跟我学spring3(8-13)

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

跟我学spring3(8-13)

目 录

1. spring[原创]

1.1 【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。

1.2 【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???

```
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】

1.3 【第八章】 对ORM的支持 之 8.3 集成iBATIS ——跟我学spring3

发表时间: 2012-03-02 关键字: spring

8.3 集成iBATIS

iBATIS是一个半自动化的ORM框架，需要通过配置方式指定映射SQL语句，而不是由框架本身生成（如Hibernate自动生成对应SQL来持久化对象），即Hibernate属于全自动ORM框架。

Spring提供对iBATIS 2.X的集成，提供一致的异常体系、一致的DAO访问支持、Spring管理事务支持。

Spring 2.5.5+版本支持iBATIS 2.3+版本，不支持低版本。

8.3.1 如何集成

Spring通过使用如下Bean进行集成iBATIS：

- SqlMapClientFactoryBean：用于集成iBATIS。

configLocation和configLocations：用于指定SQL Map XML配置文件，用于指定如数据源等配置信息；

mappingLocations：用于指定SQL Map映射文件，即半自动概念中的SQL语句定义；

sqlMapClientProperties：定义iBATIS 配置文件配置信息；

dataSource：定义数据源。

如果在Spring配置文件中指定了DataSource，就不要在iBATIS配置文件指定了，否则Spring配置文件指定的DataSource将覆盖iBATIS配置文件中定义的DataSource。

接下来示例一下如何集成iBATIS：

1、准备需要的jar包，从spring-framework-3.0.5.RELEASE-dependencies.zip中拷贝如下jar包：

com.springsource.com.ibatis-2.3.4.726.jar

2、 对象模型定义，此处使用第七章中的UserModel；

3、 iBATIS映射定义（chapter8/sqlmaps/UserSQL.xml）：

java代码：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMap PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
    "http://ibatis.apache.org/dtd/sql-map-2.dtd">
<sqlMap namespace="UserSQL">
  <statement id="createTable">
    <!--id自增主键从0开始 -->
    <![CDATA[
        create memory table test(
          id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
          name varchar(100))
    ]]>
  </statement>
  <statement id="dropTable">
    <![CDATA[  drop table test  ]]>
  </statement>
  <insert id="insert" parameterClass="cn.javass.spring.chapter7.UserModel">
    <![CDATA[
      insert into test(name) values (#myName#)
    ]]>
    <selectKey resultClass="int" keyProperty="id" type="post">
      <!-- 获取hsqldb插入的主键 -->
      call identity();
      <!-- mysql使用select last_insert_id();获取插入的主键 -->
    </selectKey>
  </insert>
</sqlMap>
```

4、 iBATIS配置文件（chapter8/sql-map-config.xml）定义：

java代码：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMapConfig PUBLIC "-//ibatis.apache.org//DTD SQL Map Config 2.0//EN"
    "http://ibatis.apache.org/dtd/sql-map-config-2.dtd">

<sqlMapConfig>
    <settings enhancementEnabled="true" useStatementNamespaces="true"
        maxTransactions="20" maxRequests="32" maxSessions="10"/>
    <sqlMap resource="chapter8/sqlmaps/UserSQL.xml"/>
</sqlMapConfig>
```

5、 数据源定义，此处使用第7章的配置文件，即“chapter7/applicationContext-resources.xml”文件。

6、 SqlMapClient配置（chapter8/applicationContext-ibatis.xml）定义：

java代码：

```
<bean id="sqlMapClient"
    class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
    <!-- 1、指定数据源 -->
    <property name="dataSource" ref="dataSource"/>
    <!-- 2、指定配置文件 -->
    <property name="configLocation" value="chapter8/sql-map-config.xml"/>
</bean>
```

7、 获取SqlMapClient：

java代码：

```
package cn.javass.spring.chapter8;
//省略import
public class IbatisTest {
    private static SqlMapClient sqlMapClient;
    @BeforeClass
    public static void setUpClass() {
      String[] configLocations = new String[] {
          "classpath:chapter7/applicationContext-resources.xml",
          "classpath:chapter8/applicationContext-ibatis.xml"};
      ApplicationContext ctx = new ClassPathXmlApplicationContext(configLocations);
      sqlMapClient = ctx.getBean(SqlMapClient.class);
  }
}
```

此处我们使用了chapter7/applicationContext-resources.xml定义的“dataSource”数据源，通过ctx.getBean(SqlMapClient.class)获取SqlMapClient。

8、 通过SqlMapClient创建和删除表：

java代码：

```
@Before
public void setUp() throws SQLException {
    sqlMapClient.update("UserSQL.createTable");
}
```

```
@After
public void tearDown() throws SQLException {
    sqlMapClient.update("UserSQL.dropTable");
}
```

9、 使用SqlMapClient进行对象持久化：

java代码：

```
@Test
public void testFirst() throws SQLException {
    UserModel model = new UserModel();
    model.setMyName("test");
    SqlMapSession session = null;
    try {
        session = sqlMapClient.openSession();
        beginTransaction(session);
        session.insert("UserSQL.insert", model);
        commitTransaction(session);
    } catch (SQLException e) {
        rollbackTransacrion(session);
        throw e;
    } finally {
        closeSession(session);
    }
}
private void closeSession(SqlMapSession session) {
   session.close();
}
private void rollbackTransacrion(SqlMapSession session) throws SQLException {
```

```
    if(session != null) {
        session.endTransaction();
    }
}
private void commitTransaction(SqlMapSession session) throws SQLException {
    session.commitTransaction();
}
private void beginTransaction(SqlMapSession session) throws SQLException {
    session.startTransaction();
}
```

同样令人心烦的事务管理和冗长代码，Spring通用提供了SqlMapClientTemplate模板类来解决这些问题。

8.3.2 使用 SqlMapClientTemplate

SqlMapClientTemplate模板类同样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SqlMapClientCallback回调接口来允许更复杂的操作。

接下来示例一下SqlMapClientTemplate的使用：

java代码：

```
@Test
public void testSqlMapClientTemplate() {
SqlMapClientTemplate sqlMapClientTemplate =
new SqlMapClientTemplate(sqlMapClient);
    final UserModel model = new UserModel();
    model.setMyName("myName");
    sqlMapClientTemplate.insert("UserSQL.insert", model);
    //通过回调允许更复杂操作
    sqlMapClientTemplate.execute(new SqlMapClientCallback<Void>() {
```

```
        @Override
        public Void doInSqlMapClient(SqlMapExecutor session) throws SQLException {
            session.insert("UserSQL.insert", model);
            return null;
    }});
}
```

通过new SqlMapClientTemplate(sqlMapClient)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SqlMapClientTemplate提供使用SqlMapClientCallback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。

8.3.3集成iBATIS及最佳实践

类似于JdbcDaoSupport类，Spring对iBATIS也提供了SqlMapClientDaoSupport类来支持一致的数据库访问。SqlMapClientDaoSupport也是DaoSupport实现：

接下来示例一下Spring集成iBATIS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iBATIS实现：

java代码：

```
package cn.javass.spring.chapter8.dao.ibatis;
//省略import
```

```
public class UserIbatisDaoImpl extends SqlMapClientDaoSupport
    implements IUserDao {
    @Override
    public void save(UserModel model) {
        getSqlMapClientTemplate().insert("UserSQL.insert", model);
    }
    @Override
    public int countAll() {
        return (Integer) getSqlMapClientTemplate().queryForObject("UserSQL.countAll");
    }
}
```

3、修改iBATS映射文件（chapter8/sqlmaps/UserSQL.xml），添加countAll查询：

java代码：

```
<select id="countAll" resultClass="java.lang.Integer">
    <![CDATA[ select count(*) from test ]]>
</select>
```

此处注意首先iBATIS实现放在dao.ibaitis包里，其次实现类命名如UserIbatisDaoImpl，即×××IbatisDaoImpl，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。

4、进行资源配置，使用resources/chapter7/applicationContext-resources.xml：

5、dao定义配置，在chapter8/applicationContext-ibatis.xml中添加如下配置：

java代码：

```
<bean id="abstractDao" abstract="true">
    <property name="sqlMapClient" ref="sqlMapClient"/>
</bean>
<bean id="userDao"
      class="cn.javass.spring.chapter8.dao.ibatis.UserIbatisDaoImpl"
      parent="abstractDao"/>
```

首先定义抽象的abstractDao，其有一个sqlMapClient属性，从而可以让继承的子类自动继承sqlMapClient属性注入；然后定义userDao，且继承abstractDao，从而继承sqlMapClient注入；我们在此给配置文件命名为applicationContext-ibatis.xml表示iBAITIS实现。

5、 最后测试一下吧（cn.javass.spring.chapter8. IbatisTest）：

java代码：

```
@Test
public void testBestPractice() {
    String[] configLocations = new String[] {
            "classpath:chapter7/applicationContext-resources.xml",
            "classpath:chapter8/applicationContext-ibatis.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-ibatis.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。

8.3.4Spring+iBATIS的CURD

Spring集成iBATIS进行CURD（增删改查），也非常简单，首先配置映射文件，然后调用SqlMapClientTemplate相应的函数进行操作即可，此处就不介绍了。

8.3.5集成MyBatis及最佳实践

(本笔记写于2010年底)

2010年4月份 iBATIS团队发布iBATIS 3.0的GA版本的候选版本，在iBATIS 3中引入了泛型、注解支持等，因此需要Java5+才能使用，但在2010年6月16日，iBATIS团队决定从apache迁出并迁移到Google Code，并更名为MyBatis。目前新网站上文档并不完善。

目前iBATIS 2.x和MyBatis 3不是100%兼容的，如配置文件的DTD变更，SqlMapClient直接由SqlSessionFactory代替了，包名也有com.ibatis变成org.ibatis等等。

ibatis 3.x和MyBatis是兼容的，只需要将DTD变更一下就可以了。

感兴趣的朋友可以到http://www.mybatis.org/官网去下载最新的文档学习，作者只使用过iBATIS2.3.4及以前版本，没在新项目使用过最新的iBATIS 3.x和Mybatis，因此如果读者需要在项目中使用最新的MyBatis，请先做好调研再使用。

接下来示例一下Spring集成MyBatis的最佳实践：

1、准备需要的jar包，到MyBatis官网下载mybatis 3.0.4版本和mybatis-spring 1.0.0版本，并拷贝如下jar包到类路径：

mybatis-3.0.4\mybatis-3.0.4.jar //核心MyBatis包

mybatis-spring-1.0.0\mybatis-spring-1.0.0.jar //集成Spring包

2、对象模型定义，此处使用第七章中的UserModel；

3、MyBatis映射定义（chapter8/sqlmaps/UserSQL-mybatis.xml）：

java代码：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="UserSQL">
    <sql id="createTable">
    <!--id自增主键从0开始 -->
    <![CDATA[
      create memory table test(
        id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
        name varchar(100))
    ]]>
    </sql>
    <sql id="dropTable">
    <![CDATA[ drop table test ]]>
    </sql>
    <insert id="insert" parameterType="cn.javass.spring.chapter7.UserModel">
```

```
    <![CDATA[ insert into test(name) values (#{myName}) ]]>
        <selectKey resultType="int" keyProperty="id" order="AFTER">
            <!-- 获取hsqldb插入的主键 -->
            call identity();
            <!-- mysql使用select last_insert_id();获取插入的主键 -->
        </selectKey>
    </insert>
    <select id="countAll" resultType="java.lang.Integer">
    <![CDATA[ select count(*) from test ]]>
    </select>
</mapper>
```

从映射定义中可以看出MyBatis与iBATIS2.3.4有如下不同：

- http://ibatis.apache.org/dtd/sql-map-2.dtd 废弃，而使用http://mybatis.org/dtd/mybatis-3-mapper.dtd。
- <sqlMap>废弃，而使用<mapper>标签；
- <statement>废弃了，而使用<sql>标签；
- parameterClass属性废弃，而使用parameterType属性；
- resultClass属性废弃，而使用resultType属性；
- #myName#方式指定命名参数废弃，而使用#{myName}方式。

3、 MyBatis配置文件（chapter8/sql-map-config-mybatis.xml）定义：

java代码：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <settings>
        <setting name="cacheEnabled" value="false"/>
```

```
    </settings>
    <mappers>
        <mapper resource="chapter8/sqlmaps/UserSQL-mybatis.xml"/>
    </mappers>
</configuration>
```

从配置定义中可以看出MyBatis与iBATIS2.3.4有如下不同：

- http://ibatis.apache.org/dtd/sql-map-config-2.dtd废弃，而使用http://mybatis.org/dtd/mybatis-3-config.dtd；
- < sqlMapConfig >废弃，而使用<configuration>；
- settings属性配置方式废弃，而改用子标签< setting name=".." value=".."/>方式指定属性，且一些属性被废弃，如maxTransactions；
- < sqlMap>废弃，而采用<mappers>标签及其子标签<mapper>定义。

