOP代理对象类型的执行方法；注意是AOP代理对象的类型匹配，这样就可能包括引入接口方法也可以匹配；注意this中使用的表达式必须是类型全限定名，不支持通配符；

模式	描述
this(cn.javass.spring.chapter6.service.IPointcutService)	当前AOP对象实现了 IPointcutService接口的任何方法
this(cn.javass.spring.chapter6.service.IIntroductionService)	当前AOP对象实现了 IIntroductionService接口的任何方法 也可能是引入接口

四、**target：**使用“target(类型全限定名)”匹配当前目标对象类型的执行方法；注意是目标对象的类型匹配，这样就不包括引入接口也类型匹配；注意target中使用的表达式必须是类型全限定名，不支持通配符；

模式	描述
target(cn.javass.spring.chapter6.service.IPointcutService)	当前目标对象（非AOP对象）实现了IPointcutService接口的任何方法
target(cn.javass.spring.chapter6.service.IIntroductionService)	当前目标对象（非AOP对象） 实现了IIntroductionService 接口的任何方法 不可能是引入接口

五、**args：**使用“args(参数类型列表)”匹配当前执行的方法传入的参数为指定类型的执行方法；注意是匹配传入的参数类型，不是匹配方法签名的参数类型；参数类型列表中的参数必须是类型全限定名，通配符不支持；args属于动态切入点，这种切入点开销非常大，非特殊情况最好不要使用；

模式	描述

args (java.io.Serializable,..)	任何一个以接受“传入参数类型为java.io.Serializable” 开头，且其后可跟任意个任意类型的参数的方法执行，args指定的参数类型是在运行时动态匹配的

六、@within：使用“@within(注解类型)”匹配所以持有指定注解类型内的方法；注解类型也必须是全限定类型名；

模式	**描述**
@within cn.javass.spring.chapter6.Secure)	任何目标对象对应的类型持有Secure注解的类方法；必须是在目标对象上声明这个注解，在接口上声明的对它不起作用

七、@target：使用“@target(注解类型)”匹配当前目标对象类型的执行方法，其中目标对象持有指定的注解；注解类型也必须是全限定类型名；

模式	描述
@target (cn.javass.spring.chapter6.Secure)	任何目标对象持有Secure注解的类方法；必须是在目标对象上声明这个注解，在接口上声明的对它不起作用

八、@args：使用“@args(注解列表)”匹配当前执行的方法传入的参数持有指定注解的执行；注解类型也必须是全限定类型名；

模式	描述
@args (cn.javass.spring.chapter6.Secure)	任何一个只接受一个参数的方法，且方法运行时传入的参数持有注解 cn.javass.spring.chapter6.Secure；动态切入点，类似于arg指示符；

九、@annotation：使用“@annotation(注解类型)”匹配当前执行方法持有指定注解的方法；注解类型也必须是全限定类型名；

模式	描述
@annotation(cn.javass.spring.chapter6.Secure)	当前执行方法上持有注解cn.javass.spring.chapter6.Secure将被匹配

十、bean：使用“bean(Bean id或名字通配符)”匹配特定名称的Bean对象的执行方法；Spring ASP扩展的，在AspectJ中无相应概念；

模式	描述
bean(*Service)	匹配所有以Service命名（id或name）结尾的Bean

十一、reference pointcut：表示引用其他命名切入点，只有@ApectJ风格支持，Schema风格不支持，如下所示：

比如我们定义如下切面：

java代码：

查看复制到剪贴板打印

```
1. package cn.javass.spring.chapter6.aop;
2. import org.aspectj.lang.annotation.Aspect;
3. import org.aspectj.lang.annotation.Pointcut;
4. @Aspect
5. public class ReferencePointcutAspect {
6.     @Pointcut(value="execution(* *())")
7.     public void pointcut() {}
8. }
```

可以通过如下方式引用：

java代码：

查看复制到剪贴板打印

```
1. @Before(value = "cn.javass.spring.chapter6.aop.ReferencePointcutAspect.pointcut()")
2. public void referencePointcutTest2(JoinPoint jp) {}
```

除了可以在@AspectJ风格的切面内引用外，也可以在Schema风格的切面定义内引用，引用方式与@AspectJ完全一样。

到此我们切入点表达式语法示例就介绍完了，我们这些示例几乎包含了日常开发中的所有情况，但当然还有更复杂的语法等等，如果以上介绍的不能满足您的需要，请参考AspectJ文档。

由于测试代码相当长，所以为了节约篇幅本示例代码在cn.javass.spring.chapter6. PointcutTest文件中，需要时请参考该文件。

6.6 通知参数

前边章节已经介绍了声明通知，但如果想获取被被通知方法参数并传递给通知方法，该如何实现呢？接下来我们将介绍两种获取通知参数的方式。

- **使用JoinPoint获取**：Spring AOP提供使用org.aspectj.lang.JoinPoint类型获取连接点数据，任何通知方法的第一个参数都可以是JoinPoint(环绕通知是ProceedingJoinPoint，JoinPoint子类)，当然第一个参数位置也可以是JoinPoint.StaticPart类型，这个只返回连接点的静态部分。

1) JoinPoint：提供访问当前被通知方法的目标对象、代理对象、方法参数等数据：

java代码：

查看复制到剪贴板打印

```
1.  package org.aspectj.lang;
2.  import org.aspectj.lang.reflect.SourceLocation;
3.  public interface JoinPoint {
4.      String toString();         //连接点所在位置的相关信息
5.      String toShortString();     //连接点所在位置的简短相关信息
6.      String toLongString();     //连接点所在位置的全部相关信息
7.      Object getThis();          //返回AOP代理对象
8.      Object getTarget();        //返回目标对象
9.      Object[] getArgs();        //返回被通知方法参数列表
10.     Signature getSignature();  //返回当前连接点签名
11.     SourceLocation getSourceLocation();//返回连接点方法所在类文件中的位置
12.     String getKind();          //连接点类型
13.     StaticPart getStaticPart(); //返回连接点静态部分
14. }
```

2) ProceedingJoinPoint：用于环绕通知，使用proceed()方法来执行目标方法：

java代码：

查看复制到剪贴板打印

```
1. public interface ProceedingJoinPoint extends JoinPoint {
2.     public Object proceed() throws Throwable;
3.     public Object proceed(Object[] args) throws Throwable;
4. }
```

3) JoinPoint.StaticPart：提供访问连接点的静态部分，如被通知方法签名、连接点类型等：

java代码：

查看复制到剪贴板打印

```
1. public interface StaticPart {
2. Signature getSignature();    //返回当前连接点签名
3. String getKind();          //连接点类型
4.     int getId();             //唯一标识
5. String toString();         //连接点所在位置的相关信息
6.     String toShortString();    //连接点所在位置的简短相关信息
7.     String toLongString();    //连接点所在位置的全部相关信息
8. }
```

使用如下方式在通知方法上声明，必须是在第一个参数，然后使用jp.getArgs()就能获取到被通知方法参数：

java代码：

查看复制到剪贴板打印

```
1. @Before(value="execution(* sayBefore(*))")
2. public void before(JoinPoint jp) {}
3.
4. @Before(value="execution(* sayBefore(*))")
```

```
5. public void before(JoinPoint.StaticPart jp) {}
```

- **自动获取：**通过切入点表达式可以将相应的参数自动传递给通知方法，例如前边章节讲过的返回值和异常是如何传递给通知方法的。

在Spring AOP中，除了execution和bean指示符不能传递参数给通知方法，其他指示符都可以将匹配的相应参数或对象自动传递给通知方法。

java代码：

查看复制到剪贴板打印

```
1. @Before(value="execution(* test(*)) && args(param)", argNames="param")
2. public void before1(String param) {
3.     System.out.println("===param:" + param);
4. }
```

切入点表达式execution(* test(*)) && args(param) ：

1）首先execution(* test(*))匹配任何方法名为test，且有一个任何类型的参数；

2）args(param)将首先查找通知方法上同名的参数，并在方法执行时（运行时）匹配传入的参数是使用该同名参数类型，即java.lang.String；如果匹配将把该被通知参数传递给通知方法上同名参数。

其他指示符（除了execution和bean指示符）都可以使用这种方式进行参数绑定。

在此有一个问题，即前边提到的类似于【3.1.2构造器注入】中的参数名注入限制：**在class文件中没生成变量调试信息是获取不到方法参数名字的。**

所以我们可以使用策略来确定参数名：

1. 如果我们通过“argNames”属性指定了参数名，那么就是要我们指定的；

java代码：

查看复制到剪贴板打印

```
1. @Before(value=" args(param)", argNames="param") //明确指定了
2. public void before1(String param) {
3.     System.out.println("===param:" + param);
4. }
```

1. 如果第一个参数类型是JoinPoint、ProceedingJoinPoint或JoinPoint.StaticPart类型，应该从“argNames”属性省略掉该参数名（可选，写上也对），这些类型对象会自动传入的，但必须作为第一个参数；

java代码：

查看复制到剪贴板打印

```
1. @Before(value=" args(param)", argNames="param") //明确指定了
2. public void before1(JoinPoint jp, String param) {
3.     System.out.println("===param:" + param);
4. }
```

1. 如果“**class文件中含有变量调试信息**”将使用这些方法签名中的参数名来确定参数名；

java代码：

查看复制到剪贴板打印

```
1. @Before(value=" args(param)") //不需要argNames了
2. public void before1(JoinPoint jp, String param) {
3.     System.out.println("===param:" + param);
4. }
```

1. 如果没有“**class文件中含有变量调试信息**”，将尝试自己的参数匹配算法，如果发现参数绑定有二义性将抛出AmbiguousBindingException异常；对于只有一个绑定变量的切入点表达式，而通知方法只接受一个参数，说明绑定参数是明确的，从而能配对成功。

java代码：

查看复制到剪贴板打印

```
1. @Before(value=" args(param)")
2. public void before1(JoinPoint jp, String param) {
3.     System.out.println("===param:" + param);
4. }
```

1. 以上策略失败将抛出IllegalArgumentException。

接下来让我们示例一下组合情况吧：

java代码：

查看复制到剪贴板打印

```
1. @Before(args(param) && target(bean) && @annotation(secure)",
2.         argNames="jp,param,bean,secure")
3. public void before5(JoinPoint jp, String param,
4.  IPointcutService pointcutService, Secure secure) {
5. ……
6. }
```

该示例的执行步骤如图6-5所示。

图6-5 参数自动获取流程

除了上边介绍的普通方式，也可以对使用命名切入点自动获取参数：

java代码：

查看复制到剪贴板打印

```
1. @Pointcut(value="args(param)", argNames="param")
2. private void pointcut1(String param){}
3. @Pointcut(value="@annotation(secure)", argNames="secure")
4. private void pointcut2(Secure secure){}
5.
6. @Before(value = "pointcut1(param) && pointcut2(secure)",
7.  argNames="param, secure")
8. public void before6(JoinPoint jp, String param, Secure secure) {
9. ……
10. }
```

自此给通知传递参数已经介绍完了，示例代码在cn.javass.spring.chapter6.ParameterTest文件中。

在Spring配置文件中，所以AOP相关定义必须放在<aop:config>标签下，该标签下可以有<aop:pointcut>、<aop:advisor>、<aop:aspect>标签，配置顺序不可变。

- <aop:pointcut>：用来定义切入点，该切入点可以重用；
- <aop:advisor>：用来定义只有一个通知和一个切入点的切面；
- <aop:aspect>：用来定义切面，该切面可以包含多个切入点和通知，而且标签内部的通知和切入点定义是无序的；和advisor的区别就在此，advisor只包含一个通知和一个切入点。

1.11 【第四章】 资源 之 4.1 基础知识 ——跟我学spring3

发表时间: 2012-02-22

4.1.1 概述

在日常程序开发中，处理外部资源是很繁琐的事情，我们可能需要处理URL资源、File资源资源、ClassPath相关资源、服务器相关资源（JBoss AS 5.x上的VFS资源）等等很多资源。因此处理这些资源需要使用不同的接口，这就增加了我们系统的复杂性；而且处理这些资源步骤都是类似的（打开资源、读取资源、关闭资源），因此如果能抽象出一个统一的接口来对这些底层资源进行统一访问，是不是很方便，而且使我们系统更加简洁，都是对不同的底层资源使用同一个接口进行访问。

Spring 提供一个Resource接口来统一这些底层资源一致的访问，而且提供了一些便利的接口，从而能提供我们的生产力。

4.1.2 Resource接口

Spring的Resource接口代表底层外部资源，提供了对底层外部资源的一致性访问接口。

java代码：

```
public interface InputStreamSource {
    InputStream getInputStream() throws IOException;
}
```

java代码：

```
public interface Resource extends InputStreamSource {
       boolean exists();
       boolean isReadable();
       boolean isOpen();
       URL getURL() throws IOException;
       URI getURI() throws IOException;
       File getFile() throws IOException;
       long contentLength() throws IOException;
```

```
        long lastModified() throws IOException;
        Resource createRelative(String relativePath) throws IOException;
        String getFilename();
        String getDescription();
}
```

1）InputStreamSource接口解析：

getInputStream：每次调用都将返回一个新鲜的资源对应的java.io. InputStream字节流，调用者在使用完毕后必须关闭该资源。

2）Resource接口继承InputStreamSource接口，并提供一些便利方法：

exists：返回当前Resource代表的底层资源是否存在，true表示存在。

isReadable：返回当前Resource代表的底层资源是否可读，true表示可读。

isOpen：返回当前Resource代表的底层资源是否已经打开，如果返回true，则只能被读取一次然后关闭以避免内存泄漏；常见的Resource实现一般返回false。

getURL：如果当前Resource代表的底层资源能由java.util.URL代表，则返回该URL，否则抛出IOException。

getURI：如果当前Resource代表的底层资源能由java.util.URI代表，则返回该URI，否则抛出IOException。

getFile：如果当前Resource代表的底层资源能由java.io.File代表，则返回该File，否则抛出IOException。

contentLength：返回当前Resource代表的底层文件资源的长度，一般是值代表的文件资源的长度。

lastModified：返回当前Resource代表的底层资源的最后修改时间。

createRelative：用于创建相对于当前Resource代表的底层资源的资源，比如当前Resource代表文件资源“d:/test/”则createRelative（ “test.txt” ）将返回表文件资源“d:/test/test.txt”Resource资源。

getFilename：返回当前Resource代表的底层文件资源的文件路径，比如File资源“file://d:/test.txt”将返回“d:/test.txt” ，而URL资源http://www.javass.cn将返回 “” ，因为只返回文件路径。

getDescription：返回当前Resource代表的底层资源的描述符，通常就是资源的全路径（实际文件名或实际URL地址）。

Resource接口提供了足够的抽象，足够满足我们日常使用。而且提供了很多内置Resource实现：ByteArrayResource、InputStreamResource 、FileSystemResource 、UrlResource 、ClassPathResource、ServletContextResource、VfsResource等。

1.12 【第四章】 资源 之 4.2 内置Resource实现 ——跟我学spring3

发表时间: 2012-02-22 关键字: spring

4.2 内置Resource实现

4.2.1 ByteArrayResource

ByteArrayResource代表byte[]数组资源，对于“getInputStream”操作将返回一个ByteArrayInputStream。

首先让我们看下使用ByteArrayResource如何处理byte数组资源：

java代码：

```
package cn.javass.spring.chapter4;
import java.io.IOException;
import java.io.InputStream;
import org.junit.Test;
import org.springframework.core.io.ByteArrayResource;
import org.springframework.core.io.Resource;
public class ResourceTest {
@Test
public void testByteArrayResource() {
Resource resource = new ByteArrayResource("Hello World!".getBytes());
        if(resource.exists()) {
            dumpStream(resource);
        }
}
}
```

是不是很简单，让我们看下“dumpStream”实现：

java代码：

```
private void dumpStream(Resource resource) {
        InputStream is = null;
        try {
            //1.获取文件资源
            is = resource.getInputStream();
            //2.读取资源
            byte[] descBytes = new byte[is.available()];
            is.read(descBytes);
            System.out.println(new String(descBytes));
        } catch (IOException e) {
            e.printStackTrace();
        }
        finally {
            try {
                //3.关闭资源
                is.close();
            } catch (IOException e) {
            }
        }
    }
```

让我们来仔细看一下代码，dumpStream方法很抽象定义了访问流的三部曲：打开资源、读取资源、关闭资源，所以dunpStrean可以再进行抽象从而能在自己项目中使用；byteArrayResourceTest测试方法，也定义了基本步骤：定义资源、验证资源存在、访问资源。

ByteArrayResource可多次读取数组资源，即isOpen ()永远返回false。

1.2.2 InputStreamResource

InputStreamResource代表java.io.InputStream字节流，对于“getInputStream ”操作将直接返回该字节流，因此只能读取一次该字节流，即“isOpen”永远返回true。

让我们看下测试代码吧：

java代码：

```
@Test
public void testInputStreamResource() {
   ByteArrayInputStream bis = new ByteArrayInputStream("Hello World!".getBytes());
   Resource resource = new InputStreamResource(bis);
    if(resource.exists()) {
       dumpStream(resource);
    }
    Assert.assertEquals(true, resource.isOpen());
}
```

测试代码几乎和ByteArrayResource测试完全一样，注意“isOpen”此处用于返回true。

4.2.3 FileSystemResource

FileSystemResource代表java.io.File资源，对于“getInputStream ”操作将返回底层文件的字节流，“isOpen”将永远返回false，从而表示可多次读取底层文件的字节流。

让我们看下测试代码吧：

java代码：

```
@Test
public void testFileResource() {
File file = new File("d:/test.txt");
    Resource resource = new FileSystemResource(file);
    if(resource.exists()) {
        dumpStream(resource);
    }
    Assert.assertEquals(false, resource.isOpen());
}
```

注意由于“isOpen”将永远返回false，所以可以多次调用dumpStream(resource)。

4.2.4 ClassPathResource

ClassPathResource代表classpath路径的资源，将使用ClassLoader进行加载资源。classpath 资源存在于类路径中的文件系统中或jar包里，且“isOpen”永远返回false，表示可多次读取资源。

ClassPathResource加载资源替代了Class类和ClassLoader类的“getResource(String name)”和“getResourceAsStream(String name)”两个加载类路径资源方法，提供一致的访问方式。

ClassPathResource提供了三个构造器：

public ClassPathResource(String path)：使用默认的ClassLoader加载“path”类路径资源；

public ClassPathResource(String path, ClassLoader classLoader)：使用指定的ClassLoader加载“path”类路径资源；

比如当前类路径是“cn.javass.spring.chapter4.ResourceTest”，而需要加载的资源路径是“cn/javass/spring/chapter4/test1.properties”，则将加载的资源在“cn/javass/spring/chapter4/test1.properties”；

public ClassPathResource(String path, Class<?> clazz)：使用指定的类加载“path”类路径资源，将加载相对于当前类的路径的资源；

比如当前类路径是“cn.javass.spring.chapter4.ResourceTest”，而需要加载的资源路径是“cn/javass/spring/chapter4/test1.properties”，则将加载的资源在“cn/javass/spring/chapter4/cn/javass/spring/chapter4/test1.properties”；

而如果需要 加载的资源路径为“test1.properties”，将加载的资源为“cn/javass/spring/chapter4/test1.properties”。

让我们直接看测试代码吧：

1）使用默认的加载器加载资源，将加载当前ClassLoader类路径上相对于根路径的资源：

java代码：

```
@Test
public void testClasspathResourceByDefaultClassLoader() throws IOException {
   Resource resource = new ClassPathResource("cn/javass/spring/chapter4/test1.properties");
    if(resource.exists()) {
        dumpStream(resource);
    }
    System.out.println("path:" + resource.getFile().getAbsolutePath());
    Assert.assertEquals(false, resource.isOpen());
}
```

2）使用指定的ClassLoader进行加载资源，将加载指定的ClassLoader类路径上相对于根路径的资源：

java代码：

```
@Test
public void testClasspathResourceByClassLoader() throws IOException {
    ClassLoader cl = this.getClass().getClassLoader();
    Resource resource = new ClassPathResource("cn/javass/spring/chapter4/test1.properties" ,
```

```
    if(resource.exists()) {
        dumpStream(resource);
    }
    System.out.println("path:" + resource.getFile().getAbsolutePath());
    Assert.assertEquals(false, resource.isOpen());
}
```

3）使用指定的类进行加载资源，将尝试加载相对于当前类的路径的资源：

java代码：

```
@Test
public void testClasspathResourceByClass() throws IOException {
   Class clazz = this.getClass();
    Resource resource1 = new ClassPathResource("cn/javass/spring/chapter4/test1.properties"
    if(resource1.exists()) {
        dumpStream(resource1);
    }
    System.out.println("path:" + resource1.getFile().getAbsolutePath());
    Assert.assertEquals(false, resource1.isOpen());

    Resource resource2 = new ClassPathResource("test1.properties" , this.getClass());
    if(resource2.exists()) {
        dumpStream(resource2);
   }
    System.out.println("path:" + resource2.getFile().getAbsolutePath());
    Assert.assertEquals(false, resource2.isOpen());
}
```

“resource1”将加载cn/javass/spring/chapter4/cn/javass/spring/chapter4/test1.properties资源；“resource2”将加载“cn/javass/spring/chapter4/test1.properties”；

4）加载jar包里的资源，首先在当前类路径下找不到，最后才到Jar包里找，而且在第一个Jar包里找到的将被返回：

java代码：

```
@Test
public void classpathResourceTestFromJar() throws IOException {
Resource resource = new ClassPathResource("overview.html");
    if(resource.exists()) {
        dumpStream(resource);
    }
    System.out.println("path:" + resource.getURL().getPath());
    Assert.assertEquals(false, resource.isOpen());
}
```

如果当前类路径包含“overview.html”，在项目的“resources”目录下，将加载该资源，否则将加载Jar包里的“overview.html”，而且不能使用“resource.getFile()”，应该使用“resource.getURL()”，因为资源不存在于文件系统而是存在于jar包里，URL类似于“file:/C:/.../***.jar!/overview.html”。

类路径一般都是相对路径，即相对于类路径或相对于当前类的路径，因此如果使用“/test1.properties”带前缀“/”的路径，将自动删除“/”得到“test1.properties”。

4.2.5 UrlResource

UrlResource代表URL资源，用于简化URL资源访问。“isOpen”永远返回false，表示可多次读取资源。

UrlResource一般支持如下资源访问：

http：通过标准的http协议访问web资源，如new UrlResource(“http://地址”)；

ftp：通过ftp协议访问资源，如new UrlResource(“ftp://地址”)；

file：通过file协议访问本地文件系统资源，如new UrlResource(“file:d:/test.txt”)；

具体使用方法在此就不演示了，可以参考cn.javass.spring.chapter4.ResourceTest中urlResourceTest测试方法。

4.2.6 ServletContextResource

ServletContextResource代表web应用资源，用于简化servlet容器的ServletContext接口的getResource操作和getResourceAsStream操作；在此就不具体演示了。

4.2.7 VfsResource

VfsResource代表Jboss 虚拟文件系统资源。

Jboss VFS(Virtual File System)框架是一个文件系统资源访问的抽象层，它能一致的访问物理文件系统、jar资源、zip资源、war资源等，VFS能把这些资源一致的映射到一个目录上，访问它们就像访问物理文件资源一样，而其实这些资源不存在于物理文件系统。

在示例之前需要准备一些jar包，在此我们使用的是Jboss VFS3版本，可以下载最新的Jboss AS 6x，拷贝lib目录下的“jboss-logging.jar”和“jboss-vfs.jar”两个jar包拷贝到我们项目的lib目录中并添加到“Java Build Path”中的“Libaries”中。

让我们看下示例（cn.javass.spring.chapter4.ResourceTest）：

java代码：

```
@Test
public void testVfsResourceForRealFileSystem() throws IOException {
//1.创建一个虚拟的文件目录
VirtualFile home = VFS.getChild("/home");
//2.将虚拟目录映射到物理的目录
VFS.mount(home, new RealFileSystem(new File("d:")));
//3.通过虚拟目录获取文件资源
VirtualFile testFile = home.getChild("test.txt");
//4.通过一致的接口访问
Resource resource = new VfsResource(testFile);
if(resource.exists()) {
```

```
        dumpStream(resource);
}
System.out.println("path:" + resource.getFile().getAbsolutePath());
Assert.assertEquals(false, resource.isOpen());
}
@Test
public void testVfsResourceForJar() throws IOException {
//1.首先获取jar包路径
    File realFile = new File("lib/org.springframework.beans-3.0.5.RELEASE.jar");
    //2.创建一个虚拟的文件目录
    VirtualFile home = VFS.getChild("/home2");
    //3.将虚拟目录映射到物理的目录
VFS.mountZipExpanded(realFile, home,
TempFileProvider.create("tmp", Executors.newScheduledThreadPool(1)));
//4.通过虚拟目录获取文件资源
    VirtualFile testFile = home.getChild("META-INF/spring.handlers");
    Resource resource = new VfsResource(testFile);
    if(resource.exists()) {
            dumpStream(resource);
    }
    System.out.println("path:" + resource.getFile().getAbsolutePath());
    Assert.assertEquals(false, resource.isOpen());
}
```

通过VFS，对于jar里的资源和物理文件系统访问都具有一致性，此处只是简单示例，如果需要请到Jboss官网深入学习。

原创内容 转自请注明出处【http://sishuok.com/forum/blogPost/list/0/2456.html】

1.13 【第四章】 资源 之 4.3 访问Resource ——跟我学spring3

发表时间: 2012-02-22 关键字: spring

4.3.1 ResourceLoader接口

ResourceLoader接口用于返回Resource对象；其实现可以看作是一个生产Resource的工厂类。

java代码：

```
public interface ResourceLoader {
	Resource getResource(String location);
	ClassLoader getClassLoader();
}
```

getResource接口用于根据提供的location参数返回相应的Resource对象；而getClassLoader则返回加载这些Resource的ClassLoader。

Spring提供了一个适用于所有环境的DefaultResourceLoader实现，可以返回ClassPathResource、UrlResource；还提供一个用于web环境的ServletContextResourceLoader，它继承了DefaultResourceLoader的所有功能，又额外提供了获取ServletContextResource的支持。

ResourceLoader在进行加载资源时需要使用前缀来指定需要加载：“classpath:path”表示返回ClasspathResource，“http://path”和“file:path”表示返回UrlResource资源，如果不加前缀则需要根据当前上下文来决定，DefaultResourceLoader默认实现可以加载classpath资源，如代码所示（cn.javass.spring.chapter4.ResourceLoaderTest）：

java代码：

```
@Test
public void testResourceLoad() {
    ResourceLoader loader = new DefaultResourceLoader();
    Resource resource = loader.getResource("classpath:cn/javass/spring/chapter4/test1.txt");
    //验证返回的是ClassPathResource
    Assert.assertEquals(ClassPathResource.class, resource.getClass());
    Resource resource2 = loader.getResource("file:cn/javass/spring/chapter4/test1.txt");
    //验证返回的是ClassPathResource
    Assert.assertEquals(UrlResource.class, resource2.getClass());
    Resource resource3 = loader.getResource("cn/javass/spring/chapter4/test1.txt");
    //验证返默认可以加载ClasspathResource
    Assert.assertTrue(resource3 instanceof ClassPathResource);
}
```

对于目前所有ApplicationContext都实现了ResourceLoader，因此可以使用其来加载资源。

ClassPathXmlApplicationContext：不指定前缀将返回默认的ClassPathResource资源，否则将根据前缀来加载资源；

FileSystemXmlApplicationContext：不指定前缀将返回FileSystemResource，否则将根据前缀来加载资源；

WebApplicationContext：不指定前缀将返回ServletContextResource，否则将根据前缀来加载资源；

其他：不指定前缀根据当前上下文返回Resource实现，否则将根据前缀来加载资源。

4.3.2 ResourceLoaderAware接口

ResourceLoaderAware是一个标记接口，用于通过ApplicationContext上下文注入ResourceLoader。

java代码：

```
public interface ResourceLoaderAware {
   void setResourceLoader(ResourceLoader resourceLoader);
}
```

让我们看下测试代码吧：

1） 首先准备测试Bean，我们的测试Bean还简单只需实现ResourceLoaderAware接口，然后通过回调将ResourceLoader保存下来就可以了：

java代码：

```
package cn.javass.spring.chapter4.bean;
import org.springframework.context.ResourceLoaderAware;
import org.springframework.core.io.ResourceLoader;
public class ResourceBean implements ResourceLoaderAware {
    private ResourceLoader resourceLoader;
    @Override
    public void setResourceLoader(ResourceLoader resourceLoader) {
        this.resourceLoader = resourceLoader;
    }
    public ResourceLoader getResourceLoader() {
        return resourceLoader;
    }
}
```

2） 配置Bean定义（chapter4/resourceLoaderAware.xml）：

java代码：

```
<bean class="cn.javass.spring.chapter4.bean.ResourceBean"/>
```

3）测试(cn.javass.spring.chapter4.ResoureLoaderAwareTest)：

java代码：

```
@Test
public void test() {
    ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter4/resourceLoaderAwar
    ResourceBean resourceBean = ctx.getBean(ResourceBean.class);
    ResourceLoader loader = resourceBean.getResourceLoader();
    Assert.assertTrue(loader instanceof ApplicationContext);
}
```

注意此处“loader instanceof ApplicationContext”，说明了ApplicationContext就是个ResoureLoader。

由于上述实现回调接口注入ResourceLoader的方式属于侵入式，所以不推荐上述方法，可以采用更好的自动注入方式，如“byType”和“constructor”，此处就不演示了。

4.3.3 注入Resource

通过回调或注入方式注入“ResourceLoader”，然后再通过“ResourceLoader”再来加载需要的资源对于只需要加载某个固定的资源是不是很麻烦，有没有更好的方法类似于前边实例中注入“java.io.File”类似方式呢？

Spring提供了一个PropertyEditor “ResourceEditor”用于在注入的字符串和Resource之间进行转换。因此可以使用注入方式注入Resource。

ResourceEditor完全使用ApplicationContext根据注入的路径字符串获取相应的Resource，说白了还是自己做还是容器帮你做的问题。

接下让我们看下示例：

1）准备Bean：

java代码：

```
package cn.javass.spring.chapter4.bean;
import org.springframework.core.io.Resource;
public class ResourceBean3 {
    private Resource resource;
    public Resource getResource() {
        return resource;
    }
    public void setResource(Resource resource) {
        this.resource = resource;
    }
}
```

2）准备配置文件（chapter4/ resourceInject.xml）：

java代码：

```
<bean id="resourceBean1" class="cn.javass.spring.chapter4.bean.ResourceBean3">
   <property name="resource" value="cn/javass/spring/chapter4/test1.properties"/>
</bean>
<bean id="resourceBean2" class="cn.javass.spring.chapter4.bean.ResourceBean3">
<property name="resource"
```

```
value="classpath:cn/javass/spring/chapter4/test1.properties"/>
</bean>
```

注意此处“resourceBean1”注入的路径没有前缀表示根据使用的ApplicationContext实现进行选择Resource实现。

3）让我们来看下测试代码（cn.javass.spring.chapter4.ResourceInjectTest）吧：

java代码：

```
@Test
public void test() {
    ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter4/resourceInject.xml
    ResourceBean3 resourceBean1 = ctx.getBean("resourceBean1", ResourceBean3.class);
    ResourceBean3 resourceBean2 = ctx.getBean("resourceBean2", ResourceBean3.class);
    Assert.assertTrue(resourceBean1.getResource() instanceof ClassPathResource);
    Assert.assertTrue(resourceBean2.getResource() instanceof ClassPathResource);
}
```

