1. **什么是控制反转IoC**？

将类中重复代码的编写，比如安全的控制，交给外围框架而不是对象本身，控制反转

控制反转使类编写时，去除了重复代码，并简化了类，使我们把工作重点放在业务逻辑上

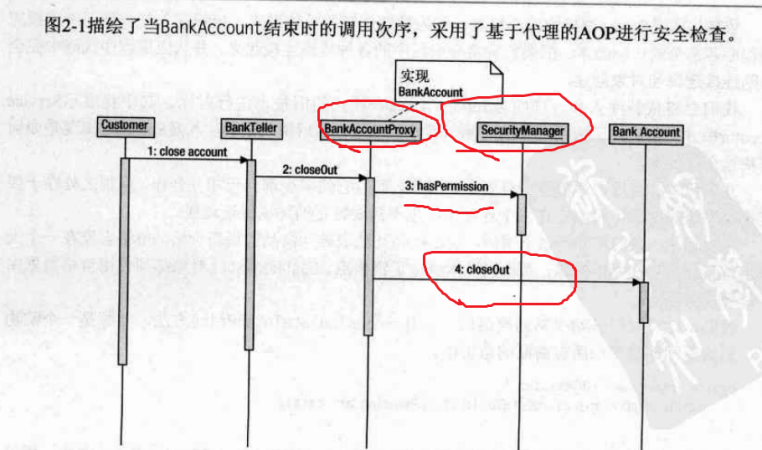
1. **什么是面向方面编程AOP**？

AOP是一类IoC的实现框架

AOP的框架分为两种：

一种在运行时插入代码（SpringAOP，基于代理的AOP，代理包装了目标对象），

一种在编译时织入代码（AspectJ，Spring也支持，比运行时高效）



1. **什么是依赖注入Dependency Injection**？

依赖注入是IoC的一个例子

Spring框架不使用ServiceLocator的查找调用（例如ServiceLocator.getPriceMartix()），而是将PriceMartix类型的**引用**提供给CashRegisterImpl类，这就使得控制反转了