4、 定义Dao接口，此处使用cn.javass.spring.chapter7.dao. IUserDao：

5、 定义Dao接口实现，此处是MyBatis实现：

java代码：

```
package cn.javass.spring.chapter8.dao.mybatis;
//省略import
public class UserMybatisDaoImpl extends SqlSessionDaoSupport
implements IUserDao {
    @Override
    public void save(UserModel model) {
        getSqlSession().insert("UserSQL.insert", model);
    }
    @Override
    public int countAll() {
        return (Integer) getSqlSession().selectOne("UserSQL.countAll");
```

```
    }
}
```

和Ibatis集成方式不同的有如下地方：

- 使用SqlSessionDaoSupport来支持一致性的DAO访问，该类位于org.mybatis.spring.support包中，非Spring提供；
- 使用getSqlSession方法获取SqlSessionTemplate，在较早版本中是getSqlSessionTemplate方法名，不知为什么改成getSqlSession方法名，因此这个地方在使用时需要注意。
- SqlSessionTemplate是SqlSession接口的实现，并且自动享受Spring管理事务好处，因此从此处可以推断出为什么把获取模板类的方法名改为getSqlSession而不是getSqlSessionTemplate。

6、进行资源配置，使用resources/chapter7/applicationContext-resources.xml：

7、dao定义配置，在chapter8/applicationContext-mybatis.xml中添加如下配置：

java代码：

```
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
  <property name="dataSource" ref="dataSource"/><!-- 1、指定数据源 -->
  <property name="configLocation" value="chapter8/sql-map-config-mybatis.xml"/>
</bean>
<bean id="abstractDao" abstract="true">
   <property name="sqlSessionFactory" ref="sqlSessionFactory"/>
</bean>
<bean id="userDao"
           class="cn.javass.spring.chapter8.dao.mybatis.UserMybatisDaoImpl"
           parent="abstractDao"/>
```

和Ibatis集成方式不同的有如下地方：

- SqlMapClient类废弃，而使用SqlSessionFactory代替；
- 使用SqlSessionFactoryBean进行集成MyBatis。

首先定义抽象的abstractDao，其有一个sqlSessionFactory属性，从而可以让继承的子类自动继承sqlSessionFactory属性注入；然后定义userDao，且继承abstractDao，从而继承sqlSessionFactory注入；我们在此给配置文件命名为applicationContext-mybatis.xml表示MyBatis实现。

8、最后测试一下吧（cn.javass.spring.chapter8. IbatisTest）：

java代码：

```
@Test
public void testMybatisBestPractice() {
    String[] configLocations = new String[] {
        "classpath:chapter7/applicationContext-resources.xml",
        "classpath:chapter8/applicationContext-mybatis.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 集成Ibatis的最佳实践完全一样，除了使用applicationContext-mybatis.xml代替了applicationContext-ibatis.xml，其他完全一样，且MyBatis 3.x与Spring整合只能运行在Spring3.x。

在写本书时，MyBatis与Spring集成所定义的API不稳定，且期待Spring能在发布新版本时将加入对MyBatis的支持。

1.4 跟我学spring3 电子书下载

发表时间: 2012-03-02 关键字: spring

感谢iteye各位网友对我的支持，在此谢过了！

《跟我学spring3》电子书下载地址：

《跟我学spring3》（1-7） http://www.iteye.com/blog/download_pdf/9383

《跟我学spring3》（8-13） http://www.iteye.com/blog/download_pdf/9619

目录：

【第七章】 对JDBC的支持 之 7.1 概述 ——跟我学spring3

【第七章】 对JDBC的支持 之 7.2 JDBC模板类 ——跟我学spring3

【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

【第七章】 对JDBC的支持 之 7.4 Spring提供的其它帮助 ——跟我学spring3

【第七章】 对JDBC的支持 之 7.5 集成Spring JDBC及最佳实践 ——跟我学spring3

（完）

正在写《跟我学Spring Web MVC》系列，希望对大家有用！

1.5 【第八章】 对ORM的支持 之 8.4 集成JPA ——跟我学spring3

发表时间: 2012-03-04 关键字: spring, jpa

8.4 集成JPA

JPA全称为Java持久性API（Java Persistence API），JPA是Java EE 5标准之一，是一个ORM规范，由厂商来实现该规范，目前有Hibernate、OpenJPA、TopLink、EclipseJPA等实现。

8.4.1 如何集成

Spring目前提供集成Hibernate、OpenJPA、TopLink、EclipseJPA四个JPA标准实现。

Spring通过使用如下Bean进行集成JPA（EntityManagerFactory）：

- **LocalEntityManagerFactoryBean：**适用于那些仅使用JPA进行数据访问的项目，该FactoryBean将根据JPA PersistenceProvider自动检测配置文件进行工作，一般从“META-INF/persistence.xml”读取配置信息，这种方式最简单，但不能设置Spring中定义的DataSource，且不支持Spring管理的全局事务，而且JPA 实现商可能在JVM启动时依赖于VM agent从而允许它们进行持久化类字节码转换（不同的实现厂商要求不同，需要时阅读其文档），不建议使用这种方式；

persistenceUnitName：指定持久化单元的名称；

使用方式：

java代码：

```
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalEntityManagerFactoryl
    <property name="persistenceUnitName" value="persistenceUnit"/>
</bean>
```

- **从JNDI中获取：**用于从Java EE服务器获取指定的EntityManagerFactory，这种方式在进行Spring事务管理时一般要使用JTA事务管理；

使用方式：

java代码：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/jee
       http://www.springframework.org/schema/jee/spring-jee-3.0.xsd">
  <jee:jndi-lookup id="entityManagerFactory"  jndi-name="persistence/persistenceUnit"/>
</beans>
```

此处需要使用“jee”命名标签，且使用<jee:jndi-lookup>标签进行JNDI查找，“jndi-name”属性用于指定JNDI名字。

- **LocalContainerEntityManagerFactoryBean：**适用于所有环境的FactoryBean，能全面控制EntityManagerFactory配置,如指定Spring定义的DataSource等等。

persistenceUnitManager：用于获取JPA持久化单元，默认实现DefaultPersistenceUnitManager用于解决多配置文件情况

dataSource：用于指定Spring定义的数据源；

persistenceXmlLocation：用于指定JPA配置文件，对于对配置文件情况请选择设置persistenceUnitManager属性来解决；

persistenceUnitName：用于指定持久化单元名字；

persistenceProvider：用于指定持久化实现厂商类；如Hibernate为org.hibernate.ejb.HibernatePersistence类；

jpaVendorAdapter：用于设置实现厂商JPA实现的特定属性，如设置Hibernate的是否自动生成DDL的属性generateDdl；这些属性是厂商特定的，因此最好在这里设置；目前Spring提供HibernateJpaVendorAdapter、OpenJpaVendorAdapter、EclipseLinkJpaVendorAdapter、TopLinkJpaVendorAdapter、OpenJpaVendorAdapter四个实现。其中最重要的属性是“**database**”，用来指定使用的数据库类型，从而能根据数据库类型来决定比如如何将数据库特定异常转换为Spring的一致性异常，目前支持如下数据库（**DB2、DERBY、H2、HSQL、INFORMIX、MYSQL、ORACLE、POSTGRESQL、SQL_SERVER、SYBASE**）。

jpaDialect：用于指定一些高级特性，如事务管理，获取具有事务功能的连接对象等，目前Spring提供HibernateJpaDialect、OpenJpaDialect 、EclipseLinkJpaDialect、TopLinkJpaDialect、和DefaultJpaDialect实现，注意DefaultJpaDialect不提供任何功能，因此在使用特定实现厂商JPA实现时需要指定JpaDialect实现，如使用Hibernate就使用HibernateJpaDialect。当指定**jpaVendorAdapter属性时可以不指定jpaDialect，会自动设置相应的JpaDialect实现；**

jpaProperties和jpaPropertyMap：指定JPA属性；如Hibernate中指定是否显示SQL的“hibernate.show_sql”属性，对于jpaProperties设置的属性自动会放进jpaPropertyMap中；

loadTimeWeaver：用于指定LoadTimeWeaver实现，从而允许JPA 加载时修改相应的类文件。具体使用得参考相应的JPA规范实现厂商文档，如Hibernate就不需要指定loadTimeWeaver。

接下来学习一下Spring如何集成JPA吧：

1、准备jar包，从下载的hibernate-distribution-3.6.0.Final包中获取如下Hibernate需要的jar包从而支持JPA：

lib\jpa\hibernate-jpa-2.0-api-1.0.0.Final.jar //用于支持JPA

2、对象模型定义，此处使用UserModel2：

java代码：

```
package cn.javass.spring.chapter8;
//省略import
@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
}
```

注意此处使用的所有注解都是位于javax.persistence包下，如使用@org.hibernate.annotations.Entity 而非@javax.persistence. Entity将导致JPA不能正常工作。

1、 JPA配置定义（chapter8/persistence.xml），定义对象和数据库之间的映射：

java代码：

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence                    http://j
    <persistence-unit name="persistenceUnit" transaction-type="RESOURCE_LOCAL"/>
</persistence>
```

在JPA配置文件中，我们指定要持久化单元名字，和事务类型，其他都将在Spring中配置。

2、 数据源定义，此处使用第7章的配置文件，即“chapter7/applicationContext-resources.xml”文件。

3、 EntityManagerFactory配置定义（chapter8/applicationContext-jpa.xml）：

java代码：

```
  <bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityMan
    <property name="dataSource" ref="dataSource"/>
    <property name="persistenceXmlLocation" value="chapter8/persistence.xml"/>
    <property name="persistenceUnitName" value="persistenceUnit"/>
    <property name="persistenceProvider" ref="persistenceProvider"/>
    <property name="jpaVendorAdapter" ref="jpaVendorAdapter"/>
    <property name="jpaDialect" ref="jpaDialect"/>
    <property name="jpaProperties">
        <props>
            <prop key="hibernate.show_sql">true</prop>
        </props>
    </property>
  </bean>
  <bean id="persistenceProvider" class="org.hibernate.ejb.HibernatePersistence"/>
```

java代码：

```
<bean id="jpaVendorAdapter" class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdap
   <property name="generateDdl" value="false" />
   <property name="database" value="HSQL"/>
</bean>
<bean id="jpaDialect" class="org.springframework.orm.jpa.vendor.HibernateJpaDialect"/>
```

- **LocalContainerEntityManagerFactoryBean:**指定使用本地容器管理EntityManagerFactory，从而进行细粒度控制；
- **dataSource**属性指定使用Spring定义的数据源；
- **persistenceXmlLocation**指定JPA配置文件为chapter8/persistence.xml，且该配置文件非常简单，具体配置完全在Spring中进行；
- **persistenceUnitName**指定持久化单元名字，即JPA配置文件中指定的;
- **persistenceProvider**:指定JPA持久化提供商，此处使用Hibernate实现HibernatePersistence类；
- **jpaVendorAdapter**：指定实现厂商专用特性，即generateDdl= false表示不自动生成DDL，database= HSQL表示使用hsqldb数据库；
- **jpaDialect**：如果指定jpaVendorAdapter此属性可选，此处为HibernateJpaDialect；
- **jpaProperties**：此处指定“hibernate.show_sql =true”表示在日志系统debug级别下将打印所有生成的SQL。

4、 获取EntityManagerFactory：

java代码：

```
package cn.javass.spring.chapter8;
//省略import
public class JPATest {
    private static EntityManagerFactory entityManagerFactory;
    @BeforeClass
    public static void setUpClass() {
        String[] configLocations = new String[] {
                "classpath:chapter7/applicationContext-resources.xml",
                "classpath:chapter8/applicationContext-jpa.xml"};
        ApplicationContext ctx = new ClassPathXmlApplicationContext(configLocations);
        entityManagerFactory = ctx.getBean(EntityManagerFactory.class);
    }
}
```

此处我们使用了chapter7/applicationContext-resources.xml定义的“dataSource”数据源，通过ctx.getBean(EntityManagerFactory.class)获取EntityManagerFactory。

5、 通过EntityManagerFactory获取EntityManager进行创建和删除表：

java代码：

```
@Before
public void setUp() throws SQLException {
   //id自增主键从0开始
   String createTableSql = "create memory table test" + "(id int GENERATED BY DEFAULT AS IDE
   executeSql(createTableSql);
}
@After
public void tearDown() throws SQLException {
    String dropTableSql = "drop table test";
    executeSql(dropTableSql);
}

private void executeSql(String sql) throws SQLException {
  EntityManager em = entityManagerFactory.createEntityManager();
  beginTransaction(em);
  em.createNativeQuery(sql).executeUpdate();
  commitTransaction(em);
  closeEntityManager(em);
}
private void closeEntityManager(EntityManager em) {
  em.close();
}
private void rollbackTransacrion(EntityManager em) throws SQLException {
  if(em != null) {
     em.getTransaction().rollback();
  }
}
private void commitTransaction(EntityManager em) throws SQLException {
   em.getTransaction().commit();
}
```