接下来一节让我们深入ApplicationContext对各种Resource的支持，及如何使用更便利的资源加载方式。

1.14 【第四章】 资源 之 4.4 Resource通配符路径 ——跟我学spring3

发表时间: 2012-02-22 关键字: spring

4.4.1 使用路径通配符加载Resource

前面介绍的资源路径都是非常简单的一个路径匹配一个资源，Spring还提供了一种更强大的Ant模式通配符匹配，从能一个路径匹配一批资源。

Ant路径通配符支持“？”、“*”、“**”，注意通配符匹配不包括目录分隔符“/”：

“?”：匹配一个字符，如“config?.xml”将匹配“config1.xml”；

“*”：匹配零个或多个字符串，如“cn/*/config.xml”将匹配“cn/javass/config.xml”，但不匹配匹配“cn/config.xml”；而“cn/config-*.xml”将匹配“cn/config-dao.xml”；

“”：匹配路径中的零个或多个目录**，如“cn/**/config.xml”将匹配“cn /config.xml”，也匹配“cn/javass/spring/config.xml”；而“cn/javass/config-**.xml”将匹配“cn/javass/config-dao.xml”，即把“**”当做两个“*”处理。

Spring提供AntPathMatcher来进行Ant风格的路径匹配。具体测试请参考cn.javass.spring.chapter4.AntPathMatcherTest。

Spring在加载类路径资源时除了提供前缀“classpath:”的来支持加载一个Resource，还提供一个前缀“classpath*:”来支持加载所有匹配的类路径Resource。

Spring提供ResourcePatternResolver接口来加载多个Resource，该接口继承了ResourceLoader并添加了“Resource[] getResources(String locationPattern)”用来加载多个Resource：

java代码：

查看 复制到剪贴板 打印

```
1. public interface ResourcePatternResolver extends ResourceLoader {
2.     String CLASSPATH_ALL_URL_PREFIX = "classpath*:";
3.     Resource[] getResources(String locationPattern) throws IOException;
4. }
```

Spring提供了一个ResourcePatternResolver实现PathMatchingResourcePatternResolver，它是基于模式匹配的，默认使用AntPathMatcher进行路径匹配，它除了支持ResourceLoader支持的前缀外，还额外支持“classpath*:”用于加载所有匹配的类路径Resource，ResourceLoader不支持前缀“classpath*:” ：

首先做下准备工作，在项目的“resources”创建“META-INF”目录，然后在其下创建一个“INDEX.LIST”文件。同时在“org.springframework.beans-3.0.5.RELEASE.jar”和“org.springframework.context-3.0.5.RELEASE.jar”两个jar包里也存在相同目录和文件。然后创建一个“LICENSE”文件，该文件存在于“com.springsource.cn.sf.cglib-2.2.0.jar”里。

一、“classpath”： 用于加载类路径（包括jar包）中的一个且仅一个资源；对于多个匹配的也只返回一个，所以如果需要多个匹配的请考虑“classpath*:”前缀；

java代码：

查看 复制到剪贴板 打印

```
@Test
public void testClasspathPrefix() throws IOException {
    ResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();
    //只加载一个绝对匹配Resource，且通过ResourceLoader.getResource进行加载
    Resource[] resources=resolver.getResources("classpath:META-INF/INDEX.LIST");
    Assert.assertEquals(1, resources.length);
    //只加载一个匹配的Resource，且通过ResourceLoader.getResource进行加载
    resources = resolver.getResources("classpath:META-INF/*.LIST");
    Assert.assertTrue(resources.length == 1);
}
```

二、“classpath*”： 用于加载类路径（包括jar包）中的所有匹配的资源。带通配符的classpath使用“ClassLoader”的“Enumeration<URL> getResources(String name)”方法来查找通配符之前的资源，然后通过模式匹配来获取匹配的资源。如“classpath:META-INF/*.LIST”将首先加载通配符之前的目录“META-INF”，然后再遍历路径进行子路径匹配从而获取匹配的资源。

java代码：

查看 复制到剪贴板 打印

```
@Test
public void testClasspathAsteriskPrefix () throws IOException {
    ResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();
    //将加载多个绝对匹配的所有Resource
    //将首先通过ClassLoader.getResources("META-INF")加载非模式路径部分
    //然后进行遍历模式匹配
    Resource[] resources=resolver.getResources("classpath*:META-INF/INDEX.LIST");
    Assert.assertTrue(resources.length > 1);
```

```
    //将加载多个模式匹配的Resource
    resources = resolver.getResources("classpath*:META-INF/*.LIST");
    Assert.assertTrue(resources.length > 1);
}
```

注意“resources.length >1”说明返回多个Resource。不管模式匹配还是非模式匹配只要匹配的都将返回。

在“com.springsource.cn.sf.cglib-2.2.0.jar”里包含“asm-license.txt”文件，对于使用“classpath*: asm-*.txt”进行通配符方式加载资源将什么也加载不了“asm-license.txt”文件，注意一定是模式路径匹配才会遇到这种问题。这是由于“ClassLoader”的“getResources(String name)”方法的限制，对于name为“”的情况将只返回文件系统的类路径，不会包换jar包根路径。

java代码：

查看 复制到剪贴板 打印

```
@Test
public void testClasspathAsteriskPrefixLimit() throws IOException {
    ResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();      //将首先通过 ClassLoader.getResources("")加载目录，
    //将只返回文件系统的类路径不返回jar的跟路径
    //然后进行遍历模式匹配
    Resource[] resources = resolver.getResources("classpath*:asm-*.txt");
    Assert.assertTrue(resources.length == 0);
    //将通过ClassLoader.getResources("asm-license.txt")加载
    //asm-license.txt存在于com.springsource.net.sf.cglib-2.2.0.jar
    resources = resolver.getResources("classpath*:asm-license.txt");
    Assert.assertTrue(resources.length > 0);
    //将只加载文件系统类路径匹配的Resource
    resources = resolver.getResources("classpath*:LICENS*");
    Assert.assertTrue(resources.length == 1);
}
```

对于“resolver.getResources("classpath*:asm-*.txt");” ，由于在项目“resources”目录下没有所以应该返回0个资源； “resolver.getResources("classpath*:asm-license.txt");”将返回jar包里的Resource； “resolver.getResources("classpath*:LICENS*");” ，因为将只返回文件系统类路径资源，所以返回1个资源。

因此在通过前缀“classpath*”加载通配符路径时，必须包含一个根目录才能保证加载的资源是所有的，而不是部分。

三、 “file” ：加载一个或多个文件系统中的Resource。如“file:D:/*.txt”将返回D盘下的所有txt文件；

四、无前缀：通过ResourceLoader实现加载一个资源。

AppliacationContext提供的getResources方法将获取资源委托给ResourcePatternResolver实现，默认使用PathMatchingResourcePatternResolver。所有在此就无需介绍其使用方法了。

4.4.2 注入Resource数组

Spring还支持注入Resource数组，直接看配置如下：

java代码：

查看 复制到剪贴板 打印

```
<bean id="resourceBean1" class="cn.javass.spring.chapter4.bean.ResourceBean4">
<property name="resources">
        <array>
            <value>cn/javass/spring/chapter4/test1.properties</value>
            <value>log4j.xml</value>
        </array>
    </property>
</bean>
<bean id="resourceBean2" class="cn.javass.spring.chapter4.bean.ResourceBean4">
<property name="resources" value="classpath*:META-INF/INDEX.LIST"/>
</bean>
<bean id="resourceBean3" class="cn.javass.spring.chapter4.bean.ResourceBean4">
<property name="resources">
        <array>
            <value>cn/javass/spring/chapter4/test1.properties</value>
            <value>classpath*:META-INF/INDEX.LIST</value>
        </array>
    </property>
</bean>
```

“resourceBean1”就不用多介绍了，传统实现方式；对于“resourceBean2”则使用前缀“classpath*” ，看到这大家应该懂的，加载匹配多个资源；“resourceBean3”是混合使用的；测试代码在“cn.javass.spring.chapter4.ResourceInjectTest.testResourceArrayInject” 。

Spring通过ResourceArrayPropertyEditor来进行类型转换的，而它又默认使用“PathMatchingResourcePatternResolver”来进行把路径解析为Resource对象。所有大家只要会使用“PathMatchingResourcePatternResolver” ，其它一些实现都是委托给它的，比如AppliacationContext的“getResources”方法等。

4.4.3 AppliacationContext实现对各种Resource的支持

一、ClassPathXmlApplicationContext：默认将通过classpath进行加载返回ClassPathResource，提供两类构造器方法：

java代码：

查看 复制到剪贴板 打印

```
public class ClassPathXmlApplicationContext {
   //1）通过ResourcePatternResolver实现根据configLocation获取资源
      public ClassPathXmlApplicationContext(String configLocation);
      public ClassPathXmlApplicationContext(String... configLocations)；
      public ClassPathXmlApplicationContext(String[] configLocations, ……);

   //2）通过直接根据path直接返回ClasspathResource
      public ClassPathXmlApplicationContext(String path, Class clazz);
      public ClassPathXmlApplicationContext(String[] paths, Class clazz);
      public ClassPathXmlApplicationContext(String[] paths, Class clazz, ……);
}
```

第一类构造器是根据提供的配置文件路径使用“ResourcePatternResolver ”的“getResources()”接口通过匹配获取资源；即如“classpath:config.xml”

第二类构造器则是根据提供的路径和clazz来构造ClassResource资源。即采用“public ClassPathResource(String path, Class<?> clazz)”构造器获取资源。

二、FileSystemXmlApplicationContext：将加载相对于当前工作目录的“configLocation”位置的资源，注意在linux系统上不管“configLocation”是否带“/” ，都作为相对路径；而在window系统上如“D:/resourceInject.xml”是绝对路径。因此在除非很必要的情况下，不建议使用该ApplicationContext。

java代码：

查看 复制到剪贴板 打印

```
public class FileSystemXmlApplicationContext{
      public FileSystemXmlApplicationContext(String configLocation);
      public FileSystemXmlApplicationContext(String... configLocations,……);
}
```

java代码：

```
查看 复制到剪贴板 打印
1. //linux系统，以下全是相对于当前vm路径进行加载
2. new FileSystemXmlApplicationContext("chapter4/config.xml");
3. new FileSystemXmlApplicationContext("/chapter4/confg.xml");
```

```
查看 复制到剪贴板 打印
1. //windows系统，第一个将相对于当前vm路径进行加载；
2. //第二个则是绝对路径方式加载
3. new FileSystemXmlApplicationContext("chapter4/config.xml");
4. new FileSystemXmlApplicationContext("d:/chapter4/confg.xml");
```

此处还需要注意：在linux系统上，构造器使用的是相对路径，而ctx.getResource()方法如果以“/”开头则表示获取绝对路径资源，而不带前导“/”将返回相对路径资源。如下：

java代码：

```
查看 复制到剪贴板 打印
1. //linux系统，第一个将相对于当前vm路径进行加载；
2. //第二个则是绝对路径方式加载
3. ctx.getResource ("chapter4/config.xml");
4. ctx.getResource ("/root/confg.xml");
5. //windows系统，第一个将相对于当前vm路径进行加载；
6. //第二个则是绝对路径方式加载
7. ctx.getResource ("chapter4/config.xml");
8. ctx.getResource ("d:/chapter4/confg.xml");
```

因此如果需要加载绝对路径资源最好选择前缀“file”方式，将全部根据绝对路径加载。如在linux系统“ctx.getResource ("file:/root/confg.xml");”

原创内容，转自请注明出处【http://sishuok.com/forum/blogPost/list/0/2458.html】

1.15 【第五章】Spring表达式语言 之 5.1 概述 5.2 SpEL基础 ——跟我学spring3

发表时间: 2012-02-22

5.1 概述

5.1.1 概述

Spring表达式语言全称为“Spring Expression Language”，缩写为“SpEL”，类似于Struts2x中使用的OGNL表达式语言，能在运行时构建复杂表达式、存取对象图属性、对象方法调用等等，并且能与Spring功能完美整合，如能用来配置Bean定义。

表达式语言给静态Java语言增加了动态功能。

SpEL是单独模块，只依赖于core模块，不依赖于其他模块，可以单独使用。

5.1.2 能干什么

表达式语言一般是用最简单的形式完成最主要的工作，减少我们的工作量。

SpEL支持如下表达式：

一、基本表达式：字面量表达式、关系，逻辑与算数运算表达式、字符串连接及截取表达式、三目运算及Elivis表达式、正则表达式、括号优先级表达式；

二、类相关表达式：类类型表达式、类实例化、instanceof表达式、变量定义及引用、赋值表达式、自定义函数、对象属性存取及安全导航表达式、对象方法调用、Bean引用；

三、集合相关表达式：内联List、内联数组、集合，字典访问、列表，字典，数组修改、集合投影、集合选择；不支持多维内联数组初始化；不支持内联字典定义；

四、其他表达式：模板表达式。

注：SpEL表达式中的关键字是不区分大小写的。

5.2 SpEL基础

5.2.1 HelloWorld

首先准备支持SpEL的Jar包：“org.springframework.expression-3.0.5.RELEASE.jar”将其添加到类路径中。

SpEL在求表达式值时一般分为四步，其中第三步可选：首先构造一个解析器，其次解析器解析字符串表达式，在此构造上下文，最后根据上下文得到表达式运算后的值。

让我们看下代码片段吧：

java代码：

```
package cn.javass.spring.chapter5;
import junit.framework.Assert;
import org.junit.Test;
import org.springframework.expression.EvaluationContext;
import org.springframework.expression.Expression;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;
public class SpELTest {
    @Test
    public void helloWorld() {
        ExpressionParser parser = new SpelExpressionParser();
        Expression expression =
parser.parseExpression("('Hello' + ' World').concat(#end)");
        EvaluationContext context = new StandardEvaluationContext();
        context.setVariable("end", "!");
        Assert.assertEquals("Hello World!", expression.getValue(context));
    }
}
```

接下来让我们分析下代码：

1）创建解析器：SpEL使用ExpressionParser接口表示解析器，提供SpelExpressionParser默认实现；

2）解析表达式：使用ExpressionParser的parseExpression来解析相应的表达式为Expression对象。

3）构造上下文：准备比如变量定义等等表达式需要的上下文数据。

4）求值：通过Expression接口的getValue方法根据上下文获得表达式值。

是不是很简单，接下来让我们看下其具体实现及原理吧。

5.2.3 SpEL原理及接口

SpEL提供简单的接口从而简化用户使用，在介绍原理前让我们学习下几个概念：

一、表达式：表达式是表达式语言的核心，所以表达式语言都是围绕表达式进行的，从我们角度来看是“干什么”；

二、解析器：用于将字符串表达式解析为表达式对象，从我们角度来看是“谁来干”；

三、上下文：表达式对象执行的环境，该环境可能定义变量、定义自定义函数、提供类型转换等等，从我们角度看是“在哪干”；

四、根对象及活动上下文对象：根对象是默认的活动上下文对象，活动上下文对象表示了当前表达式操作的对象，从我们角度看是“对谁干”。

理解了这些概念后，让我们看下SpEL如何工作的呢，如图5-1所示：

图5-1 工作原理

1）首先定义表达式：“1+2”；

2）定义解析器ExpressionParser实现，SpEL提供默认实现SpelExpressionParser；

2.1）SpelExpressionParser解析器内部使用Tokenizer类进行词法分析，即把字符串流分析为记号流，记号在SpEL使用Token类来表示；

2.2）有了记号流后，解析器便可根据记号流生成内部抽象语法树；在SpEL中语法树节点由SpelNode接口实现代表：如OpPlus表示加操作节点、IntLiteral表示int型字面量节点；使用SpelNodel实现组成了抽象语法树；

2.3）对外提供Expression接口来简化表示抽象语法树，从而隐藏内部实现细节，并提供getValue简单方法用于获取表达式值；SpEL提供默认实现为SpelExpression；

3）定义表达式上下文对象（可选），SpEL使用EvaluationContext接口表示上下文对象，用于设置根对象、自定义变量、自定义函数、类型转换器等，SpEL提供默认实现StandardEvaluationContext；

4）使用表达式对象根据上下文对象（可选）求值（调用表达式对象的getValue方法）获得结果。

接下来让我们看下SpEL的主要接口吧：

1）ExpressionParser接口：表示解析器，默认实现是org.springframework.expression.spel.standard包中的SpelExpressionParser类，使用parseExpression方法将字符串表达式转换为Expression对象，对于ParserContext接口用于定义字符串表达式是不是模板，及模板开始与结束字符：

java代码：

```
public interface ExpressionParser {
       Expression parseExpression(String expressionString);
       Expression parseExpression(String expressionString, ParserContext context);
}
```

来看下示例：

java代码：

```

@Test
public void testParserContext() {
    ExpressionParser parser = new SpelExpressionParser();
    ParserContext parserContext = new ParserContext() {
        @Override
         public boolean isTemplate() {
            return true;
        }
        @Override
        public String getExpressionPrefix() {
            return "#{";
        }
        @Override
```

```
        public String getExpressionSuffix() {
            return "}";
        }
    };
    String template = "#{'Hello '}#{'World!'}";
    Expression expression = parser.parseExpression(template, parserContext);
    Assert.assertEquals("Hello World!", expression.getValue());
}
```

在此我们演示的是使用ParserContext的情况，此处定义了ParserContext实现：定义表达式是模块，表达式前缀为“#{”，后缀为“}”；使用parseExpression解析时传入的模板必须以“#{”开头，以“}”结尾，如"#{'Hello '}#{'World!'}"。

默认传入的字符串表达式不是模板形式，如之前演示的Hello World。

2）EvaluationContext接口：表示上下文环境，默认实现是org.springframework.expression.spel.support包中的StandardEvaluationContext类，使用setRootObject方法来设置根对象，使用setVariable方法来注册自定义变量，使用registerFunction来注册自定义函数等等。

3）Expression接口：表示表达式对象，默认实现是org.springframework.expression.spel.standard包中的SpelExpression，提供getValue方法用于获取表达式值，提供setValue方法用于设置对象值。

了解了SpEL原理及接口，接下来的事情就是SpEL语法了。

1.16 【第五章】Spring表达式语言 之 5.3 SpEL语法 ——跟我学spring3

发表时间: 2012-02-23 关键字: spring

5.3 SpEL语法

5.3.1 基本表达式

一、字面量表达式： SpEL支持的字面量包括：字符串、数字类型（int、long、float、double）、布尔类型、null类型。

类型	示例
字符串	String str1 = parser.parseExpression("'Hello World!'").getValue(String.class); String str2 = parser.parseExpression("\"Hello World!\"").getValue(String.class);

数字类型	int int1 = parser.parseExpression("1").getValue(Integer.class);
	long long1 = parser.parseExpression("-1L").getValue(long.class);
	float float1 = parser.parseExpression("1.1").getValue(Float.class);
	double double1 = parser.parseExpression("1.1E+2").getValue(double.class);
	int hex1 = parser.parseExpression("0xa").getValue(Integer.class);
	long hex2 = parser.parseExpression("0xaL").getValue(long.class);

布尔类型	boolean true1 = parser.parseExpression("true").getValue(boolean.class); boolean false1 = parser.parseExpression("false").getValue(boolean.class);
null类型	Object null1 = parser.parseExpression("null").getValue(Object.class);

二、算数运算表达式： SpEL支持加(+)、减(-)、乘(*)、除(/)、求余（%）、幂（^）运算。

类型	示例

加减乘除	int result1 = parser.parseExpression("1+2-3*4/2").getValue(Integer.class);//-3
求余	int result2 = parser.parseExpression("4%3").getValue(Integer.class);//1
幂运算	int result3 = parser.parseExpression("2^3").getValue(Integer.class);//8

SpEL还提供求余（MOD）和除（DIV）而外两个运算符，与“%”和“/”等价，不区分大小写。

三、关系表达式：等于（==）、不等于(!=)、大于(>)、大于等于(>=)、小于(<)、小于等于(<=)，区间（between）运算，如“parser.parseExpression("1>2").getValue(boolean.class);”将返回false；而“parser.parseExpression("1 between {1, 2}").getValue(boolean.class);”将返回true。

between运算符右边操作数必须是列表类型，且只能包含2个元素。第一个元素为开始，第二个元素为结束，区间运算是包含边界值的，即 xxx>=list.get(0) && xxx<=list.get(1)。

SpEL同样提供了等价的“EQ”、“NE”、“GT”、“GE”、“LT”、“LE”来表示等于、不等于、大于、大于等于、小于、小于等于，不区分大小写。

四、逻辑表达式：且（and）、或(or)、非(!或NOT)。

java代码：

查看 复制到剪贴板 打印

```
1. String expression1 = "2>1 and (!true or !false)";
2. boolean result1 = parser.parseExpression(expression1).getValue(boolean.class);
3. Assert.assertEquals(true, result1);
4.
5. String expression2 = "2>1 and (NOT true or NOT false)";
6. boolean result2 = parser.parseExpression(expression2).getValue(boolean.class);
7. Assert.assertEquals(true, result2);
```

注：**逻辑运算符不支持 Java中的 && 和 || 。**

五、字符串连接及截取表达式：使用“+”进行字符串连接，使用“'String'[0] [index]”来截取一个字符，目前只支持截取一个，如“'Hello ' + 'World!'”得到“Hello World!”；而“'Hello World!'[0]”将返回“H”。

六、三目运算及Elivis运算表达式：

三目运算符 **“表达式1?表达式2:表达式3”**用于构造三目运算表达式，如“2>1?true:false”将返回true；

Elivis运算符**“表达式1?:表达式2”**从Groovy语言引入用于简化三目运算符的，当表达式1为非null时则返回表达式1，当表达式1为null时则返回表达式2，简化了三目运算符方式“表达式1? 表达式1:表达式2”，如“null?:false”将返回false，而“true?:false”将返回true；

七、正则表达式：使用“str matches regex，如“'123' matches '\\d{3}'”将返回true；

八、括号优先级表达式：使用“(表达式)”构造，括号里的具有高优先级。

5.3.3 类相关表达式

一、类类型表达式：使用“T(Type)”来表示java.lang.Class实例，“Type”必须是类全限定名，“java.lang”包除外，即该包下的类可以不指定包名；使用类类型表达式还可以进行访问类静态方法及类静态字段。

具体使用方法如下：

java代码：

查看 复制到剪贴板 打印

```
@Test
public void testClassTypeExpression() {
    ExpressionParser parser = new SpelExpressionParser();
    //java.lang包类访问
    Class<String> result1 = parser.parseExpression("T(String)").getValue(Class.class);
    Assert.assertEquals(String.class, result1);
    //其他包类访问
    String expression2 = "T(cn.javass.spring.chapter5.SpELTest)";
    Class<String> result2 = parser.parseExpression(expression2).getValue(Class.class);    Assert.assertEquals(SpELTest.class,
    //类静态字段访问
    int result3=parser.parseExpression("T(Integer).MAX_VALUE").getValue(int.class);
    Assert.assertEquals(Integer.MAX_VALUE, result3);
    //类静态方法调用
    int result4 = parser.parseExpression("T(Integer).parseInt('1')").getValue(int.class);
    Assert.assertEquals(1, result4);
}
```

对于java.lang包里的可以直接使用“T(String)”访问；其他包必须是类全限定名；可以进行静态字段访问如“T(Integer).MAX_VALUE”；也可以进行静态方法访问如“T(Integer).parseInt('1')”。

二、类实例化：类实例化同样使用java关键字“new” ，类名必须是全限定名，但java.lang包内的类型除外，如String、Integer。

java代码：

查看 复制到剪贴板 打印

```
@Test
public void testConstructorExpression() {
    ExpressionParser parser = new SpelExpressionParser();
    String result1 = parser.parseExpression("new String('haha')").getValue(String.class);
    Assert.assertEquals("haha", result1);
    Date result2 = parser.parseExpression("new java.util.Date()").getValue(Date.class);
    Assert.assertNotNull(result2);
}
```

实例化完全跟Java内方式一样。

三、instanceof表达式：SpEL支持instanceof运算符，跟Java内使用同义；如“'haha' instanceof T(String)”将返回true。

四、变量定义及引用：变量定义通过EvaluationContext接口的setVariable(variableName, value)方法定义；在表达式中使用“#variableName”引用；除了引用自定义变量，SpE还允许引用根对象及当前上下文对象，使用“#root”引用根对象，使用“#this”引用当前上下文对象；

java代码：

查看 复制到剪贴板 打印

```

@Test
public void testVariableExpression() {
    ExpressionParser parser = new SpelExpressionParser();
    EvaluationContext context = new StandardEvaluationContext();
    context.setVariable("variable", "haha");
    context.setVariable("variable", "haha");
    String result1 = parser.parseExpression("#variable").getValue(context, String.class);
    Assert.assertEquals("haha", result1);

    context = new StandardEvaluationContext("haha");
    String result2 = parser.parseExpression("#root").getValue(context, String.class);
    Assert.assertEquals("haha", result2);
    String result3 = parser.parseExpression("#this").getValue(context, String.class);
    Assert.assertEquals("haha", result3);
}
```

使用“#variable”来引用在EvaluationContext定义的变量；除了可以引用自定义变量，还可以使用“#root”引用根对象，“#this”引用当前上下文对象，此处“#this”即根对象。

五、自定义函数：目前只支持类静态方法注册为自定义函数；SpEL使用StandardEvaluationContext的registerFunction方法进行注册自定义函数，其实完全可以使用setVariable代替，两者其实本质是一样的；

java代码：

查看 复制到剪贴板 打印

```
@Test
public void testFunctionExpression() throws SecurityException, NoSuchMethodException {
    ExpressionParser parser = new SpelExpressionParser();
    StandardEvaluationContext context = new StandardEvaluationContext();
    Method parseInt = Integer.class.getDeclaredMethod("parseInt", String.class);
    context.registerFunction("parseInt", parseInt);
    context.setVariable("parseInt2", parseInt);
    String expression1 = "#parseInt('3') == #parseInt2('3')";
    boolean result1 = parser.parseExpression(expression1).getValue(context, boolean.class);
    Assert.assertEquals(true, result1);
}

```

此处可以看出“registerFunction”和“setVariable”都可以注册自定义函数，但是两个方法的含义不一样，推荐使用“registerFunction”方法注册自定义函数。

六、赋值表达式：SpEL即允许给自定义变量赋值，也允许给跟对象赋值，直接使用“#variableName=value”即可赋值：

java代码：

查看 复制到剪贴板 打印

```
@Test
public void testAssignExpression() {
    ExpressionParser parser = new SpelExpressionParser();
    //1.给root对象赋值
    EvaluationContext context = new StandardEvaluationContext("aaaa");
    String result1 = parser.parseExpression("#root='aaaaa'").getValue(context, String.class);
    Assert.assertEquals("aaaaa", result1);
    String result2 = parser.parseExpression("#this='aaaa'").getValue(context, String.class);
    Assert.assertEquals("aaaa", result2);

    //2.给自定义变量赋值
    context.setVariable("#variable", "variable");
    String result3 = parser.parseExpression("#variable=#root").getValue(context, String.class);
    Assert.assertEquals("aaaa", result3);
}
```

使用“#root='aaaaa'”给根对象赋值，使用“"#this='aaaa'”给当前上下文对象赋值，使用“#variable=#root”给自定义变量赋值，很简单。

七、对象属性存取及安全导航表达式：对象属性获取非常简单，即使用如“a.property.property”这种点缀式获取，SpEL对于属性名首字母是不区分大小写的；SpEL还引入了Groovy语言中的安全导航运算符“(对象|属性)?.属性”，用来避免但“?.”前边的表达式为null时抛出空指针异常，而是返回null；修改对象属性值则可以通过赋值表达式或Expression接口的setValue方法修改。

java代码：

查看 复制到剪贴板 打印

```
ExpressionParser parser = new SpelExpressionParser();
//1.访问root对象属性
Date date = new Date();
StandardEvaluationContext context = new StandardEvaluationContext(date);
int result1 = parser.parseExpression("Year").getValue(context, int.class);
Assert.assertEquals(date.getYear(), result1);
int result2 = parser.parseExpression("year").getValue(context, int.class);
Assert.assertEquals(date.getYear(), result2);
```

对于**当前上下文对象属性及方法访问，可以直接使用属性或方法名访问**，比如此处根对象date属性“year”，注意此处属性名首字母不区分大小写。

java代码：

查看 复制到剪贴板 打印

```

//2.安全访问
context.setRootObject(null);
Object result3 = parser.parseExpression("#root?.year").getValue(context, Object.class);
Assert.assertEquals(null, result3);

```

SpEL引入了Groovy的安全导航运算符，比如此处根对象为null，所以如果访问其属性时肯定抛出空指针异常，而采用“?.”安全访问导航运算符将不抛空指针异常，而是简单的返回null。

java代码：

```
查看 复制到剪贴板 打印
1. //3.给root对象属性赋值
2. context.setRootObject(date);
3. int result4 = parser.parseExpression("Year = 4").getValue(context, int.class);
4. Assert.assertEquals(4, result4);
5. parser.parseExpression("Year").setValue(context, 5);
6. int result5 = parser.parseExpression("Year").getValue(context, int.class);
7. Assert.assertEquals(5, result5);
8.
```

给对象属性赋值可以采用赋值表达式或Expression接口的setValue方法赋值，而且也可以采用点缀方式赋值。

八、对象方法调用：对象方法调用更简单，跟Java语法一样；如“'haha'.substring(2,4)”将返回“ha”；而对于根对象可以直接调用方法；

java代码：

```
查看 复制到剪贴板 打印
1. Date date = new Date();
2. StandardEvaluationContext context = new StandardEvaluationContext(date);
3. int result2 = parser.parseExpression("getYear()").getValue(context, int.class);
4. Assert.assertEquals(date.getYear(), result2);
```

比如根对象date方法“getYear”可以直接调用。

九、Bean引用：SpEL支持使用“@”符号来引用Bean，在引用Bean时需要使用BeanResolver接口实现来查找Bean，Spring提供BeanFactoryResolver实现；

java代码：

```
查看 复制到剪贴板 打印
1. @Test
2. public void testBeanExpression() {
3.     ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext();
4.     ctx.refresh();
5.     ExpressionParser parser = new SpelExpressionParser();
6.     StandardEvaluationContext context = new StandardEvaluationContext();
7.     context.setBeanResolver(new BeanFactoryResolver(ctx));
8.     Properties result1 = parser.parseExpression("@systemProperties").getValue(context, Properties.class);
9.     Assert.assertEquals(System.getProperties(), result1);
10. }
```

在示例中我们首先初始化了一个IoC容器，ClassPathXmlApplicationContext 实现默认会把“System.getProperties()”注册为“systemProperties”Bean，因此我们使用 “@systemProperties”来引用该Bean。

5.3.3 集合相关表达式

一、内联List：从Spring3.0.4开始支持内联List，使用{表达式，……}定义内联List，如“{1,2,3}”将返回一个整型的ArrayList，而“{}”将返回空的List，对于字面量表达式列表，SpEL会使用java.util.Collections.unmodifiableList方法将列表设置为不可修改。

java代码：

查看 复制到剪贴板 打印

```
1. //将返回不可修改的空List
2. List<Integer> result2 = parser.parseExpression("{}").getValue(List.class);
```

java代码：

查看 复制到剪贴板 打印

```
1. //对于字面量列表也将返回不可修改的List
2. List<Integer> result1 = parser.parseExpression("{1,2,3}").getValue(List.class);
3. Assert.assertEquals(new Integer(1), result1.get(0));
4. try {
5.     result1.set(0, 2);
6.     //不可能执行到这，对于字面量列表不可修改
7.     Assert.fail();
8. } catch (Exception e) {
9. }
```

java代码：

查看 复制到剪贴板 打印

```
1. //对于列表中只要有一个不是字面量表达式，将只返回原始List，
2. //不会进行不可修改处理
3. String expression3 = "{{1+2,2+4},{3,4+4}}";
4. List<List<Integer>> result3 = parser.parseExpression(expression3).getValue(List.class);
5. result3.get(0).set(0, 1);
6. Assert.assertEquals(2, result3.size());
```

java代码：

```
查看 复制到剪贴板 打印
1. //声明二维数组并初始化
2. int[] result2 = parser.parseExpression("new int[2]{1,2}").getValue(int[].class);
```

java代码：

```
查看 复制到剪贴板 打印
1. //定义一维数组并初始化
2. int[] result1 = parser.parseExpression("new int[1]").getValue(int[].class);
```

