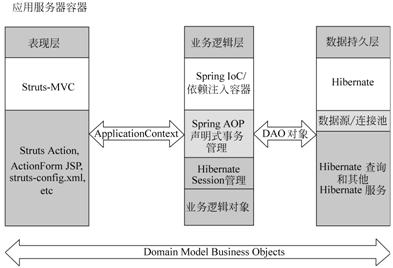
**6.5  Spring整合Hibernate**时至今日，可能极少有J2EE应用会直接以JDBC方式进行持久层访问。毕竟，用面向对象的程序设计语言来访问关系型数据库，是一件让人沮丧的事情。大部分时候，J2EE应用都会以ORM框架来进行持久层访问，在所有的ORM框架中，Hibernate以其灵巧、轻便的封装赢得了众多开发者的青睐。   
Spring具有良好的开放性，能与大部分ORM框架良好整合。下面将详细介绍Spring与Hibernate的整合。   
**6.5.1  Spring提供的DAO支持**DAO模式是一种标准的J2EE设计模式，DAO模式的核心思想是，所有的数据库访  问，都通过DAO组件完成，DAO组件封装了数据库的增、删、改等原子操作。而业务逻辑组件则依赖于DAO组件提供的数据库原子操作，完成系统业务逻辑的实现。   
对于J2EE应用的架构，有非常多的选择，但不管细节如何变换，J2EE应用都大致可分为如下3层：   
  ● 表现层。   
  ● 业务逻辑层。   
  ● 数据持久层。   
轻量级J2EE架构以Spring IoC容器为核心，承上启下。其向上管理来自表现层的Action，向下管理业务逻辑层组件，同时负责管理业务逻辑层所需的DAO对象。各层之间负责传值的是值对象，也就是JavaBean实例。   
图6.5精确地描绘了轻量级J2EE架构的大致情形。   
DAO组件是整个J2EE应用的持久层访问的重要组件，每个J2EE应用的底层实现都难以离开DAO组件的支持。Spring对实现DAO组件提供了许多工具类，系统的DAO组件可通过继承这些工具类完成，从而可以更加简便地实现DAO组件。   
Spring的DAO支持，允许使用相同的方式、不同的数据访问技术，如JDBC、Hibernate或JDO。Spring的DAO在不同的持久层访问技术上提供抽象，应用的持久层访问基于Spring的DAO抽象。因此，应用程序可以在不同的持久层技术之间切换。   
Spring提供了一系列的抽象类，这些抽象将被作为应用中DAO实现类的父类。通过继承这些抽象类，Spring简化了DAO的开发步骤，能以一致的方式使用数据库访问技术。不管底层采用JDBC、JDO或Hibernate，应用中都可采用一致的编程模型。



应用的DAO类继承这些抽象类，会大大简化应用的开发。最大的好处是，继承这些抽象类的DAO能以一致的方式访问数据库，意味着应用程序可以在不同的持久层访问技术中切换。   
除此之外，Spring提供了一致的异常抽象，将原有的Checked异常转换包装成Runtime异常，因而，编码时无须捕获各种技术中特定的异常。Spring DAO体系中的异常，都继承DataAccessException，而DataAccessException异常是Runtime的，无须显式捕捉。通过DataAccessException的子类包装原始异常信息，从而保证应用程序依然可以捕捉到原始异常信息。   
Spring提供了多种数据库访问技术的DAO支持，包括Hibernate、JDO、TopLink、iBatis、OJB等。Spring可以使用相同的访问模式、不同的数据库访问技术。就Hibernate的持久层访问技术而言，Spring提供了如下3个工具类（或接口）来支持DAO组件的实现：   
  ● HibernateDaoSupport。   
  ● HibernateTemplate。   
  ● HibernateCallBack。

**6.5.2  管理Hibernate的SessionFactory**前面介绍Hibernate时已经知道，在通过Hibernate进行持久层访问时，Hibernate的SessionFactory是一个非常重要的对象，它是单个数据库映射关系编译后的内存镜像。大部分情况下，一个J2EE应用对应一个数据库，也即对应一个SessionFactory对象。   
在纯粹的Hibernate访问中，应用程序需要手动创建SessionFactory实例，可想而知，这不是一个优秀的策略。在实际开发中，希望以一种声明式的方式管理SessionFactory实例，直接以配置文件来管理SessionFactory实例，在示范Struts的PlugIn扩展点时，大致示范了这种方式（请参阅2.12.1节的内容）。   
Spring的IoC容器则提供了更好的管理方式，它不仅能以声明式的方式配置Session- Factory实例，也可充分利用IoC容器的作用，为SessionFactory注入数据源引用。   
下面是Spring配置文件中配置Hibernate SessionFactory的示范代码：