**Spring使用两种方式进行依赖注入**：

第一种基于**构造器**注入，依赖在对象创建时由构造器提供

public CashRegisterImpl(PriceMartix pm) { this.pm = pm; } 接收框架传入的**依赖**引用

基于构造器注入的优势：利用构造器的自我验证来强制正确地初始化对象

劣势：提供不同用例时，可能导致构造器泛滥；因为随着用例增加，各个用例需要不同的依赖，构造器也将增加

第二种基于**setter方法**注入，使用适合JavaBean的setter注入代替构造器注入

public void setPriceMartix(PriceMartic pm) { this.pm = pm; }

基于setter注入的优势：不会在类一开始加载时就进行注入，使其生命周期更灵活，也符合**延迟加载**的策略

其他依赖注入方式：

基于**接口**注入，已退役

注入的工具：XML、注解

1. **Spring HelloWorld实例**

本教程介绍如何在Spring3 中创建一个简单的 Hello World 例子。

在这篇文章中使用的技术或工具：

1. Spring 3.1
2. MyEclipse 10
3. JDK 1.6

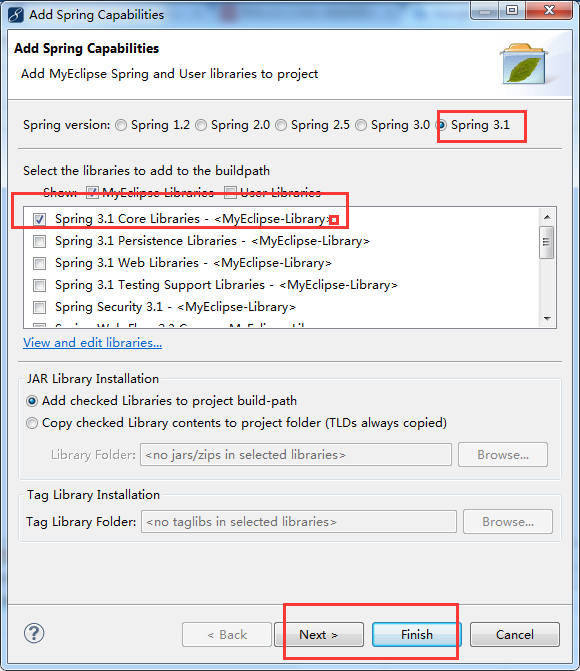
提示: Spring 3 至少JDK1.5才能正常工作。

Spring 3.0 dependencies

在Spring2.5.x中，几乎整个Spring模块分组在一个单独的 spring.jar 文件中。而Spring3中每模块被分成一个单独的 jar 文件，例如，spring-core, spring-expression, spring-context, spring-aop等

**1. 创建一个Java工程**

打开 MyEclipse 创建一个java工程：Helloword，并添加Spring支持类库，右键工程名称，选择"MyEclipse"->"Add Spring Capabilites"，如下图：



**2. Spring bean**

一个简单的 Spring bean. 无需序列化ID

package com.yiibai.core;

/\*\*

\* Spring bean

\*

\*/

public class HelloWorld {

private String name;

public void setName(String name) {

this.name = name;

}

public void printHello() {

System.out.println("Spring 3 : Hello ! " + name);

}

}

**4. Spring bean 配置文件**

创建Spring配置文件，并声明所有可用的Spring bean。

*File :*applicationContext*.xml*

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

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

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

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="helloBean" class="com.yiibai.core.HelloWorld">

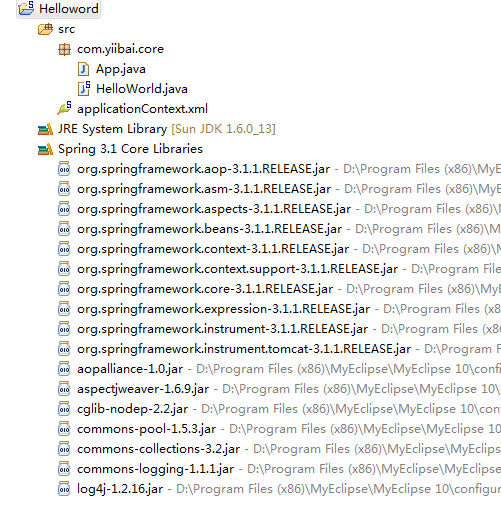
<property name="name" value="Yiibai" /> 这里指定注入的属性

</bean>

</beans>

**5. 项目结构**

查看目录结构如下:



1. **执行代码**

**通过一个**ApplicationContext类载入Spring的配置文件，通过getBean传入bean的别名，就可以进行实例化

package com.yiibai.core;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext(

"applicationContext.xml"); HelloWorld obj = (HelloWorld) context.getBean("helloBean");

obj.printHello();

}

}

**7. 输出结果**

Spring 3 : Hello ! Yiibai

**5．Spring松耦合示例**

面向对象的概念，是一个很好的设计来打破系统进入一个组可重用的对象。然而，当系统变大，尤其是在Java项目，庞大的对象依赖关系将一直紧密耦合引起对象难以管理或修改。在这种情况下，可以使用Spring框架作为一个核心模块轻松高效地**管理所有的对象依赖**。

**输出生成器的例子**

让我们来看一个例子，假设你的项目有一个函数输出的内容，以CSV或JSON格式。您的代码可能看起来像下面的例子：

File : IOutputGenerator.java – 输出生成器接口

package com.yiibai.output;

public interface IOutputGenerator

{

public void generateOutput();

}

File : CsvOutputGenerator.java – 一个CSV输出生成器用来实现IOutputGenerator接口。

package com.yiibai.output.impl;

import com.yiibai.output.IOutputGenerator;

public class CsvOutputGenerator implements IOutputGenerator

{

public void generateOutput(){

System.out.println("Csv Output Generator");