二、内联数组：和Java 数组定义类似，只是在定义时进行多维数组初始化。

java代码：

```
查看 复制到剪贴板 打印
1.
2. //定义多维数组但不初始化
3. int[][][] result3 = parser.parseExpression(expression3).getValue(int[][][].class);
```

java代码：

```
查看 复制到剪贴板 打印
1. //错误的定义多维数组，多维数组不能初始化
2. String expression4 = "new int[1][2][3]{{1}{2}{3}}";
3. try {
4.     int[][][] result4 = parser.parseExpression(expression4).getValue(int[][][].class);
5.     Assert.fail();
6. } catch (Exception e) {
```

```
7. }
```

三、集合，字典元素访问：SpEL目前支持所有集合类型和字典类型的元素访问，使用“集合[索引]”访问集合元素，使用“map[key]”访问字典元素；

java代码：

查看 复制到剪贴板 打印

```
1. //SpEL内联List访问
2. int result1 = parser.parseExpression("{1,2,3}[0]").getValue(int.class);
3. //即list.get(0)
4. Assert.assertEquals(1, result1);
5.
```

java代码：

查看 复制到剪贴板 打印

```
1. //SpEL目前支持所有集合类型的访问
2. Collection<Integer> collection = new HashSet<Integer>();
3. collection.add(1);
4. collection.add(2);
5. EvaluationContext context2 = new StandardEvaluationContext();
6. context2.setVariable("collection", collection);
7. int result2 = parser.parseExpression("#collection[1]").getValue(context2, int.class);
8. //对于任何集合类型通过Iterator来定位元素
9. Assert.assertEquals(2, result2);
```

java代码：

查看 复制到剪贴板 打印

```
1. //SpEL对Map字典元素访问的支持
2. Map<String, Integer> map = new HashMap<String, Integer>();
3. map.put("a", 1);
4. EvaluationContext context3 = new StandardEvaluationContext();
5. context3.setVariable("map", map);
6. int result3 = parser.parseExpression("#map['a']").getValue(context3, int.class);
7. Assert.assertEquals(1, result3);
```

注：**集合元素访问是通过Iterator遍历来定位元素位置的。**

四、列表，字典，数组元素修改：可以使用赋值表达式或Expression接口的setValue方法修改；

java代码：

查看 复制到剪贴板 打印

```
1.  //1.修改数组元素值
2.  int[] array = new int[] {1, 2};
3.  EvaluationContext context1 = new StandardEvaluationContext();
4.  context1.setVariable("array", array);
5.  int result1 = parser.parseExpression("#array[1] = 3").getValue(context1, int.class);
6.  Assert.assertEquals(3, result1);
```

java代码：

查看 复制到剪贴板 打印

```
1.  //2.修改集合值
2.  Collection<Integer> collection = new ArrayList<Integer>();
3.  collection.add(1);
4.  collection.add(2);
5.  EvaluationContext context2 = new StandardEvaluationContext();
6.  context2.setVariable("collection", collection);
7.  int result2 = parser.parseExpression("#collection[1] = 3").getValue(context2, int.class);
8.  Assert.assertEquals(3, result2);
9.  parser.parseExpression("#collection[1]").setValue(context2, 4);
10. result2 = parser.parseExpression("#collection[1]").getValue(context2, int.class);
11. Assert.assertEquals(4, result2);
```

java代码：

查看 复制到剪贴板 打印

```
1.  //3.修改map元素值
2.  Map<String, Integer> map = new HashMap<String, Integer>();
3.  map.put("a", 1);
4.  EvaluationContext context3 = new StandardEvaluationContext();
5.  context3.setVariable("map", map);
6.  int result3 = parser.parseExpression("#map['a'] = 2").getValue(context3, int.class);
```

```
7.  Assert.assertEquals(2, result3);
```

对数组修改直接对“#array[index]”赋值即可修改元素值，同理适用于集合和字典类型。

五、集合投影：在SQL中投影指从表中选择出列，而在SpEL指根据集合中的元素中通过选择来构造另一个集合，该集合和原集合具有相同数量的元素；SpEL使用“（list|map）.![投影表达式]”来进行投影运算：

java代码：

```
查看  复制到剪贴板  打印
1.  //1.首先准备测试数据
2.  Collection<Integer> collection = new ArrayList<Integer>();
3.  collection.add(4);   collection.add(5);
4.  Map<String, Integer> map = new HashMap<String, Integer>();
5.  map.put("a", 1);    map.put("b", 2);
```

java代码：

```
查看  复制到剪贴板  打印
1.  //2.测试集合或数组
2.  EvaluationContext context1 = new StandardEvaluationContext();
3.  context1.setVariable("collection", collection);
4.  Collection<Integer> result1 =
5.  parser.parseExpression("#collection.![#this+1]").getValue(context1, Collection.class);
6.  Assert.assertEquals(2, result1.size());
7.  Assert.assertEquals(new Integer(5), result1.iterator().next());
8.
```

对于集合或数组使用如上表达式进行投影运算，其中投影表达式中“#this”代表每个集合或数组元素，可以使用比如“#this.property”来获取集合元素的属性，其中“#this”可以省略。

java代码：

```
查看  复制到剪贴板  打印
1.  //3.测试字典
```

```
EvaluationContext context2 = new StandardEvaluationContext();
context2.setVariable("map", map);
List<Integer> result2 =
parser.parseExpression("#map.![ value+1]").getValue(context2, List.class);
Assert.assertEquals(2, result2.size());
```

SpEL投影运算还支持Map投影，但Map投影最终只能得到List结果，如上所示，对于投影表达式中的“#this”将是Map.Entry，所以可以使用“value”来获取值，使用“key”来获取键。

六、集合选择：在SQL中指使用select进行选择行数据，而在SpEL指根据原集合通过条件表达式选择出满足条件的元素并构造为新的集合，SpEL使用“(list|map).?[选择表达式]”，其中选择表达式结果必须是boolean类型，如果true则选择的元素将添加到新集合中，false将不添加到新集合中。

java代码：

查看 复制到剪贴板 打印

```
//1.首先准备测试数据
Collection<Integer> collection = new ArrayList<Integer>();
collection.add(4);  collection.add(5);
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("a", 1);   map.put("b", 2);
```

java代码：

查看 复制到剪贴板 打印

```
//2.集合或数组测试
EvaluationContext context1 = new StandardEvaluationContext();
context1.setVariable("collection", collection);
Collection<Integer> result1 =
parser.parseExpression("#collection.?[#this>4]").getValue(context1, Collection.class);
Assert.assertEquals(1, result1.size());
Assert.assertEquals(new Integer(5), result1.iterator().next());
```

对于集合或数组选择，如“#collection.?[#this>4]”将选择出集合元素值大于4的所有元素。选择表达式必须返回布尔类型，使用“#this”表示当前元素。

java代码：

查看 复制到剪贴板 打印

```
//3.字典测试
EvaluationContext context2 = new StandardEvaluationContext();
context2.setVariable("map", map);
Map<String, Integer> result2 =
parser.parseExpression("#map.?[#this.key != 'a']").getValue(context2, Map.class);
Assert.assertEquals(1, result2.size());

List<Integer> result3 =
 parser.parseExpression("#map.?[key != 'a'].![value+1]").getValue(context2, List.class);
Assert.assertEquals(new Integer(3), result3.iterator().next());
```

对于字典选择，如“#map.?[#this.key != 'a']”将选择键值不等于“a”的，其中选择表达式中“#this”是Map.Entry类型，而最终结果还是Map，这点和投影不同；集合选择和投影可以一起使用，如“#map.?[key != 'a'].![value+1]”将首先选择键值不等于“a”的，然后在选出的Map中再进行“value+1”的投影。

5.3.4 表达式模板

模板表达式就是由字面量与一个或多个表达式块组成。每个表达式块由“前缀+表达式+后缀”形式组成，如“${1+2}”即表达式块。在前边我们已经介绍了使用ParserContext接口实现来定义表达式是否是模板及前缀和后缀定义。在此就不多介绍了，如“Error ${#v0} ${#v1}”表达式表示由字面量“Error ”、模板表达式“#v0”、模板表达式“#v1”组成，其中v0和v1表示自定义变量，需要在上下文定义。

1.17 【第五章】Spring表达式语言 之 5.4在Bean定义中使用EL—跟我学spring3

发表时间: 2012-02-23 关键字: spring

5.4.1 xml风格的配置

SpEL支持在Bean定义时注入，默认使用“#{SpEL表达式}”表示，其中“#root”根对象默认可以认为是ApplicationContext，只有ApplicationContext实现默认支持SpEL，获取根对象属性其实是获取容器中的Bean。

首先看下配置方式（chapter5/el1.xml）吧：

java代码：

查看 复制到剪贴板 打印

```
<bean id="world" class="java.lang.String">
    <constructor-arg value="#{' World!'}"/>
</bean>
<bean id="hello1" class="java.lang.String">
    <constructor-arg value="#{'Hello'}#{world}"/>
</bean>
<bean id="hello2" class="java.lang.String">
    <constructor-arg value="#{'Hello' + world}"/>
    <!-- 不支持嵌套的 -->
    <!--<constructor-arg value="#{'Hello'#{world}}"/>-->
</bean>
<bean id="hello3" class="java.lang.String">
    <constructor-arg value="#{'Hello' + @world}"/>
</bean>
```

模板默认以前缀“#{”开头，以后缀“}”结尾，且不允许嵌套，如“#{'Hello'#{world}}”错误，如“#{'Hello' + world}”中“world”默认解析为Bean。当然可以使用“@bean”引用了。

接下来测试一下吧：

java代码：

查看 复制到剪贴板 打印

```
@Test
public void testXmlExpression() {
    ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter5/el1.xml");
    String hello1 = ctx.getBean("hello1", String.class);
    String hello2 = ctx.getBean("hello2", String.class);
    String hello3 = ctx.getBean("hello3", String.class);
    Assert.assertEquals("Hello World!", hello1);
    Assert.assertEquals("Hello World!", hello2);
    Assert.assertEquals("Hello World!", hello3);
}
```

是不是很简单，除了XML配置方式，Spring还提供一种注解方式@Value，接着往下看吧。

5.4.2 注解风格的配置

基于注解风格的SpEL配置也非常简单，使用@Value注解来指定SpEL表达式，该注解可以放到字段、方法及方法参数上。

测试Bean类如下，使用@Value来指定SpEL表达式：

java代码：

查看 复制到剪贴板 打印

```
1.  package cn.javass.spring.chapter5;
2.  import org.springframework.beans.factory.annotation.Value;
3.  public class SpELBean {
4.      @Value("#{'Hello' + world}")
5.      private String value;
6.      //setter和getter由于篇幅省略，自己写上
7.  }
```

首先看下配置文件(chapter5/el2.xml)：

java代码：

查看 复制到剪贴板 打印

```
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.          xmlns:context="http://www.springframework.org/schema/context"
5.          xsi:schemaLocation="
6.            http://www.springframework.org/schema/beans
7.            http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
8.            http://www.springframework.org/schema/context
9.  http://www.springframework.org/schema/context/spring-context-3.0.xsd">
10.     <context:annotation-config/>
11.     <bean id="world" class="java.lang.String">
12.         <constructor-arg value="#{' World!'}"/>
13.     </bean>
14.     <bean id="helloBean1" class="cn.javass.spring.chapter5.SpELBean"/>
15.     <bean id="helloBean2" class="cn.javass.spring.chapter5.SpELBean">
16.         <property name="value" value="haha"/>
17.     </bean>
18. </beans>
```

配置时必须使用“**<context:annotation-config/>**”来开启对注解的支持。

有了配置文件那开始测试吧：

java代码：

```
查看 复制到剪贴板 打印
1.  @Test
2.  public void testAnnotationExpression() {
3.      ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter5/el2.xml");
4.      SpELBean helloBean1 = ctx.getBean("helloBean1", SpELBean.class);
5.      Assert.assertEquals("Hello World!", helloBean1.getValue());
6.      SpELBean helloBean2 = ctx.getBean("helloBean2", SpELBean.class);
7.      Assert.assertEquals("haha", helloBean2.getValue());
8.  }
```

其中“helloBean1”值是SpEL表达式的值，而“helloBean2”是通过setter注入的值，这说明setter注入将覆盖@Value的值。

5.4.3 在Bean定义中SpEL的问题

如果有同学问“#{我不是SpEL表达式}”不是SpEL表达式，而是公司内部的模板，想换个前缀和后缀该如何实现呢？

那我们来看下Spring如何在IoC容器内使用BeanExpressionResolver接口实现来求值SpEL表达式，那如果我们通过某种方式获取该接口实现，然后把前缀后缀修改了不就可以了。

此处我们使用BeanFactoryPostProcessor接口提供postProcessBeanFactory回调方法，它是在IoC容器创建好但还未进行任何Bean初始化时被ApplicationContext实现调用，因此在这个阶段把SpEL前缀及后缀修改掉是安全的，具体代码如下：

java代码：

```
查看 复制到剪贴板 打印
1.  package cn.javass.spring.chapter5;
2.  import org.springframework.beans.BeansException;
3.  import org.springframework.beans.factory.config.BeanFactoryPostProcessor;
4.  import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;
5.  import org.springframework.context.expression.StandardBeanExpressionResolver;
6.  public class SpELBeanFactoryPostProcessor implements BeanFactoryPostProcessor {
7.      @Override
8.      public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)
9.          throws BeansException {
10.         StandardBeanExpressionResolver resolver = (StandardBeanExpressionResolver) beanFactory.getBeanExpressionResolv
11.         resolver.setExpressionPrefix("%{");
12.         resolver.setExpressionSuffix("}");
13.     }
```

```
14.  }
```

首先通过 ConfigurableListableBeanFactory的getBeanExpressionResolver方法获取BeanExpressionResolver实现，其次强制类型转换为StandardBeanExpressionResolver，其为Spring默认实现，然后改掉前缀及后缀。

开始测试吧，首先准备配置文件(chapter5/el3.xml)：

java代码：

查看 复制到剪贴板 打印

```
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.        xmlns:context="http://www.springframework.org/schema/context"
5.        xsi:schemaLocation="
6.          http://www.springframework.org/schema/beans
7.          http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
8.          http://www.springframework.org/schema/context
9.          http://www.springframework.org/schema/context/spring-context-3.0.xsd">
10.    <context:annotation-config/>
11.    <bean class="cn.javass.spring.chapter5.SpELBeanFactoryPostProcessor"/>
12.    <bean id="world" class="java.lang.String">
13.       <constructor-arg value="%{' World!'}"/>
14.    </bean>
15.    <bean id="helloBean1" class="cn.javass.spring.chapter5.SpELBean"/>
16.    <bean id="helloBean2" class="cn.javass.spring.chapter5.SpELBean">
17.       <property name="value" value="%{'Hello' + world}"/>
18.    </bean>
19. </beans>
```

配置文件和注解风格的几乎一样，只有SpEL表达式前缀变为“%{”了，并且注册了“cn.javass.spring.chapter5.SpELBeanFactoryPostProcessor”Bean，用于修改前缀和后缀的。

写测试代码测试一下吧：

java代码：

查看 复制到剪贴板 打印

```
1.  @Test
2.  public void testPrefixExpression() {
3.     ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter5/el3.xml");
4.     SpELBean helloBean1 = ctx.getBean("helloBean1", SpELBean.class);
5.     Assert.assertEquals("#{'Hello' + world}", helloBean1.getValue());
6.     SpELBean helloBean2 = ctx.getBean("helloBean2", SpELBean.class);
7.     Assert.assertEquals("Hello World!", helloBean2.getValue());
8.  }
```

此处helloBean1 中通过@Value注入的“#{'Hello' + world}”结果还是“#{'Hello' + world}”说明不对其进行SpEL表达式求值了，而helloBean2使用“%{'Hello' + world}”注入，得到正确的“"Hello World!”。

1.18 【第六章】 AOP 之 6.1 AOP基础 ——跟我学spring3

发表时间: 2012-02-23 关键字: spring

6.1.1 AOP是什么

考虑这样一个问题：需要对系统中的某些业务做日志记录，比如支付系统中的支付业务需要记录支付相关日志，对于支付系统可能相当复杂，比如可能有自己的支付系统，也可能引入第三方支付平台，面对这样的支付系统该如何解决呢？

- **传统解决方案**：

1）日志部分提前公共类LogUtils，定义“longPayBegin”方法用于记录支付开始日志，“logPayEnd”用于记录支付结果：

2）支付部分，定义IPayService接口并定义支付方法“pay”，并定义了两个实现：“PointPayService”表示积分支付，“RMBPayService”表示人民币支付；并且在每个支付实现中支付逻辑和记录日志：

3）支付实现很明显有重复代码，这个重复很明显可以使用模板设计模式消除重复：

4）到此我们设计了一个可以复用的接口；但大家觉得这样记录日志会很好吗，有没有更好的解决方案？

如果对积分支付方式添加统计功能，比如在支付时记录下用户总积分数、当前消费的积分数，那我们该如何做呢？直接修改源代码添加日志记录，这完全违背了面向对象最重要的原则之一：开闭原则（对扩展开放，对修改关闭）？

- **更好的解决方案**：在我们的支付组件中由于使用了日志组件，即日志模块横切于支付组件，在传统程序设计中很难将日志组件分离出来，即不耦合我们的支付组件；因此面向方面编程AOP就诞生了，它能分离我们的组件，使组件完全不耦合:

1）采用面向方面编程后，我们的支付组件看起来如下所示，代码中不再有日志组件的任何东西；

2）所以日志相关的提取到一个切面中，AOP实现者会在合适的时候将日志功能织入到我们的支付组件中去，从而完全解耦支付组件和日志组件。

看到这大家可能不是很理解，没关系，先往下看。

面向方面编程(AOP)：也可称为面向切面编程，是一种编程范式，提供从另一个角度来考虑程序结构从而完善面向对象编程(OOP)。

在进行OOP开发时，都是基于对组件（比如类）进行开发，然后对组件进行组合，OOP最大问题就是无法解耦组件进行开发，比如我们上边举例，而AOP就是为了克服这个问题而出现的，它来进行这种耦合的分离。

AOP为开发者提供一种进行横切关注点（比如日志关注点横切了支付关注点）分离并织入的机制，把横切关注点分离，然后通过某种技术织入到系统中，从而无耦合的完成了我们的功能。

6.1.2 能干什么

AOP主要用于横切关注点分离和织入，因此需要理解横切关注点和织入：

- **关注点**：可以认为是所关注的任何东西，比如上边的支付组件；
- **关注点分离**：将问题细化从而单独部分，即可以理解为不可再分割的组件，如上边的日志组件和支付组件；
- **横切关注点**：一个组件无法完成需要的功能，需要其他组件协作完成，如日志组件横切于支付组件；
- **织入**：横切关注点分离后，需要通过某种技术将横切关注点融合到系统中从而完成需要的功能，因此需要织入，织入可能在编译期、加载期、运行期等进行。

横切关注点可能包含很多，比如非业务的：日志、事务处理、缓存、性能统计、权限控制等等这些非业务的基础功能；还可能是业务的：如某个业务组件横切于多个模块。如图6-1

图6-1 关注点与横切关注点

传统支付形式，流水方式：

面向切面方式，先将横切关注点分离，再将横切关注点织入到支付系统中：

AOP能干什么：

- 用于横切关注点的分离和织入横切关注点到系统；比如上边提到的日志等等；

- 完善OOP；
- 降低组件和模块之间的耦合性；
- 使系统容易扩展；
- 而且由于关注点分离从而可以获得组件的更好复用。

6.1.3 AOP的基本概念

在进行AOP开发前，先熟悉几个概念：

- **连接点（Jointpoint）**：表示需要在程序中插入横切关注点的扩展点，连接点可能是类初始化、方法执行、方法调用、字段调用或处理异常等等，Spring只支持方法执行连接点，**在AOP中表示为“在哪里干”**；
- **切入点（Pointcut）**：选择一组相关连接点的模式，即可以认为连接点的集合，Spring支持perl5正则表达式和AspectJ切入点模式，Spring默认使用AspectJ语法，**在AOP中表示为“在哪里干的集合”**；
- **通知（Advice）**：在连接点上执行的行为，通知提供了在AOP中需要在切入点所选择的连接点处进行扩展现有行为的手段；包括前置通知（before advice）、后置通知(after advice)、环绕通知（around advice），在Spring中通过代理模式实现AOP，并通过拦截器模式以环绕连接点的拦截器链织入通知；**在AOP中表示为“干什么”**；
- **方面/切面（Aspect）**：横切关注点的模块化，比如上边提到的日志组件。可以认为是通知、引入和切入点的组合；在Spring中可以使用Schema和@AspectJ方式进行组织实现；**在AOP中表示为“在哪干和干什么集合”**；
- **引入（inter-type declaration）**：也称为内部类型声明，为已有的类添加额外新的字段或方法，Spring允许引入新的接口（必须对应一个实现）到所有被代理对象（目标对象），**在AOP中表示为“干什么（引入什么）”**；
- **目标对象（Target Object）**：需要被织入横切关注点的对象，即该对象是切入点选择的对象，需要被通知的对象，从而也可称为“被通知对象”；由于Spring AOP 通过代理模式实现，从而这个对象永远是被代理对象，**在AOP中表示为“对谁干”**；
- **AOP代理（AOP Proxy）**：AOP框架使用代理模式创建的对象，从而实现在连接点处插入通知（即应用切面），就是**通过代理来对目标对象应用切面**。在Spring中，AOP代理可以用JDK动态代理或CGLIB代理实现，而通过拦截器模型应用切面。
- **织入（Weaving）**：织入是一个过程，是将切面应用到目标对象从而创建出AOP代理对象的过程，织入可以在编译期、类装载期、运行期进行。

在AOP中，通过切入点选择目标对象的连接点，然后在目标对象的相应连接点处织入通知，而切入点和通知就是切面（横切关注点），而在目标对象连接点处应用切面的实现方式是通过AOP代理对象，如图6-2所示。

图6-2 概念关系

接下来再让我们具体看看Spring有哪些通知类型：

- **前置通知（Before Advice）:**在切入点选择的连接点处的方法之前执行的通知，该通知不影响正常程序执行流程（除非该通知抛出异常，该异常将中断当前方法链的执行而返回）。
- **后置通知（After Advice）:**在切入点选择的连接点处的方法之后执行的通知，包括如下类型的后置通知：
  - **后置返回通知（After returning Advice）:**在切入点选择的连接点处的方法正常执行完毕时执行的通知，必须是连接点处的方法没抛出任何异常正常返回时才调用后置通知。

- **后置异常通知（After throwing Advice）：**在切入点选择的连接点处的方法抛出异常返回时执行的通知，必须是连接点处的方法抛出任何异常返回时才调用异常通知。
- **后置最终通知（After finally Advice）：**在切入点选择的连接点处的方法返回时执行的通知，不管抛没抛出异常都执行，类似于Java中的finally块。
- **环绕通知（Around Advices）：**环绕着在切入点选择的连接点处的方法所执行的通知，环绕通知可以在方法调用之前和之后自定义任何行为，并且可以决定是否执行连接点处的方法、替换返回值、抛出异常等等。

各种通知类型在UML序列图中的位置如图6-3所示：

图6-3 通知类型

6.1.4 AOP代理

AOP代理就是AOP框架通过代理模式创建的对象，Spring使用JDK动态代理或CGLIB代理来实现，Spring缺省使用JDK动态代理来实现，从而任何接口都可别代理，如果被代理的对象实现不是接口将默认使用CGLIB代理，不过CGLIB代理当然也可应用到接口。

AOP代理的目的就是将切面织入到目标对象。

概念都将完了，接下来让我们看一下AOP的 HelloWorld!吧。

原创内容 转自请注明出处【http://sishuok.com/forum/blogPost/list/2466.html】

1.19 【第六章】 AOP 之 6.2 AOP的HelloWorld ——跟我学spring3

发表时间: 2012-02-23 关键字: spring

6.2.1 准备环境

首先准备开发需要的jar包，请到spring-framework-3.0.5.RELEASE-dependencies.zip和spring-framework-3.0.5.RELEASE-with-docs中查

org.springframework.aop-3.0.5.RELEASE.jar

com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar

com.springsource.org.aopalliance-1.0.0.jar

com.springsource.net.sf.cglib-2.2.0.jar

将这些jar包添加到“Build Path”下。

6.2.2 定义目标类

1）定义目标接口：

java代码：

```
查看 复制到剪贴板 打印
1. package cn.javass.spring.chapter6.service;
2. public interface IHelloWorldService {
3.     public void sayHello();
4. }
```

2）定义目标接口实现：

java代码：

```
查看 复制到剪贴板 打印
1. package cn.javass.spring.chapter6.service.impl;
2. import cn.javass.spring.chapter6.service.IHelloWorldService;
3. public class HelloWorldService implements IHelloWorldService {
4.     @Override
5.     public void sayHello() {
6.         System.out.println("============Hello World!");
7.     }
8. }
9.
```

注：在日常开发中最后将业务逻辑定义在一个专门的service包下，而实现定义在service包下的impl包中，服务接口以IXXXService形式，而服务实现就是XXXService，这就是规约设计，见名知义。当然可以使用公司内部更好的形式，只要大家都好理解就可以了。

6.2.2 定义切面支持类

有了目标类，该定义切面了，切面就是通知和切入点的组合，而切面是通过配置方式定义的，因此这定义切面前，我们需要定义切面支持类，切面支持类提供了通知实现：

java代码：

```
查看 复制到剪贴板 打印
1. package cn.javass.spring.chapter6.aop;
2. public class HelloWorldAspect {
3.       //前置通知
4.     public void beforeAdvice() {
5.         System.out.println("============before advice");
6. }
7. //后置最终通知
8.     public void afterFinallyAdvice() {
9.         System.out.println("============after finally advice");
10.     }
11. }
```

此处HelloWorldAspect类不是真正的切面实现，只是定义了通知实现的类，在此我们可以把它看作就是缺少了切入点的切面。

注：对于AOP相关类最后专门放到一个包下，如“aop”包，因为AOP是动态织入的，所以如果某个目标类被AOP拦截了并应用了通知，可能很难发现这个通知实现在哪个包里，因此推荐使用规约命名，方便以后维护人员查找相应的AOP实现。

6.2.3 在XML中进行配置

有了通知实现，那就让我们来配置切面吧：

1）首先配置AOP需要aop命名空间，配置头如下：

java代码：

```
查看 复制到剪贴板 打印
<?xml version="1.0" encoding="UTF-8"?>
<beans  xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:aop="http://www.springframework.org/schema/aop"
      xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
</beans>
```

2）配置目标类：

java代码：

```
查看 复制到剪贴板 打印
<bean id="helloWorldService"
class="cn.javass.spring.chapter6.service.impl.HelloWorldService"/>
```

3）配置切面：

java代码：

查看 复制到剪贴板 打印

```
<bean id="aspect" class="cn.javass.spring.chapter6.aop.HelloWorldAspect"/>
<aop:config>
<aop:pointcut id="pointcut" expression="execution(* cn.javass..*.*(..))"/>
   <aop:aspect ref="aspect">
      <aop:before pointcut-ref="pointcut" method="beforeAdvice"/>
      <aop:after pointcut="execution(* cn.javass..*.*(..))" method="afterFinallyAdvice"/>
   </aop:aspect>
</aop:config>
```

切入点使用<aop:config>标签下的<aop:pointcut>配置，expression属性用于定义切入点模式，默认是AspectJ语法，“execution(* cn.javass..*.*(..))”表示匹配cn.javass包及子包下的任何方法执行。

切面使用<aop:config>标签下的<aop:aspect>标签配置，其中“ref”用来引用切面支持类的方法。

前置通知使用<aop:aspect>标签下的<aop:before>标签来定义，pointcut-ref属性用于引用切入点Bean，而method用来引用切面通知实现类中的方法，该方法就是通知实现，即在目标类方法执行之前调用的方法。

最终通知使用<aop:aspect>标签下的<aop:after >标签来定义，切入点除了使用pointcut-ref属性来引用已经存在的切入点，也可以使用pointcut属性来定义，如pointcut="execution(* cn.javass..*.*(..))"，method属性同样是指定通知实现，即在目标类方法执行之后调用的方法。

6.2.4 运行测试

测试类非常简单，调用被代理Bean跟调用普通Bean完全一样，Spring AOP将为目标对象创建AOP代理，具体测试代码如下：

java代码：

查看 复制到剪贴板 打印

```
package cn.javass.spring.chapter6;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import cn.javass.spring.chapter6.service.IHelloWorldService;
```

```
6.  import cn.javass.spring.chapter6.service.IPayService;
7.  public class AopTest {
8.      @Test
9.      public void testHelloworld() {
10.         ApplicationContext ctx =  new ClassPathXmlApplicationContext("chapter6/helloworld.xml");
11.         IHelloWorldService helloworldService =
12.         ctx.getBean("helloWorldService", IHelloWorldService.class);
13.         helloworldService.sayHello();
14.     }
15. }
16.
```

该测试将输出如下如下内容：

java代码：

```
查看  复制到剪贴板  打印
1.  ===========before advice
2.  ============Hello World!
3.  ===========after finally advice
```

从输出我们可以看出：前置通知在切入点选择的连接点（方法）之前允许，而后置通知将在连接点（方法）之后执行，具体生成AOP代理及执行过程如图6-4所示。

图6-4 Spring AOP框架生成AOP代理过程

1.20 【第六章】 AOP 之 6.3 基于Schema的AOP ——跟我学spring3

发表时间: 2012-02-23

6.3 基于Schema的AOP

基于Schema的AOP从Spring2.0之后通过“aop”命名空间来定义切面、切入点及声明通知。

在Spring配置文件中，所以AOP相关定义必须放在<aop:config>标签下，该标签下可以有<aop:pointcut>、<aop:advisor>、<aop:aspect>标签，配置顺序不可变。

- <aop:pointcut>：用来定义切入点，该切入点可以重用；
- <aop:advisor>：用来定义只有一个通知和一个切入点的切面；
- <aop:aspect>：用来定义切面，该切面可以包含多个切入点和通知，而且标签内部的通知和切入点定义是无序的；和advisor的区别就在此，advisor只包含一个通知和一个切入点。

6.3.1 声明切面

切面就是包含切入点和通知的对象，在Spring容器中将被定义为一个Bean，Schema方式的切面需要一个切面支持Bean，该支持Bean的字段和方法提供了切面的状态和行为信息，并通过配置方式来指定切入点和通知实现。