**Xml代码  收藏代码**

1. **<?xml** version="1.0" encoding="GBK"**?>**
2. <!-- beans是Spring配置文件的根元素，并且指定了Schema信息 -->
3. **<beans** xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>**
6. <!-- 定义数据源Bean，使用C3P0数据源实现 -->
7. **<bean** id="dataSource" class="com.mchange.v2.c3p0. ComboPooledDataSource"
8. destroy-method="close"**>**
9. <!-- 指定连接数据库的驱动 -->
10. **<property** name="driverClass" value="com.mysql.jdbc.Driver"**/>**
11. <!-- 指定连接数据库的URL -->
12. **<property** name="jdbcUrl" value="jdbc:mysql://localhost/j2ee"**/>**
13. <!-- 指定连接数据库的用户名 -->
14. **<property** name="user" value="root"**/>**
15. <!-- 指定连接数据库的密码 -->
16. **<property** name="password" value="32147"**/>**
17. <!-- 指定连接数据库连接池的最大连接数 -->
18. **<property** name="maxPoolSize" value="40"**/>**
19. <!-- 指定连接数据库连接池的最小连接数 -->
20. **<property** name="minPoolSize" value="1"**/>**
21. <!-- 指定连接数据库连接池的初始化连接数 -->
22. **<property** name="initialPoolSize" value="1"**/>**
23. <!-- 指定连接数据库连接池的连接最大空闲时间 -->
24. **<property** name="maxIdleTime" value="20"**/>**
25. **</bean>**
26. <!-- 定义Hibernate的SessionFactory -->
27. **<bean** id="sessionFactory" class="org.springframework.orm.hibernate3.
28. LocalSessionFactoryBean"**>**
29. <!-- 依赖注入数据源，正是上文定义的dataSource -->
30. **<property** name="dataSource" ref="dataSource"**/>**
31. <!-- mappingResources属性用来列出全部映射文件 -->
32. **<property** name="mappingResources"**>**
33. **<list>**
34. <!-- 以下用来列出所有的PO映射文件 -->
35. **<value>**lee/MyTest.hbm.xml**</value>**
36. **</list>**
37. **</property>**
38. <!-- 定义Hibernate的SessionFactory属性 -->
39. **<property** name="hibernateProperties"**>**
40. **<props>**
41. <!-- 指定Hibernate的连接方言 -->
42. **<prop** key="hibernate.dialect"**>**org.hibernate.dialect.
43. MySQLDialect**</prop>**
44. <!-- 配置启动应用时，是否根据Hibernate映射自动创建数据表 -->
45. **<prop** key="hibernate.hbm2ddl.auto"**>**update**</prop>**
46. **</props>**
47. **</property>**
48. **</bean>**
49. **</beans>**

 一旦在Spring的IoC容器中配置了SessionFactory Bean，它将随应用的启动而加载，并可以充分利用IoC容器的功能，将SessionFactory Bean注入任何Bean，比如DAO组件。一旦DAO组件获得了SessionFactory Bean的引用，就可以完成实际的数据库访问。   
当然，Spring也支持访问容器数据源。如果需要使用容器数据源，可将数据源Bean修改成如下配置：

**Xml代码  收藏代码**

1. <!-- 此处配置JNDI数据源 -->
2. **<bean** id="myDataSource" class="org.springframework.jndi.JndiObjectFactoryBean"**>**
3. **<property** name="jndiName"**>**
4. <!-- 指定数据源的JNDI -->
5. **<value>**java:comp/env/jdbc/myds**</value>**
6. **</property>**
7. **</bean>**

 可见，以声明式的方式管理SessionFactory实例，可以让应用在不同数据源之间切换。如果应用更换数据库等持久层资源，只需对配置文件进行简单修改即可。   
提示：以声明式的方式管理SessionFactory，非常类似于早期将数据库服务的相关信息放在web.xml文件中进行配置。这种方式是为了提供更好的适应性，当持久层服务需要更改时，应用代码无须任何改变。