}

}

File : JsonOutputGenerator.java – 一个JSON输出生成器用来实现IOutputGenerator接口。

package com.yiibai.output.impl;

import com.yiibai.output.IOutputGenerator;

public class JsonOutputGenerator implements IOutputGenerator

{

public void generateOutput(){

System.out.println("Json Output Generator");

}

}

有几个方法来调用IOutputGenerator，以及如何使用 Spring 来避免对象相互结合紧密。

保证了多态复用的同时又解耦合

**1. 方法1 – 直接调用**

正常的方式，直接调用它。

package com.yiibai.common;

import com.yiibai.output.IOutputGenerator;

import com.yiibai.output.impl.CsvOutputGenerator;

public class App

{

public static void main( String[] args )

{

IOutputGenerator output = new CsvOutputGenerator();

output.generateOutput();

}

}

存在问题

以这种方式，这个问题是“output”紧密到CsvOutputGenerator耦合，输出生成的每一个变化可能涉及代码变化。如果此代码分散在你的项目中，输出生成的每一次变化都会让你受苦。

**方法2 – 用辅助类调用它**

可能想创建一个辅助类将所有输出实现在类的内部。

package com.yiibai.output;

import com.yiibai.output.IOutputGenerator;

import com.yiibai.output.impl.CsvOutputGenerator;

public class OutputHelper

{

IOutputGenerator outputGenerator;

public OutputHelper(){

outputGenerator = new CsvOutputGenerator();

}

public void generateOutput(){

outputGenerator.generateOutput();

}

}

通过辅助类调用它

package com.yiibai.common;

import com.yiibai.output.OutputHelper;

public class App

{

public static void main( String[] args )

{

OutputHelper output = new OutputHelper();

output.generateOutput();

}

}

存在问题

这看起来比之前的更优雅，只需要管理一个辅助类，但是辅助类仍是紧耦合 CsvOutputGenerator，输出生成的每一个变化仍然涉及小的代码更改。

**方法3 – Spring**

在这种情况下，Spring 依赖注入(DI)是一个不错的选择。Spring 可以让输出生成松散的耦合到输出发生器。

OutputHelper类更小的修改。

package com.yiibai.output;

import com.yiibai.output.IOutputGenerator;

public class OutputHelper

{

IOutputGenerator outputGenerator;

//辅助调用

public void generateOutput(){

outputGenerator.generateOutput();

}

public void setOutputGenerator(IOutputGenerator outputGenerator){

this.outputGenerator = outputGenerator;

}

}

创建一个 Spring bean 的配置文件，并在这里声明所有的Java对象的依赖。

<!-- **Spring-Common.xml** --> //自定义的名字

<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">

<bean id="OutputHelper" class="com.yiibai.output.OutputHelper">

<property name="outputGenerator" ref="CsvOutputGenerator" /> //ref指向下面的bean id对应的class（注意这里的class被ApplicationContext读取会进行注入）

</bean>

<bean id="CsvOutputGenerator" class="com.yiibai.output.impl.CsvOutputGenerator" />

<bean id="JsonOutputGenerator" class="com.yiibai.output.impl.JsonOutputGenerator" />

</beans>

**通过Spring来调用它**

package com.yiibai.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.yiibai.output.OutputHelper;

public class App

{

public static void main( String[] args )

{

ApplicationContext context =

//注意这个ApplicationContext的实现类

new **ClassPathXmlApplicationContext**(new String[] {"**Spring-Common.xml**"});

OutputHelper output = (OutputHelper)context.getBean("OutputHelper");

output.generateOutput();