切面使用<aop:aspect>标签指定，ref属性用来引用切面支持Bean。

切面支持Bean“aspectSupportBean”跟普通Bean完全一样使用，切面使用“ref”属性引用它。

6.3.2 声明切入点

切入点在Spring中也是一个Bean，Bean定义方式可以有很三种方式：

1）在<aop:config>标签下使用<aop:pointcut>声明一个切入点Bean，该切入点可以被多个切面使用，对于需要共享使用的切入点最好使用该方式，该切入点使用id属性指定Bean名字，在通知定义时使用pointcut-ref属性通过该id引用切入点，expression属性指定切入点表达式：

java代码：

查看 复制到剪贴板 打印

```
1. <aop:config>
2.  <aop:pointcut id="pointcut" expression="execution(* cn.javass..*.*(..))"/>
3.  <aop:aspect ref="aspectSupportBean">
4.    <aop:before pointcut-ref="pointcut" method="before"/>
```

```
5.  </aop:aspect>
6. </aop:config>
```

2）在<aop:aspect>标签下使用<aop:pointcut>声明一个切入点Bean，该切入点可以被多个切面使用，但一般该切入点只被该切面使用，当然也可以被其他切面使用，但最好不要那样使用，该切入点使用id属性指定Bean名字，在通知定义时使用pointcut-ref属性通过该id引用切入点，expression属性指定切入点表达式：

java代码：

```
查看 复制到剪贴板 打印
1. <aop:config>
2.  <aop:aspect ref="aspectSupportBean">
3.    <aop:pointcut id=" pointcut" expression="execution(* cn.javass..*.*(..))"/>
4.    <aop:before pointcut-ref="pointcut" method="before"/>
5.  </aop:aspect>
6. </aop:config>
```

3）匿名切入点Bean，可以在声明通知时通过pointcut属性指定切入点表达式，该切入点是匿名切入点，只被该通知使用：

java代码：

```
查看 复制到剪贴板 打印
1. <aop:config>
2.  <aop:aspect ref="aspectSupportBean">
3.    <aop:after pointcut="execution(* cn.javass..*.*(..))" method="afterFinallyAdvice"/>
4.  </aop:aspect>
5. </aop:config>
```

6.3.3 声明通知

基于Schema方式支持前边介绍的5中通知类型：

一、前置通知：在切入点选择的方法之前执行，通过<aop:aspect>标签下的<aop:before>标签声明：

java代码：

```
查看 复制到剪贴板 打印
1. <aop:before pointcut="切入点表达式" pointcut-ref="切入点Bean引用"
2. method="前置通知实现方法名"
3. arg-names="前置通知实现方法参数列表参数名字"/>
```

pointcut和pointcut-ref：二者选一，指定切入点；

method：指定前置通知实现方法名，如果是多态需要加上参数类型，多个用“，”隔开，如beforeAdvice(java.lang.String)；

arg-names：指定通知实现方法的参数名字，多个用“，”分隔，可选，类似于【3.1.2 构造器注入】中的参数名注入限制：**在class文件中没生成变量调试信息是获取不到方法参数名字的，因此只有在类没生成变量调试信息时才需要使用arg-names属性来指定参数名，如**arg-names="param"表示通知实现方法的参数列表的第一个参数名字为“param”。

首先在cn.javass.spring.chapter6.service.IhelloWorldService定义一个测试方法：

java代码：

```
查看 复制到剪贴板 打印
1. public void sayBefore(String param);
```

其次在cn.javass.spring.chapter6.service.impl. HelloWorldService定义实现

java代码：

```
查看 复制到剪贴板 打印
1. @Override
2. public void sayBefore(String param) {
3.     System.out.println("============say " + param);
4. }
```

第三在cn.javass.spring.chapter6.aop. HelloWorldAspect定义通知实现：

java代码：

```
查看 复制到剪贴板 打印
1. public void beforeAdvice(String param) {
2.     System.out.println("============before advice param:" + param);
3. }
```

最后在chapter6/advice.xml配置文件中进行如下配置：

java代码：

```
查看 复制到剪贴板 打印
1. <bean id="helloWorldService" class="cn.javass.spring.chapter6.service.impl.HelloWorldService"/>
2. <bean id="aspect" class="cn.javass.spring.chapter6.aop.HelloWorldAspect"/>
3. <aop:config>
4.     <aop:aspect ref="aspect">
5.         <aop:before pointcut="execution(* cn.javass..*.sayBefore(..)) and args(param)"
6.                     method="beforeAdvice(java.lang.String)"
7.                     arg-names="param"/>
8.     </aop:aspect>
9. </aop:config>
```

测试代码cn.javass.spring.chapter6.AopTest:

java代码：

```
查看 复制到剪贴板 打印
1. @Test
2. public void testSchemaBeforeAdvice(){
3.     System.out.println("========================================");
4.     ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter6/advice.xml");
5.     IHelloWorldService helloworldService = ctx.getBean("helloWorldService", IHelloWorldService.class);
6.     helloworldService.sayBefore("before");
7.     System.out.println("========================================");
8. }
```

将输入：

```
==========================================

==========before advice param:before

============say before

==========================================
```

分析一下吧：

1）切入点匹配：在配置中使用“execution(* cn.javass..*.sayBefore(..)) ”匹配目标方法sayBefore，且使用“args(param)”匹配目标方法只有一个参数且传入的参数类型为通知实现方法中同名的参数类型；

2）目标方法定义：使用method=" beforeAdvice(java.lang.String) "指定前置通知实现方法，且该通知有一个参数类型为java.lang.String参数；

3）目标方法参数命名：其中使用arg-names=" param "指定通知实现方法参数名为“param” ，切入点中使用“args(param)”匹配的目标方法参数将自动传递给通知实现方法同名参数。

二、后置返回通知：在切入点选择的方法正常返回时执行，通过<aop:aspect>标签下的<aop:after-returning>标签声明：

java代码：

查看 复制到剪贴板 打印

```
<aop:after-returning pointcut="切入点表达式" pointcut-ref="切入点Bean引用"
    method="后置返回通知实现方法名"
    arg-names="后置返回通知实现方法参数列表参数名字"
```

```
    returning="返回值对应的后置返回通知实现方法参数名"
/>
```

pointcut和pointcut-ref：同前置通知同义；

method：同前置通知同义；

arg-names：同前置通知同义；

returning：定义一个名字，该名字用于匹配通知实现方法的一个参数名，当目标方法执行正常返回后，将把目标方法返回值传给通知方法；returning限定了只有目标方法返回值匹配与通知方法相应参数类型时才能执行后置返回通知，否则不执行，对于returning对应的通知方法参数为Object类型将匹配任何目标返回值。

首先在cn.javass.spring.chapter6.service.IhelloWorldService定义一个测试方法：

java代码：

查看 复制到剪贴板 打印

```
public boolean sayAfterReturning();

```

其次在cn.javass.spring.chapter6.service.impl. HelloWorldService定义实现

java代码：

查看 复制到剪贴板 打印

```
@Override
public boolean sayAfterReturning() {
    System.out.println("============after returning");
    return true;
}
```

第三在cn.javass.spring.chapter6.aop. HelloWorldAspect定义通知实现：

java代码：

```
查看 复制到剪贴板 打印
1. public void afterReturningAdvice(Object retVal) {
2.     System.out.println("===========after returning advice retVal:" + retVal);
3. }
```

最后在chapter6/advice.xml配置文件中接着前置通知配置的例子添加如下配置：

java代码：

```
查看 复制到剪贴板 打印
1. <aop:after-returning pointcut="execution(* cn.javass..*.sayAfterReturning(..))"
2.                      method="afterReturningAdvice"
3.                      arg-names="retVal"
4.                      returning="retVal"/>
```

测试代码cn.javass.spring.chapter6.AopTest:

java代码：

```
查看 复制到剪贴板 打印
1. @Test
2. public void testSchemaAfterReturningAdvice() {
3.     System.out.println("========================================");
4.     ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter6/advice.xml");
5.     IHelloWorldService helloworldService = ctx.getBean("helloWorldService", IHelloWorldService.class);
6.     helloworldService.sayAfterReturning();
7.     System.out.println("========================================");
8. }
```

将输入：

```
========================================
===========after returning
===========after returning advice retVal:true
```

==========

分析一下吧：

1）切入点匹配：在配置中使用“execution(* cn.javass..*.sayAfterReturning(..))”匹配目标方法sayAfterReturning，该方法返回true；

2）目标方法定义：使用method="afterReturningAdvice"指定后置返回通知实现方法；

3）目标方法参数命名：其中使用arg-names="retVal"指定通知实现方法参数名为“retVal”；

4）返回值命名：returning="retVal"用于将目标返回值赋值给通知实现方法参数名为“retVal”的参数上。

三、后置异常通知：在切入点选择的方法抛出异常时执行，通过<aop:aspect>标签下的<aop:after-throwing>标签声明：

java代码：

查看 复制到剪贴板 打印

```
<aop:after-throwing pointcut="切入点表达式" pointcut-ref="切入点Bean引用"
                    method="后置异常通知实现方法名"
                    arg-names="后置异常通知实现方法参数列表参数名字"
                    throwing="将抛出的异常赋值给的通知实现方法参数名"/>
```

pointcut和pointcut-ref：同前置通知同义；

method：同前置通知同义；

arg-names：同前置通知同义；

throwing：定义一个名字，该名字用于匹配通知实现方法的一个参数名，当目标方法抛出异常返回后，将把目标方法抛出的异常传给通知方法；throwing限定了只有目标方法抛出的异常匹配与通知方法相应参数异常类型时才能执行后置异常通知，否则不执行，对于throwing对应的通知方法参数为Throwable类型将匹配任何异常。

首先在cn.javass.spring.chapter6.service.IhelloWorldService定义一个测试方法：

java代码：

查看 复制到剪贴板 打印

```
1. public void sayAfterThrowing();
2.
```

其次在cn.javass.spring.chapter6.service.impl. HelloWorldService定义实现

java代码：

查看 复制到剪贴板 打印

```
1. @Override
2. public void sayAfterThrowing() {
3.     System.out.println("============before throwing");
4.     throw new RuntimeException();
5. }
```

第三在cn.javass.spring.chapter6.aop. HelloWorldAspect定义通知实现：

java代码：

查看 复制到剪贴板 打印

```
1. public void afterThrowingAdvice(Exception exception) {
2.   System.out.println("===========after throwing advice exception:" + exception);
3. }
```

最后在chapter6/advice.xml配置文件中接着前置通知配置的例子添加如下配置：

java代码：

```
查看 复制到剪贴板 打印
<aop:after-throwing pointcut="execution(* cn.javass..*.sayAfterThrowing(..))"
                    method="afterThrowingAdvice"
                    arg-names="exception"
                    throwing="exception"/>
```

测试代码cn.javass.spring.chapter6.AopTest:

java代码：

```
查看 复制到剪贴板 打印
@Test(expected = RuntimeException.class)
public void testSchemaAfterThrowingAdvice() {
    System.out.println("========================================");
    ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter6/advice.xml");
    IHelloWorldService helloworldService = ctx.getBean("helloWorldService", IHelloWorldService.class);
    helloworldService.sayAfterThrowing();
    System.out.println("========================================");
}
```

将输入：

```
========================================

============before throwing

===========after throwing advice
exception:java.lang.RuntimeException

========================================
```

分析一下吧：

1）切入点匹配：在配置中使用“execution(* cn.javass..*.sayAfterThrowing(..))”匹配目标方法sayAfterThrowing，该方法将抛出RuntimeException异常；

2）目标方法定义：使用method="afterThrowingAdvice"指定后置异常通知实现方法；

3）目标方法参数命名：其中使用arg-names="exception"指定通知实现方法参数名为“exception”；

4）异常命名：returning="exception"用于将目标方法抛出的异常赋值给通知实现方法参数名为“exception”的参数上。

四、后置最终通知：在切入点选择的方法返回时执行，不管是正常返回还是抛出异常都执行，通过<aop:aspect>标签下的<aop:after >标签声明：

java代码：

查看 复制到剪贴板 打印

```
<aop:after pointcut="切入点表达式" pointcut-ref="切入点Bean引用"
           method="后置最终通知实现方法名"
           arg-names="后置最终通知实现方法参数列表参数名字"/>
```

pointcut和pointcut-ref：同前置通知同义；

method：同前置通知同义；

arg-names：同前置通知同义；

首先在cn.javass.spring.chapter6.service.IhelloWorldService定义一个测试方法：

java代码：

```
查看 复制到剪贴板 打印
1. public boolean sayAfterFinally();
```

其次在cn.javass.spring.chapter6.service.impl. HelloWorldService定义实现

java代码：

```
查看 复制到剪贴板 打印
1. @Override
2. public boolean sayAfterFinally() {
3.     System.out.println("============before finally");
4.     throw new RuntimeException();
5. }
```

第三在cn.javass.spring.chapter6.aop. HelloWorldAspect定义通知实现：

java代码：

```
查看 复制到剪贴板 打印
1. public void afterFinallyAdvice() {
2.     System.out.println("===========after finally advice");
3. }
```

最后在chapter6/advice.xml配置文件中接着前置通知配置的例子添加如下配置：

java代码：

```
查看 复制到剪贴板 打印
1. <aop:after pointcut="execution(* cn.javass..*.sayAfterFinally(..))"
2.     method="afterFinallyAdvice"/>
```

测试代码cn.javass.spring.chapter6.AopTest:

java代码：

查看 复制到剪贴板 打印

```
@Test(expected = RuntimeException.class)
public void testSchemaAfterFinallyAdvice() {
    System.out.println("========================================");
    ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter6/advice.xml");
    IHelloWorldService helloworldService = ctx.getBean("helloWorldService", IHelloWorldService.class);
    helloworldService.sayAfterFinally();
    System.out.println("========================================");
}
```

将输入：

```
========================================
============before finally
===========after finally advice
========================================
```

分析一下吧：

1）切入点匹配：在配置中使用“execution(* cn.javass..*.sayAfterFinally(..))”匹配目标方法sayAfterFinally，该方法将抛出RuntimeException异常；

2）目标方法定义：使用method=" afterFinallyAdvice "指定后置最终通知实现方法。

五、环绕通知：环绕着在切入点选择的连接点处的方法所执行的通知，环绕通知非常强大，可以决定目标方法是否执行，什么时候执行，执行时是否需要替换方法参数，执行完毕是否需要替换返回值，可通过<aop:aspect>标签下的<aop:around >标签声明：

java代码：

```
查看 复制到剪贴板 打印
<aop:around pointcut="切入点表达式" pointcut-ref="切入点Bean引用"
            method="后置最终通知实现方法名"
            arg-names="后置最终通知实现方法参数列表参数名字"/>
```

pointcut和pointcut-ref：同前置通知同义；

method：同前置通知同义；

arg-names：同前置通知同义；

环绕通知第一个参数必须是org.aspectj.lang.ProceedingJoinPoint类型，在通知实现方法内部使用ProceedingJoinPoint的proceed()方法使目标方法执行，proceed 方法可以传入可选的Object[]数组，该数组的值将被作为目标方法执行时的参数。

首先在cn.javass.spring.chapter6.service.IhelloWorldService定义一个测试方法：

java代码：

```
查看 复制到剪贴板 打印
public void sayAround(String param);
```

其次在cn.javass.spring.chapter6.service.impl. HelloWorldService定义实现

java代码：

```
查看 复制到剪贴板 打印
1. @Override
2. public void sayAround(String param) {
3.     System.out.println("============around param:" + param);
4. }
```

第三在cn.javass.spring.chapter6.aop. HelloWorldAspect定义通知实现：

java代码：

```
查看 复制到剪贴板 打印
1. public Object aroundAdvice(ProceedingJoinPoint pjp) throws Throwable {
2.     System.out.println("===========around before advice");
3.     Object retVal = pjp.proceed(new Object[] {"replace"});
4.     System.out.println("===========around after advice");
5.     return retVal;
6. }
```

最后在chapter6/advice.xml配置文件中接着前置通知配置的例子添加如下配置：

java代码：

```
查看 复制到剪贴板 打印
1. <aop:around pointcut="execution(* cn.javass..*.sayAround(..))"
2.             method="aroundAdvice"/>
```

测试代码cn.javass.spring.chapter6.AopTest:

java代码：

```
查看 复制到剪贴板 打印
1. @Test
2. public void testSchemaAroundAdvice() {
3.     System.out.println("=====================================");
4.     ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter6/advice.xml");
```

```
5.      IHelloWorldService helloworldService =
6.      ctx.getBean("helloWorldService", IHelloWorldService.class);
7.      helloworldService.sayAround("haha");
8.      System.out.println("=====================================");
9.  }
```

将输入：

==========

==========around before advice

==========around param:replace

==========around after advice

==========

分析一下吧：

1）切入点匹配：在配置中使用“execution(* cn.javass..*.sayAround(..))”匹配目标方法sayAround；

2）目标方法定义：使用method="aroundAdvice"指定环绕通知实现方法，在该实现中，第一个方法参数为pjp，类型为ProceedingJoinPoint，其中“Object retVal = pjp.proceed(new Object[] {"replace"});”，用于执行目标方法，且目标方法参数被“new Object[] {"replace"}”替换，最后返回“retVal ”返回值。

3）测试：我们使用“helloworldService.sayAround("haha");”传入参数为“haha”，但最终输出为“replace”，说明参数被替换了。

6.3.4 引入

Spring引入允许为目标对象引入新的接口，通过在< aop:aspect>标签内使用< aop:declare-parents>标签进行引入，定义方式如下：

java代码：

查看 复制到剪贴板 打印

```
<aop:declare-parents
        types-matching="AspectJ语法类型表达式"
        implement-interface=引入的接口"
        default-impl="引入接口的默认实现"
        delegate-ref="引入接口的默认实现Bean引用"/>
```

types-matching：匹配需要引入接口的目标对象的AspectJ语法类型表达式；

implement-interface：定义需要引入的接口；

default-impl和delegate-ref：定义引入接口的默认实现，二者选一，default-impl是接口的默认实现类全限定名，而delegate-ref是默认的实现的委托Bean名；

接下来让我们练习一下吧：

首先定义引入的接口及默认实现：

java代码：

查看 复制到剪贴板 打印

```
package cn.javass.spring.chapter6.service;
public interface IIntroductionService {
    public void induct();
}
```

java代码：

```
查看 复制到剪贴板 打印
1. package cn.javass.spring.chapter6.service.impl;
2. import cn.javass.spring.chapter6.service.IIntroductionService;
3. public class IntroductiondService implements IIntroductionService {
4.     @Override
5.     public void induct() {
6.         System.out.println("=========introduction");
7.     }
8. }
```

其次在chapter6/advice.xml配置文件中接着前置通知配置的例子添加如下配置：

java代码：

```
查看 复制到剪贴板 打印
1. <aop:declare-parents
2.     types-matching="cn.javass..*IHelloWorldService+"
3.     implement-interface="cn.javass.spring.chapter6.service.IIntroductionService"
4.     default-impl="cn.javass.spring.chapter6.service.impl.IntroductiondService"/>
5.
```

最后测试一下吧，测试代码cn.javass.spring.chapter6.AopTest：

java代码：

```
查看 复制到剪贴板 打印
1. @Test
2. public void testSchemaIntroduction() {
3.     System.out.println("========================================");
4.     ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter6/advice.xml");
5.     IIntroductionService introductionService =
6.     ctx.getBean("helloWorldService", IIntroductionService.class);
7.     introductionService.induct();
8.     System.out.println("========================================");
9. }
10.
```

将输入：

```
=======================================

=========introduction

=======================================
```

分析一下吧：

1）目标对象类型匹配：使用types-matching="cn.javass..*.IHelloWorldService+"匹配IHelloWorldService接口的子类型，如HelloWorldService实现；

2）引入接口定义：通过implement-interface属性表示引入的接口，如“cn.javass.spring.chapter6.service.IIntroductionService”。

3）引入接口的实现：通过default-impl属性指定，如“cn.javass.spring.chapter6.service.impl.IntroductiondService”，也可以使用“delegate-ref”来指定实现的Bean。

4）获取引入接口：如使用“ctx.getBean("helloWorldService", IIntroductionService.class);”可直接获取到引入的接口。

6.3.5 Advisor

Advisor表示只有一个通知和一个切入点的切面，由于Spring AOP都是基于AOP联盟的拦截器模型的环绕通知的，所以引入Advisor来支持各种通知类型（如前置通知等5种），Advisor概念来自于Spring1.2对AOP的支持，在AspectJ中没有相应的概念对应。

Advisor可以使用<aop:config>标签下的<aop:advisor>标签定义：

java代码：

```
查看 复制到剪贴板 打印
1. <aop:advisor pointcut="切入点表达式" pointcut-ref="切入点Bean引用"
2.              advice-ref="通知API实现引用"/>
```

pointcut和pointcut-ref：二者选一，指定切入点表达式；

advice-ref：引用通知API实现Bean，如前置通知接口为MethodBeforeAdvice；

接下来让我们看一下示例吧：

首先在cn.javass.spring.chapter6.service.IhelloWorldService定义一个测试方法：

java代码：

```
查看 复制到剪贴板 打印
1. public void sayAdvisorBefore(String param);
```

其次在cn.javass.spring.chapter6.service.impl. HelloWorldService定义实现

java代码：

```
查看 复制到剪贴板 打印
1. @Override
2. public void sayAdvisorBefore(String param) {
3.     System.out.println("============say " + param);
4. }
```

第三定义前置通知API实现：

java代码：

```
查看 复制到剪贴板 打印
1. package cn.javass.spring.chapter6.aop;
2. import java.lang.reflect.Method;
3. import org.springframework.aop.MethodBeforeAdvice;
```

```
4. public class BeforeAdviceImpl implements MethodBeforeAdvice {
5.     @Override
6.     public void before(Method method, Object[] args, Object target) throws Throwable {
7.         System.out.println("===========before advice");
8.     }
9. }
```

在chapter6/advice.xml配置文件中先添加通知实现Bean定义：

java代码：

```
查看 复制到剪贴板 打印
1.
2. <bean id="beforeAdvice" class="cn.javass.spring.chapter6.aop.BeforeAdviceImpl"/>
3.
```

然后在<aop:config>标签下，添加Advisor定义，添加时注意顺序：

java代码：

```
查看 复制到剪贴板 打印
1. <aop:advisor pointcut="execution(* cn.javass..*.sayAdvisorBefore(..))"
2.                 advice-ref="beforeAdvice"/>
3.
```

测试代码cn.javass.spring.chapter6.AopTest:

java代码：

```
查看 复制到剪贴板 打印
1. @Test
2. public void testSchemaAdvisor() {
3.     System.out.println("=======================================");
4.     ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter6/advice.xml");
5.     IHelloWorldService helloworldService =
6.     ctx.getBean("helloWorldService", IHelloWorldService.class);
7.     helloworldService.sayAdvisorBefore("haha");
8.     System.out.println("=======================================");
```

```
9. }
```

将输入：

```
==========================================
===========before advice
============say haha
==========================================
```

在此我们只介绍了前置通知API，其他类型的在后边章节介绍。

不推荐使用Advisor，除了在进行事务控制的情况下，其他情况一般不推荐使用该方式，该方式属于侵入式设计，必须实现通知API。

1.21 源代码下载 ——跟我学spring3

发表时间: 2012-02-23 关键字: spring

附件太大，，请到http://sishuok.com/forum/blogPost/list/2561.html下载

1.22 【第六章】 AOP 之 6.4 基于@AspectJ的AOP ——跟我学spring3

发表时间: 2012-02-24 关键字: spring

Spring除了支持Schema方式配置AOP，还支持注解方式：使用@AspectJ风格的切面声明。

6.4.1 启用对@AspectJ的支持

Spring默认不支持@AspectJ风格的切面声明，为了支持需要使用如下配置：

java代码：

```
<aop:aspectj-autoproxy/>
```

这样Spring就能发现@AspectJ风格的切面并且将切面应用到目标对象。

6.4.2 声明切面

@AspectJ风格的声明切面非常简单，使用@Aspect注解进行声明：

java代码：

```
@Aspect()
Public class Aspect{
……
}
```

然后将该切面在配置文件中声明为Bean后，Spring就能自动识别并进行AOP方面的配置：

java代码：

```
<bean id="aspect" class="……Aspect"/>
```

该切面就是一个POJO，可以在该切面中进行切入点及通知定义，接着往下看吧。

6.4.3 声明切入点

@AspectJ风格的命名切入点使用org.aspectj.lang.annotation包下的@Pointcut+方法（方法必须是返回void类型）实现。

java代码：

```
@Pointcut(value="切入点表达式", argNames = "参数名列表")
public void pointcutName(……) {}
```

value：指定切入点表达式；

argNames：指定命名切入点方法参数列表参数名字，可以有多个用“，”分隔，这些参数将传递给通知方法同名的参数，同时比如切入点表达式“args(param)”将匹配参数类型为命名切入点方法同名参数指定的参数类型。

pointcutName：切入点名字，可以使用该名字进行引用该切入点表达式。

java代码：

```
@Pointcut(value="execution(* cn.javass..*.sayAdvisorBefore(..)) && args(param)", argNames =
public void beforePointcut(String param) {}
```

定义了一个切入点，名字为“beforePointcut”，该切入点将匹配目标方法的第一个参数类型为通知方法实现中参数名为“param”的参数类型。

6.4.4 声明通知

@AspectJ风格的声明通知也支持5种通知类型：

一、前置通知：使用org.aspectj.lang.annotation 包下的@Before注解声明；

java代码：

```
@Before(value = "切入点表达式或命名切入点", argNames = "参数列表参数名")
```

value：指定切入点表达式或命名切入点；

argNames：与Schema方式配置中的同义。

接下来示例一下吧：

1、定义接口和实现，在此我们就使用Schema风格时的定义；

2、定义切面：

java代码：

```
package cn.javass.spring.chapter6.aop;
import org.aspectj.lang.annotation.Aspect;
@Aspect
public class HelloWorldAspect2 {

}
```

3、定义切入点：

java代码：

```
@Pointcut(value="execution(* cn.javass..*.sayAdvisorBefore(..)) && args(param)", argNames =
public void beforePointcut(String param) {}
```

4、定义通知：

java代码：

```
@Before(value = "beforePointcut(param)", argNames = "param")
public void beforeAdvice(String param) {
    System.out.println("===========before advice param:" + param);
}
```

5、在chapter6/advice2.xml配置文件中进行如下配置：

java代码：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans  xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:aop="http://www.springframework.org/schema/aop"
        xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
```

```
  <aop:aspectj-autoproxy/>
  <bean id="helloWorldService"
            class="cn.javass.spring.chapter6.service.impl.HelloWorldService"/>

  <bean id="aspect"
             class="cn.javass.spring.chapter6.aop.HelloWorldAspect2"/>

</beans>
```

6、测试代码cn.javass.spring.chapter6.AopTest:

java代码：

```
@Test
public void testAnnotationBeforeAdvice() {
    System.out.println("=====================================");
    ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter6/advice2.xml");
    IHelloWorldService helloworldService = ctx.getBean("helloWorldService", IHelloWorldServi
    helloworldService.sayBefore("before");
    System.out.println("=====================================");
}
```

将输出：

==========

==========before advice param:before

==========say before

==========

切面、切入点、通知全部使用注解完成：

1）使用@Aspect将POJO声明为切面；

2）使用@Pointcut进行命名切入点声明，同时指定目标方法第一个参数类型必须是java.lang.String，对于其他匹配的方法但参数类型不一致的将也是不匹配的，通过argNames = "param"指定了将把该匹配的目标方法参数传递给通知同名的参数上；

3）使用@Before进行前置通知声明，其中value用于定义切入点表达式或引用命名切入点；

4）配置文件需要使用<aop:aspectj-autoproxy/>来开启注解风格的@AspectJ支持；

5）需要将切面注册为Bean，如“aspect”Bean；

6）测试代码完全一样。

二、后置返回通知：使用org.aspectj.lang.annotation 包下的@AfterReturning注解声明；

java代码：

```
@AfterReturning(
value="切入点表达式或命名切入点",
pointcut="切入点表达式或命名切入点",
argNames="参数列表参数名",
returning="返回值对应参数名")
```

value：指定切入点表达式或命名切入点；

pointcut：同样是指定切入点表达式或命名切入点，如果指定了将覆盖value属性指定的，pointcut具有高优先级；

argNames：与Schema方式配置中的同义；

returning：与Schema方式配置中的同义。

java代码：

```
@AfterReturning(
    value="execution(* cn.javass..*.sayBefore(..))",
    pointcut="execution(* cn.javass..*.sayAfterReturning(..))",
    argNames="retVal", returning="retVal")
public void afterReturningAdvice(Object retVal) {
    System.out.println("===========after returning advice retVal:" + retVal);
}
```

其中测试代码与Schema方式几乎一样，在此就不演示了，如果需要请参考AopTest.java中的testAnnotationAfterReturningAdvice测试方法。

三、后置异常通知：使用org.aspectj.lang.annotation 包下的@AfterThrowing注解声明；

java代码：

```
@AfterThrowing (
value="切入点表达式或命名切入点",
pointcut="切入点表达式或命名切入点",
argNames="参数列表参数名",
throwing="异常对应参数名")
```

value：指定切入点表达式或命名切入点；

pointcut：同样是指定切入点表达式或命名切入点，如果指定了将覆盖value属性指定的，pointcut具有高优先级；

argNames：与Schema方式配置中的同义；

throwing：与Schema方式配置中的同义。

java代码：

```
@AfterThrowing(
    value="execution(* cn.javass..*.sayAfterThrowing(..))",
    argNames="exception", throwing="exception")
public void afterThrowingAdvice(Exception exception) {
    System.out.println("===========after throwing advice exception:" + exception);
}
```

其中测试代码与Schema方式几乎一样，在此就不演示了，如果需要请参考AopTest.java中的testAnnotationAfterThrowingAdvice测试方法。

四、后置最终通知：使用org.aspectj.lang.annotation 包下的@After注解声明；

java代码：

```
@After (
value="切入点表达式或命名切入点",
argNames="参数列表参数名")
```

value：指定切入点表达式或命名切入点；

argNames：与Schema方式配置中的同义；

java代码：

```
@After(value="execution(* cn.javass..*.sayAfterFinally(..))")
public void afterFinallyAdvice() {
    System.out.println("===========after finally advice");
}
```

其中测试代码与Schema方式几乎一样，在此就不演示了，如果需要请参考AopTest.java中的testAnnotationAfterFinallyAdvice测试方法。

五、环绕通知：使用org.aspectj.lang.annotation 包下的@Around注解声明；

java代码：

```
@Around (
value="切入点表达式或命名切入点",
argNames="参数列表参数名")
```

value：指定切入点表达式或命名切入点；

argNames：与Schema方式配置中的同义；

java代码：

```
@Around(value="execution(* cn.javass..*.sayAround(..))")
public Object aroundAdvice(ProceedingJoinPoint pjp) throws Throwable {
    System.out.println("===========around before advice");
    Object retVal = pjp.proceed(new Object[] {"replace"});
    System.out.println("===========around after advice");
    return retVal;
}
```

其中测试代码与Schema方式几乎一样，在此就不演示了，如果需要请参考AopTest.java中的annotationAroundAdviceTest测试方法。

6.4.5 引入

@AspectJ风格的引入声明在切面中使用org.aspectj.lang.annotation包下的@DeclareParents声明：

java代码：

```
@DeclareParents(
value=" AspectJ语法类型表达式",
defaultImpl=引入接口的默认实现类)
private Interface interface;
```

value：匹配需要引入接口的目标对象的AspectJ语法类型表达式；与Schema方式中的types-matching属性同义；

private Interface interface：指定需要引入的接口；

defaultImpl：指定引入接口的默认实现类，没有与Schema方式中的delegate-ref属性同义的定义方式；

java代码：