**6.5.3  使用HibernateTemplateHibernateTemplate提供持久层访问模板.**使用HibernateTemplate无须实现特定接口，它只需要提供一个SessionFactory的引用就可执行持久化操作。SessionFactory对象既可通过构造参数传入，也可通过设值方式传入。HibernateTemplate提供如下3个构造函数：  
  ● HibernateTemplate()。   
  ● HibernateTemplate(org.hibernate.SessionFactory sessionFactory)。   
  ● HibernateTemplate(org.hibernate.SessionFactory sessionFactory, boolean allowCreate)。   
第一个构造函数，构造一个默认的HibernateTemplate实例。因此，使用Hibernate- Template实例之前，还必须使用方法setSessionFactory(SessionFactory sessionFactory)来为HibernateTemplate传入SessionFactory的引用。   
第二个构造函数，在构造时已经传入SessionFactory引用。   
第三个构造函数，其boolean型参数表明，如果当前线程已经存在一个非事务性的Session，是否直接返回此非事务性的Session。   
在Web应用中，通常启动时自动加载ApplicationContext，SessionFactory和DAO对象都处在Spring上下文管理下，因此无须在代码中显式设置，可采用依赖注入完成Session- Factory和DAO的解耦，依赖关系通过配置文件来设置，如下所示：

**Xml代码  收藏代码**

1. **<?xml** version="1.0" encoding="GBK"**?>**
2. <!-- beans是Spring配置文件的根元素，并且指定了Schema信息 -->
3. **<beans** xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>**
6. <!-- 定义数据源Bean，使用C3P0数据源实现 -->
7. **<bean** id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
8. destroy-method="close"**>**
9. <!-- 指定连接数据库的驱动 -->
10. **<property** name="driverClass" value="com.mysql.jdbc.Driver"**/>**
11. <!-- 指定连接数据库的URL -->
12. **<property** name="jdbcUrl" value="jdbc:mysql://localhost/j2ee"**/>**
13. <!-- 指定连接数据库的用户名 -->
14. **<property** name="user" value="root"**/>**
15. <!-- 指定连接数据库的密码 -->
16. **<property** name="password" value="32147"**/>**
17. <!-- 指定连接数据库连接池的最大连接数 -->
18. **<property** name="maxPoolSize" value="40"**/>**
19. <!-- 指定连接数据库连接池的最小连接数 -->
20. **<property** name="minPoolSize" value="1"**/>**
21. <!-- 指定连接数据库连接池的初始化连接数 -->
22. **<property** name="initialPoolSize" value="1"**/>**
23. <!-- 指定连接数据库连接池的连接最大空闲时间 -->
24. **<property** name="maxIdleTime" value="20"**/>**
25. **</bean>**
26. <!-- 定义Hibernate的SessionFactory Bean -->
27. **<bean** id="sessionFactory" class="org.springframework.orm.hibernate3.
28. LocalSessionFactoryBean"**>**
29. <!-- 依赖注入数据源，注入的正是上文中定义的dataSource -->
30. **<property** name="dataSource" ref="dataSource"**/>**
31. <!-- mappingResources属性用来列出全部映射文件 -->
32. **<property** name="mappingResources"**>**
33. **<list>**
34. <!-- 以下用来列出所有的PO映射文件 -->
35. **<value>**lee/Person.hbm.xml**</value>**
36. **</list>**
37. **</property>**
38. <!-- 定义Hibernate的SessionFactory属性 -->
39. **<property** name="hibernateProperties"**>**
40. **<props>**
41. <!-- 指定Hibernate的连接方言 -->
42. **<prop** key="hibernate.dialect"**>**org.hibernate.dialect.
43. MySQLDialect**</prop>**
44. <!-- 指定启动应用时，是否根据Hibernate映射文件创建数据表 -->
45. **<prop** key="hibernate.hbm2ddl.auto"**>**update**</prop>**
46. **</props>**
47. **</property>**
48. **</bean>**
49. <!-- 配置Person持久化类的DAO bean -->
50. **<bean** id="personDao" class="lee.PersonDaoImpl"**>**
51. <!-- 采用依赖注入来传入SessionFactory的引用 -->
52. **<property** name="sessionFactory" ref="sessionFactory"**/>**
53. **</bean>**
54. **</beans>**