}

}

现在，只需要改变 Spring XML 文件使用不同的输出生成器。只修改 Spring XML 文件而不需要无码修改，这意味着更少的错误。

**结论**

有了Spring框架 - 这种依赖注入(DI)为对象的依赖关系管理有用的特性，使大型Java项目开发管理中更优雅的，高度灵活和便于维护。

6．Spring结合JDBC示例

Spring使用JNDI来获取数据库连接，即使用DataSource

**1. Customer 表**

在这个例子中，我们使用的是MySQL数据库。

CREATE TABLE `customer` (

`CUST\_ID` int(10) unsigned NOT NULL AUTO\_INCREMENT,

`NAME` varchar(100) NOT NULL,

`AGE` int(10) unsigned NOT NULL,

PRIMARY KEY (`CUST\_ID`)

) ENGINE=InnoDB AUTO\_INCREMENT=1 DEFAULT CHARSET=utf8;

**2. Customer模型**

添加一个客户模型用来存储用户的数据。

package com.yiibai.customer.model;

import java.sql.Timestamp;

public class Customer

{

int custId;

String name;

int age;

//getter and setter methods

}

**3. 数据访问对象 (DAO) 模式**

Customer Dao 接口.

package com.yiibai.customer.dao;

import com.yiibai.customer.model.Customer;

public interface CustomerDAO

{

public void insert(Customer customer);

public Customer findByCustomerId(int custId);

}

客户的DAO实现，使用 JDBC 发出简单的 insert 和 select SQL语句。

package com.yiibai.customer.dao.impl;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import javax.sql.DataSource;

import com.yiibai.customer.dao.CustomerDAO;

import com.yiibai.customer.model.Customer;

public class JdbcCustomerDAO implements CustomerDAO

{

private DataSource dataSource;

public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

}

public void insert(Customer customer){

String sql = "INSERT INTO CUSTOMER " +

"(CUST\_ID, NAME, AGE) VALUES (?, ?, ?)";

Connection conn = null;

try {

conn = dataSource.getConnection();

PreparedStatement ps = conn.prepareStatement(sql);

ps.setInt(1, customer.getCustId());

ps.setString(2, customer.getName());

ps.setInt(3, customer.getAge());

ps.executeUpdate();

ps.close();

} catch (SQLException e) {

throw new RuntimeException(e);

} finally {

if (conn != null) {

try {

conn.close();

} catch (SQLException e) {}

}

}

}

public Customer findByCustomerId(int custId){

String sql = "SELECT \* FROM CUSTOMER WHERE CUST\_ID = ?";

Connection conn = null;

try {

conn = dataSource.getConnection();

PreparedStatement ps = conn.prepareStatement(sql);

ps.setInt(1, custId);

Customer customer = null;

ResultSet rs = ps.executeQuery();

if (rs.next()) {

customer = new Customer(

rs.getInt("CUST\_ID"),

rs.getString("NAME"),

rs.getInt("Age")

);

}

rs.close();

ps.close();

return customer;

} catch (SQLException e) {

throw new RuntimeException(e);

} finally {

if (conn != null) {

try {

conn.close();

} catch (SQLException e) {}

}

}

}

}

**4. Spring bean配置**

创建 customerDAO 和数据源在 Spring bean 配置文件中。 让框架来注入DataSource到customerDAO中

*File :* ***Spring-Customer.xml***

<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">

<bean id="customerDAO" class="com.yiibai.customer.dao.impl.JdbcCustomerDAO">

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

**//ref指向的是存在同一XML中的bean id，这里要知道Spring-Customer.xml和Spring-Datasource.xml都导入到Spring-Module.xml中才会进行引用**

**如果把ref写成子节点的形式，就要给ref子节点多加一个bean属性：<ref bean=”dataSource” />**

</bean>

</beans>

*File :* ***Spring-Datasource.xml***

<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">

<bean id="dataSource"

class="org.springframework.jdbc.datasource.DriverManagerDataSource"> //找到这个类

<property name="driverClassName" value="com.mysql.jdbc.Driver" /> //进行数据库配置注入

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

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

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

</bean>

</beans>

*File :* applicationContext***.xml* 这个XML用来导入我们自定义的Spring配置XML，引入其中的Bean，名字无限定，主要是要**ApplicationContext对象创建时引入相关XML文件