```
@DeclareParents(
    value="cn.javass..*.IHelloWorldService+", defaultImpl=cn.javass.spring.chapter6.service.
private IIntroductionService introductionService;
```

其中测试代码与Schema方式几乎一样，在此就不演示了，如果需要请参考AopTest.java中的testAnnotationIntroduction测试方法。

1.23 【第六章】 AOP 之 6.6 通知参数 ——跟我学spring3

发表时间: 2012-02-24 关键字: spring

Spring除了支持Schema方式配置AOP，还支持注解方式：使用@AspectJ风格的切面声明。

6.4.1 启用对@AspectJ的支持

Spring默认不支持@AspectJ风格的切面声明，为了支持需要使用如下配置：

java代码：

```
<aop:aspectj-autoproxy/>
```

这样Spring就能发现@AspectJ风格的切面并且将切面应用到目标对象。

6.4.2 声明切面

@AspectJ风格的声明切面非常简单，使用@Aspect注解进行声明：

java代码：

```
@Aspect()
Public class Aspect{
……
}
```

然后将该切面在配置文件中声明为Bean后，Spring就能自动识别并进行AOP方面的配置：

java代码：

```
<bean id="aspect" class="……Aspect"/>
```

该切面就是一个POJO，可以在该切面中进行切入点及通知定义，接着往下看吧。

6.4.3 声明切入点

@AspectJ风格的命名切入点使用org.aspectj.lang.annotation包下的@Pointcut+方法（方法必须是返回void类型）实现。

java代码：

```
@Pointcut(value="切入点表达式", argNames = "参数名列表")
public void pointcutName(……) {}
```

value：指定切入点表达式；

argNames：指定命名切入点方法参数列表参数名字，可以有多个用“，”分隔，这些参数将传递给通知方法同名的参数，同时比如切入点表达式“args(param)”将匹配参数类型为命名切入点方法同名参数指定的参数类型。

pointcutName：切入点名字，可以使用该名字进行引用该切入点表达式。

java代码：

```
@Pointcut(value="execution(* cn.javass..*.sayAdvisorBefore(..)) && args(param)", argNames =
public void beforePointcut(String param) {}
```

定义了一个切入点，名字为“beforePointcut”，该切入点将匹配目标方法的第一个参数类型为通知方法实现中参数名为“param”的参数类型。

6.4.4 声明通知

@AspectJ风格的声明通知也支持5种通知类型：

一、前置通知：使用org.aspectj.lang.annotation 包下的@Before注解声明；

java代码：

```
@Before(value = "切入点表达式或命名切入点", argNames = "参数列表参数名")
```

value：指定切入点表达式或命名切入点；

argNames：与Schema方式配置中的同义。

接下来示例一下吧：

1、定义接口和实现，在此我们就使用Schema风格时的定义；

2、定义切面：

java代码：

```
package cn.javass.spring.chapter6.aop;
import org.aspectj.lang.annotation.Aspect;
@Aspect
public class HelloWorldAspect2 {

}
```

3、定义切入点：

java代码：

```
@Pointcut(value="execution(* cn.javass..*.sayAdvisorBefore(..)) && args(param)", argNames =
public void beforePointcut(String param) {}
```

4、定义通知：

java代码：

```
@Before(value = "beforePointcut(param)", argNames = "param")
public void beforeAdvice(String param) {
    System.out.println("===========before advice param:" + param);
}
```

5、在chapter6/advice2.xml配置文件中进行如下配置：

java代码：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans  xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:aop="http://www.springframework.org/schema/aop"
        xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
```

```
  <aop:aspectj-autoproxy/>
  <bean id="helloWorldService"
            class="cn.javass.spring.chapter6.service.impl.HelloWorldService"/>

  <bean id="aspect"
             class="cn.javass.spring.chapter6.aop.HelloWorldAspect2"/>

</beans>
```

6、测试代码cn.javass.spring.chapter6.AopTest:

java代码：

```
@Test
public void testAnnotationBeforeAdvice() {
    System.out.println("======================================");
    ApplicationContext ctx = new ClassPathXmlApplicationContext("chapter6/advice2.xml");
    IHelloWorldService helloworldService = ctx.getBean("helloWorldService", IHelloWorldServi
    helloworldService.sayBefore("before");
    System.out.println("======================================");
}
```

将输出：

==========

==========before advice param:before

==========say before

==========

切面、切入点、通知全部使用注解完成：

1）使用@Aspect将POJO声明为切面；

2）使用@Pointcut进行命名切入点声明，同时指定目标方法第一个参数类型必须是java.lang.String，对于其他匹配的方法但参数类型不一致的将也是不匹配的，通过argNames = "param"指定了将把该匹配的目标方法参数传递给通知同名的参数上；

3）使用@Before进行前置通知声明，其中value用于定义切入点表达式或引用命名切入点；

4）配置文件需要使用<aop:aspectj-autoproxy/>来开启注解风格的@AspectJ支持；

5）需要将切面注册为Bean，如“aspect”Bean；

6）测试代码完全一样。

二、后置返回通知：使用org.aspectj.lang.annotation 包下的@AfterReturning注解声明；

java代码：

```
@AfterReturning(
value="切入点表达式或命名切入点",
pointcut="切入点表达式或命名切入点",
argNames="参数列表参数名",
returning="返回值对应参数名")
```

value：指定切入点表达式或命名切入点；

pointcut：同样是指定切入点表达式或命名切入点，如果指定了将覆盖value属性指定的，pointcut具有高优先级；

argNames：与Schema方式配置中的同义；

returning：与Schema方式配置中的同义。

java代码：

```
@AfterReturning(
    value="execution(* cn.javass..*.sayBefore(..))",
    pointcut="execution(* cn.javass..*.sayAfterReturning(..))",
    argNames="retVal", returning="retVal")
public void afterReturningAdvice(Object retVal) {
    System.out.println("===========after returning advice retVal:" + retVal);
}
```

其中测试代码与Schema方式几乎一样，在此就不演示了，如果需要请参考AopTest.java中的testAnnotationAfterReturningAdvice测试方法。

三、后置异常通知：使用org.aspectj.lang.annotation 包下的@AfterThrowing注解声明；

java代码：

```
@AfterThrowing (
value="切入点表达式或命名切入点",
pointcut="切入点表达式或命名切入点",
argNames="参数列表参数名",
throwing="异常对应参数名")
```

value：指定切入点表达式或命名切入点；

pointcut：同样是指定切入点表达式或命名切入点，如果指定了将覆盖value属性指定的，pointcut具有高优先级；

argNames：与Schema方式配置中的同义；

throwing：与Schema方式配置中的同义。

java代码：

```
@AfterThrowing(
    value="execution(* cn.javass..*.sayAfterThrowing(..))",
    argNames="exception", throwing="exception")
public void afterThrowingAdvice(Exception exception) {
    System.out.println("===========after throwing advice exception:" + exception);
}
```

其中测试代码与Schema方式几乎一样，在此就不演示了，如果需要请参考AopTest.java中的testAnnotationAfterThrowingAdvice测试方法。

四、后置最终通知：使用org.aspectj.lang.annotation 包下的@After注解声明；

java代码：

```
@After (
value="切入点表达式或命名切入点",
argNames="参数列表参数名")
```

value：指定切入点表达式或命名切入点；

argNames：与Schema方式配置中的同义；

java代码：

```
@After(value="execution(* cn.javass..*.sayAfterFinally(..))")
public void afterFinallyAdvice() {
    System.out.println("===========after finally advice");
}
```

其中测试代码与Schema方式几乎一样，在此就不演示了，如果需要请参考AopTest.java中的testAnnotationAfterFinallyAdvice测试方法。

五、环绕通知：使用org.aspectj.lang.annotation 包下的@Around注解声明；

java代码：

```
@Around (
value="切入点表达式或命名切入点",
argNames="参数列表参数名")
```

value：指定切入点表达式或命名切入点；

argNames：与Schema方式配置中的同义；

java代码：

```
@Around(value="execution(* cn.javass..*.sayAround(..))")
public Object aroundAdvice(ProceedingJoinPoint pjp) throws Throwable {
    System.out.println("===========around before advice");
    Object retVal = pjp.proceed(new Object[] {"replace"});
    System.out.println("===========around after advice");
    return retVal;
}
```

其中测试代码与Schema方式几乎一样，在此就不演示了，如果需要请参考AopTest.java中的annotationAroundAdviceTest测试方法。

6.4.5 引入

@AspectJ风格的引入声明在切面中使用org.aspectj.lang.annotation包下的@DeclareParents声明：

java代码：

```
@DeclareParents(
value=" AspectJ语法类型表达式",
defaultImpl=引入接口的默认实现类)
private Interface interface;
```

value：匹配需要引入接口的目标对象的AspectJ语法类型表达式；与Schema方式中的types-matching属性同义；

private Interface interface：指定需要引入的接口；

defaultImpl：指定引入接口的默认实现类，没有与Schema方式中的delegate-ref属性同义的定义方式；

java代码：

```
@DeclareParents(
    value="cn.javass..*.IHelloWorldService+", defaultImpl=cn.javass.spring.chapter6.service.
private IIntroductionService introductionService;
```

其中测试代码与Schema方式几乎一样，在此就不演示了，如果需要请参考AopTest.java中的testAnnotationIntroduction测试方法。

1.24 【第六章】 AOP 之 6.5 AspectJ切入点语法详解 ——跟我学spring3

发表时间: 2012-02-24 关键字: spring

6.5.1 Spring AOP支持的AspectJ切入点指示符

切入点指示符用来指示切入点表达式目的，，在Spring AOP中目前只有执行方法这一个连接点，Spring AOP支持的AspectJ切入点指示符如下：

execution：用于匹配方法执行的连接点；

within：用于匹配指定类型内的方法执行；

this：用于匹配当前AOP代理对象类型的执行方法；注意是AOP代理对象的类型匹配，这样就可能包括引入接口也类型匹配；

target：用于匹配当前目标对象类型的执行方法；注意是目标对象的类型匹配，这样就不包括引入接口也类型匹配；

args：用于匹配当前执行的方法传入的参数为指定类型的执行方法；

@within：用于匹配所以持有指定注解类型内的方法；

@target：用于匹配当前目标对象类型的执行方法，其中目标对象持有指定的注解；

@args：用于匹配当前执行的方法传入的参数持有指定注解的执行；

@annotation：用于匹配当前执行方法持有指定注解的方法；

bean：Spring AOP扩展的，AspectJ没有对于指示符，用于匹配特定名称的Bean对象的执行方法；

reference pointcut：表示引用其他命名切入点，只有@ApectJ风格支持，Schema风格不支持。

AspectJ切入点支持的切入点指示符还有： call、get、set、preinitialization、staticinitialization、initialization、handler、adviceexecution、withincode、cflow、

cflowbelow、if、@this、@withincode；但Spring AOP目前不支持这些指示符，使用这些指示符将抛出IllegalArgumentException异常。这些指示符Spring AOP可能会在以后进行扩展。

6.5.1 命名及匿名切入点

命名切入点可以被其他切入点引用，而匿名切入点是不可以的。

只有@AspectJ支持命名切入点，而Schema风格不支持命名切入点。

如下所示，@AspectJ使用如下方式引用命名切入点：

6.5.2 ；类型匹配语法

首先让我们来了解下AspectJ类型匹配的通配符：

* ：匹配任何数量字符；

.. ：（两个点）匹配任何数量字符的重复，如在类型模式中匹配任何数量子包；而在方法参数模式中匹配任何数量参数。

+ ：匹配指定类型的子类型；仅能作为后缀放在类型模式后边。

java.lang.String 匹配String类型；

java.*.String 匹配java包下的任何“一级子包”下的String类型；

如匹配java.lang.String，但不匹配java.lang.ss.String

java..* 匹配java包及任何子包下的任何类型;

如匹配java.lang.String、java.lang.annotation.Annotation

java.lang.*ing 匹配任何java.lang包下的以ing结尾的类型；

java.lang.Number+ 匹配java.lang包下的任何Number的自类型；

如匹配java.lang.Integer，也匹配java.math.BigInteger

接下来再看一下具体的匹配表达式类型吧：

匹配类型：使用如下方式匹配

注解？ 类的全限定名字

- **注解：**可选，类型上持有的注解，如@Deprecated；
- **类的全限定名：**必填，可以是任何类全限定名。

匹配方法执行：使用如下方式匹配：

注解？ 修饰符? 返回值类型 类型声明?方法名(参数列表) 异常列表？

-
- **注解：**可选，方法上持有的注解，如@Deprecated；
- **修饰符：**可选，如public、protected；
- **返回值类型：**必填，可以是任何类型模式；“*”表示所有类型；
- **类型声明：**可选，可以是任何类型模式；

- **方法名：**必填，可以使用“*”进行模式匹配；
- **参数列表：**“()”表示方法没有任何参数；“(..)”表示匹配接受任意个参数的方法，“(..,java.lang.String)”表示匹配接受java.lang.String类型的参数结束，且其前边可以接受有任意个参数的方法；“(java.lang.String,..)” 表示匹配接受java.lang.String类型的参数开始，且其后边可以接受任意个参数的方法；“(*,java.lang.String)” 表示匹配接受java.lang.String类型的参数结束，且其前边接受有一个任意类型参数的方法；
- **异常列表：**可选，以“throws 异常全限定名列表”声明，异常全限定名列表如有多个以“，”分割，如throws java.lang.IllegalArgumentException, java.lang.ArrayIndexOutOfBoundsException。

匹配Bean名称：可以使用Bean的id或name进行匹配，并且可使用通配符“*” ；

6.5.3 组合切入点表达式

AspectJ使用 且（&&）、或（||）、非（！）来组合切入点表达式。

在Schema风格下，由于在XML中使用“&&”需要使用转义字符“&”来代替之，所以很不方便，因此Spring ASP 提供了and、or、not来代替&&、||、！。

6.5.4 切入点使用示例

一、execution：使用“execution(方法表达式)”匹配方法执行；

模式	描述
public * *(..)	任何公共方法的执行
* cn.javass..IPointcutService.*()	cn.javass包及所有子包下IPointcutService接口中的任何无参方法
* cn.javass..*.*(..)	cn.javass包及所有子包下任何类的任何方法

* cn.javass..IPointcutService.*(*)	cn.javass包及所有子包下IPointcutService接口的任何只有一个参数方法
* (!cn.javass..IPointcutService+).*(..)	非“cn.javass包及所有子包下IPointcutService接口及子类型”的任何方法
* cn.javass..IPointcutService+.*()	cn.javass包及所有子包下IPointcutService接口及子类型的的任何无参方法
* cn.javass..IPointcut*.test*(java.util.Date)	cn.javass包及所有子包下IPointcut前缀类型的的以test开头的只有一个参数类型为java.util.Date的方法，注意该匹配是根据方法签名的参数类型进行匹配的，而不是根据执行时传入的参数类型决定的 如定义方法：public void test(Object obj);即使执行时传入java.util.Date，也不会匹配的；
* cn.javass..IPointcut*.test*(..) throws IllegalArgumentException, ArrayIndexOutOfBoundsException	cn.javass包及所有子包下IPointcut前缀类型的的任何方法，且抛出IllegalArgumentException和ArrayIndexOutOfBoundsException异常
* (cn.javass..IPointcutService+ && java.io.Serializable+).*(..)	任何实现了cn.javass包及所有子包下IPointcutService接口和java.io.Serializable接口的类型的任何方法
@java.lang.Deprecated * *(..)	任何持有@java.lang.Deprecated注解的方法
@java.lang.Deprecated @cn.javass..Secure * *(..)	任何持有@java.lang.Deprecated和@cn.javass..Secure注解的方法
@(java.lang.Deprecated || cn.javass..Secure) * *(..)	任何持有@java.lang.Deprecated或@ cn.javass..Secure注解的方法
(@cn.javass..Secure *) *(..)	任何返回值类型持有@cn.javass..Secure的方法

* (@cn.javass..Secure *).*(..)	任何定义方法的类型持有@cn.javass..Secure的方法
* *(@cn.javass..Secure (*) , @cn.javass..Secure (*))	任何签名带有两个参数的方法，且这个两个参数都被@Secure标记了， 如public void test(@Secure String str1, @Secure String str1);
* *((@ cn.javass..Secure *))或 * *(@ cn.javass..Secure *)	任何带有一个参数的方法，且该参数类型持有@cn.javass..Secure； 如public void test(Model model);且Model类上持有@Secure注解
* *(@cn.javass..Secure (@cn.javass..Secure *) , @ cn.javass..Secure (@cn.javass..Secure *))	任何带有两个参数的方法，且这两个参数都被@cn.javass..Secure标记了；且这两个参数的类型上都持有@ cn.javass..Secure；
* *(java.util.Map<cn.javass..Model, cn.javass..Model> , ..)	任何带有一个java.util.Map参数的方法，且该参数类型是以< cn.javass..Model, cn.javass..Model >为泛型参数；注意只匹配第一个参数为java.util.Map,不包括子类型； 如public void test(HashMap<Model, Model> map, String str);将不匹配，必须使用“* *(java.util.HashMap<cn.javass..Model,cn.javass..Model> , ..)”进行匹配； 而public void test(Map map, int i);也将不匹配，因为泛型参数不匹配

* *(java.util.Collection<@cn.javass..Secure *>)	任何带有一个参数（类型为java.util.Collection）的方法，且该参数类型是有一个泛型参数，该泛型参数类型上持有@cn.javass..Secure注解； 如public void test(Collection<Model> collection);Model类型上持有@cn.javass..Secure
* *(java.util.Set<? extends HashMap>)	任何带有一个参数的方法，且传入的参数类型是有一个泛型参数，该泛型参数类型继承与HashMap； **Spring AOP目前测试不能正常工作**
* *(java.util.List<? super HashMap>)	任何带有一个参数的方法，且传入的参数类型是有一个泛型参数，该泛型参数类型是HashMap的基类型；如public voi test(Map map)； **Spring AOP目前测试不能正常工作**
* *(*<@cn.javass..Secure *>)	任何带有一个参数的方法，且该参数类型是有一个泛型参数，该泛型参数类型上持有@cn.javass..Secure注解； **Spring AOP目前测试不能正常工作**

二、within：使用“within(类型表达式)”匹配指定类型内的方法执行；

模式	描述
within(cn.javass..*)	cn.javass包及子包下的任何方法执行
within(cn.javass..IPointcutService+)	cn.javass包或所有子包下IPointcutService类型及子类型的任何方法
within(@cn.javass..Secure *)	持有cn.javass..Secure注解的任何类型的任何方法

	必须是在目标对象上声明这个注解，在接口上声明的对它不起作用

三、this：使用“this(类型全限定名)”匹配当前AOP代理对象类型的执行方法；注意是AOP代理对象的类型匹配，这样就可能包括引入接口方法也可以匹配；注意this中使用的表达式必须是类型全限定名，不支持通配符；

模式	**描述**
this(cn.javass.spring.chapter6.service.IPointcutService)	当前AOP对象实现了 IPointcutService接口的任何方法
this(cn.javass.spring.chapter6.service.IIntroductionService)	当前AOP对象实现了 IIntroductionService接口的任何方法 也可能是引入接口

四、**target：**使用“target(类型全限定名)”匹配当前目标对象类型的执行方法；注意是目标对象的类型匹配，这样就不包括引入接口也类型匹配；注意target中使用的表达式必须是类型全限定名，不支持通配符；

模式	描述
target(cn.javass.spring.chapter6.service.IPointcutService)	当前目标对象（非AOP对象）实现了IPointcutService接口的任何方法
target(cn.javass.spring.chapter6.service.IIntroductionService)	当前目标对象（非AOP对象） 实现了IIntroductionService 接口的任何方法 不可能是引入接口

五、**args：**使用“args(参数类型列表)”匹配当前执行的方法传入的参数为指定类型的执行方法；注意是匹配传入的参数类型，不是匹配方法签名的参数类型；参数类型列表中的参数必须是类型全限定名，通配符不支持；args属于动态切入点，这种切入点开销非常大，非特殊情况最好不要使用；

模式	描述
args (java.io.Serializable,..)	任何一个以接受“传入参数类型为java.io.Serializable” 开头，且其后可跟任意个任意类型的参数的方法执行，args指定的参数类型是在运行时动态匹配的

六、@within：使用“@within(注解类型)”匹配所以持有指定注解类型内的方法；注解类型也必须是全限定类型名；

模式	描述
@within cn.javass.spring.chapter6.Secure)	任何目标对象对应的类型持有Secure注解的类方法；必须是在目标对象上声明这个注解，在接口上声明的对它不起作用

七、@target：使用“@target(注解类型)”匹配当前目标对象类型的执行方法，其中目标对象持有指定的注解；注解类型也必须是全限定类型名；

模式	描述

@target (cn.javass.spring.chapter6.Secure)	任何目标对象持有Secure注解的类方法；必须是在目标对象上声明这个注解，在接口上声明的对它不起作用

八、@args：使用“@args(注解列表)”匹配当前执行的方法传入的参数持有指定注解的执行；注解类型也必须是全限定类型名；

模式	**描述**
@args (cn.javass.spring.chapter6.Secure)	任何一个只接受一个参数的方法，且方法运行时传入的参数持有注解 cn.javass.spring.chapter6.Secure；动态切入点，类似于arg指示符；

九、@annotation：使用“@annotation(注解类型)”匹配当前执行方法持有指定注解的方法；注解类型也必须是全限定类型名；

模式	描述
@annotation(cn.javass.spring.chapter6.Secure)	当前执行方法上持有注解cn.javass.spring.chapter6.Secure将被匹配

十、bean：使用“bean(Bean id或名字通配符)”匹配特定名称的Bean对象的执行方法；Spring ASP扩展的，在AspectJ中无相应概念；

模式	描述
bean(*Service)	匹配所有以Service命名（id或name）结尾的Bean

十一、reference pointcut：表示引用其他命名切入点，只有@ApectJ风格支持，Schema风格不支持，如下所示：

比如我们定义如下切面：

java代码：

```
package cn.javass.spring.chapter6.aop;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
@Aspect
public class ReferencePointcutAspect {
    @Pointcut(value="execution(* *())")
    public void pointcut() {}
}
```

可以通过如下方式引用：

java代码：

```
@Before(value = "cn.javass.spring.chapter6.aop.ReferencePointcutAspect.pointcut()")
public void referencePointcutTest2(JoinPoint jp) {}
```

除了可以在@AspectJ风格的切面内引用外，也可以在Schema风格的切面定义内引用，引用方式与@AspectJ完全一样。

到此我们切入点表达式语法示例就介绍完了，我们这些示例几乎包含了日常开发中的所有情况，但当然还有更复杂的语法等等，如果以上介绍的不能满足您的需要，请参考AspectJ文档。

由于测试代码相当长，所以为了节约篇幅本示例代码在cn.javass.spring.chapter6. PointcutTest文件中，需要时请参考该文件。

原创内容，转自请注明出处【http://sishuok.com/forum/blogPost/list/0/2472.html】

1.25 【第六章】 AOP 之 6.6 通知参数 ——跟我学spring3

发表时间: 2012-02-25 关键字: spring

前边章节已经介绍了声明通知，但如果想获取被被通知方法参数并传递给通知方法，该如何实现呢？接下来我们将介绍两种获取通知参数的方式。

- **使用JoinPoint获取**：Spring AOP提供使用org.aspectj.lang.JoinPoint类型获取连接点数据，任何通知方法的第一个参数都可以是JoinPoint(环绕通知是ProceedingJoinPoint，JoinPoint子类)，当然第一个参数位置也可以是JoinPoint.StaticPart类型，这个只返回连接点的静态部分。

1) JoinPoint：提供访问当前被通知方法的目标对象、代理对象、方法参数等数据：

java代码：

```
package org.aspectj.lang;
import org.aspectj.lang.reflect.SourceLocation;
public interface JoinPoint {
    String toString();          //连接点所在位置的相关信息
    String toShortString();     //连接点所在位置的简短相关信息
    String toLongString();      //连接点所在位置的全部相关信息
    Object getThis();           //返回AOP代理对象
    Object getTarget();         //返回目标对象
    Object[] getArgs();         //返回被通知方法参数列表
    Signature getSignature();   //返回当前连接点签名
    SourceLocation getSourceLocation();//返回连接点方法所在类文件中的位置
    String getKind();           //连接点类型
    StaticPart getStaticPart(); //返回连接点静态部分
}
```

2) ProceedingJoinPoint：用于环绕通知，使用proceed()方法来执行目标方法：

java代码：

```
public interface ProceedingJoinPoint extends JoinPoint {
    public Object proceed() throws Throwable;
    public Object proceed(Object[] args) throws Throwable;
}
```

3) JoinPoint.StaticPart：提供访问连接点的静态部分，如被通知方法签名、连接点类型等：

java代码：

```
public interface StaticPart {
    Signature getSignature();    //返回当前连接点签名
    String getKind();           //连接点类型
    int getId();                //唯一标识
    String toString();          //连接点所在位置的相关信息
    String toShortString();      //连接点所在位置的简短相关信息
    String toLongString();      //连接点所在位置的全部相关信息
}
```

使用如下方式在通知方法上声明，必须是在第一个参数，然后使用jp.getArgs()就能获取到被通知方法参数：

java代码：

```
@Before(value="execution(* sayBefore(*))")
public void before(JoinPoint jp) {}

@Before(value="execution(* sayBefore(*))")
public void before(JoinPoint.StaticPart jp) {}
```

- **自动获取：**通过切入点表达式可以将相应的参数自动传递给通知方法，例如前边章节讲过的返回值和异常是如何传递给通知方法的。

在Spring AOP中，除了execution和bean指示符不能传递参数给通知方法，其他指示符都可以将匹配的相应参数或对象自动传递给通知方法。

java代码：

```
@Before(value="execution(* test(*)) && args(param)", argNames="param")
public void before1(String param) {
    System.out.println("===param:" + param);
}
```

切入点表达式execution(* test(*)) && args(param) ：

1）首先execution(* test(*))匹配任何方法名为test，且有一个任何类型的参数；

2）args(param)将首先查找通知方法上同名的参数，并在方法执行时（运行时）匹配传入的参数是使用该同名参数类型，即java.lang.String；如果匹配将把该被通知参数传递给通知方法上同名参数。

其他指示符（除了execution和bean指示符）都可以使用这种方式进行参数绑定。

在此有一个问题，即前边提到的类似于【3.1.2构造器注入】中的参数名注入限制：**在class文件中没生成变量调试信息是获取不到方法参数名字的。**

所以我们可以使用策略来确定参数名：

1、如果我们通过“argNames”属性指定了参数名，那么就是要我们指定的；

java代码：

```
@Before(value=" args(param)", argNames="param") //明确指定了
public void before1(String param) {
    System.out.println("===param:" + param);
}
```

2、如果第一个参数类型是JoinPoint、ProceedingJoinPoint或JoinPoint.StaticPart类型，应该从“argNames”属性省略掉该参数名（可选，写上也对），这些类型对象会自动传入的，但必须作为第一个参数；

java代码：

```
@Before(value=" args(param)", argNames="param") //明确指定了
public void before1(JoinPoint jp, String param) {
    System.out.println("===param:" + param);
}
```

3、如果“**class文件中含有变量调试信息**”将使用这些方法签名中的参数名来确定参数名；

java代码：

```
@Before(value=" args(param)") //不需要argNames了
public void before1(JoinPoint jp, String param) {
    System.out.println("===param:" + param);
}
```

4、如果没有“**class文件中含有变量调试信息**”，将尝试自己的参数匹配算法，如果发现参数绑定有二义性将抛出AmbiguousBindingException异常；对于只有一个绑定变量的切入点表达式，而通知方法只接受一个参数，说明绑定参数是明确的，从而能配对成功。

java代码：

```
@Before(value=" args(param)")
public void before1(JoinPoint jp, String param) {
    System.out.println("===param:" + param);
}
```

5、以上策略失败将抛出IllegalArgumentException。

接下来让我们示例一下组合情况吧：

java代码：

```
@Before(args(param) && target(bean) && @annotation(secure)",
        argNames="jp,param,bean,secure")
public void before5(JoinPoint jp, String param,
 IPointcutService pointcutService, Secure secure) {
……
}
```

该示例的执行步骤如图6-5所示。

图6-5 参数自动获取流程

除了上边介绍的普通方式，也可以对使用命名切入点自动获取参数：

java代码：

```
@Pointcut(value="args(param)", argNames="param")
private void pointcut1(String param){}
@Pointcut(value="@annotation(secure)", argNames="secure")
private void pointcut2(Secure secure){}

@Before(value = "pointcut1(param) && pointcut2(secure)",
 argNames="param, secure")
public void before6(JoinPoint jp, String param, Secure secure) {
……
}
```

自此给通知传递参数已经介绍完了，示例代码在cn.javass.spring.chapter6.ParameterTest文件中。

1.26 【第六章】 AOP 之 6.7 通知顺序 ——跟我学spring3

发表时间: 2012-02-25 关键字: spring

如果我们有多个通知想要在同一连接点执行，那执行顺序如何确定呢？Spring AOP使用AspectJ的优先级规则来确定通知执行顺序。总共有两种情况：同一切面中通知执行顺序、不同切面中的通知执行顺序。

首先让我们看下

1）同一切面中通知执行顺序：如图6-6所示。

图6-6 同一切面中的通知执行顺序

而如果在同一切面中定义两个相同类型通知（如同是前置通知或环绕通知（proceed之前））并在同一连接点执行时，其执行顺序是未知的，如果确实需要指定执行顺序需要将通知重构到两个切面，然后定义切面的执行顺序。

java代码：

```
错误“Advice precedence circularity error”：说明AspectJ无法决定通知的执行顺序，只要将通知方法
```

2）**不同切面中的通知执行顺序：**当定义在不同切面的相同类型的通知需要在同一个连接点执行，如果没指定切面的执行顺序，这两个通知的执行顺序将是未知的。

如果需要他们顺序执行，可以通过指定切面的优先级来控制通知的执行顺序。

Spring中可以通过在切面实现类上实现org.springframework.core.Ordered接口或使用Order注解来指定切面优先级。在多个切面中，Ordered.getValue()方法返回值（或者注解值）较小值的那个切面拥有较高优先级，如图6-7所示。

图6-7 两个切面指定了优先级

对于@AspectJ风格和注解风格可分别用以下形式指定优先级：

在此我们不推荐使用实现Ordered接口方法，所以没介绍，示例代码在cn.javass.spring.chapter6.OrderAopTest文件中。

1.27 【第六章】 AOP 之 6.8 切面实例化模型 ——跟我学spring3

发表时间: 2012-02-25 关键字: spring

所谓切面实例化模型指何时实例化切面。

Spring AOP支持AspectJ的singleton、perthis、pertarget实例化模型（目前不支持percflow、percflowbelow 和pertypewithin）。

- **singleton**：即切面只会有一个实例；
- **perthis**：每个切入点表达式匹配的连接点对应的AOP对象都会创建一个新切面实例；
- **pertarget**：每个切入点表达式匹配的连接点对应的目标对象都会创建一个新的切面实例；

默认是singleton实例化模型，Schema风格只支持singleton实例化模型，而@AspectJ风格支持这三种实例化模型。

singleton：使用@Aspect()指定，即默认就是单例实例化模式，在此就不演示示例了。

perthis：每个切入点表达式匹配的连接点对应的AOP对象都会创建一个新的切面实例，使用@Aspect("perthis(切入点表达式)")指定切入点表达式；

如@Aspect("perthis(this(cn.javass.spring.chapter6.service.IIntroductionService))")将对每个匹配“this(cn.javass.spring.chapter6.service.IIntroductionService)”切入点表达式的AOP代理对象创建一个切面实例，注意“IIntroductionService”可能是引入接口。

pertarget：每个切入点表达式匹配的连接点对应的目标对象都会创建一个新的切面实例，使用@Aspect("pertarget(切入点表达式)")指定切入点表达式；