```
private void beginTransaction(EntityManager em) throws SQLException {
   em.getTransaction().begin();
}
```

使用EntityManagerFactory创建EntityManager，然后通过EntityManager对象的createNativeQuery创建本地SQL执行创建和删除表。

6、 使用EntityManagerFactory获取EntityManager对象进行持久化数据：

java代码：

```
@Test
public void testFirst() throws SQLException {
    UserModel2 model = new UserModel2();
    model.setMyName("test");
    EntityManager em = null;
    try {
        em = entityManagerFactory.createEntityManager();
        beginTransaction(em);
        em.persist(model);
        commitTransaction(em);
    } catch (SQLException e) {
        rollbackTransacrion(em);
        throw e;
    } finally {
      closeEntityManager(em);
    }
}
```

使用**EntityManagerFactory**获取**EntityManager**进行操作，看到这还能忍受冗长的代码和事务管理吗？Spring同样提供JpaTemplate模板类来简化这些操作。

大家有没有注意到此处的模型对象能自动映射到数据库，这是因为Hibernate JPA实现默认自动扫描类路径中的@Entity注解类及*.hbm.xml映射文件，可以通过更改Hibernate JPA属性“hibernate.ejb.resource_scanner”，并指定org.hibernate.ejb.packaging.Scanner接口实现来定制新的扫描策略。

8.4.2 使用JpaTemplate

JpaTemplate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JpaCallback回调接口来允许更复杂的操作。

接下来示例一下JpaTemplate的使用：

1、修改Spring配置文件（chapter8/applicationContext-jpa.xml），添加JPA事务管理器：

java代码：

```
<bean id="txManager" class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>
```

- txManager：指定事务管理器，JPA使用JpaTransactionManager事务管理器实现，通过entityManagerFactory指定EntityManagerFactory；用于支持Java SE环境的**JPA扩展的持久化上下文（EXTENDED Persistence Context）**。

2、修改JPATest类，添加类变量ctx，用于后边使用其获取事务管理器使用：

java代码：

```
package cn.javass.spring.chapter8;
public class JPATest {
    private static EntityManagerFactory entityManagerFactory;
```

```
    private static ApplicationContext ctx;
    @BeforeClass
    public static void beforeClass() {
        String[] configLocations = new String[] {
                "classpath:chapter7/applicationContext-resources.xml",
                "classpath:chapter8/applicationContext-jpa.xml"};
        ctx = new ClassPathXmlApplicationContext(configLocations);
        entityManagerFactory = ctx.getBean(EntityManagerFactory.class);
}
}
```

3）JpaTemplate模板类使用：

java代码：

```
@Test
public void testJpaTemplate() {
final JpaTemplate jpaTemplate = new JpaTemplate(entityManagerFactory);
  final UserModel2 model = new UserModel2();
  model.setMyName("test1");
  PlatformTransactionManager txManager = ctx.getBean(PlatformTransactionManager.class);
  new TransactionTemplate(txManager).execute(
    new TransactionCallback<Void>() {
      @Override
      public Void doInTransaction(TransactionStatus status) {
        jpaTemplate.persist(model);
        return null;
      }
  });
  String COUNT_ALL = "select count(*) from UserModel";
  Number count = (Number) jpaTemplate.find(COUNT_ALL).get(0);
  Assert.assertEquals(1, count.intValue());
```

```
}
```

- **jpaTemplate**：可通过new JpaTemplate(entityManagerFactory)方式创建；
- **txManager**：通过ctx.getBean(PlatformTransactionManager.class)获取事务管理器；
- **TransactionTemplate**：通过new TransactionTemplate(txManager)创建事务模板对象，并通过execute方法执行TransactionCallback回调中的doInTransaction方法中定义需要执行的操作，从而将由模板类通过txManager事务管理器来进行事务管理，此处是调用jpaTemplate对象的persist方法进行持久化；
- **jpaTemplate.persist()**：根据JPA规范，在JPA扩展的持久化上下文，该操作必须运行在事务环境，还有persist()、 merge()、remove()操作也必须运行在事务环境；
- **jpaTemplate.find()**：根据JPA规范，该操作无需运行在事务环境，还有find()、getReference()、refresh()、detach()和查询操作都无需运行在事务环境。

此实例与Hibernate和Ibatis有所区别，通过JpaTemplate模板类进行如持久化等操作时必须有运行在事务环境中，否则可能抛出如下异常或警告：

- **“javax.persistence.TransactionRequiredException：Executing an update/delete query”**：表示没有事务支持，不能执行更新或删除操作；
- **警告“delaying identity-insert due to no transaction in progress”**：需要在日志系统启动debug模式才能看到，表示在无事务环境中无法进行持久化，而选择了延迟标识插入。

以上异常和警告是没有事务造成的，也是最让人困惑的问题，需要大家注意。

8.4.3 集成JPA及最佳实践

类似于JdbcDaoSupport类，Spring对JPA也提供了JpaDaoSupport类来支持一致的数据库访问。JpaDaoSupport也是DaoSupport实现：

接下来示例一下Spring集成JPA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JPA实现：

java代码：

```
package cn.javass.spring.chapter8.dao.jpa;
//省略import
@Transactional(propagation = Propagation.REQUIRED)
public class UserJpaDaoImpl extends JpaDaoSupport implements IUserDao {
    private static final String COUNT_ALL_JPAQL = "select count(*) from UserModel";
    @Override
    public void save(UserModel model) {
        getJpaTemplate().persist(model);
    }
    @Override
    public int countAll() {
        Number count =
           (Number) getJpaTemplate().find(COUNT_ALL_JPAQL).get(0);
        return count.intValue();
    }
}
```

此处注意首先JPA实现放在dao.jpa包里，其次实现类命名如UserJpaDaoImpl，即×××JpaDaoImpl，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。

另外在类上添加了**@Transactional**注解表示该类的所有方法将在调用时需要事务支持，propagation传播属性为Propagation.REQUIRED表示事务是必需的，如果执行该类的方法没有开启事务，将开启一个新的事务。

3、进行资源配置，使用resources/chapter7/applicationContext-resources.xml：

4、dao定义配置，在chapter8/applicationContext-jpa.xml中添加如下配置：

4.1、首先添加tx命名空间用于支持事务：

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:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
```

4.2、为**@Transactional**注解事务开启事务支持：

java代码：

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

只为类添加**@Transactional** 注解是不能支持事务的，需要通过<tx:annotation-driven>标签来开启事务支持，其中txManager属性指定事务管理器。

4.3、配置DAO Bean：

java代码：

```
<bean id="abstractDao" abstract="true">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>
<bean id="userDao"
      class="cn.javass.spring.chapter8.dao.jpa.UserJpaDaoImpl"
      parent="abstractDao"/>
```

首先定义抽象的abstractDao，其有一个entityManagerFactory属性，从而可以让继承的子类自动继承entityManagerFactory属性注入；然后定义userDao，且继承abstractDao，从而继承entityManagerFactory注入；我们在此给配置文件命名为applicationContext-jpa.xml表示JPA实现。

5、最后测试一下吧（cn.javass.spring.chapter8. JPATest）：

java代码：

```
@Test
public void testBestPractice() {
    String[] configLocations = new String[] {
            "classpath:chapter7/applicationContext-resources.xml",
            "classpath:chapter8/applicationContext-jpa.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-jpa.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。

还有与集成其他ORM框架不同的是JPA在进行持久化或更新数据库操作时需要事务支持。

8.4.4 Spring+JPA的CRUD

Spring+JPA CRUD（增删改查）也相当简单，让我们直接看具体示例吧：

java代码：

```
@Test
public void testCRUD() {
    PlatformTransactionManager txManager = ctx.getBean(PlatformTransactionManager.class);
    final JpaTemplate jpaTemplate = new JpaTemplate(entityManagerFactory);
    TransactionTemplate tansactionTemplate = new TransactionTemplate(txManager);
    tansactionTemplate.execute(new TransactionCallback<Void>() {
        @Override
        public Void doInTransaction(TransactionStatus status) {
            UserModel model = new UserModel();
            model.setMyName("test");
            //新增
            jpaTemplate.persist(model);
            //修改
            model.setMyName("test2");
            jpaTemplate.flush();//可选
            //查询
            String sql = "from UserModel where myName=?";
            List result = jpaTemplate.find(sql, "test2");
            Assert.assertEquals(1, result.size());
            //删除
```

```
            jpaTemplate.remove(model);
            return null;
        }
    });
}
```

- 对于增删改必须运行在事务环境，因此我们使用TransactionTemplate事务模板类来支持事务。
- 持久化：使用JpaTemplate 类的persist方法持久化模型对象；
- 更新：对于持久化状态的模型对象直接修改属性，调用flush方法即可更新到数据库，在一些场合时flush方法调用可选，如执行一个查询操作等，具体请参考相关文档；
- 查询：可以使用find方法执行JPA QL查询；
- 删除：使用remove方法删除一个持久化状态的模型对象。

Spring集成JPA进行增删改查也相当简单，但本文介绍的稍微复杂一点，因为牵扯到编程式事务，如果采用声明式事务将和集成Hibernate方式一样简洁。

1.6 【第九章】 Spring的事务 之 9.1 数据库事务概述 ——跟我学spring3

发表时间: 2012-03-04 关键字: spring, 事务

9.1 数据库事务概述

事务首先是一系列操作组成的工作单元，该工作单元内的操作是不可分割的，即要么所有操作都做，要么所有操作都不做，这就是事务。

事务必需满足ACID（原子性、一致性、隔离性和持久性）特性，缺一不可：

- **原子性（Atomicity）：**即事务是不可分割的最小工作单元，事务内的操作要么全做，要么全不做；
- **一致性（Consistency）：**在事务执行前数据库的数据处于正确的状态，而事务执行完成后数据库的数据还是处于正确的状态，即数据完整性约束没有被破坏；如银行转帐，A转帐给B，必须保证A的钱一定转给B，一定不会出现A的钱转了但B没收到，否则数据库的数据就处于不一致（不正确）的状态。
- **隔离性（Isolation）：**并发事务执行之间无影响，在一个事务内部的操作对其他事务是不产生影响，这需要事务隔离级别来指定隔离性；
- **持久性（Durability）：**事务一旦执行成功，它对数据库的数据的改变必须是永久的，不会因比如遇到系统故障或断电造成数据不一致或丢失。

在实际项目开发中数据库操作一般都是并发执行的，即有多个事务并发执行，并发执行就可能遇到问题，目前常见的问题如下：

- 丢失更新：两个事务同时更新一行数据，最后一个事务的更新会覆盖掉第一个事务的更新，从而导致第一个事务更新的数据丢失，这是由于没有加锁造成的；
- 脏读：一个事务看到了另一个事务未提交的更新数据；
- 不可重复读：在同一事务中，多次读取同一数据却返回不同的结果；也就是有其他事务更改了这些数据；
- 幻读：一个事务在执行过程中读取到了另一个事务已提交的插入数据；即在第一个事务开始时读取到一批数据，但此后另一个事务又插入了新数据并提交，此时第一个事务又读取这批数据但发现多了一条，即好像发生幻觉一样。

为了解决这些并发问题，需要通过数据库隔离级别来解决，在标准SQL规范中定义了四种隔离级别：

- **未提交读（Read Uncommitted）**：最低隔离级别，一个事务能读取到别的事务未提交的更新数据，很不安全，可能出现丢失更新、脏读、不可重复读、幻读；

- **提交读（Read Committed）**：一个事务能读取到别的事务提交的更新数据，不能看到未提交的更新数据，不可能可能出现丢失更新、脏读，但可能出现不可重复读、幻读；
- **可重复读（Repeatable Read）**：保证同一事务中先后执行的多次查询将返回同一结果，不受其他事务影响，可能可能出现丢失更新、脏读、不可重复读，但可能出现幻读；
- **序列化（Serializable）**：最高隔离级别，不允许事务并发执行，而必须串行化执行，最安全，不可能出现更新、脏读、不可重复读、幻读。

隔离级别越高，数据库事务并发执行性能越差，能处理的操作越少。因此在实际项目开发中为了考虑并发性能一般使用**提交读**隔离级别，它能避免丢失更新和脏读，尽管不可重复读和幻读不能避免，但可以在可能出现的场合使用**悲观锁或乐观锁**来解决这些问题。

9.1.1 事务类型

数据库事务类型有本地事务和分布式事务：

- 本地事务：就是普通事务，能保证单台数据库上的操作的ACID，被限定在一台数据库上；
- 分布式事务：涉及两个或多个数据库源的事务，即跨越多台同类或异类数据库的事务（由每台数据库的本地事务组成的），分布式事务旨在保证这些本地事务的所有操作的ACID，使事务可以跨越多台数据库；

Java事务类型有JDBC事务和JTA事务：

- JDBC事务：就是数据库事务类型中的本地事务，通过Connection对象的控制来管理事务；
- JTA事务：JTA指Java事务API(Java Transaction API)，是Java EE数据库事务规范， JTA只提供了事务管理接口，由应用程序服务器厂商（如WebSphere Application Server）提供实现，JTA事务比JDBC更强大，支持分布式事务。

Java EE事务类型有本地事务和全局事务：

- 本地事务：使用JDBC编程实现事务；
- 全局事务：由应用程序服务器提供，使用JTA事务；

按是否通过编程实现事务有声明式事务和编程式事务；

- 声明式事务： 通过注解或XML配置文件指定事务信息；
- 编程式事务：通过编写代码实现事务。

9.1.2 Spring提供的事务管理

Spring框架最核心功能之一就是事务管理，而且提供一致的事务管理抽象，这能帮助我们：

- 提供一致的编程式事务管理API，不管使用Spring JDBC框架还是集成第三方框架使用该API进行事务编程；
- 无侵入式的声明式事务支持。

Spring支持声明式事务和编程式事务事务类型。

1.7 【第九章】 Spring的事务 之 9.2 事务管理器 ——跟我学spring3

发表时间: 2012-03-05 关键字: spring

9.2.1 概述

Spring框架支持事务管理的核心是事务管理器抽象，对于不同的数据访问框架（如Hibernate）通过实现策略接口PlatformTransactionManager，从而能支持各种数据访问框架的事务管理，PlatformTransactionManager接口定义如下：

java代码：