<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

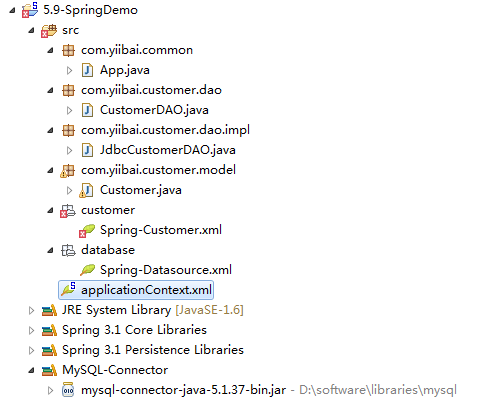
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

<**import resource**="database/Spring-Datasource.xml" /> //相对地址

<**import resource**="customer/Spring-Customer.xml" />

</beans>

**6.项目结构**

本实例完整目录结构。  


**7.运行它**

package com.yiibai.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.yiibai.customer.dao.CustomerDAO;

import com.yiibai.customer.model.Customer;

public class App

{

public static void main( String[] args )

{

ApplicationContext context =

new ClassPathXmlApplicationContext("**applicationContext**.xml");

CustomerDAO customerDAO = (CustomerDAO) context.getBean("customerDAO");

Customer customer = new Customer(1, "yiibai",29);

customerDAO.insert(customer);

Customer customer1 = customerDAO.findByCustomerId(1);

System.out.println(customer1);

}

}

输出结果：

Customer [age=29, custId=1, name=yiibai]

7．Spring JDBC Template 和 JdbcDaoSupport

在Spring JDBC开发中，可以使用 JdbcTemplate 和 JdbcDaoSupport 类来简化整个数据库的操作过程。

在本教程中，我们将重复上一篇文章[Spring+JDBC例子](http://www.yiibai.com/spring/maven-spring-jdbc-example.html)，看之前(无JdbcTemplate支持)和之后(含JdbcTemplate的支持)之间不同的例子。

**1. 不使用JdbcTemplate示例**

如果不用JdbcTemplate，必须创建大量的冗余代码(创建连接，关闭连接，处理异常)中的所有DAO数据库的操作方法 - 插入，更新和删除。它的效率并不是很高，容易出错和乏味。

private DataSource dataSource;

public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

}

public void insert(Customer customer){

String sql = "INSERT INTO CUSTOMER " +

"(CUST\_ID, NAME, AGE) VALUES (?, ?, ?)";

Connection conn = null;

try {

conn = **dataSource.getConnection()**;

PreparedStatement ps = conn.prepareStatement(sql);

ps.setInt(1, customer.getCustId());

ps.setString(2, customer.getName());

ps.setInt(3, customer.getAge());

ps.executeUpdate();

ps.close();

} catch (SQLException e) {

throw new RuntimeException(e);

} finally {

if (conn != null) {

try {

conn.close();

} catch (SQLException e) {}

}

}

}

**2. 使用JdbcTemplate示例**

使用JdbcTemplate可节省大量的冗余代码，因为JdbcTemplate类会自动处理它。

private DataSource dataSource;

private JdbcTemplate jdbcTemplate;

public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

}

public void insert(Customer customer){

String sql = "INSERT INTO CUSTOMER " +

"(CUST\_ID, NAME, AGE) VALUES (?, ?, ?)";

jdbcTemplate = new **JdbcTemplate(dataSource)**;

jdbcTemplate.**update**(sql, new Object[] { customer.getCustId(),

customer.getName(),customer.getAge() });

}

看看有什么不同？

Srping的JDBC Template封装了连接管理、异常判断和传参方式

**3. 使用JdbcDaoSupport示例**

通过**扩展 JdbcDaoSupport**，设置数据源，并且 JdbcTemplate 在你的**类中不再是必需**的，**只需要正确的数据源注入JdbcCustomerDAO**。就**可以使用 getJdbcTemplate()方法得到 JdbcTemplate。** 这样就**延迟加载**了JdbcTemplate