如@Aspect("pertarget(target(cn.javass.spring.chapter6. service.IPointcutService))")将对每个匹配“target(cn.javass.spring.chapter6.service. IPointcutService)”切入点表达式的目标对象创建一个切面，注意“IPointcutService”不可能是引入接口。

在进行切面定义时必须将切面scope定义为“prototype”，如“<bean class="……Aspect" scope="prototype"/>”，否则不能为每个匹配的连接点的目标对象或AOP代理对象创建一个切面。

示例请参考cn.javass.spring.chapter6. InstanceModelTest。

1.28 【第六章】 AOP 之 6.9 代理机制 ——跟我学spring3

发表时间: 2012-02-25 关键字: spring

Spring AOP通过代理模式实现，目前支持两种代理：JDK动态代理、CGLIB代理来创建AOP代理，Spring建议优先使用JDK动态代理。

- **JDK动态代理：**使用java.lang.reflect.Proxy动态代理实现，即提取目标对象的接口，然后对接口创建AOP代理。
- **CGLIB代理：**CGLIB代理不仅能进行接口代理，也能进行类代理，CGLIB代理需要注意以下问题：

不能通知final方法，因为final方法不能被覆盖（CGLIB通过生成子类来创建代理）。

会产生两次构造器调用，第一次是目标类的构造器调用，第二次是CGLIB生成的代理类的构造器调用。如果需要CGLIB代理方法，请确保两次构造器调用不影响应用。

Spring AOP默认首先使用JDK动态代理来代理目标对象，如果目标对象没有实现任何接口将使用CGLIB代理，如果需要强制使用CGLIB代理，请使用如下方式指定：

对于Schema风格配置切面使用如下方式来指定使用CGLIB代理：

java代码：

```
<aop:config proxy-target-class="true">
</aop:config>
```

而如果使用@AspectJ风格使用如下方式来指定使用CGLIB代理：

java代码：

```
<aop:aspectj-autoproxy proxy-target-class="true"/>
```

1.29 【第七章】 对JDBC的支持 之 7.1 概述 ——跟我学spring3

发表时间: 2012-02-26 关键字: spring

7.1 概述

7.1.1 JDBC回顾

传统应用程序开发中，进行JDBC编程是相当痛苦的，如下所示：

java代码：

```
//cn.javass.spring.chapter7. TraditionalJdbcTest
@Test
public void test() throws Exception {
    Connection conn = null;
    PreparedStatement pstmt = null;
    try {
      conn = getConnection();              //1.获取JDBC连接
                                          //2.声明SQL
      String sql = "select * from INFORMATION_SCHEMA.SYSTEM_TABLES";
      pstmt = conn.prepareStatement(sql);    //3.预编译SQL
      ResultSet rs = pstmt.executeQuery();   //4.执行SQL
      process(rs);                          //5.处理结果集
      closeResultSet(rs);                    //5.释放结果集
      closeStatement(pstmt);                 //6.释放Statement
      conn.commit();                        //8.提交事务
    } catch (Exception e) {
      //9.处理异常并回滚事务
      conn.rollback();
      throw e;
    } finally {
      //10.释放JDBC连接，防止JDBC连接不关闭造成的内存泄漏
      closeConnection(conn);
    }
}
```

以上代码片段具有冗长、重复、容易忘记某一步骤从而导致出错、显示控制事务、显示处理受检查异常等等。

有朋友可能重构出自己的一套JDBC模板，从而能简化日常开发，但自己开发的JDBC模板不够通用，而且对于每一套JDBC模板实现都差不多，从而导致开发人员必须掌握每一套模板。

Spring JDBC提供了一套JDBC抽象框架，用于简化JDBC开发，而且如果各个公司都使用该抽象框架，开发人员首先减少了学习成本，直接上手开发，如图7-1所示。

图7-1 Spring JDBC与传统JDBC编程对比

7.1.2 Spring对JDBC的支持

Spring通过抽象JDBC访问并提供一致的API来简化JDBC编程的工作量。我们只需要**声明SQL、调用合适的Spring JDBC框架API、处理结果集**即可。事务由Spring管理，并将JDBC受查异常转换为Spring一致的非受查异常，从而简化开发。

Spring主要提供**JDBC模板方式、关系数据库对象化方式和SimpleJdbc**方式三种方式来简化JDBC编程，这三种方式就是Spring JDBC的工作模式：

- **JDBC模板方式：**Spring JDBC框架提供以下几种模板类来简化JDBC编程，实现GoF模板设计模式，将可变部分和非可变部分分离，可变部分采用回调接口方式由用户来实现：如JdbcTemplate、NamedParameterJdbcTemplate、SimpleJdbcTemplate。
- **关系数据库操作对象化方式：**Spring JDBC框架提供了将关系数据库操作对象化的表示形式，从而使用户可以采用面向对象编程来完成对数据库的访问；如MappingSqlQuery、SqlUpdate、SqlCall、SqlFunction、StoredProcedure等类。这些类的实现一旦建立即可重用并且是线程安全的。
- **SimpleJdbc方式：**Spring JDBC框架还提供了**SimpleJdbc方式**来简化JDBC编程，SimpleJdbcInsert 、SimpleJdbcCall用来简化数据库表插入、存储过程或函数访问。

Spring JDBC还提供了一些强大的工具类，如DataSourceUtils来在必要的时候手工获取数据库连接等。

7.1.4 Spring的JDBC架构

Spring JDBC抽象框架由四部分组成：datasource、support、core、object。如图7-2所示。

图7-2 Spring JDBC架构图

support包：提供将JDBC异常转换为DAO非检查异常转换类、一些工具类如JdbcUtils等。

datasource包：提供简化访问JDBC 数据源（javax.sql.DataSource实现）工具类，并提供了一些DataSource简单实现类从而能使从这些DataSource获取的连接能自动得到Spring管理事务支持。

core包：提供JDBC模板类实现及可变部分的回调接口，还提供SimpleJdbcInsert等简单辅助类。

object包：提供关系数据库的对象表示形式，如MappingSqlQuery、SqlUpdate、SqlCall、SqlFunction、StoredProcedure等类，该包是基于core包JDBC模板类实现。

1.30 【第七章】 对JDBC的支持 之 7.2 JDBC模板类 ——跟我学spring3

发表时间: 2012-02-26 关键字: spring

7.2 JDBC模板类

7.2.1 概述

Spring JDBC抽象框架core包提供了JDBC模板类，其中JdbcTemplate是core包的核心类，所以其他模板类都是基于它封装完成的，JDBC模板类是第一种工作模式。

JdbcTemplate类通过模板设计模式帮助我们消除了冗长的代码，只做需要做的事情（即可变部分），并且帮我们做哪些固定部分，如连接的创建及关闭。

JdbcTemplate类对可变部分采用回调接口方式实现，如ConnectionCallback通过回调接口返回给用户一个连接，从而可以使用该连接做任何事情、StatementCallback通过回调接口返回给用户一个Statement，从而可以使用该Statement做任何事情等等，还有其他一些回调接口如图7-3所示。

图7-3 JdbcTemplate支持的回调接口

Spring除了提供JdbcTemplate核心类，还提供了基于JdbcTemplate实现的NamedParameterJdbcTemplate类用于支持命名参数绑定、 SimpleJdbcTemplate类用于支持Java5+的可变参数及自动装箱拆箱等特性。

7.2.3 传统JDBC编程替代方案

前边我们已经使用过传统JDBC编程方式，接下来让我们看下Spring JDBC框架提供的更好的解决方案。

1）准备需要的jar包并添加到类路径中：

java代码：

```
//JDBC抽象框架模块
org.springframework.jdbc-3.0.5.RELEASE.jar
//Spring事务管理及一致的DAO访问及非检查异常模块
org.springframework.transaction-3.0.5.RELEASE.jar
//hsqldb驱动，hsqldb是一个开源的Java实现数据库，请下载hsqldb2.0.0+版本
hsqldb.jar
```

2）传统JDBC编程替代方案：

在使用JdbcTemplate模板类时必须通过DataSource获取数据库连接，Spring JDBC提供了DriverManagerDataSource实现，它通过包装“DriverManager.getConnection”获取数据库连接，具体DataSource相关请参考【7.5.1控制数据库连接】。

java代码：

```
package cn.javass.spring.chapter7;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowCallbackHandler;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
public class JdbcTemplateTest {
    private static JdbcTemplate jdbcTemplate;
    @BeforeClass
    public static void setUpClass() {
        String url = "jdbc:hsqldb:mem:test";
        String username = "sa";
```

```
        String password = "";
        DriverManagerDataSource dataSource = new DriverManagerDataSource(url, username, passw
        dataSource.setDriverClassName("org.hsqldb.jdbcDriver");
        jdbcTemplate = new JdbcTemplate(dataSource);
    }
    @Test
    public void test() {
        //1.声明SQL
        String sql = "select * from INFORMATION_SCHEMA.SYSTEM_TABLES";
        jdbcTemplate.query(sql, new RowCallbackHandler() {
            @Override
            public void processRow(ResultSet rs) throws SQLException {
                //2.处理结果集
                String value = rs.getString("TABLE_NAME");
                System.out.println("Column TABLENAME:" + value);
            }
        });
    }
}
```

接下来让我们具体分析一下：

1） **jdbc:hsqldb:mem:test：**表示使用hsqldb内存数据库，数据库名为“test”。

2） **public static void setUpClass()：**使用junit的@BeforeClass注解，表示在所以测试方法之前执行，且只执行一次。在此方法中定义了DataSource并使用DataSource对象创建了JdbcTemplate对象。JdbcTemplate对象是线程安全的。

3） **JdbcTemplate执行流程：**首先定义SQL，其次调用JdbcTemplate方法执行SQL，最后通过RowCallbackHandler回调处理ResultSet结果集。

Spring JDBC解决方法相比传统JDBC编程方式是不是简单多了，是不是只有可变部分需要我们来做，其他的都由Spring JDBC框架来实现了。

接下来让我们深入JdbcTemplate及其扩展吧。

7.2.4 JdbcTemplate

首先让我们来看下如何使用JdbcTemplate来实现增删改查。

一、首先创建表结构：

java代码：

```
//代码片段(cn.javass.spring.chapter7.JdbcTemplateTest)
@Before
public void setUp() {
   String createTableSql = "create memory table test" + "(id int GENERATED BY DEFAULT AS IDE
   jdbcTemplate.update(createTableSql);
}
@After
public void tearDown() {
  String dropTableSql = "drop table test";
  jdbcTemplate.execute(dropTableSql);
}
```

1） org.junit包下的**@Before和@After**分别表示在测试方法之前和之后执行的方法，对于每个测试方法都将执行一次；

2） **create memory table test**表示创建hsqldb内存表，包含两个字段id和name，其中id是具有自增功能的主键，如果有朋友对此不熟悉hsqldb可以换成熟悉的数据库。

二、定义测试骨架，该测试方法将用于实现增删改查测试：

java代码：

```
@Test
public void testCURD() {
    insert();
    delete();
    update();
    select();
}
```

三、新增测试：

java代码：

```
private void insert() {
  jdbcTemplate.update("insert into test(name) values('name1')");
  jdbcTemplate.update("insert into test(name) values('name2')");
  Assert.assertEquals(2, jdbcTemplate.queryForInt("select count(*) from test"));
}
```

四、删除测试：

java代码：

```
private void delete() {
  jdbcTemplate.update("delete from test where name=?", new Object[]{"name2"});
  Assert.assertEquals(1, jdbcTemplate.queryForInt("select count(*) from test"));
}
```

五、更新测试：

java代码：

```
private void update() {
  jdbcTemplate.update("update test set name='name3' where name=?", new Object[]{"name1"});
  Assert.assertEquals(1, jdbcTemplate.queryForInt("select count(*) from test where name='nam
}
```

六、查询测试：

java代码：

```
private void select() {
  jdbcTemplate.query("select * from test", new RowCallbackHandler(){
    @Override
    public void processRow(ResultSet rs) throws SQLException {
      System.out.print("====id:" + rs.getInt("id"));
      System.out.println(",name:" + rs.getString("name"));
    }
  });
}
```

看完以上示例，大家是否觉得JdbcTemplate简化了我们很多劳动力呢？接下来让我们深入学习一下JdbcTemplate提供的方法。

JdbcTemplate主要提供以下五类方法：

- **execute方法：**可以用于执行任何SQL语句，一般用于执行DDL语句；

- **update方法及batchUpdate方法：**update方法用于执行新增、修改、删除等语句；batchUpdate方法用于执行批处理相关语句；
- **query方法及queryForXXX方法：**用于执行查询相关语句；
- **call方法：**用于执行存储过程、函数相关语句。

JdbcTemplate类支持的回调类：

- **预编译语句及存储过程创建回调：**用于根据JdbcTemplate提供的连接创建相应的语句；

PreparedStatementCreator：通过回调获取JdbcTemplate提供的Connection，由用户使用该Conncetion创建相关的PreparedStatement；

CallableStatementCreator：通过回调获取JdbcTemplate提供的Connection，由用户使用该Conncetion创建相关的CallableStatement；

- **预编译语句设值回调：**用于给预编译语句相应参数设值；

PreparedStatementSetter：通过回调获取JdbcTemplate提供的PreparedStatement，由用户来对相应的预编译语句相应参数设值；

BatchPreparedStatementSetter：；类似于PreparedStatementSetter，但用于批处理，需要指定批处理大小；

- **自定义功能回调：**提供给用户一个扩展点，用户可以在指定类型的扩展点执行任何数量需要的操作；

ConnectionCallback：通过回调获取JdbcTemplate提供的Connection，用户可在该Connection执行任何数量的操作；

StatementCallback：通过回调获取JdbcTemplate提供的Statement，用户可以在该Statement执行任何数量的操作；

PreparedStatementCallback：通过回调获取JdbcTemplate提供的PreparedStatement，用户可以在该PreparedStatement执行任何数量的操作；

CallableStatementCallback：通过回调获取JdbcTemplate提供的CallableStatement，用户可以在该CallableStatement执行任何数量的操作；

- **结果集处理回调：**通过回调处理ResultSet或将ResultSet转换为需要的形式；

RowMapper：用于将结果集每行数据转换为需要的类型，用户需实现方法mapRow(ResultSet rs, int rowNum)来完成将每行数据转换为相应的类型。

RowCallbackHandler：用于处理ResultSet的每一行结果，用户需实现方法processRow(ResultSet rs)来完成处理，在该回调方法中无需执行rs.next()，该操作由JdbcTemplate来执行，用户只需按行获取数据然后处理即可。

ResultSetExtractor：用于结果集数据提取，用户需实现方法extractData(ResultSet rs)来处理结果集，用户必须处理整个结果集；

接下来让我们看下具体示例吧，在示例中不可能介绍到JdbcTemplate全部方法及回调类的使用方法，我们只介绍代表性的，其余的使用都是类似的；

1）预编译语句及存储过程创建回调、自定义功能回调使用：

java代码：

```
@Test
public void testPpreparedStatement1() {
  int count = jdbcTemplate.execute(new PreparedStatementCreator() {
     @Override
     public PreparedStatement createPreparedStatement(Connection conn)
         throws SQLException {
         return conn.prepareStatement("select count(*) from test");
     }}, new PreparedStatementCallback<Integer>() {
     @Override
     public Integer doInPreparedStatement(PreparedStatement pstmt)
         throws SQLException, DataAccessException {
         pstmt.execute();
         ResultSet rs = pstmt.getResultSet();
         rs.next();
         return rs.getInt(1);
      }});
   Assert.assertEquals(0, count);
```

```
}
```

首先使用PreparedStatementCreator创建一个预编译语句，其次由JdbcTemplate通过PreparedStatementCallback回调传回，由用户决定如何执行该PreparedStatement。此处我们使用的是execute方法。

2）预编译语句设值回调使用：

java代码：

```
@Test
public void testPreparedStatement2() {
  String insertSql = "insert into test(name) values (?)";
  int count = jdbcTemplate.update(insertSql, new PreparedStatementSetter() {
      @Override
      public void setValues(PreparedStatement pstmt) throws SQLException {
          pstmt.setObject(1, "name4");
  }});
  Assert.assertEquals(1, count);
  String deleteSql = "delete from test where name=?";
  count = jdbcTemplate.update(deleteSql, new Object[] {"name4"});
  Assert.assertEquals(1, count);
}
```

通过JdbcTemplate的int update(String sql, PreparedStatementSetter pss)执行预编译sql，其中sql参数为“insert into test(name) values (?) ”，该sql有一个占位符需要在执行前设值，PreparedStatementSetter实现就是为了设值，使用setValues(PreparedStatement pstmt)回调方法设值相应的占位符位置的值。JdbcTemplate也提

供一种更简单的方式“update(String sql, Object... args)”来实现设值，所以只要当使用该种方式不满足需求时才应使用PreparedStatementSetter。

3）结果集处理回调：

java代码：

```
@Test
public void testResultSet1() {
  jdbcTemplate.update("insert into test(name) values('name5')");
  String listSql = "select * from test";
  List result = jdbcTemplate.query(listSql, new RowMapper<Map>() {
      @Override
      public Map mapRow(ResultSet rs, int rowNum) throws SQLException {
          Map row = new HashMap();
          row.put(rs.getInt("id"), rs.getString("name"));
          return row;
  }});
  Assert.assertEquals(1, result.size());
  jdbcTemplate.update("delete from test where name='name5'");
}
```

RowMapper接口提供mapRow(ResultSet rs, int rowNum)方法将结果集的每一行转换为一个Map，当然可以转换为其他类，如表的对象画形式。

java代码：

```
@Test
public void testResultSet2() {
  jdbcTemplate.update("insert into test(name) values('name5')");
  String listSql = "select * from test";
```

```
  final List result = new ArrayList();
  jdbcTemplate.query(listSql, new RowCallbackHandler() {
      @Override
      public void processRow(ResultSet rs) throws SQLException {
          Map row = new HashMap();
          row.put(rs.getInt("id"), rs.getString("name"));
          result.add(row);
  }});
  Assert.assertEquals(1, result.size());
  jdbcTemplate.update("delete from test where name='name5'");
}
```

RowCallbackHandler接口也提供方法processRow(ResultSet rs)，能将结果集的行转换为需要的形式。

java代码：

```
@Test
public void testResultSet3() {
  jdbcTemplate.update("insert into test(name) values('name5')");
  String listSql = "select * from test";
  List result = jdbcTemplate.query(listSql, new ResultSetExtractor<List>() {
      @Override
      public List extractData(ResultSet rs)
     throws SQLException, DataAccessException {
          List result = new ArrayList();
          while(rs.next()) {
              Map row = new HashMap();
              row.put(rs.getInt("id"), rs.getString("name"));
              result.add(row);
           }
           return result;
  }});
  Assert.assertEquals(0, result.size());
```

```
  jdbcTemplate.update("delete from test where name='name5'");
}
```

ResultSetExtractor使用回调方法extractData(ResultSet rs)提供给用户整个结果集，让用户决定如何处理该结果集。

当然JdbcTemplate提供更简单的queryForXXX方法，来简化开发：

java代码：

```
//1.查询一行数据并返回int型结果
jdbcTemplate.queryForInt("select count(*) from test");
//2. 查询一行数据并将该行数据转换为Map返回
jdbcTemplate.queryForMap("select * from test where name='name5'");
//3.查询一行任何类型的数据，最后一个参数指定返回结果类型
jdbcTemplate.queryForObject("select count(*) from test", Integer.class);
//4.查询一批数据，默认将每行数据转换为Map
jdbcTemplate.queryForList("select * from test");
//5.只查询一列数据列表，列类型是String类型，列名字是name
jdbcTemplate.queryForList("
select name from test where name=?", new Object[]{"name5"}, String.class);
//6.查询一批数据，返回为SqlRowSet，类似于ResultSet，但不再绑定到连接上
SqlRowSet rs = jdbcTemplate.queryForRowSet("select * from test");
```

3）存储过程及函数回调：

首先修改JdbcTemplateTest的setUp方法，修改后如下所示：

java代码：

```
@Before
public void setUp() {
    String createTableSql = "create memory table test" +
    "(id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY, " +
    "name varchar(100))";
    jdbcTemplate.update(createTableSql);

    String createHsqldbFunctionSql =
      "CREATE FUNCTION FUNCTION_TEST(str CHAR(100)) " +
      "returns INT begin atomic return length(str);end";
    jdbcTemplate.update(createHsqldbFunctionSql);
    String createHsqldbProcedureSql =
      "CREATE PROCEDURE PROCEDURE_TEST" +
      "(INOUT inOutName VARCHAR(100), OUT outId INT) " +
      "MODIFIES SQL DATA " +
      "BEGIN ATOMIC " +
      "  insert into test(name) values (inOutName); " +
      "  SET outId = IDENTITY(); " +
      "  SET inOutName = 'Hello,' + inOutName; " +
    "END";
    jdbcTemplate.execute(createHsqldbProcedureSql);
}
```

其中CREATE FUNCTION FUNCTION_TEST用于创建自定义函数，CREATE PROCEDURE PROCEDURE_TEST用于创建存储过程，注意这些创建语句是数据库相关的，本示例中的语句只适用于HSQLDB数据库。

其次修改JdbcTemplateTest的tearDown方法，修改后如下所示：

java代码：

```
@After
public void tearDown() {
    jdbcTemplate.execute("DROP FUNCTION FUNCTION_TEST");
    jdbcTemplate.execute("DROP PROCEDURE PROCEDURE_TEST");
    String dropTableSql = "drop table test";
    jdbcTemplate.execute(dropTableSql);
}
```

其中drop语句用于删除创建的存储过程、自定义函数及数据库表。

接下来看一下hsqldb如何调用自定义函数：

java代码：

```
@Test
public void testCallableStatementCreator1() {
    final String callFunctionSql = "{call FUNCTION_TEST(?)}";
    List<SqlParameter> params = new ArrayList<SqlParameter>();
    params.add(new SqlParameter(Types.VARCHAR));
    params.add(new SqlReturnResultSet("result",
       new ResultSetExtractor<Integer>() {
           @Override
           public Integer extractData(ResultSet rs) throws SQLException,
               DataAccessException {
               while(rs.next()) {
                   return rs.getInt(1);
               }
              return 0;
       }));
    Map<String, Object> outValues = jdbcTemplate.call(
       new CallableStatementCreator() {
            @Override
```

```
            public CallableStatement createCallableStatement(Connection conn) throws SQLExce
              CallableStatement cstmt = conn.prepareCall(callFunctionSql);
              cstmt.setString(1, "test");
              return cstmt;
    }}, params);
    Assert.assertEquals(4, outValues.get("result"));
}
```

- **{call FUNCTION_TEST(?)}：**定义自定义函数的sql语句，注意hsqldb {?= call ...}和{call ...}含义是一样的，而比如mysql中两种含义是不一样的；
- params：用于描述自定义函数占位符参数或命名参数类型；SqlParameter用于描述IN类型参数、SqlOutParameter用于描述OUT类型参数、SqlInOutParameter用于描述INOUT类型参数、SqlReturnResultSet用于描述调用存储过程或自定义函数返回的ResultSet类型数据，其中SqlReturnResultSet需要提供结果集处理回调用于将结果集转换为相应的形式，hsqldb自定义函数返回值是ResultSet类型。
- **CallableStatementCreator：**提供Connection对象用于创建CallableStatement对象
- outValues：调用call方法将返回类型为Map<String, Object>对象；
- **outValues.get("result")：**获取结果，即通过SqlReturnResultSet对象转换过的数据；其中SqlOutParameter、SqlInOutParameter、SqlReturnResultSet指定的name用于从call执行后返回的Map中获取相应的结果，即name是Map的键。

注：因为hsqldb {?= call ...}和{call ...}含义是一样的，因此调用自定义函数将返回一个包含结果的ResultSet。

最后让我们示例下mysql如何调用自定义函数：

java代码：

```
@Test
public void testCallableStatementCreator2() {
    JdbcTemplate mysqlJdbcTemplate = new JdbcTemplate(getMysqlDataSource);
    //2.创建自定义函数
```

```
String createFunctionSql =
    "CREATE FUNCTION FUNCTION_TEST(str VARCHAR(100)) " +
     "returns INT return LENGTH(str)";
String dropFunctionSql = "DROP FUNCTION IF EXISTS FUNCTION_TEST";
mysqlJdbcTemplate.update(dropFunctionSql);
mysqlJdbcTemplate.update(createFunctionSql);
//3.准备sql,mysql支持{?= call …}
final String callFunctionSql = "{?= call FUNCTION_TEST(?)}";
//4.定义参数
List<SqlParameter> params = new ArrayList<SqlParameter>();
params.add(new SqlOutParameter("result", Types.INTEGER));
params.add(new SqlParameter("str", Types.VARCHAR));
Map<String, Object> outValues = mysqlJdbcTemplate.call(
new CallableStatementCreator() {
    @Override
    public CallableStatement createCallableStatement(Connection conn) throws SQLException {
      CallableStatement cstmt = conn.prepareCall(callFunctionSql);
      cstmt.registerOutParameter(1, Types.INTEGER);
      cstmt.setString(2, "test");
        return cstmt;
    }}, params);
   Assert.assertEquals(4, outValues.get("result"));
}
public DataSource getMysqlDataSource() {
    String url = "jdbc:mysql://localhost:3306/test";
    DriverManagerDataSource dataSource =
        new DriverManagerDataSource(url, "root", "");     dataSource.setDriverClassName("com
    return dataSource;
}
```

- **getMysqlDataSource**：首先启动mysql（本书使用5.4.3版本），其次登录mysql创建test数据库（“create database test;”），在进行测试前，请先下载并添加mysql-connector-java-5.1.10.jar到classpath；
- **{?= call FUNCTION_TEST(?)}**：可以使用{?= call …}形式调用自定义函数；
- **params**：无需使用SqlReturnResultSet提取结果集数据，而是使用SqlOutParameter来描述自定义函数返回值；

- **CallableStatementCreator：**同上个例子含义一样；
- **cstmt.registerOutParameter(1, Types.INTEGER)：**将OUT类型参数注册为JDBC类型Types.INTEGER，此处即返回值类型为Types.INTEGER。
- **outValues.get("result")：**获取结果，直接返回Integer类型，比hsqldb简单多了吧。

最后看一下如何如何调用存储过程：

java代码：

```
@Test
public void testCallableStatementCreator3() {
    final String callProcedureSql = "{call PROCEDURE_TEST(?, ?)}";
    List<SqlParameter> params = new ArrayList<SqlParameter>();
    params.add(new SqlInOutParameter("inOutName", Types.VARCHAR));
    params.add(new SqlOutParameter("outId", Types.INTEGER));
    Map<String, Object> outValues = jdbcTemplate.call(
      new CallableStatementCreator() {
        @Override
        public CallableStatement createCallableStatement(Connection conn) throws SQLExceptio
          CallableStatement cstmt = conn.prepareCall(callProcedureSql);
          cstmt.registerOutParameter(1, Types.VARCHAR);
          cstmt.registerOutParameter(2, Types.INTEGER);
          cstmt.setString(1, "test");
          return cstmt;
    }}, params);
    Assert.assertEquals("Hello,test", outValues.get("inOutName"));
    Assert.assertEquals(0, outValues.get("outId"));
}
```

- **{call PROCEDURE_TEST(?, ?)}：**定义存储过程sql；
- **params：**定义存储过程参数；SqlInOutParameter描述INOUT类型参数、SqlOutParameter描述OUT类型参数；
- CallableStatementCreator：用于创建CallableStatement，并设值及注册OUT参数类型；
- outValues：通过SqlInOutParameter及SqlOutParameter参数定义的name来获取存储过程结果。

JdbcTemplate类还提供了很多便利方法，在此就不一一介绍了，但这些方法是由规律可循的，第一种就是提供回调接口让用户决定做什么，第二种可以认为是便利方法（如queryForXXX），用于那些比较简单的操作。

7.2.4 NamedParameterJdbcTemplate

NamedParameterJdbcTemplate类是基于JdbcTemplate类，并对它进行了封装从而支持命名参数特性。

NamedParameterJdbcTemplate主要提供以下三类方法：execute方法、query及queryForXXX方法、update及batchUpdate方法。

首先让我们看个例子吧：

java代码：

```
@Test
public void testNamedParameterJdbcTemplate1() {
NamedParameterJdbcTemplate namedParameterJdbcTemplate = null;
//namedParameterJdbcTemplate =
//    new NamedParameterJdbcTemplate(dataSource);
namedParameterJdbcTemplate =
new NamedParameterJdbcTemplate(jdbcTemplate);
    String insertSql = "insert into test(name) values(:name)";
    String selectSql = "select * from test where name=:name";
    String deleteSql = "delete from test where name=:name";
    Map<String, Object> paramMap = new HashMap<String, Object>();
    paramMap.put("name", "name5");
    namedParameterJdbcTemplate.update(insertSql, paramMap);
    final List<Integer> result = new ArrayList<Integer>();
namedParameterJdbcTemplate.query(selectSql, paramMap,
new RowCallbackHandler() {
        @Override
        public void processRow(ResultSet rs) throws SQLException {
            result.add(rs.getInt("id"));
```

```
        }
    });
Assert.assertEquals(1, result.size());
SqlParameterSource paramSource = new MapSqlParameterSource(paramMap);
namedParameterJdbcTemplate.update(deleteSql, paramSource);
}
```

接下来让我们分析一下代码吧：

1）NamedParameterJdbcTemplate初始化：可以使用DataSource或JdbcTemplate 对象作为构造器参数初始化；

2）insert into test(name) values(:name)：其中“:name”就是命名参数；

3） update(insertSql, paramMap)：其中paramMap是一个Map类型，包含键为“name” ，值为“name5”的键值对，也就是为命名参数设值的数据；

4）query(selectSql, paramMap, new RowCallbackHandler()……)：类似于JdbcTemplate中介绍的，唯一不同是需要传入paramMap来为命名参数设值；

5）update(deleteSql, paramSource)：类似于“update(insertSql, paramMap)” ，但使用SqlParameterSource参数来为命名参数设值，此处使用MapSqlParameterSource实现，其就是简单封装java.util.Map。

NamedParameterJdbcTemplate类为命名参数设值有两种方式：java.util.Map和SqlParameterSource：

1）java.util.Map：使用Map键数据来对于命名参数，而Map值数据用于设值；

2）SqlParameterSource：可以使用SqlParameterSource实现作为来实现为命名参数设值，默认有MapSqlParameterSource和BeanPropertySqlParameterSource实现；MapSqlParameterSource实现非常简单，只是封装了java.util.Map；而BeanPropertySqlParameterSource封装了一个JavaBean对象，通过JavaBean对象属性来决定命名参数的值。

java代码：

```
package cn.javass.spring.chapter7;
public class UserModel {
    private int id;
    private String myName;
    //省略getter和setter
}
```

java代码：

```
@Test
public void testNamedParameterJdbcTemplate2() {
    NamedParameterJdbcTemplate namedParameterJdbcTemplate = null;
    namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(jdbcTemplate);
    UserModel model = new UserModel();
    model.setMyName("name5");
    String insertSql = "insert into test(name) values(:myName)";
    SqlParameterSource paramSource = new BeanPropertySqlParameterSource(model);
    namedParameterJdbcTemplate.update(insertSql, paramSource);
}
```