 在PersonDao组件中，所有的持久化操作都通过HibernateTemplate实例完成，而HibernateTemplate操作数据库非常简洁，大部分CRUD操作都可通过一行代码解决问题。下面介绍如何通过HibernateTemplate进行持久层访问。   
HibernateTemplate提供了非常多的常用方法来完成基本的操作，比如通常的增加、删除、修改、查询等操作，Spring 2.0更增加了对命名SQL查询的支持，也增加了对分页的支持。大部分情况下，使用Hibernate的常规用法，就可完成大多数DAO对象的CRUD操作。下面是HibernateTemplate的常用方法简介：   
  ● void delete(Object entity)，删除指定持久化实例。   
  ● deleteAll(Collection entities)，删除集合内全部持久化类实例。   
  ● find(String queryString)，根据HQL查询字符串来返回实例集合。   
  ● findByNamedQuery(String queryName)，根据命名查询返回实例集合。   
  ● get(Class entityClass, Serializable id)，根据主键加载特定持久化类的实例。   
  ● save(Object entity)，保存新的实例。   
  ● saveOrUpdate(Object entity)，根据实例状态，选择保存或者更新。   
  ● update(Object entity)，更新实例的状态，要求entity是持久状态。   
  ● setMaxResults(int maxResults)，设置分页的大小。   
下面是一个完整DAO类的源代码：

**Java代码  收藏代码**

1. **public** **class** PersonDaoImpl **implements** PersonDao
2. {
3. //执行持久化操作的HibernateTemplate实例
4. **private** HibernateTemplate ht = **null**;
5. **private** SessionFactory sessionFactory;
6. //该DAO组件持久化操作所需的SessionFactory对象
7. **public** **void** setSessionFactory(SessionFactory sessionFactory)
8. {
9. **this**.sessionFactory = sessionFactory;
10. }
11. //用于根据SessionFactory实例返回HibernateTemplate实例的方法
12. **private**  HibernateTemplate getHibernateTemplate()
13. {
14. **if** (ht == **null**)
15. {
16. ht = **new** HibernateTemplate(sessionFactory);
17. }
18. **return** ht;
19. }
20. /\*\*
21. \* 加载人实例
22. \* @param id 需要加载的Person实例的主键值
23. \* @return 返回加载的Person实例
24. \*/
25. **public** Person get(**int** id)
26. {
27. **return** (Person)getHibernateTemplate().get(Person.**class**, **new**
28. Integer(id));
29. }
30. /\*\*
31. \* 保存人实例
32. \* @param person 需要保存的Person实例
33. \*/
34. **public** **void** save(Person person)
35. {
36. getHibernateTemplate().save(person);
37. }
38. /\*\*
39. \* 修改Person实例
40. \* @param person 需要修改的Person实例
41. \*/
42. **public** **void** update(Person person)
43. {
44. getHibernateTemplate().update(person);
45. }
46. /\*\*
47. \* 删除Person实例
48. \* @param id 需要删除的Person的id
49. \*/
50. **public** **void** delete(**int** id)
51. {
52. getHibernateTemplate().delete(getHibernateTemplate().get(Person.
53. **class**,**new** Integer(id)));
54. }
55. /\*\*
56. \* 删除Person实例
57. \* @param person 需要删除的Person实例
58. \*/
59. **public** **void** delete(Person person)
60. {
61. getHibernateTemplate().delete(person);
62. }
63. /\*\*
64. \* 根据用户名查找Person
65. \* @param name 用户名
66. \* @return 用户名对应的全部用户
67. \*/
68. **public** List findByName(String name)
69. {
70. **return** getHibernateTemplate().find("from Person p where p.name
71. like ?" , name);
72. }
73. /\*\*
74. \* 返回全部的Person实例
75. \* @return 全部的Person实例
76. \*/
77. **public** List findAllPerson()
78. {
79. **return** getHibernateTemplate().find("from Person ");
80. }
81. }

 通过上面实现DAO组件的代码可以看出，通过HibernateTemplate进行持久层访问的代码如此清晰，大部分CRUD操作一行代码即可完成，完全无须Hibernate访问那些繁琐的步骤。而且，一旦DAO组件获得了SessionFactory的引用，即可很轻易地创建HibernateTemplate实例。   
提示：HibernateTemplate是Spring众多模板工具类之一，Spring正是通过这种简便地封装，完成了开发中大量需要重复执行的工作。