```
public interface PlatformTransactionManager {
       TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionE
       void commit(TransactionStatus status) throws TransactionException;
       void rollback(TransactionStatus status) throws TransactionException;
}
```

- **getTransaction()**：返回一个已经激活的事务或创建一个新的事务（根据给定的TransactionDefinition类型参数定义的事务属性），返回的是TransactionStatus对象代表了当前事务的状态，其中该方法抛出TransactionException（未检查异常）表示事务由于某种原因失败。
- **commit()**：用于提交TransactionStatus参数代表的事务，具体语义请参考Spring Javadoc；
- **rollback()**：用于回滚TransactionStatus参数代表的事务，具体语义请参考Spring Javadoc。

TransactionDefinition接口定义如下：

java代码：

```
public interface TransactionDefinition {
       int getPropagationBehavior();
       int getIsolationLevel();
       int getTimeout();
       boolean isReadOnly();
```

```
        String getName();
}
```

- **getPropagationBehavior()**：返回定义的事务传播行为；
- **getIsolationLevel()**：返回定义的事务隔离级别；
- **getTimeout()**：返回定义的事务超时时间；
- **isReadOnly()**：返回定义的事务是否是只读的；
- **getName()**：返回定义的事务名字。

TransactionStatus接口定义如下：

java代码：

```
public interface TransactionStatus extends SavepointManager {
        boolean isNewTransaction();
        boolean hasSavepoint();
        void setRollbackOnly();
        boolean isRollbackOnly();
        void flush();
        boolean isCompleted();
}
```

- **isNewTransaction()：返回**当前事务状态是否是新事务；
- **hasSavepoint()：返回当前事务是否有保存点**；
- **setRollbackOnly()**：设置当前事务应该回滚；
- **isRollbackOnly(()**：返回当前事务是否应该回滚；
- **flush()**：用于刷新底层会话中的修改到数据库，一般用于刷新如Hibernate/JPA的会话，可能对如JDBC类型的事务无任何影响；
- **isCompleted():**当前事务否已经完成。

9.2.2 内置事务管理器实现

Spring提供了许多内置事务管理器实现：

- **DataSourceTransactionManager：**位于org.springframework.jdbc.datasource包中，数据源事务管理器，提供对单个javax.sql.DataSource事务管理，用于Spring JDBC抽象框架、iBATIS或MyBatis框架的事务管理；
- **JdoTransactionManager：**位于org.springframework.orm.jdo包中，提供对单个javax.jdo.PersistenceManagerFactory事务管理，用于集成JDO框架时的事务管理；
- **JpaTransactionManager：**位于org.springframework.orm.jpa包中，提供对单个javax.persistence.EntityManagerFactory事务支持，用于集成JPA实现框架时的事务管理；
- **HibernateTransactionManager：**位于org.springframework.orm.hibernate3包中，提供对单个org.hibernate.SessionFactory事务支持，用于集成Hibernate框架时的事务管理；该事务管理器只支持Hibernate3+版本，且Spring3.0+版本只支持Hibernate 3.2+版本；
- **JtaTransactionManager：**位于org.springframework.transaction.jta包中，提供对分布式事务管理的支持，并将事务管理委托给Java EE应用服务器事务管理器；
- **OC4JjtaTransactionManager：**位于org.springframework.transaction.jta包中，Spring提供的对OC4J10.1.3+应用服务器事务管理器的适配器，此适配器用于对应用服务器提供的高级事务的支持；
- **WebSphereUowTransactionManager：**位于org.springframework.transaction.jta包中，Spring提供的对WebSphere 6.0+应用服务器事务管理器的适配器，此适配器用于对应用服务器提供的高级事务的支持；
- **WebLogicJtaTransactionManager：**位于org.springframework.transaction.jta包中，Spring提供的对WebLogic 8.1+应用服务器事务管理器的适配器，此适配器用于对应用服务器提供的高级事务的支持。

Spring不仅提供这些事务管理器，还提供对如JMS事务管理的管理器等，Spring提供一致的事务抽象如图9-1所示。

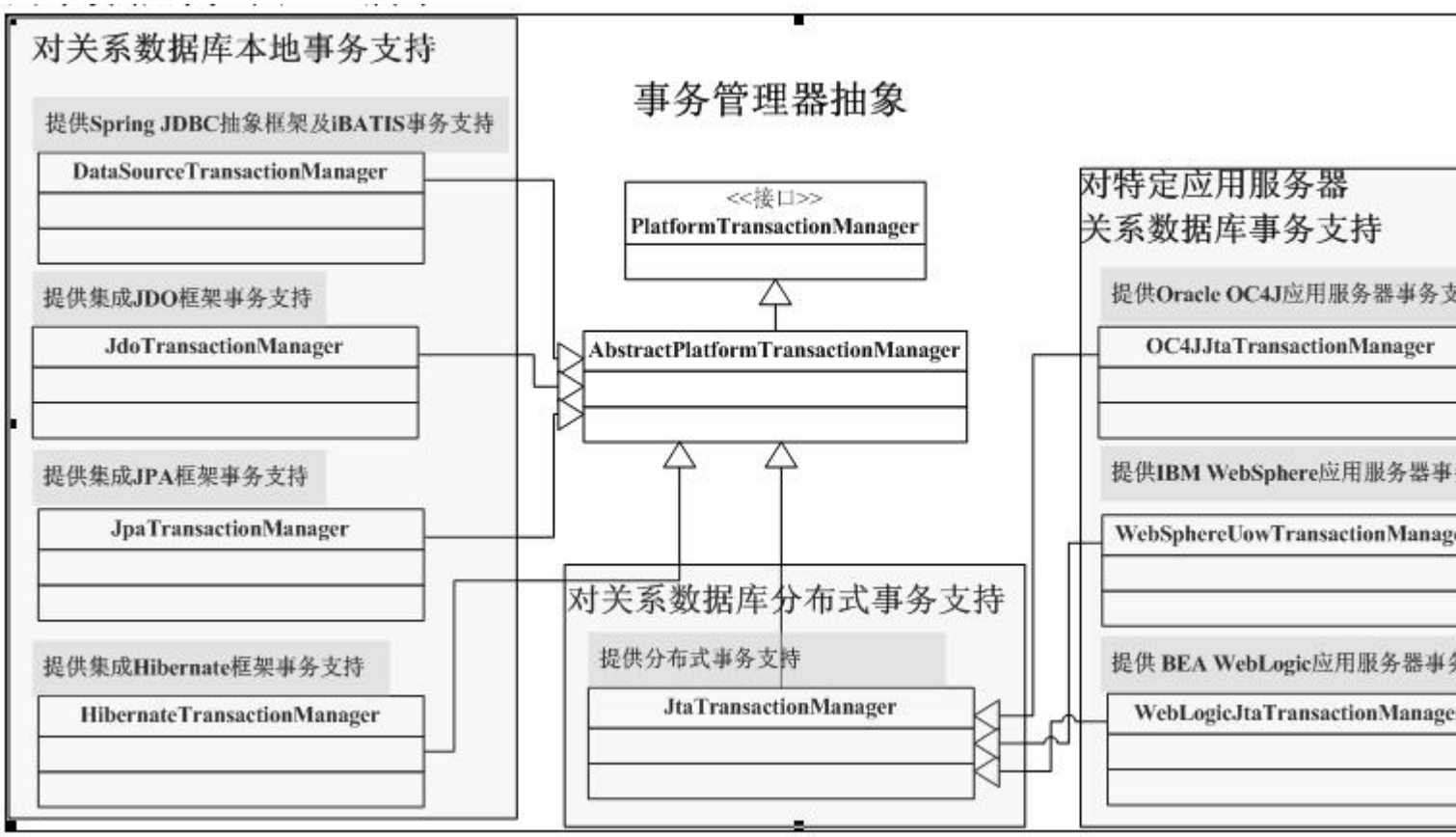


图9-1 Spring事务管理器

接下来让我们学习一下如何在Spring配置文件中定义事务管理器：

一、声明对本地事务的支持：

a)JDBC及iBATIS、MyBatis框架事务管理器

java代码：

```
<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager
    <property name="dataSource" ref="dataSource"/>
</bean>
```

通过dataSource属性指定需要事务管理的单个javax.sql.DataSource对象。

b)Jdo事务管理器

java代码：

```
<bean id="txManager" class="org.springframework.orm.jdo.JdoTransactionManager">
    <property name="persistenceManagerFactory" ref="persistenceManagerFactory"/>
</bean>
```

通过persistenceManagerFactory属性指定需要事务管理的javax.jdo.PersistenceManagerFactory对象。

c)Jpa事务管理器

java代码：

```
<bean id="txManager" class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>
```

通过entityManagerFactory属性指定需要事务管理的javax.persistence.EntityManagerFactory对象。

还需要为entityManagerFactory对象指定jpaDialect属性，该属性所对应的对象指定了如何获取连接对象、开启事务、关闭事务等事务管理相关的行为。

java代码：

```
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManag
        ……
```

```
        <property name="jpaDialect" ref="jpaDialect"/>
</bean>
<bean id="jpaDialect" class="org.springframework.orm.jpa.vendor.HibernateJpaDialect"/>
```

d)Hibernate事务管理器

java代码：

```
<bean id="txManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

通过entityManagerFactory属性指定需要事务管理的org.hibernate.SessionFactory对象。

二、Spring对全局事务的支持：

a)Jta事务管理器

java代码：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="
```

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

  <jee:jndi-lookup id="dataSource" jndi-name="jdbc/test"/>
  <bean id="txManager" class="org.springframework.transaction.jta.JtaTransactionManager">
    <property name="transactionManagerName" value=" java:comp/TransactionManager"/>
  </bean>
</beans>
```

“dataSource”Bean表示从JNDI中获取的数据源，而txManager是JTA事务管理器，其中属性transactionManagerName指定了JTA事务管理器的JNDI名字，从而将事务管理委托给该事务管理器。

这只是最简单的配置方式，更复杂的形式请参考Spring Javadoc。

在此我们再介绍两个不依赖于应用服务器的开源JTA事务实现：JOTM和Atomikos Transactions Essentials。

- **JOTM：**即基于Java开放事务管理器（Java Open Transaction Manager），实现JTA规范，能够运行在非应用服务器环境中，Web容器或独立Java SE环境，官网地址: http://jotm.objectweb.org/。
- **Atomikos Transactions Essentials：**其为Atomikos开发的事务管理器，该产品属于开源产品，另外还一个商业的Extreme Transactions。官网地址为：http://www.atomikos.com。

对于以上JTA事务管理器使用，本文作者只是做演示使用，如果在实际项目中需要不依赖于应用服务器的JTA事务支持，需详细测试并选择合适的。

在本文中将使用Atomikos Transactions Essentials来进行演示JTA事务使用，由于Atomikos对hsqldb分布式支持不是很好，在Atomikos官网中列出如下兼容的数据库：Oracle、Informix、FirstSQL、DB2、MySQL、SQLServer、Sybase，这不代表其他数据库不支持，而是Atomikos团队没完全测试，在此作者决定使用derby内存数据库来演示JTA分布式事务。