可以看出BeanPropertySqlParameterSource使用能减少很多工作量，但命名参数必须和JavaBean属性名称相对应才可以。

7.2.5 SimpleJdbcTemplate

SimpleJdbcTemplate类也是基于JdbcTemplate类，但利用Java5+的可变参数列表和自动装箱和拆箱从而获取更简洁的代码。

SimpleJdbcTemplate主要提供两类方法：query及queryForXXX方法、update及batchUpdate方法。

首先让我们看个例子吧：

java代码：

```
//定义UserModel的RowMapper
package cn.javass.spring.chapter7;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;
public class UserRowMapper implements RowMapper<UserModel> {
    @Override
    public UserModel mapRow(ResultSet rs, int rowNum) throws SQLException {
        UserModel model = new UserModel();
        model.setId(rs.getInt("id"));
        model.setMyName(rs.getString("name"));
        return model;
    }
}
```

java代码：

```
@Test
public void testSimpleJdbcTemplate() {
    //还支持DataSource和NamedParameterJdbcTemplate作为构造器参数
    SimpleJdbcTemplate simpleJdbcTemplate = new SimpleJdbcTemplate(jdbcTemplate);
    String insertSql = "insert into test(id, name) values(?, ?)";
    simpleJdbcTemplate.update(insertSql, 10, "name5");
    String selectSql = "select * from test where id=? and name=?";
    List<Map<String, Object>> result = simpleJdbcTemplate.queryForList(selectSql, 10, "name5
    Assert.assertEquals(1, result.size());
```

```
    RowMapper<UserModel> mapper = new UserRowMapper();
    List<UserModel> result2 = simpleJdbcTemplate.query(selectSql, mapper, 10, "name5");
    Assert.assertEquals(1, result2.size());
}
```

1）SimpleJdbcTemplate初始化：可以使用DataSource、JdbcTemplate或NamedParameterJdbcTemplate对象作为构造器参数初始化；

2）update(insertSql, 10, "name5")：采用Java5+可变参数列表从而代替new Object[]{10, "name5"}方式；

3）query(selectSql, mapper, 10, "name5")：使用Java5+可变参数列表及RowMapper回调并利用泛型特性来指定返回值类型（List<UserModel>）。

SimpleJdbcTemplate类还支持命名参数特性，如queryForList(String sql, SqlParameterSource args)和queryForList(String sql, Map<String, ?> args) ，类似于NamedParameterJdbcTemplate中使用，在此就不介绍了。

注：SimpleJdbcTemplate还提供类似于ParameterizedRowMapper 用于泛型特性的支持，ParameterizedRowMapper是RowMapper的子类，但从Spring 3.0由于允许环境需要Java5+，因此不再需要ParameterizedRowMapper ，而可以直接使用RowMapper；

~~**query(String sql, ParameterizedRowMapper<T> rm, SqlParameterSource args)**~~

query(String sql, RowMapper<T> rm, Object... args) //直接使用该语句

SimpleJdbcTemplate还提供如下方法用于获取JdbcTemplate和NamedParameterJdbcTemplate：

1）获取JdbcTemplate对象方法：JdbcOperations是JdbcTemplate的接口

JdbcOperations getJdbcOperations()

2）获取NamedParameterJdbcTemplate对象方法：NamedParameterJdbcOperations是NamedParameterJdbcTemplate的接口

NamedParameterJdbcOperations getNamedParameterJdbcOperations()

1.31 SpringMVC + spring3.1.1 + hibernate4.1.0 集成及常见问题总结

发表时间: 2012-02-26 关键字: spring, ssh, hibernate, 企业应用

下载地址

一 开发环境

1、动态web工程

2、部分依赖

java代码：

```
hibernate-release-4.1.0.Final.zip
hibernate-validator-4.2.0.Final.jar
spring-framework-3.1.1.RELEASE-with-docs.zip
proxool-0.9.1.jar
log4j 1.2.16
slf4j -1.6.1
mysql-connector-java-5.1.10.jar
hamcrest 1.3.0RC2
ehcache 2.4.3
```

3、为了方便学习，暂没有使用maven构建工程

二 工程主要包括内容

1、springMVC + spring3.1.1 + hibernate4.1.0集成

2、通用DAO层 和 Service层

3、二级缓存 Ehcache

4、REST风格的表现层

5、通用分页（两个版本）

5.1、首页 上一页,下一页 尾页 跳转

5.2、上一页 1 2 3 4 5 下一页

6、数据库连接池采用proxool

7、spring集成测试

8、表现层的 java validator框架验证（采用hibernate-validator-4.2.0实现）

9、视图采用JSP，并进行组件化分离

三 TODO LIST 将本项目做成脚手架方便以后新项目查询

1、Service层进行AOP缓存（缓存使用Memcached实现）

2、单元测试（把常见的桩测试、伪实现、模拟对象演示一遍 区别集成测试）

3、监控功能

后台查询hibernate二级缓存 hit/miss率功能

后台查询当前服务器状态功能（如 线程信息、服务器相关信息）

4、spring RPC功能

5、spring集成 quartz 进行任务调度

6、spring集成 java mail进行邮件发送

7、DAO层将各种常用框架集成进来（方便查询）

8、把工作中经常用的东西 融合进去，作为脚手架，方便以后查询

四 集成重点及常见问题

1、spring-config.xml 配置文件：

1.1、该配置文件只加载除表现层之外的所有bean，因此需要如下配置：

java代码：

```
    <context:component-scan base-package="cn.javass">
        <context:exclude-filter type="annotation" expression="org.springframework.stereotype
    </context:component-scan>
```

通过exclude-filter 把所有 @Controller注解的表现层控制器组件排除

1.2、国际化消息文件配置

java代码：

```
<!-- 国际化的消息资源文件 -->
    <bean id="messageSource" class="org.springframework.context.support.ReloadableResourceBu
        <property name="basenames">
            <list>
                <!-- 在web环境中一定要定位到classpath 否则默认到当前web应用下找  -->
                <value>classpath:messages</value>
            </list>
        </property>
        <property name="defaultEncoding" value="UTF-8"/>
        <property name="cacheSeconds" value="60"/>
    </bean>
```

此处basenames内一定是 classpath:messages ，如果你写出“messages” ，将会到你的web应用的根下找 即你的messages.properties一定在 web应用/messages.propertis。

1.3、hibernate的sessionFactory配置 需要使用org.springframework.orm.hibernate4.LocalSessionFactoryBean，其他都是类似的，具体看源代码。

1．4、<aop:aspectj-autoproxy expose-proxy="true"/> 实现@AspectJ注解的，默认使用AnnotationAwareAspectJAutoProxyCreator进行AOP代理，它是BeanPostProcessor的子类，在容器启动时Bean初始化开始和结束时调用进行AOP代理的创建，因此只对当容器启动时有效，使用时注意此处。

1.5、声明式容器管理事务

建议使用声明式容器管理事务，而不建议使用注解容器管理事务（虽然简单），但太分布式了，采用声明式容器管理事务一般只对service层进行处理。

java代码：

```
<tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
        <tx:method name="save*" propagation="REQUIRED" />
        <tx:method name="add*" propagation="REQUIRED" />
        <tx:method name="create*" propagation="REQUIRED" />
        <tx:method name="insert*" propagation="REQUIRED" />
        <tx:method name="update*" propagation="REQUIRED" />
        <tx:method name="merge*" propagation="REQUIRED" />
        <tx:method name="del*" propagation="REQUIRED" />
        <tx:method name="remove*" propagation="REQUIRED" />
        <tx:method name="put*" propagation="REQUIRED" />
        <tx:method name="use*" propagation="REQUIRED"/>
        <!--hibernate4必须配置为开启事务 否则 getCurrentSession()获取不到-->
        <tx:method name="get*" propagation="REQUIRED" read-only="true" />
        <tx:method name="count*" propagation="REQUIRED" read-only="true" />
        <tx:method name="find*" propagation="REQUIRED" read-only="true" />
        <tx:method name="list*" propagation="REQUIRED" read-only="true" />
        <tx:method name="*" read-only="true" />
    </tx:attributes>
</tx:advice>
<aop:config expose-proxy="true">
```

```
        <!-- 只对业务逻辑层实施事务 -->
        <aop:pointcut id="txPointcut" expression="execution(* cn.javass..service..*.*(..))" 
        <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut"/>
    </aop:config>
```

此处一定注意 使用 hibernate4，在不使用OpenSessionInView模式时，在使用getCurrentSession()时会有如下问题：

当有一个方法list 传播行为为Supports，当在另一个方法getPage()（无事务）调用list方法时会抛出org.hibernate.HibernateException: No Session found for current thread 异常。

这是因为getCurrentSession()在没有session的情况下不会自动创建一个，不知道这是不是Spring3.1实现的bug，欢迎大家讨论下。

因此最好的解决方案是使用REQUIRED的传播行为。

二、spring-servlet.xml：

2.1、表现层配置文件，只应加装表现层Bean，否则可能引起问题。

java代码：

```
    <!-- 开启controller注解支持 -->
    <!-- 注：如果base-package=cn.javass 则注解事务不起作用-->
    <context:component-scan base-package="cn.javass.demo.web.controller">
        <context:include-filter type="annotation" expression="org.springframework.stereotype
    </context:component-scan>
```

此处只应该加载表现层组件，如果此处还加载dao层或service层的bean会将之前容器加载的替换掉，而且此处不会进行AOP织入，所以会造成AOP失效问题（如事务不起作用），再回头看我们的1.4讨论的。

2.2、<mvc:view-controller path="/" view-name="forward:/index"/> 表示当访问主页时自动转发到index控制器。

2.3、静态资源映射

java代码：

```
<!-- 当在web.xml 中   DispatcherServlet使用     <url-pattern>/</url-pattern> 映射时，能映射
<mvc:default-servlet-handler/>
<!-- 静态资源映射 -->
<mvc:resources mapping="/images/**" location="/WEB-INF/images/" />
<mvc:resources mapping="/css/**" location="/WEB-INF/css/" />
<mvc:resources mapping="/js/**" location="/WEB-INF/js/" />
```

以上是配置文件部分，接下来来看具体代码。

三、通用DAO层Hibernate4实现

为了减少各模块实现的代码量，实际工作时都会有通用DAO层实现，以下是部分核心代码：

java代码：

```
public abstract class BaseHibernateDao<M extends java.io.Serializable, PK extends java.io.Se

    protected static final Logger LOGGER = LoggerFactory.getLogger(BaseHibernateDao.class);

    private final Class<M> entityClass;
    private final String HQL_LIST_ALL;
    private final String HQL_COUNT_ALL;
    private final String HQL_OPTIMIZE_PRE_LIST_ALL;
    private final String HQL_OPTIMIZE_NEXT_LIST_ALL;
    private String pkName = null;

    @SuppressWarnings("unchecked")
    public BaseHibernateDao() {
        this.entityClass = (Class<M>) ((ParameterizedType) getClass().getGenericSuperclass()
        Field[] fields = this.entityClass.getDeclaredFields();
        for(Field f : fields) {
            if(f.isAnnotationPresent(Id.class)) {
                this.pkName = f.getName();
            }
        }

        Assert.notNull(pkName);
        //TODO @Entity name not null
        HQL_LIST_ALL = "from " + this.entityClass.getSimpleName() + " order by " + pkName +
        HQL_OPTIMIZE_PRE_LIST_ALL = "from " + this.entityClass.getSimpleName() + " where " +
        HQL_OPTIMIZE_NEXT_LIST_ALL = "from " + this.entityClass.getSimpleName() + " where "
        HQL_COUNT_ALL = " select count(*) from " + this.entityClass.getSimpleName();
    }

    @Autowired
    @Qualifier("sessionFactory")
    private SessionFactory sessionFactory;

    public Session getSession() {
        //事务必须是开启的，否则获取不到
        return sessionFactory.getCurrentSession();
```

```
    }
……
}
```

Spring3.1集成Hibernate4不再需要HibernateDaoSupport和HibernateTemplate了，直接使用原生API即可。

四、通用Service层代码 此处省略，看源代码，有了通用代码后CURD就不用再写了。

java代码：

```
@Service("UserService")
public class UserServiceImpl extends BaseService<UserModel, Integer> implements UserService

    private static final Logger LOGGER = LoggerFactory.getLogger(UserServiceImpl.class);

    private UserDao userDao;

    @Autowired
    @Qualifier("UserDao")
    @Override
    public void setBaseDao(IBaseDao<UserModel, Integer> userDao) {
        this.baseDao = userDao;
        this.userDao = (UserDao) userDao;
    }



    @Override
    public Page<UserModel> query(int pn, int pageSize, UserQueryModel command) {
        return PageUtil.getPage(userDao.countQuery(command) ,pn, userDao.query(pn, pageSize,
    }
```

```
}
```

五、表现层 Controller实现

采用SpringMVC支持的REST风格实现，具体看代码，此处我们使用了java Validator框架 来进行 表现层数据验证

在Model实现上加验证注解

java代码：

```
    @Pattern(regexp = "[A-Za-z0-9]{5,20}", message = "{username.illegal}") //java validator验
    private String username;

    @NotEmpty(message = "{email.illegal}")
    @Email(message = "{email.illegal}") //错误消息会自动到MessageSource中查找
    private String email;

    @Pattern(regexp = "[A-Za-z0-9]{5,20}", message = "{password.illegal}")
    private String password;

    @DateFormat( message="{register.date.error}")//自定义的验证器
    private Date registerDate;
```

在Controller中相应方法的需要验证的参数上加@Valid即可

java代码：

```
    @RequestMapping(value = "/user/add", method = {RequestMethod.POST})
    public String add(Model model, @ModelAttribute("command") @Valid UserModel command, Bind
```

六、Spring集成测试

使用Spring集成测试能很方便的进行Bean的测试，而且使用@TransactionConfiguration(transactionManager = "txManager", defaultRollback = true)能自动回滚事务，清理测试前后状态。

java代码：

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {"classpath:spring-config.xml"})
@Transactional
@TransactionConfiguration(transactionManager = "txManager", defaultRollback = true)
public class UserServiceTest {

    AtomicInteger counter = new AtomicInteger();

    @Autowired
    private UserService userService;
    ……
}
```

其他部分请直接看源码，欢迎大家讨论。

1.32 【第七章】 对JDBC的支持 之 7.3 关系数据库操作对象化 ——跟我学spring3

发表时间: 2012-02-27 关键字: spring, jdbc

7.3.1 概述

所谓**关系数据库对象化其实就是用面向对象方式表示关系数据库操作，从而可以复用。**

Spring JDBC框架将数据库操作封装为一个RdbmsOperation，该对象是线程安全的、可复用的对象，是所有数据库对象的父类。而SqlOperation继承了RdbmsOperation，代表了数据库SQL操作，如select、update、call等，如图7-4所示。

图7-4 关系数据库操作对象化支持类

数据库操作对象化只要有以下几种类型，所以类型是线程安全及可复用的：

- **查询：**将数据库操作select封装为对象，查询操作的基类是SqlQuery，所有查询都可以使用该类表示，Spring JDBC还提供了一些更容易使用的MappingSqlQueryWithParameters和MappingSqlQuery用于将结果集映射为Java对象，查询对象类还提供了两个扩展UpdatableSqlQuery和SqlFunction；
- **更新：**即增删改操作，将数据库操作insert 、update、delete封装为对象，增删改基类是SqlUpdate，当然还提供了BatchSqlUpdate用于批处理；
- **存储过程及函数：**将存储过程及函数调用封装为对象，基类是SqlCall类，提供了StoredProcedure实现。

7.3.2 查询

1）SqlQuery：需要覆盖如下方法来定义一个RowMapper，其中parameters参数表示命名参数或占位符参数值列表，而context是由用户传入的上下文数据。

java代码：

```
RowMapper<T> newRowMapper(Object[] parameters, Map context)
```

SqlQuery提供两类方法：

- execute及executeByNamedParam方法：用于查询多行数据，其中executeByNamedParam用于支持命名参数绑定参数；
- findObject及findObjectByNamedParam方法：用于查询单行数据，其中findObjectByNamedParam用于支持命名参数绑定。

演示一下SqlQuery如何使用：

java代码：

```
@Test
public void testSqlQuery() {
    SqlQuery query = new UserModelSqlQuery(jdbcTemplate);
    List<UserModel> result = query.execute("name5");
    Assert.assertEquals(0, result.size());
}
```

从测试代码可以SqlQuery使用非常简单，创建SqlQuery实现对象，然后调用相应的方法即可，接下来看一下SqlQuery实现：

java代码：

```
package cn.javass.spring.chapter7;
//省略import
public class UserModelSqlQuery extends SqlQuery<UserModel> {
    public UserModelSqlQuery(JdbcTemplate jdbcTemplate) {
```

```
        //super.setDataSource(jdbcTemplate.getDataSource());
        super.setJdbcTemplate(jdbcTemplate);
        super.setSql("select * from test where name=?");
        super.declareParameter(new SqlParameter(Types.VARCHAR));
        compile();
    }
    @Override
    protected RowMapper<UserModel> newRowMapper(Object[] parameters, Map context) {
        return new UserRowMapper();
    }
}
```

从测试代码可以看出，具体步骤如下：

一、setJdbcTemplate/ setDataSource：首先设置数据源或JdbcTemplate；

二、setSql("select * from test where name=?")：定义sql语句，所以定义的sql语句都将被编译为PreparedStatement；

三、declareParameter(new SqlParameter(Types.VARCHAR))：对PreparedStatement参数描述，使用SqlParameter来描述参数类型，支持命名参数、占位符描述；

对于命名参数可以使用如new SqlParameter("name", Types.VARCHAR)描述；注意占位符参数描述必须按占位符参数列表的顺序进行描述；

四、编译：可选，当执行相应查询方法时会自动编译，用于将sql编译为PreparedStatement，对于编译的SqlQuery不能再对参数进行描述了。

五、以上步骤是不可变的，必须按顺序执行。

2）MappingSqlQuery：用于简化SqlQuery中RowMapper创建，可以直接在实现mapRow(ResultSet rs, int rowNum)来将行数据映射为需要的形式；

MappingSqlQuery所有查询方法完全继承于SqlQuery。

演示一下MappingSqlQuery如何使用：

java代码：

```
@Test
public void testMappingSqlQuery() {
    jdbcTemplate.update("insert into test(name) values('name5')");
    SqlQuery<UserModel> query = new UserModelMappingSqlQuery(jdbcTemplate);
    Map<String, Object> paramMap = new HashMap<String, Object>();
    paramMap.put("name", "name5");
    UserModel result = query.findObjectByNamedParam(paramMap);
    Assert.assertNotNull(result);
}
```

MappingSqlQuery使用和SqlQuery完全一样，创建MappingSqlQuery实现对象，然后调用相应的方法即可，接下来看一下MappingSqlQuery实现，findObjectByNamedParam方法用于执行命名参数查询：

java代码：

```
package cn.javass.spring.chapter7;
//省略import
public class UserModelMappingSqlQuery extends MappingSqlQuery<UserModel> {
  public UserModelMappingSqlQuery(JdbcTemplate jdbcTemplate) {
    super.setDataSource(jdbcTemplate.getDataSource());
    super.setSql("select * from test where name=:name");
    super.declareParameter(new SqlParameter("name", Types.VARCHAR));
    compile();
  }
  @Override
  protected UserModel mapRow(ResultSet rs, int rowNum) throws SQLException {
    UserModel model = new UserModel();
    model.setId(rs.getInt("id"));
```

```
    model.setMyName(rs.getString("name"));
    return model;
  }
}
```

和SqlQuery唯一不同的是使用mapRow来讲每行数据转换为需要的形式，其他地方完全一样。

1） **UpdatableSqlQuery**：提供可更新结果集查询支持，子类实现updateRow(ResultSet rs, int rowNum, Map context)对结果集进行更新。

2） **GenericSqlQuery**：提供setRowMapperClass(Class rowMapperClass)方法用于指定RowMapper实现，在此就不演示了。具体请参考testGenericSqlQuery()方法。

3） **SqlFunction**：SQL“函数”包装器，用于支持那些返回单行结果集的查询。该类主要用于返回单行单列结果集。

java代码：

```
@Test
public void testSqlFunction() {
    jdbcTemplate.update("insert into test(name) values('name5')");
    String countSql = "select count(*) from test";
    SqlFunction<Integer> sqlFunction1 = new SqlFunction<Integer>(jdbcTemplate.getDataSource(
    Assert.assertEquals(1, sqlFunction1.run());
    String selectSql = "select name from test where name=?";
    SqlFunction<String> sqlFunction2 = new SqlFunction<String>(jdbcTemplate.getDataSource(),
    sqlFunction2.declareParameter(new SqlParameter(Types.VARCHAR));
    String name = (String) sqlFunction2.runGeneric(new Object[] {"name5"});
    Assert.assertEquals("name5", name);
}
```

如代码所示，SqlFunction初始化时需要DataSource和相应的sql语句，如果有参数需要使用declareParameter对参数类型进行描述；run方法默认返回int型，当然也可以使用runGeneric返回其他类型，如String等。

7.3.3 更新

SqlUpdate类用于支持数据库更新操作，即增删改（insert、delete、update）操作，该方法类似于SqlQuery，只是职责不一样。

SqlUpdate提供了update及updateByNamedParam方法用于数据库更新操作，其中updateByNamedParam用于命名参数类型更新。

演示一下SqlUpdate如何使用：

java代码：

```
package cn.javass.spring.chapter7;
//省略import
public class InsertUserModel extends SqlUpdate {
    public InsertUserModel(JdbcTemplate jdbcTemplate) {
        super.setJdbcTemplate(jdbcTemplate);
        super.setSql("insert into test(name) values(?)");
        super.declareParameter(new SqlParameter(Types.VARCHAR));
        compile();
    }
}
```

java代码：

```
@Test
public void testSqlUpdate() {
    SqlUpdate insert = new InsertUserModel(jdbcTemplate);
    insert.update("name5");
```

```
    String updateSql = "update test set name=? where name=?";
    SqlUpdate update = new SqlUpdate(jdbcTemplate.getDataSource(), updateSql, new int[]{Type
    update.update("name6", "name5");
    String deleteSql = "delete from test where name=:name";

    SqlUpdate delete = new SqlUpdate(jdbcTemplate.getDataSource(), deleteSql, new int[]{Type
    Map<String, Object> paramMap = new HashMap<String, Object>();
    paramMap.put("name", "name5");
    delete.updateByNamedParam(paramMap);
}
```

InsertUserModel类实现类似于SqlQuery实现，用于执行数据库插入操作，SqlUpdate还提供一种更简洁的构造器SqlUpdate(DataSource ds, String sql, int[] types)，其中types用于指定占位符或命名参数类型；SqlUpdate还支持命名参数，使用updateByNamedParam方法来进行命名参数操作。

7.3.4 存储过程及函数

StoredProcedure用于支持存储过程及函数，该类的使用同样类似于SqlQuery。

StoredProcedure提供execute方法用于执行存储过程及函数。

一、StoredProcedure如何调用自定义函数：

java代码：

```
@Test
public void testStoredProcedure1() {
    StoredProcedure lengthFunction = new HsqldbLengthFunction(jdbcTemplate);
    Map<String,Object> outValues = lengthFunction.execute("test");
    Assert.assertEquals(4, outValues.get("result"));
}
```

StoredProcedure使用非常简单，定义StoredProcedure实现HsqldbLengthFunction，并调用execute方法执行即可，接下来看一下HsqldbLengthFunction实现：

java代码：

```
package cn.javass.spring.chapter7;
//省略import
public class HsqldbLengthFunction extends StoredProcedure {
    public HsqldbLengthFunction(JdbcTemplate jdbcTemplate) {
        super.setJdbcTemplate(jdbcTemplate);
        super.setSql("FUNCTION_TEST");
        super.declareParameter(
          new SqlReturnResultSet("result", new ResultSetExtractor<Integer>() {
            @Override
            public Integer extractData(ResultSet rs) throws SQLException, DataAccessExceptio
                while(rs.next()) {
                    return rs.getInt(1);
                }
                return 0;
            }
        }));
        super.declareParameter(new SqlParameter("str", Types.VARCHAR));
        compile();
    }
}
```

StoredProcedure自定义函数使用类似于SqlQuery，首先设置数据源或JdbcTemplate对象，其次定义自定义函数，然后使用declareParameter进行参数描述，最后调用compile（可选）编译自定义函数。

接下来看一下mysql自定义函数如何使用：

java代码：

```
@Test
public void testStoredProcedure2() {
    JdbcTemplate mysqlJdbcTemplate = new JdbcTemplate(getMysqlDataSource());
    String createFunctionSql =
        "CREATE FUNCTION FUNCTION_TEST(str VARCHAR(100)) " +
        "returns INT return LENGTH(str)";
    String dropFunctionSql = "DROP FUNCTION IF EXISTS FUNCTION_TEST";
    mysqlJdbcTemplate.update(dropFunctionSql);
    mysqlJdbcTemplate.update(createFunctionSql);
    StoredProcedure lengthFunction = new MysqlLengthFunction(mysqlJdbcTemplate);
    Map<String,Object> outValues = lengthFunction.execute("test");
    Assert.assertEquals(4, outValues.get("result"));
}
```

MysqlLengthFunction自定义函数使用与HsqldbLengthFunction使用完全一样，只是内部实现稍有差别：

java代码：

```
package cn.javass.spring.chapter7;
//省略import
public class MysqlLengthFunction extends StoredProcedure {
    public MysqlLengthFunction(JdbcTemplate jdbcTemplate) {
        super.setJdbcTemplate(jdbcTemplate);
        super.setSql("FUNCTION_TEST");
        super.setFunction(true);
        super.declareParameter(new SqlOutParameter("result", Types.INTEGER));
        super.declareParameter(new SqlParameter("str", Types.VARCHAR));
        compile();
    }
}
```

MysqlLengthFunction与HsqldbLengthFunction实现不同的地方有两点：

- **setFunction(true)**：表示是自定义函数调用，即编译后的sql为{?= call ...}形式；如果使用hsqldb不能设置为true，因为在hsqldb中{?= call ...}和{call ...}含义一样；
- **declareParameter(new SqlOutParameter("result", Types.INTEGER))**：将自定义函数返回值类型直接描述为Types.INTEGER；SqlOutParameter必须指定name，而不用使用SqlReturnResultSet首先获取结果集，然后再从结果集获取返回值，这是mysql与hsqldb的区别；

一、StoredProcedure如何调用存储过程：

java代码：

```
@Test
public void testStoredProcedure3() {
    StoredProcedure procedure = new HsqldbTestProcedure(jdbcTemplate);
    Map<String,Object> outValues = procedure.execute("test");
    Assert.assertEquals(0, outValues.get("outId"));
    Assert.assertEquals("Hello,test", outValues.get("inOutName"));
}
```

StoredProcedure存储过程实现HsqldbTestProcedure调用与HsqldbLengthFunction调用完全一样，不同的是在实现时，参数描述稍有不同：

java代码：

```
package cn.javass.spring.chapter7;
//省略import
public class HsqldbTestProcedure extends StoredProcedure {
    public HsqldbTestProcedure(JdbcTemplate jdbcTemplate) {
        super.setJdbcTemplate(jdbcTemplate);
```

```
        super.setSql("PROCEDURE_TEST");
        super.declareParameter(new SqlInOutParameter("inOutName", Types.VARCHAR));
        super.declareParameter(new SqlOutParameter("outId", Types.INTEGER));
        compile();
    }
}
```

- **declareParameter**：使用SqlInOutParameter描述INOUT类型参数，使用SqlOutParameter描述OUT类型参数，必须按顺序定义，不能颠倒。

原创内容，转载请注明出处【http://sishuok.com/forum/blogPost/list/0/2491.html】

1.33 【第七章】 对JDBC的支持 之 7.4 Spring提供的其它帮助 ——跟我学spring3【私塾在线原创】

发表时间: 2012-02-28 关键字: spring, jdbc

7.4 Spring提供的其它帮助

7.4.1 SimpleJdbc方式

Spring JDBC抽象框架提供SimpleJdbcInsert和SimpleJdbcCall类，这两个类通过利用JDBC驱动提供的数据库元数据来简化JDBC操作。

1、SimpleJdbcInsert： 用于插入数据，根据数据库元数据进行插入数据，本类用于简化插入操作，提供三种类型方法：execute方法用于普通插入、executeAndReturnKey及executeAndReturnKeyHolder方法用于插入时获取主键值、executeBatch方法用于批处理。

java代码：

```
@Test
public void testSimpleJdbcInsert() {
    SimpleJdbcInsert insert = new SimpleJdbcInsert(jdbcTemplate);
    insert.withTableName("test");
    Map<String, Object> args = new HashMap<String, Object>();
    args.put("name", "name5");
    insert.compile();
    //1.普通插入
    insert.execute(args);
    Assert.assertEquals(1, jdbcTemplate.queryForInt("select count(*) from test"));
    //2.插入时获取主键值
    insert = new SimpleJdbcInsert(jdbcTemplate);
    insert.withTableName("test");
    insert.setGeneratedKeyName("id");
    Number id = insert.executeAndReturnKey(args);
    Assert.assertEquals(1, id);
    //3.批处理
```

```
    insert = new SimpleJdbcInsert(jdbcTemplate);
    insert.withTableName("test");
    insert.setGeneratedKeyName("id");
    int[] updateCount = insert.executeBatch(new Map[] {args, args, args});
    Assert.assertEquals(1, updateCount[0]);
    Assert.assertEquals(5, jdbcTemplate.queryForInt("select count(*) from test"));
}
```

- **new SimpleJdbcInsert(jdbcTemplate) ：** 首次通过DataSource对象或JdbcTemplate对象初始化SimpleJdbcInsert；
- **insert.withTableName("test") ：** 用于设置数据库表名；
- **args ：** 用于指定插入时列名及值，如本例中只有name列名，即编译后的sql类似于“insert into test(name) values(?)” ；
- **insert.compile() ：** 可选的编译步骤，在调用执行方法时自动编译，编译后不能再对insert对象修改；
- **执行：** execute方法用于执行普通插入；executeAndReturnKey用于执行并获取自动生成主键（注意是Number类型），必须首先通过setGeneratedKeyName设置主键然后才能获取，如果想获取复合主键请使用setGeneratedKeyNames描述主键然后通过executeReturningKeyHolder获取复合主键KeyHolder对象；executeBatch用于批处理；

2、SimpleJdbcCall： 用于调用存储过程及自定义函数，本类用于简化存储过程及自定义函数调用。

java代码：

```
@Test
public void testSimpleJdbcCall1() {
    //此处用mysql,因为hsqldb调用自定义函数和存储过程一样
    SimpleJdbcCall call = new SimpleJdbcCall(getMysqlDataSource());
    call.withFunctionName("FUNCTION_TEST");
    call.declareParameters(new SqlOutParameter("result", Types.INTEGER));
    call.declareParameters(new SqlParameter("str", Types.VARCHAR));
    Map<String, Object> outVlaues = call.execute("test");
    Assert.assertEquals(4, outVlaues.get("result"));
}
```

- **new SimpleJdbcCall(getMysqlDataSource())** ：通过DataSource对象或JdbcTemplate对象初始化SimpleJdbcCall；
- **withFunctionName("FUNCTION_TEST")** ： 定义自定义函数名；自定义函数sql语句将被编译为类似于{?= call ...}形式；
- **declareParameters** ： 描述参数类型，使用方式与StoredProcedure对象一样；
- **执行**： 调用execute方法执行自定义函数；

java代码：