**6.5.4  使用HibernateCallBackHibernateTemplate还提供了一种更加灵活的方式来操作数据库，通过这种方式可以完全使用Hibernate的操作方式。**

HibernateTemplate的灵活访问方式可通过如下两个方法完成：   
  ● Object execute(HibernateCallback action)。   
  ● List execute(HibernateCallback action)。   
这两个方法都需要一个HibernateCallback的实例，HibernateCallback实例可在任何有效的Hibernate数据访问中使用。程序开发者通过HibernateCallback，可以完全使用Hibernate灵活的方式来访问数据库，解决Spring封装Hibernate后灵活性不足的缺陷。   
HibernateCallback是一个接口，该接口包含一个方法doInHibernate(org.hibernate. Session session)，该方法只有一个参数Session。在开发中提供HibernateCallback实现类时，必须实现接口里包含的doInHibernate方法，在该方法体内即可获得Hibernate Session的引用，一旦获得了Hibernate Session的引用，就可以完全以Hibernate的方式进行数据库访问。   
注意：doInHibernate方法内可以访问Session，该Session对象是绑定在该线程的Session实例。该方法内的持久层操作，与不使用Spring时的持久层操作完全相同。这保证了对于复杂的持久层访问，依然可以使用Hibernate的访问方式。   
下面的代码对HibernateDaoSupport类进行扩展（虽然Spring 2.0的HibernateTemplate提供了一个分页方法setMaxResults，但仅此一个方法依然不能实现分页查询），这种扩展主要是为该类增加了3个分页查询的方法，分页查询时必须直接调用Hibernate的Session完成，因此，必须借助于HibernateCallBack的帮助。

**Java代码  收藏代码**

1. **public** **class** YeekuHibernateDaoSupport **extends** HibernateDaoSupport
2. {
3. /\*\*
4. \* 使用hql 语句进行分页查询操作
5. \* @param hql 需要查询的hql语句
6. \* @param offset 第一条记录索引
7. \* @param pageSize 每页需要显示的记录数
8. \* @return 当前页的所有记录
9. \*/
10. **public** List findByPage(**final** String hql,
11. **final** **int** offset, **final** **int** pageSize)
12. {
13. //HibernateDaoSupport已经包含了getHibernateTemplate()方法
14. List list = getHibernateTemplate().executeFind(**new**
15. HibernateCallback()
16. {
17. **public** Object doInHibernate(Session session)
18. **throws** HibernateException, SQLException
19. //该方法体内以Hibernate方法进行持久层访问
20. {
21. List result = session.createQuery(hql)
22. .setFirstResult(offset)
23. .setMaxResults(pageSize)
24. .list();
25. **return** result;
26. }
27. });
28. **return** list;
29. }
30. /\*\*
31. \* 使用hql 语句进行分页查询操作
32. \* @param hql 需要查询的hql语句
33. \* @param value 如果hql有一个参数需要传入，value就是传入的参数
34. \* @param offset 第一条记录索引
35. \* @param pageSize 每页需要显示的记录数
36. \* @return 当前页的所有记录
37. \*/
38. **public** List findByPage(**final** String hql , **final** Object value ,
39. **final** **int** offset, **final** **int** pageSize)
40. {
41. List list = getHibernateTemplate().executeFind(**new**
42. HibernateCallback()
43. {
44. **public** Object doInHibernate(Session session)
45. **throws** HibernateException, SQLException
46. {
47. //下面查询的是最简单的Hiberante HQL查询
48. List result = session.createQuery(hql)
49. .setParameter(0, value)
50. .setFirstResult(offset)
51. .setMaxResults(pageSize)
52. .list();
53. **return** result;
54. }
55. });
56. **return** list;
57. }
58. /\*\*
59. \* 使用hql 语句进行分页查询操作
60. \* @param hql 需要查询的hql语句
61. \* @param values 如果hql有多个参数需要传入，values就是传入的参数数组
62. \* @param offset 第一条记录索引
63. \* @param pageSize 每页需要显示的记录数
64. \* @return 当前页的所有记录
65. \*/
66. **public** List findByPage(**final** String hql, **final** Object[] values,
67. **final** **int** offset, **final** **int** pageSize)
68. {
69. List list = getHibernateTemplate().executeFind(**new**
70. HibernateCallback()
71. {
72. **public** Object doInHibernate(Session session)
73. **throws** HibernateException, SQLException
74. {
75. Query query = session.createQuery(hql);
76. **for** (**int** i = 0 ; i < values.length ; i++)
77. {
78. query.setParameter( i, values);
79. }
80. List result = query.setFirstResult(offset)
81. .setMaxResults(pageSize)
82. .list();
83. **return** result;
84. }
85. });
86. **return** list;
87. }
88. }