1、首先准备jar包：

1.1、准备derby数据jar包，到下载的spring-framework-3.0.5.RELEASE-dependencies.zip中拷贝如下jar包：

com.springsource.org.apache.derby-10.5.1000001.764942.jar

1．2、准备Atomikos Transactions Essentials 对JTA事务支持的JTA包，此处使用AtomikosTransactionsEssentials3.5.5版本，到官网下载AtomikosTransactionsEssentials-3.5.5.zip，拷贝如下jar包到类路径：

atomikos-util.jar

transactions-api.jar

transactions-essentials-all.jar

transactions-jdbc.jar

transactions-jta.jar

transactions.jar

将如上jar包放在lib\atomikos目录下，并添加到类路径中。

2、接下来看一下在Spring中如何配置AtomikosTransactionsEssentials JTA事务：

2.1、配置分布式数据源：

java代码：

```
<bean id="dataSource1" class="com.atomikos.jdbc.AtomikosDataSourceBean" init-method="init" d
    <property name="uniqueResourceName" value="jdbc/test1"/>
    <property name="xaDataSourceClassName" value="org.apache.derby.jdbc.EmbeddedXADataSource
    <property name="poolSize" value="5"/>
    <property name="xaProperties">
        <props>
            <prop key="databaseName">test1</prop>
            <prop key="createDatabase">create</prop>
        </props>
    </property>
</bean>

<bean id="dataSource2" class="com.atomikos.jdbc.AtomikosDataSourceBean"
       init-method="init" destroy-method="close">
    ……
</bean>
```

在此我们配置两个分布式数据源：使用com.atomikos.jdbc.AtomikosDataSourceBean来配置AtomikosTransactionsEssentials分布式数据源：

- uniqueResourceName表示唯一资源名，如有多个数据源不可重复；
- xaDataSourceClassName是具体分布式数据源厂商实现；
- poolSize是数据连接池大小；
- xaProperties属性指定具体厂商数据库属性，如databaseName指定数据库名，createDatabase表示启动derby内嵌数据库时创建databaseName指定的数据库。

在此我们还有定义了一个“dataSource2”Bean，其属性和“DataSource1”除以下不一样其他完全一样：

- uniqueResourceName：因为不能重复，因此此处使用jdbc/test2；
- databaseName：我们需要指定两个数据库，因此此处我们指定为test2。

2.2、配置事务管理器：

java代码：

```
<bean id="atomikosTransactionManager" class = "com.atomikos.icatch.jta.UserTransactionManage
      <property name="forceShutdown" value="true"/>
</bean>
<bean id="atomikosUserTransaction" class="com.atomikos.icatch.jta.UserTransactionImp">  </be

<bean id="transactionManager" class="org.springframework.transaction.jta.JtaTransactionManag
    <property name="transactionManager">
        <ref bean="atomikosTransactionManager"/>
    </property>
    <property name="userTransaction">
        <ref bean="atomikosUserTransaction"/>
    </property>
</bean>
```

- atomikosTransactionManager：定义了AtomikosTransactionsEssentials事务管理器；
- atomikosUserTransaction：定义UserTransaction，该Bean是线程安全的；
- transactionManager：定义Spring事务管理器，transactionManager属性指定外部事务管理器（真正的事务管理者），使用userTransaction指定UserTransaction，该属性一般用于本地JTA实现，如果使用应用服务器事务管理器，该属性将自动从JNDI获取。

配置完毕，是不是也挺简单的，但是如果确实需要使用JTA事务，请首先选择应用服务器事务管理器，本示例不适合生产环境，如果非要运用到生产环境，可以考虑购买AtomikosTransactionsEssentials商业支持。

b)特定服务器事务管理器

Spring还提供了对特定应用服务器事务管理器集成的支持，目前提供对IBM WebSphere、BEA WebLogic、 Oracle OC4J应用服务器高级事务的支持，具体使用请参考Spring Javadoc。

现在我们已经学习如何配置事务管理器了，但是只有事务管理器Spring能自动进行事务管理吗？当然不能了，这需要我们来控制，目前Spring支持两种事务管理方式：编程式和声明式事务管理。接下来先看一下如何进行编程式事务管理吧。

原创内容，转载请注明出处【http://sishuok.com/forum/blogPost/list/0/2503.html】

1.8 【第九章】 Spring的事务 之 9.3 编程式事务 ——跟我学spring3

发表时间: 2012-03-06 关键字: spring

9.3 编程式事务

9.3.1 编程式事务概述

所谓编程式事务指的是通过编码方式实现事务，即类似于JDBC编程实现事务管理。

Spring框架提供一致的事务抽象，因此对于JDBC还是JTA事务都是采用相同的API进行编程。

java代码：

```
Connection conn = null;
UserTransaction tx = null;
try {
    tx = getUserTransaction();                          //1.获取事务
    tx.begin();                                        //2.开启JTA事务
    conn = getDataSource().getConnection();             //3.获取JDBC
    //4.声明SQL
    String sql = "select * from INFORMATION_SCHEMA.SYSTEM_TABLES";
    PreparedStatement pstmt = conn.prepareStatement(sql);//5.预编译SQL
    ResultSet rs = pstmt.executeQuery();                //6.执行SQL
    process(rs);                                       //7.处理结果集
    closeResultSet(rs);                                //8.释放结果集
    tx.commit();                                       //7.提交事务
} catch (Exception e) {
    tx.rollback();                                     //8.回滚事务
    throw e;
} finally {
   conn.close();                                      //关闭连接
}
```

此处可以看到使用UserTransaction而不是Connection连接进行控制事务，从而对于JDBC事务和JTA事务是采用不同API进行编程控制的，并且JTA和JDBC事务管理的异常也是不一样的。

具体如何使用JTA编程进行事务管理请参考cn.javass.spring.chapter9包下的TranditionalTransactionTest类。

而在Spring中将采用一致的事务抽象进行控制和一致的异常控制，即面向PlatformTransactionManager接口编程来控制事务。

9.3.1 Spring对编程式事务的支持

Spring中的事务分为物理事务和逻辑事务；

- **物理事务**：就是底层数据库提供的事务支持，如JDBC或JTA提供的事务；
- **逻辑事务**：是Spring管理的事务，不同于物理事务，逻辑事务提供更丰富的控制，而且如果想得到Spring事务管理的好处，必须使用逻辑事务，因此在Spring中如果没特别强调一般就是逻辑事务；

逻辑事务即支持非常低级别的控制，也有高级别解决方案：

- **低级别解决方案：**

工具类：使用工具类获取连接（会话）和释放连接（会话），如使用org.springframework.jdbc.datasource包中的 DataSourceUtils 类来获取和释放具有逻辑事务功能的连接。当然对集成第三方ORM框架也提供了类似的工具类，如对Hibernate提供了SessionFactoryUtils工具类，JPA的EntityManagerFactoryUtils等，其他工具类都是使用类似***Utils命名；

java代码：

```
//获取具有Spring事务（逻辑事务）管理功能的连接
DataSourceUtils. getConnection(DataSource dataSource)
//释放具有Spring事务（逻辑事务）管理功能的连接
DataSourceUtils. releaseConnection(Connection con, DataSource dataSource)
```

TransactionAwareDataSourceProxy：使用该数据源代理类包装需要Spring事务管理支持的数据源，该包装类必须位于最外层，主要用于遗留项目中可能直接使用数据源获取连接和释放连接支持或希望在Spring中进行混合使用各种持久化框架时使用，其内部实际使用 DataSourceUtils 工具类获取和释放真正连接；

java代码：

```
<!--使用该方式包装数据源，必须在最外层，targetDataSource 知道目标数据源-->
<bean id="dataSourceProxy"
class="org.springframework.jdbc.datasource.
TransactionAwareDataSourceProxy">
    <property name="targetDataSource" ref="dataSource"/>
</bean>
```

通过如上方式包装数据源后，可以在项目中使用物理事务编码的方式来获得逻辑事务的支持，即支持直接从DataSource获取连接和释放连接，且这些连接自动支持Spring逻辑事务；

- **高级别解决方案：**

模板类：使用Spring提供的模板类，如JdbcTemplate、HibernateTemplate和JpaTemplate模板类等，而这些模板类内部其实是使用了低级别解决方案中的工具类来管理连接或会话；

Spring提供两种编程式事务支持：直接使用PlatformTransactionManager实现和使用TransactionTemplate模板类，用于支持逻辑事务管理。

如果采用编程式事务推荐使用TransactionTemplate模板类和高级别解决方案。

9.3.3 使用PlatformTransactionManager

首先让我们看下如何使用PlatformTransactionManager实现来进行事务管理：

1、数据源定义，此处使用第7章的配置文件，即“chapter7/ applicationContext-resources.xml”文件。

2、事务管理器定义（chapter9/applicationContext-jdbc.xml）：

java代码：

```
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransacti
    <property name="dataSource" ref="dataSource"/>
</bean>
```

3、 准备测试环境：

3.1、首先准备测试时使用的SQL：

java代码：

```
package cn.javass.spring.chapter9;
//省略import
public class TransactionTest {
    //id自增主键从0开始
    private static final String CREATE_TABLE_SQL = "create table test" +
    "(id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY, " +
    "name varchar(100))";
    private static final String DROP_TABLE_SQL = "drop table test";
    private static final String INSERT_SQL = "insert into test(name) values(?)";
    private static final String COUNT_SQL = "select count(*) from test";
    ……
}
```

3.2、初始化Spring容器

java代码：

```
package cn.javass.spring.chapter9;
//省略import
public class TransactionTest {
```

```
    private static ApplicationContext ctx;
    private static PlatformTransactionManager txManager;
    private static DataSource dataSource;
    private static JdbcTemplate jdbcTemplate;
    ……
    @BeforeClass
    public static void setUpClass() {
        String[] configLocations = new String[] {
                "classpath:chapter7/applicationContext-resources.xml",
                "classpath:chapter9/applicationContext-jdbc.xml"};
        ctx = new ClassPathXmlApplicationContext(configLocations);
        txManager = ctx.getBean(PlatformTransactionManager.class);
        dataSource = ctx.getBean(DataSource.class);
        jdbcTemplate = new JdbcTemplate(dataSource);
    }
    ……
}
```

3.3、使用高级别方案JdbcTemplate来进行事务管理器测试：

java代码：

```
@Test
public void testPlatformTransactionManager() {
    DefaultTransactionDefinition def = new DefaultTransactionDefinition();
    def.setIsolationLevel(TransactionDefinition.ISOLATION_READ_COMMITTED);
    def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);
    TransactionStatus status = txManager.getTransaction(def);
    jdbcTemplate.execute(CREATE_TABLE_SQL);
    try {
        jdbcTemplate.update(INSERT_SQL, "test");
        txManager.commit(status);
    } catch (RuntimeException e) {
```

```
        txManager.rollback(status);
    }
    jdbcTemplate.execute(DROP_TABLE_SQL);
}
```

- **DefaultTransactionDefinition**：事务定义，定义如隔离级别、传播行为等，即在本示例中隔离级别为ISOLATION_READ_COMMITTED（提交读），传播行为为PROPAGATION_REQUIRED（必须有事务支持，即如果当前没有事务，就新建一个事务，如果已经存在一个事务中，就加入到这个事务中）。
- **TransactionStatus**：事务状态类，通过PlatformTransactionManager的getTransaction方法根据事务定义获取；获取事务状态后，Spring根据传播行为来决定如何开启事务；
- JdbcTemplate：通过JdbcTemplate对象执行相应的SQL操作，且自动享受到事务支持，注意事务是线程绑定的，因此事务管理器可以运行在多线程环境；
- txManager.commit(status)：提交status对象绑定的事务；
- txManager.rollback(status)：当遇到异常时回滚status对象绑定的事务。

3.4、使用低级别解决方案来进行事务管理器测试：

java代码：

```
@Test
public void testPlatformTransactionManagerForLowLevel1() {
DefaultTransactionDefinition def = new DefaultTransactionDefinition();      def.setIsolation
  TransactionStatus status = txManager.getTransaction(def);
  Connection conn = DataSourceUtils.getConnection(dataSource);
  try {
      conn.prepareStatement(CREATE_TABLE_SQL).execute();
      PreparedStatement pstmt = conn.prepareStatement(INSERT_SQL);
      pstmt.setString(1, "test");
      pstmt.execute();
      conn.prepareStatement(DROP_TABLE_SQL).execute();
      txManager.commit(status);
  } catch (Exception e) {
      status.setRollbackOnly();
```

```
        txManager.rollback(status);
    } finally {
        DataSourceUtils.releaseConnection(conn, dataSource);
}
}
```