```
@Test
public void testSimpleJdbcCall2() {
    //调用hsqldb自定义函数得使用如下方式
    SimpleJdbcCall call = new SimpleJdbcCall(jdbcTemplate);
    call.withProcedureName("FUNCTION_TEST");
call.declareParameters(new SqlReturnResultSet("result",
new ResultSetExtractor<Integer>() {
        @Override
        public Integer extractData(ResultSet rs)
throws SQLException, DataAccessException {
          while(rs.next()) {
            return rs.getInt(1);
          }
          return 0;
    }}));
    call.declareParameters(new SqlParameter("str", Types.VARCHAR));
    Map<String, Object> outVlaues = call.execute("test");
    Assert.assertEquals(4, outVlaues.get("result"));
}
```

调用hsqldb数据库自定义函数与调用mysql自定义函数完全不同，详见StoredProcedure中的解释。

java代码：

```
@Test
public void testSimpleJdbcCall3() {
  SimpleJdbcCall call = new SimpleJdbcCall(jdbcTemplate);
  call.withProcedureName("PROCEDURE_TEST");
  call.declareParameters(new SqlInOutParameter("inOutName", Types.VARCHAR));
  call.declareParameters(new SqlOutParameter("outId", Types.INTEGER));
  SqlParameterSource params =
  new MapSqlParameterSource().addValue("inOutName", "test");
  Map<String, Object> outVlaues = call.execute(params);
  Assert.assertEquals("Hello,test", outVlaues.get("inOutName"));
  Assert.assertEquals(0, outVlaues.get("outId"));
}
```

与自定义函数调用不同的是使用withProcedureName来指定存储过程名字；其他参数描述等完全一样。

7.4.2 控制数据库连接

Spring JDBC通过DataSource控制数据库连接，即通过DataSource实现获取数据库连接。

Spring JDBC提供了一下DataSource实现：

- **DriverManagerDataSource** ：简单封装了DriverManager获取数据库连接；通过DriverManager的getConnection方法获取数据库连接；
- **SingleConnectionDataSource** ：内部封装了一个连接，该连接使用后不会关闭，且不能在多线程环境中使用，一般用于测试；
- **LazyConnectionDataSourceProxy** ：包装一个DataSource，用于延迟获取数据库连接，只有在真正创建Statement等时才获取连接，因此再说实际项目中最后使用该代理包装原始DataSource从而使得只有在真正需要连接时才去获取。

第三方提供的DataSource实现主要有C3P0、Proxool、DBCP等，这些实现都具有数据库连接池能力。

DataSourceUtils： Spring JDBC抽象框架内部都是通过它的getConnection(DataSource dataSource)方法获取数据库连接，releaseConnection(Connection con, DataSource dataSource) 用于释放数据库连接，DataSourceUtils用于支持Spring管理事务，只有使用DataSourceUtils获取的连接才具有Spring管理事务。

7.4.3 获取自动生成的主键

有许多数据库提供自动生成主键的能力，因此我们可能需要获取这些自动生成的主键，JDBC 3.0标准支持获取自动生成的主键，且必须数据库支持自动生成键获取。

1 ）JdbcTemplate 获取自动生成主键方式：

java代码：

```
@Test
public void testFetchKey1() throws SQLException {
    final String insertSql = "insert into test(name) values('name5')";
    KeyHolder generatedKeyHolder = new GeneratedKeyHolder();
    jdbcTemplate.update(new PreparedStatementCreator() {
        @Override
       public PreparedStatement createPreparedStatement(Connection conn)
            throws SQLException {
            return conn.prepareStatement(insertSql, new String[]{"ID"});
      }}, generatedKeyHolder);
    Assert.assertEquals(0, generatedKeyHolder.getKey());
}
```

使用JdbcTemplate的update(final PreparedStatementCreator psc, final KeyHolder generatedKeyHolder)方法执行需要返回自动生成主键的插入语句，其中psc用于创建PreparedStatement并指定自动生成键，如“prepareStatement(insertSql, new String[]{"ID"})” ；generatedKeyHolder是KeyHolder类型，用于获取自动生成的主键或复合主键；如使用getKey方法获取自动生成的主键。

2 ）SqlUpdate 获取自动生成主键方式：

java代码：

```
@Test
public void testFetchKey2() {
    final String insertSql = "insert into test(name) values('name5')";
    KeyHolder generatedKeyHolder = new GeneratedKeyHolder();
    SqlUpdate update = new SqlUpdate();
    update.setJdbcTemplate(jdbcTemplate);
    update.setReturnGeneratedKeys(true);
    //update.setGeneratedKeysColumnNames(new String[]{"ID"});
    update.setSql(insertSql);
    update.update(null, generatedKeyHolder);
    Assert.assertEquals(0, generatedKeyHolder.getKey());
}
```

SqlUpdate获取自动生成主键方式和JdbcTemplate完全一样，可以使用setReturnGeneratedKeys（true）表示要获取自动生成键；也可以使用setGeneratedKeysColumnNames指定自动生成键列名。

3 ）SimpleJdbcInsert ： 前边示例已介绍，此处就不演示了。

7.4.4 JDBC批量操作

JDBC批处理用于减少与数据库交互的次数来提升性能，Spring JDBC抽象框架通过封装批处理操作来简化批处理操作

1 ）JdbcTemplate 批处理： 支持普通的批处理及占位符批处理；

java代码：

```
@Test
public void testBatchUpdate1() {
    String insertSql = "insert into test(name) values('name5')";
```

```
    String[] batchSql = new String[] {insertSql, insertSql};
    jdbcTemplate.batchUpdate(batchSql);
    Assert.assertEquals(2, jdbcTemplate.queryForInt("select count(*) from test"));
}
```

直接调用batchUpdate方法执行需要批处理的语句即可。

java代码：

```
@Test
public void testBatchUpdate2() {
    String insertSql = "insert into test(name) values(?)";
    final String[] batchValues = new String[] {"name5", "name6"};
    jdbcTemplate.batchUpdate(insertSql, new BatchPreparedStatementSetter() {
        @Override
        public void setValues(PreparedStatement ps, int i) throws SQLException {
            ps.setString(1, batchValues[i]);
        }
        @Override
        public int getBatchSize() {
            return batchValues.length;
        }
    });
    Assert.assertEquals(2, jdbcTemplate.queryForInt("select count(*) from test"));
}
```

JdbcTemplate还可以通过batchUpdate(String sql, final BatchPreparedStatementSetter pss)方法进行批处理，该方式使用预编译语句，然后通过BatchPreparedStatementSetter实现进行设值（setValues）及指定批处理大小（getBatchSize）。

2 ）NamedParameterJdbcTemplate 批处理： 支持命名参数批处理；

java代码：

```
@Test
public void testBatchUpdate3() {
    NamedParameterJdbcTemplate namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(j
    String insertSql = "insert into test(name) values(:myName)";
    UserModel model = new UserModel();
    model.setMyName("name5");
    SqlParameterSource[] params = SqlParameterSourceUtils.createBatch(new Object[] {model, m
    namedParameterJdbcTemplate.batchUpdate(insertSql, params);
    Assert.assertEquals(2, jdbcTemplate.queryForInt("select count(*) from test"));
}
```

通过batchUpdate(String sql, SqlParameterSource[] batchArgs)方法进行命名参数批处理，batchArgs指定批处理数据集。SqlParameterSourceUtils.createBatch用于根据JavaBean对象或者Map创建相应的BeanPropertySqlParameterSource或MapSqlParameterSource。

3) SimpleJdbcTemplate 批处理： 已更简单的方式进行批处理；

java代码：

```
@Test
public void testBatchUpdate4() {
    SimpleJdbcTemplate simpleJdbcTemplate = new SimpleJdbcTemplate(jdbcTemplate);
    String insertSql = "insert into test(name) values(?)";
    List<Object[]> params = new ArrayList<Object[]>();
```

```
    params.add(new Object[]{"name5"});
    params.add(new Object[]{"name5"});
    simpleJdbcTemplate.batchUpdate(insertSql, params);
    Assert.assertEquals(2, jdbcTemplate.queryForInt("select count(*) from test"));
}
```

本示例使用batchUpdate(String sql, List<Object[]> batchArgs)方法完成占位符批处理，当然也支持命名参数批处理等。

4 ）SimpleJdbcInsert 批处理：

java代码：

```
@Test
public void testBatchUpdate5() {
    SimpleJdbcInsert insert = new SimpleJdbcInsert(jdbcTemplate);
    insert.withTableName("test");
    Map<String, Object> valueMap = new HashMap<String, Object>();
    valueMap.put("name", "name5");
    insert.executeBatch(new Map[] {valueMap, valueMap});
   Assert.assertEquals(2, jdbcTemplate.queryForInt("select count(*) from test"));
}
```

如代码所示，使用executeBatch(Map<String, Object>[] batch)方法执行批处理。

1.34 【第七章】 对JDBC的支持 之 7.5 集成Spring JDBC及最佳实践 ——跟我学spring3

发表时间: 2012-02-29 关键字: spring, jdbc

7.5 集成Spring JDBC及最佳实践

大多数情况下Spring JDBC都是与IOC容器一起使用。通过配置方式使用Spring JDBC。

而且大部分时间都是使用JdbcTemplate类（或SimpleJdbcTemplate和NamedParameterJdbcTemplate）进行开发，即可能80%时间使用JdbcTemplate类，而只有20%时间使用其他类开发，符合**80/20法则。**

Spring JDBC通过实现DaoSupport来支持一致的数据库访问。

Spring JDBC提供如下DaoSupport实现：

- **JdbcDaoSupport：**用于支持一致的JdbcTemplate访问；
- **NamedParameterJdbcDaoSupport:**继承JdbcDaoSupport，同时提供NamedParameterJdbcTemplate访问；
- **SimpleJdbcDaoSupport：**继承JdbcDaoSupport，同时提供SimpleJdbcTemplate访问。

由于JdbcTemplate、NamedParameterJdbcTemplate、SimpleJdbcTemplate类使用DataSourceUtils获取及释放连接，而且连接是与线程绑定的，因此这些JDBC模板类是线程安全的，即JdbcTemplate对象可以在多线程中重用。

接下来看一下Spring JDBC框架的最佳实践：

1)首先定义Dao接口

java代码：

```
package cn.javass.spring.chapter7.dao;
import cn.javass.spring.chapter7.UserModel;
public interface IUserDao {
    public void save(UserModel model);
    public int countAll();
}
```

2）定义Dao实现，此处是使用Spring JDBC实现：

java代码：

```
package cn.javass.spring.chapter7.dao.jdbc;
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;
import cn.javass.spring.chapter7.UserModel;
import cn.javass.spring.chapter7.dao.IUserDao;
public class UserJdbcDaoImpl extends SimpleJdbcDaoSupport implements IUserDao {
  private static final String INSERT_SQL = "insert into test(name) values(:myName)";
  private static final String COUNT_ALL_SQL = "select count(*) from test";

  @Override
  public void save(UserModel model) {
      getSimpleJdbcTemplate().update(INSERT_SQL, new BeanPropertySqlParameterSource(model));
  }
  @Override
  public int countAll() {
      return getJdbcTemplate().queryForInt(COUNT_ALL_SQL);
  }
}
```

此处注意首先Spring JDBC实现放在dao.jdbc包里，如果有hibernate实现就放在dao.hibernate包里；其次实现类命名如UserJdbcDaoImpl，即×××JdbcDaoImpl，当然如果自己有更好的命名规范可以遵循自己的，此处只是提个建议。

3）进行资源配置（resources/chapter7/applicationContext-resources.xml）：

java代码：

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
      <list>
          <value>classpath:chapter7/resources.properties</value>
      </list>
    </property>
</bean>
```

PropertyPlaceholderConfigurer用于替换配置元数据，如本示例中将对bean定义中的${...}占位符资源用“classpath:chapter7/resources.properties”中相应的元素替换。

java代码：

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.LazyConnectionDataSourcePro
    <property name="targetDataSource">
      <bean class="org.logicalcobwebs.proxool.ProxoolDataSource">
          <property name="driver" value="${db.driver.class}" />
          <property name="driverUrl" value="${db.url}" />
          <property name="user" value="${db.username}" />
          <property name="password" value="${db.password}" />
          <property name="maximumConnectionCount"
                 value="${proxool.maxConnCount}" />
```

```
        <property name="minimumConnectionCount"
               value="${proxool.minConnCount}" />
        <property name="statistics" value="${proxool.statistics}" />
        <property name="simultaneousBuildThrottle"
                 value="${proxool.simultaneousBuildThrottle}" />
        <property name="trace" value="${proxool.trace}" />
      </bean>
    </property>
</bean>
```

dataSource定义数据源，本示例使用proxool数据库连接池，并使用LazyConnectionDataSourceProxy包装它，从而延迟获取数据库连接；${db.driver.class}将被“classpath:chapter7/resources.properties”中的“db.driver.class”元素属性值替换。

proxool数据库连接池：本示例使用proxool-0.9.1版本，请到proxool官网下载并添加proxool-0.9.1.jar和proxool-cglib.jar到类路径。

ProxoolDataSource属性含义如下：

- driver：指定数据库驱动；
- driverUrl：数据库连接；
- username：用户名；
- password：密码；
- maximumConnectionCount：连接池最大连接数量；
- minimumConnectionCount：连接池最小连接数量；
- statistics：连接池使用样本状况统计；如1m,15m,1h,1d表示没1分钟、15分钟、1小时及1天进行一次样本统计；
- simultaneousBuildThrottle：一次可以创建连接的最大数量；
- trace：true表示被执行的每个sql都将被记录（DEBUG级别时被打印到相应的日志文件）；

4）定义资源文件（classpath:chapter7/resources.properties）：

java代码：

```
proxool.maxConnCount=10
proxool.minConnCount=5
proxool.statistics=1m,15m,1h,1d
proxool.simultaneousBuildThrottle=30
proxool.trace=false
db.driver.class=org.hsqldb.jdbcDriver
db.url=jdbc:hsqldb:mem:test
db.username=sa
db.password=
```

用于替换配置元数据中相应的占位符数据，如${db.driver.class}将被替换为“org.hsqldb.jdbcDriver”。

5）dao定义配置（chapter7/applicationContext-jdbc.xml）：

java代码：

```
<bean id="abstractDao" abstract="true">
    <property name="dataSource" ref="dataSource"/>
</bean>
<bean id="userDao"
     class="cn.javass.spring.chapter7.dao.jdbc.UserJdbcDaoImpl"
    parent="abstractDao"/>
```

首先定义抽象的abstractDao，其有一个dataSource属性，从而可以让继承的子类自动继承dataSource属性注入；然后定义userDao，且继承abstractDao，从而继承dataSource注入；我们在此给配置文件命名为applicationContext-jdbc.xml表示Spring JDBC DAO实现；如果使用hibernate实现可以给配置文件命名为applicationContext-hibernate.xml。

6）最后测试一下吧（cn.javass.spring.chapter7. JdbcTemplateTest）：

java代码：

```
@Test
public void testBestPractice() {
    String[] configLocations = new String[] {
            "classpath:chapter7/applicationContext-resources.xml",
            "classpath:chapter7/applicationContext-jdbc.xml"};
    ApplicationContext ctx = new ClassPathXmlApplicationContext(configLocations);
    IUserDao userDao = ctx.getBean(IUserDao.class);
    UserModel model = new UserModel();
    model.setMyName("test");
    userDao.save(model);
     Assert.assertEquals(1, userDao.countAll());
}
```

首先读取配置文件，获取IUserDao接口实现，然后再调用IUserDao接口方法，进行数据库操作，这样对于开发人员使用来说，只面向接口，不关心实现，因此很容易更换实现，比如像更换为hibernate实现非常简单。

1.35 【第八章】 对ORM的支持 之 8.1 概述 ——跟我学spring3

发表时间: 2012-03-01 关键字: spring

8.1 概述

8.1.1 ORM框架

ORM全称对象关系映射（Object/Relation Mapping），指将Java对象状态自动映射到关系数据库中的数据上，从而提供透明化的持久化支持，即把一种形式转化为另一种形式。

对象与关系数据库之间是不匹配，我们把这种不匹配称为阻抗失配，主要表现在：

- 关系数据库首先不支持面向对象技术如继承、多态，如何使关系数据库支持它们；
- 关系数据库是由表来存放数据，而面向对象使用对象来存放状态；其中表的列称为属性，而对象的属性就是属性，因此需要通过解决这种不匹配；
- 如何将对象透明的持久化到关系数据库表中；
- 如果一个对象存在横跨多个表的数据，应该如何为对象建模和映射。

其中这些阻抗失配只是其中的一小部分，比如还有如何将SQL集合函数结果集映射到对象，如何在对象中处理主键等。

ORM框架就是用来解决这种阻抗失配，提供关系数据库的对象化支持。

ORM框架不是万能的，同样符合**80/20法则**，应解决的最核心问题是如何在关系数据库表中的行和对象进行映射，并自动持久化对象到关系数据库。

ORM解决方案适用于解决透明持久化、小结果集查询等；对于复杂查询，大结果集数据处理还是没有任何帮助的。

目前已经有许多ORM框架产生，如Hibernate、JDO、JPA、iBATIS等等，这些ORM框架各有特色，Spring对这些ORM框架提供了很好的支持，接下来首先让我们看一下Spring如何支持这些ORM框架。

8.1.2 Spring对ORM的支持

Spring对ORM的支持主要表现在以下方面：

- 一致的异常体系结构，对第三方ORM框架抛出的专有异常进行包装，从而在使我们在Spring中只看到DataAccessException异常体系；

- 一致的DAO抽象支持：提供类似与JdbcSupport的DAO支持类HibernateDaoSupport，使用HibernateTemplate模板类来简化常用操作，HibernateTemplate提供回调接口来支持复杂操作；
- Spring事务管理：Spring对所有数据访问提供一致的事务管理，通过配置方式，简化事务管理。

Spring还在测试、数据源管理方面提供支持，从而允许方便测试，简化数据源使用。

接下来让我们学习一下Spring如何集成ORM框架—Hibernate。

1.36 【第八章】 对ORM的支持 之 8.2 集成Hibernate3 ——跟我学spring3

发表时间: 2012-03-01 关键字: spring

8.2 集成Hibernate3

Hibernate是全自动的ORM框架，能自动为对象生成相应SQL并透明的持久化对象到数据库。

Spring2.5+版本支持Hibernate 3.1+版本，不支持低版本，Spring3.0.5版本提供对Hibernate 3.6.0 Final版本支持。

8.2.1 如何集成

Spring通过使用如下Bean进行集成Hibernate：

- LocalSessionFactoryBean ：用于支持XML映射定义读取：

configLocation和configLocations：用于定义Hibernate配置文件位置，一般使用如classpath:hibernate.cfg.xml形式指定；

mappingLocations ：用于指定Hibenate映射文件位置，如chapter8/hbm/user.hbm.xml；

hibernateProperties：用于定义Hibernate属性，即Hibernate配置文件中的属性；

dataSource：定义数据源；

hibernateProperties、dataSource用于消除Hibernate配置文件，因此如果使用configLocations指定配置文件，就不要设置这两个属性了，否则会产生重复配置。推荐使用dataSource来指定数据源，而使用hibernateProperties指定Hibernate属性。

- AnnotationSessionFactoryBean：用于支持注解风格映射定义读取，该类继承LocalSessionFactoryBean并额外提供自动查找注解风格配置模型的能力：

annotatedClasses：设置注解了模型类，通过注解指定映射元数据。

packagesToScan：通过扫描指定的包获取注解模型类，而不是手工指定，如“cn.javass.**.model”将扫描cn.javass包及子包下的model包下的所有注解模型类。

接下来学习一下Spring如何集成Hibernate吧：

1、准备jar包：

首先准备Spring对ORM框架支持的jar包：

org.springframework.orm-3.0.5.RELEASE.jar //提供对ORM框架集成

下载hibernate-distribution-3.6.0.Final包，获取如下Hibernate需要的jar包：

hibernate3.jar //核心包

lib\required\antlr-2.7.6.jar //HQL解析时使用的包

lib\required\javassist-3.9.0.GA.jar //字节码类库，类似于cglib

lib\required\commons-collections-3.1.jar //对集合类型支持包，前边测试时已经提供过了，无需再拷贝该包了

lib\required\dom4j-1.6.1.jar //xml解析包，用于解析配置使用

lib\required\jta-1.1.jar //JTA事务支持包

lib\jpa\hibernate-jpa-2.0-api-1.0.0.Final.jar //用于支持JPA

下载slf4j-1.6.1.zip（http://www.slf4j.org/download.html），slf4j是日志系统门面（Simple Logging Facade for Java），用于对各种日志框架提供给一致的日志访问接口，从而能随时替换日志框架（如log4j、java.util.logging）：

slf4j-api-1.6.1.jar //核心API

slf4j-log4j12-1.6.1.jar //log4j实现

将这些jar包添加到类路径中。

2、对象模型定义，此处使用第七章中的UserModel：

java代码：

```
package cn.javass.spring.chapter7;
public class UserModel {
    private int id;
    private String myName;
    //省略getter和setter
}
```

3、Hibernate映射定义（chapter8/hbm/user.hbm.xml），定义对象和数据库之间的映射：

java代码：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="cn.javass.spring.chapter7.UserModel" table="test">
        <id name="id" column="id"><generator class="native"/></id>
        <property name="myName" column="name"/>
    </class>
</hibernate-mapping>
```

4、 数据源定义，此处使用第7章的配置文件，即“chapter7/applicationContext-resources.xml”文件。

5、 SessionFactory配置定义（chapter8/applicationContext-hibernate.xml）：

java代码：

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean"
    <property name="dataSource" ref="dataSource"/> <!-- 指定数据源 -->
      <property name="mappingResources">     <!-- 指定映射定义 -->
        <list>
          <value>chapter8/hbm/user.hbm.xml</value>
        </list>
      </property>
      <property name="hibernateProperties">   <!--指定Hibernate属性 -->
        <props>
          <prop key="hibernate.dialect">
              org.hibernate.dialect.HSQLDialect
          </prop>
        </props>
      </property>
</bean>
```

6、 获取SessionFactory：

java代码：

```
package cn.javass.spring.chapter8;
//省略import
public class HibernateTest {
    private static SessionFactory sessionFactory;
    @BeforeClass
    public static void beforeClass() {
        String[] configLocations = new String[] {
```

```
                "classpath:chapter7/applicationContext-resources.xml",
                "classpath:chapter8/applicationContext-hibernate.xml"};
        ApplicationContext ctx = new ClassPathXmlApplicationContext(configLocations);
        sessionFactory = ctx.getBean("sessionFactory", SessionFactory.class);
}
}
```

此处我们使用了chapter7/applicationContext-resources.xml定义的“dataSource”数据源，通过ctx.getBean("sessionFactory", SessionFactory.class)获取SessionFactory。

7、通过SessionFactory获取Session对象进行创建和删除表：

java代码：

```
@Before
public void setUp() {
  //id自增主键从0开始
  final String createTableSql = "create memory table test" + "(id int GENERATED BY DEFAULT A
  sessionFactory.openSession().
  createSQLQuery(createTableSql).executeUpdate();
}
@After
public void tearDown() {
    final String dropTableSql = "drop table test";
    sessionFactory.openSession().
    createSQLQuery(dropTableSql).executeUpdate();
}
```

使用SessionFactory创建Session，然后通过Session对象的createSQLQuery创建本地SQL执行创建和删除表。

8、使用SessionFactory获取Session对象进行持久化数据：

java代码：

```
@Test
public void testFirst() {
    Session session = sessionFactory.openSession();
    Transaction transaction = null;
    try {
        transaction = beginTransaction(session);
        UserModel model = new UserModel();
        model.setMyName("myName");
        session.save(model);
    } catch (RuntimeException e) {
        rollbackTransaction(transaction);
        throw e;
    } finally {
        commitTransaction(session);
    }
}
```

java代码：

```
private Transaction beginTransaction(Session session) {
    Transaction transaction = session.beginTransaction();
    transaction.begin();
    return transaction;
}
private void rollbackTransaction(Transaction transaction) {
   if(transaction != null) {
        transaction.rollback();
    }
```

```
}
private void commitTransaction(Session session) {
    session.close();
}
```

使用SessionFactory获取Session进行操作，必须自己控制事务，而且还要保证各个步骤不会出错，有没有更好的解决方案把我们从编程事务中解脱出来？Spring提供了HibernateTemplate模板类用来简化事务处理和常见操作。

8.2.2 使用HibernateTemplate

HibernateTimplate模板类用于简化事务管理及常见操作，类似于JdbcTemplate模板类，对于复杂操作通过提供HibernateCallback回调接口来允许更复杂的操作。

接下来示例一下HibernateTemplate的使用：

java代码：

```
@Test
public void testHibernateTemplate() {
HibernateTemplate hibernateTemplate =
new HibernateTemplate(sessionFactory);
    final UserModel model = new UserModel();
    model.setMyName("myName");
    hibernateTemplate.save(model);
    //通过回调允许更复杂操作
    hibernateTemplate.execute(new HibernateCallback<Void>() {
        @Override
        public Void doInHibernate(Session session)
```

```
            throws HibernateException, SQLException {
            session.save(model);
            return null;
        }});
}
```

通过new HibernateTemplate(sessionFactory) 创建HibernateTemplate模板类对象，通过调用模板类的save方法持久化对象，并且自动享受到Spring管理事务的好处。

而且HibernateTemplate 提供使用HibernateCallback回调接口的方法execute用来支持复杂操作，当然也自动享受到Spring管理事务的好处。

8.2.3 集成Hibernate及最佳实践

类似于JdbcDaoSupport类，Spring对Hibernate也提供了HibernateDaoSupport类来支持一致的数据库访问。HibernateDaoSupport也是DaoSupport实现：

接下来示例一下Spring集成Hibernate的最佳实践：

1、 定义Dao接口，此处使用cn.javass.spring.chapter7.dao. IUserDao：

2、 定义Dao接口实现，此处是Hibernate实现：

java代码：

```
package cn.javass.spring.chapter8.dao.hibernate;
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;
import cn.javass.spring.chapter7.UserModel;
import cn.javass.spring.chapter7.dao.IUserDao;
public class UserHibernateDaoImpl extends HibernateDaoSupport implements IUserDao {
```

```
    private static final String COUNT_ALL_HQL = "select count(*) from UserModel";
    @Override
    public void save(UserModel model) {
        getHibernateTemplate().save(model);
    }
    @Override
    public int countAll() {
        Number count = (Number) getHibernateTemplate().find(COUNT_ALL_HQL).get(0);
        return count.intValue();
    }
}
```

此处注意首先Hibernate实现放在dao.hibernate包里，其次实现类命名如UserHibernateDaoImpl，即×××HibernateDaoImpl，当然如果自己有更好的命名规范可以遵循自己的，此处只是提个建议。

3、进行资源配置，使用resources/chapter7/applicationContext-resources.xml：

4、dao定义配置，在chapter8/applicationContext-hibernate.xml中添加如下配置：

java代码：

```
<bean id="abstractDao" abstract="true">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
<bean id="userDao"  class="cn.javass.spring.chapter8.dao.hibernate.UserHibernateDaoImpl" par
```

首先定义抽象的abstractDao，其有一个sessionFactory属性，从而可以让继承的子类自动继承sessionFactory属性注入；然后定义userDao，且继承abstractDao，从而继承sessionFactory注入；我们在此给配置文件命名为applicationContext-hibernate.xml表示Hibernate实现。

5、 最后测试一下吧（cn.javass.spring.chapter8. HibernateTest）：

java代码：

```
@Test
public void testBestPractice() {
    String[] configLocations = new String[] {
            "classpath:chapter7/applicationContext-resources.xml",
            "classpath:chapter8/applicationContext-hibernate.xml"};
    ApplicationContext ctx = new ClassPathXmlApplicationContext(configLocations);
    IUserDao userDao = ctx.getBean(IUserDao.class);
    UserModel model = new UserModel();
    model.setMyName("test");
    userDao.save(model);
    Assert.assertEquals(1, userDao.countAll());
}
```

和Spring JDBC框架的最佳实践完全一样，除了使用applicationContext-hibernate.xml代替了applicationContext-jdbc.xml，其他完全一样。也就是说，DAO层的实现替换可以透明化。

8.2.4 Spring+Hibernate的CRUD

Spring+Hibernate CRUD（增删改查）我们使用注解类来示例，让我们看具体示例吧：

1、首先定义带注解的模型对象UserModel2：

- 使用JPA注解@Table指定表名映射；

- 使用注解@Id指定主键映射；
- 使用注解@ Column指定数据库列映射；

java代码：

```
package cn.javass.spring.chapter8;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
@Entity
@Table(name = "test")
public class UserModel2 {
    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    @Column(name = "name")
    private String myName;
    //省略getter和setter
}
```

2、 定义配置文件（chapter8/applicationContext-hibernate2.xml）：

2.1、 定义SessionFactory：

此处使用AnnotationSessionFactoryBean通过annotatedClasses属性指定注解模型来定义映射元数据；

java代码：

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.annotation.AnnotationSes
<property name="dataSource" ref="dataSource"/>  <!-- 1、指定数据源 -->
```

```
  <property name="annotatedClasses">           <!-- 2、指定注解类 -->
     <list><value>cn.javass.spring.chapter8.UserModel2</value></list>
</property>
  <property name="hibernateProperties"><!-- 3、指定Hibernate属性 -->
    <props>
     <prop key="hibernate.dialect">
        org.hibernate.dialect.HSQLDialect
     </prop>
    </props>
  </property>
</bean>
```

2.2、定义HibernateTemplate ：

java代码：

```
<bean id="hibernateTemplate" class="org.springframework.orm.hibernate3.HibernateTemplate">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

3、最后进行CURD测试吧：

java代码：

```
@Test
public void testCURD() {
    String[] configLocations = new String[] {
            "classpath:chapter7/applicationContext-resources.xml",
            "classpath:chapter8/applicationContext-hibernate2.xml"};
    ApplicationContext ctx =  new ClassPathXmlApplicationContext(configLocations);
```

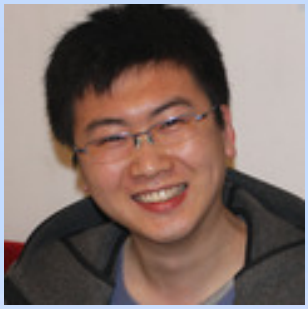
```
    HibernateTemplate hibernateTemplate =  ctx.getBean(HibernateTemplate.class);
    UserModel2 model = new UserModel2();
    model.setMyName("test");
    insert(hibernateTemplate, model);
    select(hibernateTemplate, model);
    update(hibernateTemplate, model);
    delete(hibernateTemplate, model);
}

private void insert(HibernateTemplate hibernateTemplate, UserModel2 model) {
    hibernateTemplate.save(model);
}
private void select(HibernateTemplate hibernateTemplate, UserModel2 model) {
    UserModel2 model2 = hibernateTemplate.get(UserModel2.class, 0);
    Assert.assertEquals(model2.getMyName(), model.getMyName());
    List<UserModel2> list = hibernateTemplate.find("from UserModel2");
    Assert.assertEquals(list.get(0).getMyName(), model.getMyName());
}
private void update(HibernateTemplate hibernateTemplate, UserModel2 model) {
    model.setMyName("test2");
    hibernateTemplate.update(model);
}
private void delete(HibernateTemplate hibernateTemplate, UserModel2 model) {
    hibernateTemplate.delete(model);
}
```

Spring集成Hibernate进行增删改查是不是比Spring JDBC方式简单许多，而且支持注解方式配置映射元数据，从而减少映射定义配置文件数量。

私塾在线原创内容 转载请注明出处【http://sishuok.com/forum/blogPost/list/0/2497.html】

私塾在线原创 http://sishuok.com



跟我学spring3(1-7)

作者: jinnianshilongnian
http://jinnianshilongnian.iteye.com