 在上面的代码实现中，直接使用了getHibernateTemplate()方法，这个方法由Hibernate- DaoSupport提供。而YeekuHibernateDaoSupport是HibernateDaoSupport的子类，因此，可以直接使用该方法。   
当实现doInHibernate(Session session)方法时，完全以Hibernate的方式进行数据库访问，这样保证了Hibernate进行数据库访问的灵活性。   
注意：Spring提供的XxxTemplate和XxxCallBack互为补充，二者体现了Spring框架设计的用心良苦：XxxTemplate对通用操作进行封装，而XxxCallBack解决了封装后灵活性不足的缺陷。

**6.5.5  实现DAO组件为了实现DAO组件，Spring提供了大量的XxxDaoSupport类，这些DAO支持类对于实现DAO组件大有帮助，因为这些DAO支持类已经完成了大量基础性工作。**

Spring为Hibernate的DAO提供了工具类HibernateDaoSupport。该类主要提供如下两个方法以方便DAO的实现：   
  ● public final HibernateTemplate getHibernateTemplate()。   
  ● public final void setSessionFactory(SessionFactory sessionFactory)。   
其中，setSessionFactory方法可用于接收Spring的ApplicationContext的依赖注入，可接收配置在Spring的SessionFactory实例，getHibernateTemplate方法用于返回通过SessionFactory产生的HibernateTemplate实例，持久层访问依然通过HibernateTemplate实例完成。   
下面实现的DAO组件继承了Spring提供的HibernateDaoSupport类，依然实现了PersonDao接口，其功能与前面提供的PersonDao实现类完全相同。其代码如下：

**Java代码  收藏代码**

1. **public** **class** PersonDaoHibernate **extends** HibernateDaoSupport **implements** PersonDao
2. {
3. /\*\*
4. \* 加载人实例
5. \* @param id 需要加载的Person实例的主键值
6. \* @return 返回加载的Person实例
7. \*/
8. **public** Person get(**int** id)
9. {
10. **return** (Person)getHibernateTemplate().get(Person.**class**, **new**
11. Integer(id));
12. }
13. /\*\*
14. \* 保存人实例
15. \* @param person 需要保存的Person实例
16. \*/
17. **public** **void** save(Person person)
18. {
19. getHibernateTemplate().save(person);
20. }
21. /\*\*
22. \* 修改Person实例
23. \* @param person 需要修改的Person实例
24. \*/
25. **public** **void** update(Person person)
26. {
27. getHibernateTemplate().update(person);
28. }
29. /\*\*
30. \* 删除Person实例
31. \* @param id 需要删除的Person的id
32. \*/
33. **public** **void** delete(**int** id)
34. {
35. getHibernateTemplate().delete(getHibernateTemplate().
36. get(Person.**class**, **new** Integer(id)));
37. }
38. /\*\*
39. \* 删除Person实例
40. \* @param person 需要删除的Person实例
41. \*/
42. **public** **void** delete(Person person)
43. {
44. getHibernateTemplate().delete(person);
45. }
46. /\*\*
47. \* 根据用户名查找Person
48. \* @param name 用户名
49. \* @return 用户名对应的全部用户
50. \*/
51. **public** List findByPerson(String name)
52. {
53. **return** getHibernateTemplate().find("from Person p where p.name
54. like ?" , name);
55. }
56. /\*\*
57. \* 返回全部的Person实例
58. \* @return 全部的Person实例
59. \*/
60. **public** List findAllPerson()
61. {
62. **return** getHibernateTemplate().find("from Person ");
63. }
64. }

 上面的代码与前面的PersonDAOImpl对比会发现，代码量大大减少。事实上，DAO的实现依然借助于HibernateTemplate的模板访问方式，只是HibernateDaoSupport将依赖注入SessionFactory的工作已经完成，获取HibernateTemplate的工作也已完成。该DAO的配置必须依赖于SessionFactory，配置文件与前面部署DAO组件的方式完全相同，此处不再赘述。   
在继承HibernateDaoSupport的DAO实现里，Hibernate Session的管理完全不需要打开代码，而由Spring来管理。Spring会根据实际的操作，采用“每次事务打开一次session”的策略，自动提高数据库访问的性能。

**6.5.6  使用IoC容器组装各种组件至此为止，J2EE应用所需要的各种组件都已经出现了，从MVC层的控制器组件，到业务逻辑组件，以及持久层的DAO组件，已经全部成功实现。应用程序代码并未将这些组件耦合在一起，代码中都是面向接口编程，因此必须利用Spring的IoC容器将他们组合在一起。**