//JdbcDaoSupport类实现了获取JdbcTemplate的方法getJdbcTemplate()，这使得我们可以延迟加载JdbcTemplate

public class JdbcCustomerDAO **extends JdbcDaoSupport** implements CustomerDAO

{

//no need to set datasource here

public void insert(Customer customer){

String sql = "INSERT INTO CUSTOMER " +

"(CUST\_ID, NAME, AGE) VALUES (?, ?, ?)";

**getJdbcTemplate()**.**update**(sql, new Object[] { customer.getCustId(),

customer.getName(),customer.getAge()

});

}

<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">

<bean id="dataSource"

class="org.springframework.jdbc.datasource.DriverManagerDataSource">

//**Spring框架实现了DataSource**

<property name="driverClassName" value="com.mysql.jdbc.Driver" />

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

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

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

</bean>

</beans>

<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">

<bean id="customerDAO" class="com.yiibai.customer.dao.impl.JdbcCustomerDAO">

<property name="dataSource" ref="dataSource" /> //引用bean注入

</bean>

</beans>

**注:**在Spring JDBC开发，它总是建议使用，总是建议使用 JdbcTemplate和JdbcDaoSupport，而不使用自己的JDBC编程代码。

8．Spring Jdbc Template的三种用法

一、首先配置JdbcTemplate；

要使用Jdbctemplate 对象来完成jdbc 操作。通常情况下，有三种种方式**得到JdbcTemplate 对象**。   
      第一种方式：我们可以在自己定义的DAO 实现类中**注入**一个**DataSource 引用**来完 成JdbcTemplate 的实例化。也就是它是从外部“注入” DataSource 到DAO 中，然后 自己实例化JdbcTemplate，然后将DataSource 设置到JdbcTemplate 对象中。这样把DataSource和JdbcTemplate耦合在DAO里面了。

      第二种方式： 在 Spring 的 IoC 容器中配置一个 JdbcTemplate 的 bean，将 DataSource 注入进来，然后再把JdbcTemplate 注入到自定义DAO 中。这样解耦了JDBCTemplate和DAO，但把DataSource耦合在JdbcTemplate里面了，根据逻辑二者应该低耦合

      第三种方式: Spring 提供了 org.springframework.jdbc.core.support.**JdbcDaoSupport** 类 ， 这 个 类 中 定 义 了 **JdbcTemplate 属性**，也定义了**DataSource 属性**，**当设置DataSource 属性的时候，会创 建jdbcTemplate 的实例**，符合**延迟加载**的原则。所以我们自己编写的DAO 只需要**继承JdbcDaoSupport 类**， 然后注入DataSource 即可。提倡采用第三种方法。

也就是说，配置方法有3种：

注入方法1、

1. **public** **class** UserServiceImpl **implements** UserService {
3. **private** JdbcTemplate jdbcTemplate;
5. **public** JdbcTemplate getJdbcTemplate() {
6. **return** jdbcTemplate;
7. }
9. //注入方法1
10. **public** **void** setJdbcTemplate(JdbcTemplate jdbcTemplate) {
11. **this**.jdbcTemplate = jdbcTemplate;
12. }
14. //其它方法这里省略……
15. }

  配置方法1：spring配置文件为：

1. **<bean** id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate"**>**
2. **<property** name = "dataSource" ref="dataSource"**>**
3. **</bean>**
4. **<bean** id="userService" class="com.hxzy.account.jdbcTemplate.UserServiceImpl"**>**
5. **<property** name="jdbcTemplate" ref="jdbcTemplate"**/>**
6. **</bean>**