低级别方案中使用DataSourceUtils获取和释放连接，使用txManager开管理事务，而且面向JDBC编程，比起模板类方式来繁琐和复杂的多，因此不推荐使用该方式。在此就不介绍数据源代理类使用了，需要请参考platformTransactionManagerForLowLevelTest2测试方法。

到此事务管理是不是还很繁琐？必须手工提交或回滚事务，有没有更好的解决方案呢？Spring提供了TransactionTemplate模板类来简化事务管理。

9.3.4 使用TransactionTemplate

TransactionTemplate模板类用于简化事务管理，事务管理由模板类定义，而具体操作需要通过TransactionCallback回调接口或TransactionCallbackWithoutResult回调接口指定，通过调用模板类的参数类型为TransactionCallback或TransactionCallbackWithoutResult的execute方法来自动享受事务管理。

TransactionTemplate模板类使用的回调接口：

- **TransactionCallback**：通过实现该接口的“T doInTransaction(TransactionStatus status)”方法来定义需要事务管理的操作代码；
- **TransactionCallbackWithoutResult**：继承TransactionCallback接口，提供“void doInTransactionWithoutResult(TransactionStatus status)”便利接口用于方便那些不需要返回值的事务操作代码。

1、接下来演示一下TransactionTemplate模板类如何使用：

java代码：

```
@Test
public void testTransactionTemplate() {//位于TransactionTest类中
  jdbcTemplate.execute(CREATE_TABLE_SQL);
  TransactionTemplate transactionTemplate = new TransactionTemplate(txManager);
  transactionTemplate.setIsolationLevel(TransactionDefinition.ISOLATION_READ_COMMITTED);
  transactionTemplate.execute(new TransactionCallbackWithoutResult() {
      @Override
      protected void doInTransactionWithoutResult(TransactionStatus status) {
         jdbcTemplate.update(INSERT_SQL, "test");
  }});
  jdbcTemplate.execute(DROP_TABLE_SQL);
}
```

- **TransactionTemplate** ：通过new TransactionTemplate(txManager)创建事务模板类，其中构造器参数为PlatformTransactionManager实现，并通过其相应方法设置事务定义，如事务隔离级别、传播行为等，此处未指定传播行为，其默认为PROPAGATION_REQUIRED；
- TransactionCallbackWithoutResult：此处使用不带返回的回调实现，其doInTransactionWithoutResult方法实现中定义了需要事务管理的操作；
- transactionTemplate.execute()：通过该方法执行需要事务管理的回调。

这样是不是简单多了，没有事务管理代码，而是由模板类来完成事务管理。

注：对于抛出Exception类型的异常且需要回滚时，需要捕获异常并通过调用status对象的setRollbackOnly()方法告知事务管理器当前事务需要回滚，如下所示：

java代码：

```
try {
    //业务操作
} catch (Exception e) { //可使用具体业务异常代替
    status.setRollbackOnly();
}
```

2、前边已经演示了JDBC事务管理，接下来演示一下JTA分布式事务管理：

java代码：

```
@Test
public void testJtaTransactionTemplate() {
    String[] configLocations = new String[] {
      "classpath:chapter9/applicationContext-jta-derby.xml"};
    ctx = new ClassPathXmlApplicationContext(configLocations);
    final PlatformTransactionManager jtaTXManager = ctx.getBean(PlatformTransactionManager.c
    final DataSource derbyDataSource1 = ctx.getBean("dataSource1", DataSource.class);
    final DataSource derbyDataSource2 = ctx.getBean("dataSource2", DataSource.class);
    final JdbcTemplate jdbcTemplate1 = new JdbcTemplate(derbyDataSource1);
    final JdbcTemplate jdbcTemplate2 = new JdbcTemplate(derbyDataSource2);
    TransactionTemplate transactionTemplate = new TransactionTemplate(jtaTXManager);
    transactionTemplate.setIsolationLevel(TransactionDefinition.ISOLATION_READ_COMMITTED);
    jdbcTemplate1.update(CREATE_TABLE_SQL);
    int originalCount = jdbcTemplate1.queryForInt(COUNT_SQL);
    try {
        transactionTemplate.execute(new TransactionCallbackWithoutResult() {
            @Override
            protected void doInTransactionWithoutResult(TransactionStatus status) {
                jdbcTemplate1.update(INSERT_SQL, "test");
               //因为数据库2没有创建数据库表因此会回滚事务
              jdbcTemplate2.update(INSERT_SQL, "test");
          }});
    } catch (RuntimeException e) {
        int count = jdbcTemplate1.queryForInt(COUNT_SQL);
        Assert.assertEquals(originalCount, count);
    }
    jdbcTemplate1.update(DROP_TABLE_SQL);
}
```

- **配置文件**：使用此前定义的chapter9/applicationContext-jta-derby.xml；

- **jtaTXManager**： JTA事务管理器；
- **derbyDataSource1和derbyDataSource2**：derby数据源1和derby数据源2；
- **jdbcTemplate1和jdbcTemplate2**：分别使用derbyDataSource1和derbyDataSource2构造的JDBC模板类；
- **transactionTemplate**：使用jtaTXManager事务管理器的事务管理模板类，其隔离级别为提交读，传播行为默认为PROPAGATION_REQUIRED（必须有事务支持，即如果当前没有事务，就新建一个事务，如果已经存在一个事务中，就加入到这个事务中）；
- **jdbcTemplate1.update(CREATE_TABLE_SQL)**：此处只有derbyDataSource1所代表的数据库创建了“test”表，而derbyDataSource2所代表的数据库没有此表；
- **TransactionCallbackWithoutResult**：在此接口实现中定义了需要事务支持的操作：

jdbcTemplate1.update(INSERT_SQL, "test")：表示向数据库1中的test表中插入数据；

jdbcTemplate2.update(INSERT_SQL, "test")：表示向数据库2中的test表中插入数据，但数据库2没有此表将抛出异常，且JTA分布式事务将回滚；

- **Assert.assertEquals(originalCount, count)**：用来验证事务是否回滚，验证结果返回为true，说明分布式事务回滚了。

到此我们已经会使用PlatformTransactionManager和TransactionTemplate进行简单事务处理了，那如何应用到实际项目中去呢？接下来让我们看下如何在实际项目中应用Spring管理事务。

接下来看一下如何将Spring管理事务应用到实际项目中，为简化演示，此处只定义最简单的模型对象和不完整的Dao层接口和Service层接口：

1、 首先定义项目中的模型对象，本示例使用用户模型和用户地址模型：

模型对象一般放在项目中的model包里。

java代码：

```
package cn.javass.spring.chapter9.model;
public class UserModel {
```

```
    private int id;
    private String name;
    private AddressModel address;
    //省略getter和setter
}
```

java代码：

```
package cn.javass.spring.chapter9.model;
public class AddressModel {
    private int id;
    private String province;
    private String city;
    privateString street;
    private int userId;
    //省略getter和setter
}
```

2.1、定义Dao层接口：

java代码：

```
package cn.javass.spring.chapter9.service;
import cn.javass.spring.chapter9.model.UserModel;
public interface IUserService {
    public void save(UserModel user);
    public int countAll();
}
```

java代码：

```
package cn.javass.spring.chapter9.service;
import cn.javass.spring.chapter9.model.AddressModel;
public interface IAddressService {
    public void save(AddressModel address);
    public int countAll();
}
```

2.2、定义Dao层实现：

java代码：

```
package cn.javass.spring.chapter9.dao.jdbc;
//省略import，注意model要引用chapter包里的
public class UserJdbcDaoImpl extends NamedParameterJdbcDaoSupport implements IUserDao {
    private final String INSERT_SQL = "insert into user(name) values(:name)";
    private final String COUNT_ALL_SQL = "select count(*) from user";
    @Override
    public void save(UserModel user) {
        KeyHolder generatedKeyHolder = new GeneratedKeyHolder();
        SqlParameterSource paramSource = new BeanPropertySqlParameterSource(user);
        getNamedParameterJdbcTemplate().update(INSERT_SQL, paramSource, generatedKeyHolder);
        user.setId(generatedKeyHolder.getKey().intValue());
    }
    @Override
    public int countAll() {
       return getJdbcTemplate().queryForInt(COUNT_ALL_SQL);
    }
}
```

java代码：

```
package cn.javass.spring.chapter9.dao.jdbc;
//省略import，注意model要引用chapter包里的
public class AddressJdbcDaoImpl extends NamedParameterJdbcDaoSupport implements IAddressDao {
    private final String INSERT_SQL = "insert into address(province, city, street, user_id)"
    private final String COUNT_ALL_SQL = "select count(*) from address";
    @Override
    public void save(AddressModel address) {
        KeyHolder generatedKeyHolder = new GeneratedKeyHolder();
        SqlParameterSource paramSource = new BeanPropertySqlParameterSource(address);
        getNamedParameterJdbcTemplate().update(INSERT_SQL, paramSource, generatedKeyHolder);
        address.setId(generatedKeyHolder.getKey().intValue());
}
    @Override
    public int countAll() {
        return getJdbcTemplate().queryForInt(COUNT_ALL_SQL);
    }
}
```

3.1、定义Service层接口，一般使用“I×××Service”命名：

java代码：

```
package cn.javass.spring.chapter9.service;
import cn.javass.spring.chapter9.model.UserModel;
public interface IUserService {
    public void save(UserModel user);
    public int countAll();
}
```

```
package cn.javass.spring.chapter9.service;
import cn.javass.spring.chapter9.model.AddressModel;
public interface IAddressService {
    public void save(AddressModel address);
    public int countAll();
}
```

3.2、定义Service层实现，一般使用“×××ServiceImpl”或“×××Service”命名：

java代码：

```
package cn.javass.spring.chapter9.service.impl;
//省略import，注意model要引用chapter包里的
public class AddressServiceImpl implements IAddressService {
    private IAddressDao addressDao;
    private PlatformTransactionManager txManager;
    public void setAddressDao(IAddressDao addressDao) {
        this.addressDao = addressDao;
    }
    public void setTxManager(PlatformTransactionManager txManager) {
        this.txManager = txManager;
    }
    @Override
    public void save(final AddressModel address) {
        TransactionTemplate transactionTemplate = TransactionTemplateUtils.getDefaultTransac
        transactionTemplate.execute(new TransactionCallbackWithoutResult() {
           @Override
           protected void doInTransactionWithoutResult(TransactionStatus status) {
                addressDao.save(address);
           }
        });
```

```
    }
    @Override
    public int countAll() {
        return addressDao.countAll();
    }
}
```

java代码：

```
package cn.javass.spring.chapter9.service.impl;
//省略import，注意model要引用chapter包里的
public class UserServiceImpl implements IUserService {
    private IUserDao userDao;
    private IAddressService addressService;
    private PlatformTransactionManager txManager;
    public void setUserDao(IUserDao userDao) {
        this.userDao = userDao;
    }
    public void setTxManager(PlatformTransactionManager txManager) {
        this.txManager = txManager;
    }
    public void setAddressService(IAddressService addressService) {
        this.addressService = addressService;
    }
    @Override
    public void save(final UserModel user) {
        TransactionTemplate transactionTemplate =
            TransactionTemplateUtils.getDefaultTransactionTemplate(txManager);
        transactionTemplate.execute(new TransactionCallbackWithoutResult() {
        @Override
```

```
            protected void doInTransactionWithoutResult(TransactionStatus status) {
                 userDao.save(user);
                 user.getAddress().setUserId(user.getId());
                 addressService.save(user.getAddress());
            }
        });
    }
    @Override
    public int countAll() {
        return userDao.countAll();
    }
}
```

Service实现中需要Spring事务管理的部分应该使用TransactionTemplate模板类来包装执行。

4、定义TransactionTemplateUtils，用于简化获取TransactionTemplate模板类，工具类一般放在util包中：

java代码：

```
package cn.javass.spring.chapter9.util;
//省略import
public class TransactionTemplateUtils {
    public static TransactionTemplate getTransactionTemplate(
            PlatformTransactionManager txManager,
            int propagationBehavior,
            int isolationLevel) {

        TransactionTemplate transactionTemplate = new TransactionTemplate(txManager);
        transactionTemplate.setPropagationBehavior(propagationBehavior);
        transactionTemplate.setIsolationLevel(isolationLevel);
```

```
        return transactionTemplate;
    }

    public static TransactionTemplate getDefaultTransactionTemplate(PlatformTransactionManag
        return getTransactionTemplate(
                txManager,
                TransactionDefinition.PROPAGATION_REQUIRED,
                TransactionDefinition.ISOLATION_READ_COMMITTED);
    }
}
```

getDefaultTransactionTemplate用于获取传播行为为PROPAGATION_REQUIRED，隔离级别为ISOLATION_READ_COMMITTED的模板类。