从用户角度来看，用户发出HTTP请求，当MVC框架的控制器组件拦截到用户请求时，将调用系统的业务逻辑组件，而业务逻辑组件则调用系统的DAO组件，而DAO组件则依赖于SessionFactory和DataSource等底层组件实现数据库访问。   
从系统实现角度来看，IoC容器先创建SessionFactory和DataSource等底层组件，然后将这些底层组件注入给DAO组件，提供一个完整的DAO组件，并将此DAO组件注入给业务逻辑组件，从而提供一个完整的业务逻辑组件，而业务逻辑组件又被注入给控制器组件，控制器组件负责拦截用户请求，并将处理结果呈现给用户——这一系列的衔接都由Spring的IoC容器提供实现。   
下面给出关于如何在容器中配置J2EE组件的大致模板，其模板代码如下：

**Xml代码  收藏代码**

1. **<?xml** version="1.0" encoding="GBK"**?>**
2. <!-- beans是Spring配置文件的根元素，并且指定了Schema信息 -->
3. **<beans** xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>**
6. <!-- 定义数据源Bean，使用C3P0数据源实现 -->
7. **<bean** id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
8. destroy-method="close"**>**
9. <!-- 指定连接数据库的驱动 -->
10. **<property** name="driverClass" value="com.mysql.jdbc.Driver"**/>**
11. <!-- 指定连接数据库的URL -->
12. **<property** name="jdbcUrl" value="jdbc:mysql://localhost/j2ee"**/>**
13. <!-- 指定连接数据库的用户名 -->
14. **<property** name="user" value="root"**/>**
15. <!-- 指定连接数据库的密码 -->
16. **<property** name="password" value="32147"**/>**
17. <!-- 指定连接数据库连接池的最大连接数 -->
18. **<property** name="maxPoolSize" value="40"**/>**
19. <!-- 指定连接数据库连接池的最小连接数 -->
20. **<property** name="minPoolSize" value="1"**/>**
21. <!-- 指定连接数据库连接池的初始化连接数 -->
22. **<property** name="initialPoolSize" value="1"**/>**
23. <!-- 指定连接数据库连接池的连接最大空闲时间 -->
24. **<property** name="maxIdleTime" value="20"**/>**
25. **</bean>**
26. <!-- 定义Hibernate的SessionFactory Bean -->
27. **<bean** id="sessionFactory" class="org.springframework.orm.hibernate3.
28. LocalSessionFactoryBean"**>**
29. <!-- 依赖注入数据源，注入的正是上文中定义的dataSource -->
30. **<property** name="dataSource" ref="dataSource"**/>**
31. <!-- mappingResources属性用来列出全部映射文件 -->
32. **<property** name="mappingResources"**>**
33. **<list>**
34. <!-- 以下用来列出所有的PO映射文件 -->
35. **<value>**lee/Person.hbm.xml**</value>**
36. <!-- 此处还可列出更多的PO映射文件 -->
37. **</list>**
38. **</property>**
39. <!-- 定义Hibernate的SessionFactory属性 -->
40. **<property** name="hibernateProperties"**>**
41. **<props>**
42. <!-- 指定Hibernate的连接方言 -->
43. **<prop** key="hibernate.dialect"**>**org.hibernate.dialect.
44. MySQLDialect**</prop>**
45. <!-- 指定启动应用时，是否根据Hibernate映射文件创建数据表 -->
46. **<prop** key="hibernate.hbm2ddl.auto"**>**update**</prop>**
47. **</props>**
48. **</property>**
49. **</bean>**
50. <!-- 配置Person持久化类的DAO Bean -->
51. **<bean** id="personDao" class="lee.PersonDaoImpl"**>**
52. <!-- 采用依赖注入来传入SessionFactory的引用 -->
53. **<property** name="sessionFactory" ref="sessionFactory"**/>**
54. **</bean>**
55. <!-- 下面能以相同的方式配置更多的持久化Bean -->
56. ...
57. **<bean** id="myService" class="lee.MyServiceImp"**>**
58. <!-- 注入业务逻辑组件所必需的DAO组件 -->
59. **<property** name="peronDdao" ref=" personDao "**/>**
60. <!-- 此处可采用依赖注入更多的DAO组件 -->
61. ...
62. **</bean>**
63. <!-- 配置控制器Bean，设置起作用域为Request -->
64. **<bean** name="/login" class="lee.LoginAction" scope="request"**>**
65. <!-- 依赖注入控制器所必需的业务逻辑组件 -->
66. **<property** name="myService" ref=" myService "**/>**
67. **</bean>**
68. **</beans>**