注入方法2、

1. **public** **class** UserServiceImpl **implements** UserService {
3. **private** JdbcTemplate jdbcTemplate;
5. //注入方法2
6. **public** **void** setDataSource(DataSource dataSource) {
7. **this**.jdbcTemplate = **new** JdbcTemplate(dataSource);
8. }
10. //其它方法省略……
11. }

配置方法2：spring配置文件为：

1. **<bean** id="userService" class="com.hxzy.account.jdbcTemplate.UserServiceImpl"**>**
2. **<property** name="dataSource" ref="dataSource"**/>**
3. **</bean>**

注入方法3：**继承JdbcDaoSupport**，其内部有个JdbcTemplate 属性，调用getJdbcTemplate就可实例化，而且需要注入DataSource 属性来实例化。

1. **public** **class** UserDaoImpl **extends** JdbcDaoSupport **implements** UserDao {
3. @Override
4. **public** **void** save(User user) {
5. String sql = **null**;
6. **this**.getJdbcTemplate().update(sql);
7. }
8. //其它方法省略……
9. }

配置方法3：spring配置文件：

1. **<bean** id="userDao" class="com.hxzy.account.jdbcTemplate.UserDaoImpl"**>**
2. **<property** name="dataSource" ref="dataSource"**/>**
3. **</bean>**

**9.Spring的ApplicationContext的两种加载方式**

Spring为此提供了两种解决方案，

一种是基于**ContextLoaderListener**实现的（此方案只适用于Servlet2.4及以上规范的Servlet容器）。例如，

在**web.xml**中加入如下代码：

<!-- 指定Spring的配置文件，多个配置文件以逗号分隔 -->

<context-param>

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

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

</context-param>

<!-- 指定以Listener方式启动Spring容器 -->

<listener>

**<listener-class>**org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

另外一种方案则是基于**ContextLoaderServlet**实现的。

例如，

在**web.xml**中加入如下代码：

<!-- 指定Spring的配置文件，多个配置文件以逗号分隔 -->

<context-param>

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

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

</context-param>

<!-- 指定以Servlet方式启动Spring容器 -->

<servlet>

<servlet-name>context</servlet-name>

<servlet-class>org.springframework.web.context.ContextLoaderServlet</servlet-class>

<load-on-startup>1</load-on-startup> //最高优先级

</servlet>

从servlet容器启动时加载组件的顺序来看，Listener组件是优先于Servlet组件的。基于Servlet方式的加载方案主要是为了兼容Servlet2.3及以下规范的Servlet容器。以Tomcat为例，**Tomcat 5.x都已经支持Servlet2.4规范了，因此，基于Listener方式启动Spring容器是目前的主流选择**

**10.Spring JdbcTemplate查询示例**

这里有几个例子向您展示如何使用JdbcTemplate的query()方法来查询或从数据库提取数据。整个项目的目录结构如下：

**1.查询单行数据 RowMapper接口对象传入queryForObject**

这里有两种方法来查询或从数据库中提取单行记录，并将其转换成一个模型类。

**1.1 自定义RowMapper**

在一般情况下，它总是建议实现 **RowMapper 接口**来创建自定义的RowMapper，以满足您的需求。

package com.yiibai.customer.model;

import java.sql.ResultSet;

import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

public class CustomerRowMapper implements RowMapper

{

**//这里通过结果集来封装一个对象**

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

Customer customer = new Customer();

customer.setCustId(rs.getInt("CUST\_ID"));

customer.setName(rs.getString("NAME"));

customer.setAge(rs.getInt("AGE"));

return customer;

}

}

将RowMapper对象传递给 queryForObject()方法，返回的结果将调用自定义 mapRow()方法的值匹配到属性。

**queryForObject**

**参数**:**预编译sql语句**、**填入的?参数（可变的Object参数）**、**得到的结果集所要传入的bean**

**返回:封装后的bean**

public Customer findByCustomerId(int custId){

String **sql** = "SELECT \* FROM CUSTOMER WHERE CUST\_ID = ?";