5、数据源配置定义，此处使用第7章的配置文件，即“chapter7/ applicationContext-resources.xml”文件。

6、Dao层配置定义（chapter9/dao/applicationContext-jdbc.xml）：

java代码：

```
<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager
    <property name="dataSource" ref="dataSource"/>
</bean>
<bean id="abstractDao" abstract="true">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

java代码：

```
<bean id="userDao" class="cn.javass.spring.chapter9.dao.jdbc.UserJdbcDaoImpl" parent="abstra
<bean id="addressDao" class="cn.javass.spring.chapter9.dao.jdbc.AddressJdbcDaoImpl" parent="
```

7、Service层配置定义（chapter9/service/applicationContext-service.xml）：

java代码：

```
<bean id="userService" class="cn.javass.spring.chapter9.service.impl.UserServiceImpl">
    <property name="userDao" ref="userDao"/>
    <property name="txManager" ref="txManager"/>
    <property name="addressService" ref="addressService"/>
</bean>
<bean id="addressService" class="cn.javass.spring.chapter9.service.impl.AddressServiceImpl">
    <property name="addressDao" ref="addressDao"/>
    <property name="txManager" ref="txManager"/>
</bean>
```

8、准备测试需要的表创建语句，在TransactionTest测试类中添加如下静态变量：

java代码：

```
private static final String CREATE_USER_TABLE_SQL =
    "create table user" +
    "(id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY, " +
    "name varchar(100))";
```

```
private static final String DROP_USER_TABLE_SQL = "drop table user";

private static final String CREATE_ADDRESS_TABLE_SQL =
    "create table address" +
    "(id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY, " +
    "province varchar(100), city varchar(100), street varchar(100), user_id int)";
private static final String DROP_ADDRESS_TABLE_SQL = "drop table address";
```

9、 测试一下吧：

java代码：

```
@Test
    public void testServiceTransaction() {
        String[] configLocations = new String[] {
        "classpath:chapter7/applicationContext-resources.xml",
        "classpath:chapter9/dao/applicationContext-jdbc.xml",
        "classpath:chapter9/service/applicationContext-service.xml"};
        ApplicationContext ctx2 = new ClassPathXmlApplicationContext(configLocations);

        DataSource dataSource2 = ctx2.getBean(DataSource.class);
        JdbcTemplate jdbcTemplate2 = new JdbcTemplate(dataSource2);
        jdbcTemplate2.update(CREATE_USER_TABLE_SQL);
        jdbcTemplate2.update(CREATE_ADDRESS_TABLE_SQL);

        IUserService userService = ctx2.getBean("userService", IUserService.class);
        IAddressService addressService = ctx2.getBean("addressService", IAddressService.clas
        UserModel user = createDefaultUserModel();
        userService.save(user);
        Assert.assertEquals(1, userService.countAll());
        Assert.assertEquals(1, addressService.countAll());
        jdbcTemplate2.update(DROP_USER_TABLE_SQL);
        jdbcTemplate2.update(DROP_ADDRESS_TABLE_SQL);
```

```
}
private UserModel createDefaultUserModel() {
    UserModel user = new UserModel();
    user.setName("test");
    AddressModel address = new AddressModel();
    address.setProvince("beijing");
    address.setCity("beijing");
    address.setStreet("haidian");
    user.setAddress(address);
    return user;
}
```

从Spring容器中获取Service层对象，调用Service层对象持久化对象，大家有没有注意到Spring事务全部在Service层定义，为什么会在Service层定义，而不是Dao层定义呢？这是因为在服务层可能牵扯到业务逻辑，而每个业务逻辑可能调用多个Dao层方法，为保证这些操作的原子性，必须在Service层定义事务。

还有大家有没有注意到如果Service层的事务管理相当令人头疼，而且是侵入式的，有没有办法消除这些冗长的事务管理代码呢？这就需要Spring声明式事务支持，下一节将介绍无侵入式的声明式事务。

可能大家对事务定义中的各种属性有点困惑，如传播行为到底干什么用的？接下来将详细讲解一下事务属性。

9.3.5 事务属性

事务属性通过TransactionDefinition接口实现定义，主要有事务隔离级别、事务传播行为、事务超时时间、事务是否只读。

Spring提供TransactionDefinition接口默认实现DefaultTransactionDefinition，可以通过该实现类指定这些事务属性。

- **事务隔离级别：**用来解决并发事务时出现的问题，其使用TransactionDefinition中的静态变量来指定：

ISOLATION_DEFAULT：默认隔离级别，即使用底层数据库默认的隔离级别；

ISOLATION_READ_UNCOMMITTED：未提交读；

ISOLATION_READ_COMMITTED：提交读，一般情况下我们使用这个；

ISOLATION_REPEATABLE_READ：可重复读；

ISOLATION_SERIALIZABLE：序列化。

可以使用DefaultTransactionDefinition类的setIsolationLevel(TransactionDefinition.ISOLATION_READ_COMMITTED)来指定隔离级别，其中此处表示隔离级别为提交读，也可以使用或setIsolationLevelName(“ISOLATION_READ_COMMITTED”)方式指定，其中参数就是隔离级别静态变量的名字，但不推荐这种方式。

- **事务传播行为：**Spring管理的事务是逻辑事务，而且物理事务和逻辑事务最大差别就在于事务传播行为，事务传播行为用于指定在多个事务方法间调用时，事务是如何在这些方法间传播的，Spring共支持7种传播行为：

Required：必须有逻辑事务，否则新建一个事务，使用PROPAGATION_REQUIRED指定，表示如果当前存在一个逻辑事务，则加入该逻辑事务，否则将新建一个逻辑事务，如图9-2和9-3所示；

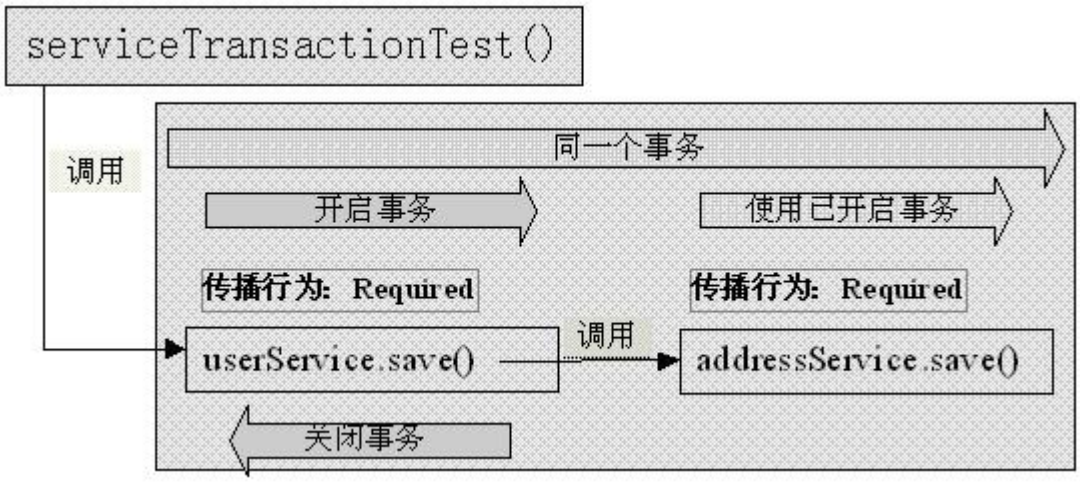


图9-2 Required传播行为

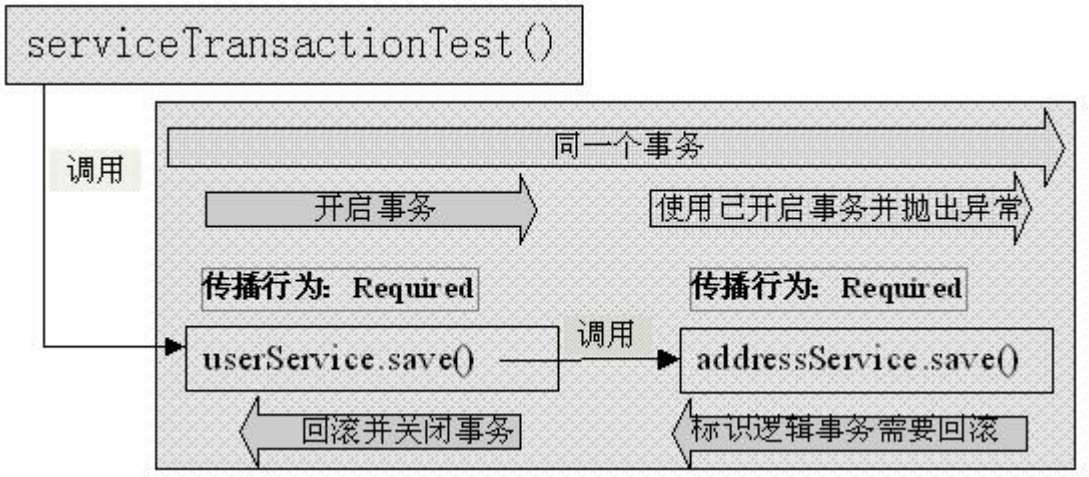


图9-3 Required传播行为抛出异常情况

在前边示例中就是使用的Required传播行为：

一、在调用userService对象的save方法时，此方法用的是Required传播行为且此时Spring事务管理器发现还没开启逻辑事务，因此Spring管理器觉得开启逻辑事务，

二、在此逻辑事务中调用了addressService对象的save方法，而在save方法中发现同样用的是Required传播行为，因此使用该已经存在的逻辑事务；

三、在返回到addressService对象的save方法，当事务模板类执行完毕，此时提交并关闭事务。

因此userService对象的save方法和addressService的save方法属于同一个物理事务，如果发生回滚，则两者都回滚。

接下来测试一下该传播行为如何执行吧：

一、正确提交测试，如上一节的测试，在此不再演示；

二、回滚测试，修改AddressServiceImpl的save方法片段：

java代码：

```
addressDao.save(address);
```

为

java代码：

```
addressDao.save(address);
//抛出异常，将标识当前事务需要回滚
throw new RuntimeException();
```

二、修改UserServiceImpl的save方法片段：

java代码：

```
addressService.save(user.getAddress());
```

为

java代码：

```
try {
    addressService.save(user.getAddress());//将在同一个事务内执行
} catch (RuntimeException e) {
}
```

如果该业务方法执行时事务被标记为回滚，则不管在此是否捕获该异常都将发生回滚，因为处于同一逻辑事务。

三、修改测试方法片段:

java代码：

```
userService.save(user);
Assert.assertEquals(1, userService.countAll());
Assert.assertEquals(1, addressService.countAll());
```

为如下形式：

java代码：