 在上面的配置文件中，同时配置了控制器Bean、业务逻辑组件Bean、DAO组件Bean以及一些基础资源Bean。各组件的组织被解耦到配置文件中，而不是在代码层次的低级耦合。   
当客户端的HTTP请求向/login.do发送请求时，将被容器中的lee.LoginAction拦截，LoginAction调用myService Bean，myService Bean则调用personDao等系列DAO组件，整个流程将系统中的各组件有机地组织在一起。   
注意：在实际应用中，很少会将DAO组件、业务逻辑组件以及控制组件都配置在同一个文件中。而是在不同配置文件中，配置相同一组J2EE应用组件。

**6.5.7  使用声明式事务在上面的配置文件中，部署了控制器组件、业务逻辑组件、DAO组件，几乎可以形成一个完整的J2EE应用。**

但有一个小小的问题：事务控制。系统没有任何事务逻辑，没有事务逻辑的应用是不可想象的。   
Spring提供了非常简洁的声明式事务控制，只需要在配置文件中增加事务控制片段，业务逻辑代码无须任何改变。Spring的声明式事务逻辑，甚至支持在不同事务策略之间切换。   
配置Spring声明式事务时，通常推荐使用BeanNameAutoProxyCreator自动创建事务代理。通过这种自动事务代理的配置策略，增加业务逻辑组件，只需要在BeanNameAutoProxyCreator Bean配置中增加一行即可，从而避免了增量式配置。   
在上面的配置模板文件中增加如下配置片段，系统的myService业务逻辑组件将变成事务代理Bean，从而为业务逻辑方法增加事务逻辑。 

**Xml代码  收藏代码**

1. <!-- 配置Hibernate的局部事务管理器 -->
2. <!-- 使用HibernateTransactionManager类，该类是PlatformTransactionManager
3. 接口，针对采用Hibernate持久化连接的特定实现 --**>**
4. **<bean** id="transactionManager"
5. class="org.springframework.orm.hibernate3.
6. HibernateTransactionManager"**>**
7. <!-- HibernateTransactionManager  Bean需要依赖注入一个SessionFactory
8. bean的引用 --**>**
9. **<property** name="sessionFactory" ref="sessionFactory"**/>**
10. **</bean>**
11. <!-- 配置事务拦截器Bean -->
12. **<bean** id="transactionInterceptor"
13. class="org.springframework.transaction.interceptor.
14. TransactionInterceptor"**>**
15. <!-- 事务拦截器bean需要依赖注入一个事务管理器 -->
16. **<property** name="transactionManager" ref="transactionManager"**/>**
17. **<property** name="transactionAttributes"**>**
18. <!-- 下面定义事务传播属性 -->
19. **<props>**
20. **<prop** **>**PROPAGATION\_REQUIRED**</prop>**
21. **<prop** **>**PROPAGATION\_REQUIRED,readOnly**</prop>**
22. **<prop** key="\*"**>**PROPAGATION\_REQUIRED**</prop>**
23. **</props>**
24. **</property>**
25. **</bean>**
26. <!-- 定义BeanNameAutoProxyCreator的Bean后处理器 -->
27. **<bean** class="org.springframework.aop.framework.autoproxy.
28. BeanNameAutoProxyCreator"**>**
29. <!-- 指定对满足哪些bean name的bean自动生成业务代理 -->
30. **<property** name="beanNames"**>**
31. <!-- 下面是所有需要自动创建事务代理的Bean -->
32. **<list>**
33. **<value>**myService**</value>**
34. <!-- 下面还可增加需要增加事务逻辑的业务逻辑Bean -->
35. ...
36. **</list>**
37. <!-- 此处可增加其他需要自动创建事务代理的Bean -->
38. **</property>**
39. <!-- 下面定义BeanNameAutoProxyCreator所需的拦截器 -->
40. **<property** name="interceptorNames"**>**
41. **<list>**
42. **<value>**transactionInterceptor**</value>**
43. <!-- 此处可增加其他新的Interceptor -->
44. **</list>**
45. **</property>**
46. **</bean>**

 一旦增加了如上的配置片段，系统中的业务逻辑方法就有了事务逻辑。这种声明式事务配置方式可以在不同的事务策略之间自由切换。   
提示：尽量使用声明式事务配置方式，而不要在代码中完成事务逻辑。

来源： <http://greatwqs.iteye.com/blog/1044271>