Customer customer = (Customer)**getJdbcTemplate**().**queryForObject**(

**sql**, new Object[] { custId }, new **CustomerRowMapper**());

return customer;

}

**1.2 BeanPropertyRowMapper ，Spring中一个RowMapper接口的实现类**

在Spring2.5中，**有一个 RowMapper 实现，即**所谓“**BeanPropertyRowMapper**”，它可以通过匹配行的名字的列值映射到一个属性。只要确保这两个属性和列具有相同的名称，如属性“CUSTID'将匹配到列名为：”CUSTID'或下划线“CUST\_ID”。

public Customer findByCustomerId2(int custId){

String sql = "SELECT \* FROM CUSTOMER WHERE CUST\_ID = ?";

Customer customer = (Customer)getJdbcTemplate().queryForObject(

sql, new Object[] { custId },

new BeanPropertyRowMapper(Customer.class));

return customer;

}

**2，查询多行**

现在，查询或从数据库中提取多行，并且将它转换成一个列表。

**2.1手动映射它**

返回多行，RowMapper 不支持 queryForList()方法，需要手动映射它。

public List<Customer> findAll(){

String sql = "SELECT \* FROM CUSTOMER";

List<Customer> customers = new ArrayList<Customer>();

List<Map> rows = getJdbcTemplate().queryForList(sql);

for (Map row : rows) {

Customer customer = new Customer();

customer.setCustId((Long)(row.get("CUST\_ID")));

customer.setName((String)row.get("NAME"));

customer.setAge((Integer)row.get("AGE"));

customers.add(customer);

}

return customers;

}

**2.2 BeanPropertyRowMapper**

最简单的解决方案是使用 BeanPropertyRowMapper 类。

public List<Customer> findAll(){

String sql = "SELECT \* FROM CUSTOMER";

List<Customer> customers = getJdbcTemplate().query(sql,

new BeanPropertyRowMapper(Customer.class));

return customers;

}

**3.查询单值**

在这个例子中，展示了如何从数据库中查询或提取单个列值。

**3.1单列名**

它显示了如何查询单个列名作为字符串。

public String findCustomerNameById(int custId){

String sql = "SELECT NAME FROM CUSTOMER WHERE CUST\_ID = ?";

String name = (String)getJdbcTemplate().queryForObject(

sql, new Object[] { custId }, String.class);

return name;

}

**3.2、行总数**

它展示了如何从数据库中查询行的总数。

public int findTotalCustomer(){

String sql = "SELECT COUNT(\*) FROM CUSTOMER";

int total = getJdbcTemplate().queryForInt(sql);

return total;

}

运行它

package com.yiibai.common;

import java.util.ArrayList;

import java.util.List;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.yiibai.customer.dao.CustomerDAO;

import com.yiibai.customer.model.Customer;

public class JdbcTemplateApp

{

public static void main( String[] args )

{

ApplicationContext context =

new ClassPathXmlApplicationContext("Spring-Customer.xml");

CustomerDAO customerDAO = (CustomerDAO) context.getBean("customerDAO");

Customer customerA = customerDAO.findByCustomerId(1);

System.out.println("Customer A : " + customerA);

Customer customerB = customerDAO.findByCustomerId2(1);

System.out.println("Customer B : " + customerB);

List<Customer> customerAs = customerDAO.findAll();

for(Customer cust: customerAs){

System.out.println("Customer As : " + customerAs);

}

List<Customer> customerBs = customerDAO.findAll2();

for(Customer cust: customerBs){

System.out.println("Customer Bs : " + customerBs);

}

String customerName = customerDAO.findCustomerNameById(1);

System.out.println("Customer Name : " + customerName);

int total = customerDAO.findTotalCustomer();

System.out.println("Total : " + total);

}

}

**总结**

JdbcTemplate类，附带了很多有用的重载查询方法。它提醒参考现有的查询方法在创建自己的自定义查询方法之前，因为 Spring 已经做给你了。