```
try {
    userService.save(user);
    Assert.fail();
} catch (RuntimeException e) {
}
Assert.assertEquals(0, userService.countAll());
Assert.assertEquals(0, addressService.countAll());
```

Assert断言中countAll方法都返回0，说明事务回滚了，即说明两个业务方法属于同一个物理事务，即使在userService对象的save方法中将异常捕获，由于addressService对象的save方法抛出异常，即事务管理器将自动标识当前事务为需要回滚。

RequiresNew：创建新的逻辑事务，使用PROPAGATION_REQUIRES_NEW指定，表示每次都创建新的逻辑事务（物理事务也是不同的）如图9-4和9-5所示：

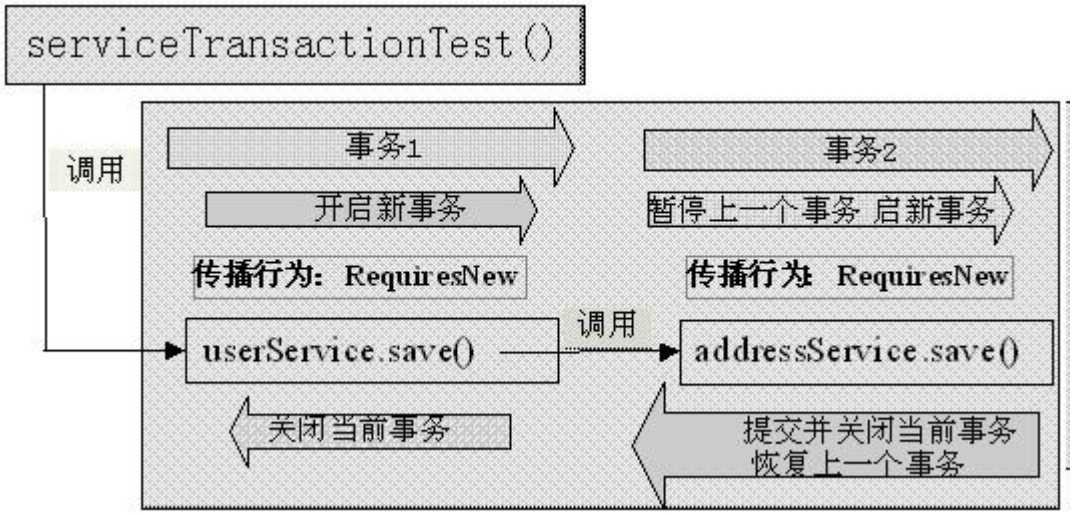


图9-4 RequiresNew传播行为

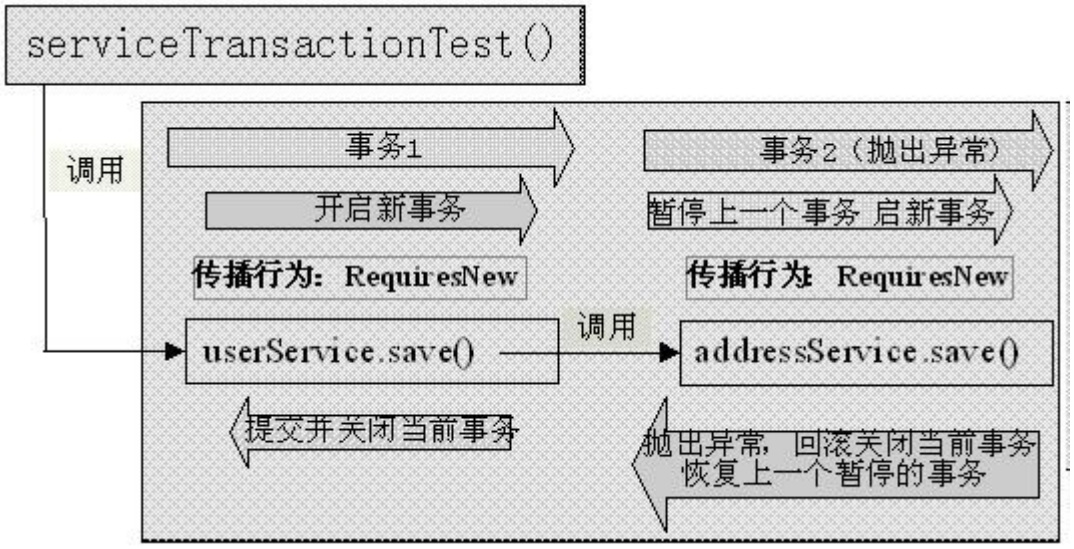


图9-5 RequiresNew传播行为并抛出异常

接下来测试一个该传播行为如何执行吧：

1、将如下获取事务模板方式

java代码：

```
TransactionTemplate transactionTemplate = TransactionTemplateUtils.getDefaultTransactionTemp
```

替换为如下形式，表示传播行为为RequiresNew：

java代码：

```
TransactionTemplate transactionTemplate = TransactionTemplateUtils.getTransactionTemplate(
        txManager,
        TransactionDefinition.PROPAGATION_REQUIRES_NEW,
        TransactionDefinition.ISOLATION_READ_COMMITTED);
```

2、执行如下测试，发现执行结果是正确的：

java代码：

```
userService.save(user);
Assert.assertEquals(1, userService.countAll());
Assert.assertEquals(1, addressService.countAll());
```

3、修改UserServiceImpl的save方法片段

java代码：

```
userDao.save(user);
user.getAddress().setUserId(user.getId());
addressService.save(user.getAddress());
```

为如下形式，表示userServiceImpl类的save方法将发生回滚，而AddressServiceImpl类的方法由于在抛出异常前执行，将成功提交事务到数据库：

java代码：

```
userDao.save(user);
user.getAddress().setUserId(user.getId());
addressService.save(user.getAddress());
throw new RuntimeException();
```

4、修改测试方法片段：

java代码：

```
userService.save(user);
Assert.assertEquals(1, userService.countAll());
Assert.assertEquals(1, addressService.countAll());
```

为如下形式：

java代码：

```
try {
    userService.save(user);
    Assert.fail();
} catch (RuntimeException e) {
}
Assert.assertEquals(0, userService.countAll());
Assert.assertEquals(1, addressService.countAll());
```

Assert断言中调用userService对象countAll方法返回0，说明该逻辑事务作用域回滚，而调用addressService对象的countAll方法返回1，说明该逻辑事务作用域正确提交。因此这是不正确的行为，因为用户和地址应该是一一对应的，不应该发生这种情况，因此此处正确的传播行为应该是Required。

该传播行为执行流程（正确提交情况）：

一、当执行userService对象的save方法时，由于传播行为是RequiresNew，因此创建一个新的逻辑事务（物理事务也是不同的）；

二、当执行到addressService对象的save方法时，由于传播行为是RequiresNew，因此首先暂停上一个逻辑事务并创建一个新的逻辑事务（物理事务也是不同的）；

三、addressService对象的save方法执行完毕后，提交逻辑事务（并提交物理事务）并重新恢复上一个逻辑事务，继续执行userService对象的save方法内的操作；

四、最后userService对象的save方法执行完毕，提交逻辑事务（并提交物理事务）；

五、userService对象的save方法和addressService对象的save方法不属于同一个逻辑事务且也不属于同一个物理事务。

Supports：支持当前事务，使用PROPAGATION_SUPPORTS指定，指如果当前存在逻辑事务，就加入到该逻辑事务，如果当前没有逻辑事务，就以非事务方式执行，如图9-6和9-7所示：

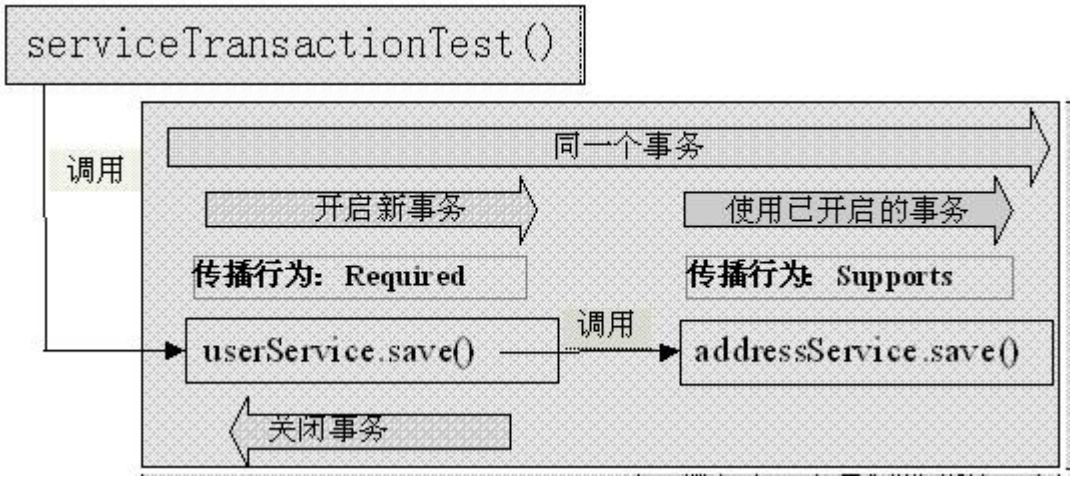


图9-6 Required+Supports传播行为

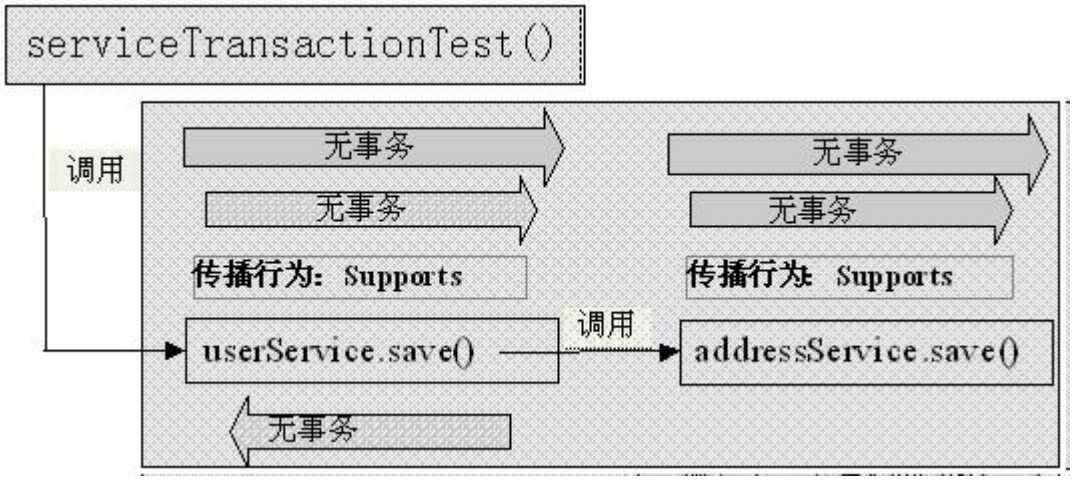


图9-7 Supports+Supports传播行为

NotSupported：不支持事务，如果当前存在事务则暂停该事务，使用PROPAGATION_NOT_SUPPORTED指定，即以非事务方式执行，如果当前存在逻辑事务，就把当前事务暂停，以非事务方式执行，如图9-8和9-9所示：

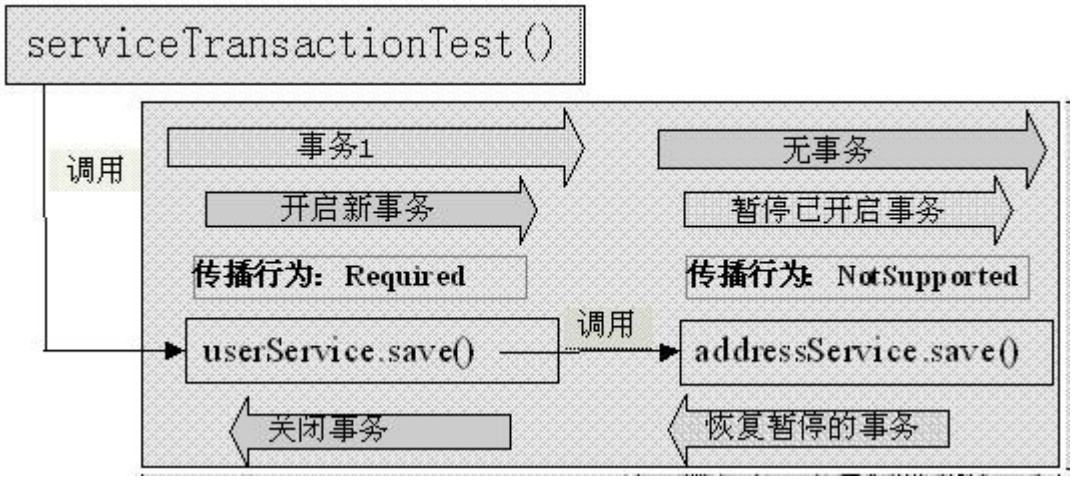


图9-8 Required+NotSupported传播行为

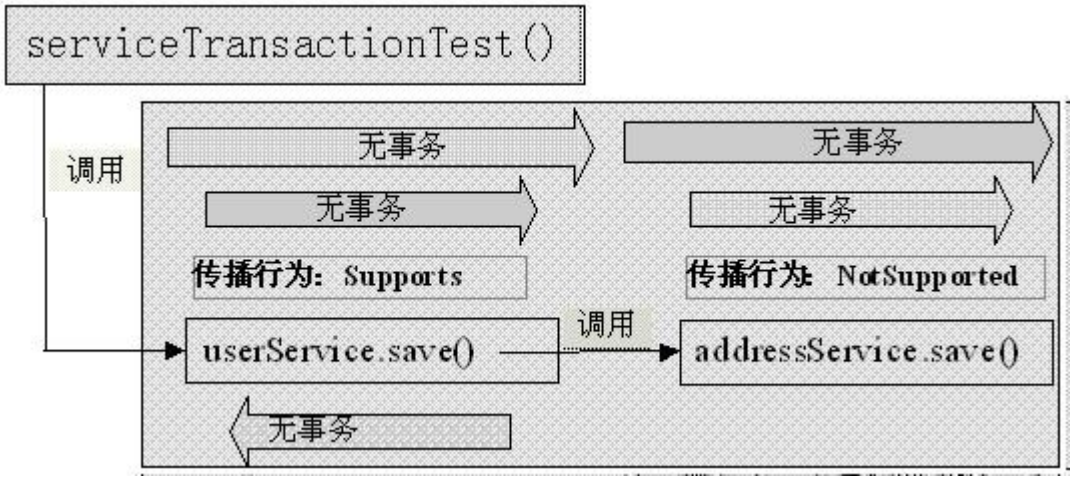


图9-9 Supports+NotSupported传播行为

Mandatory：必须有事务，否则抛出异常，使用PROPAGATION_MANDATORY指定，使用当前事务执行，如果当前没有事务，则抛出异常（IllegalTransactionStateException），如图9-10和9-11所示：

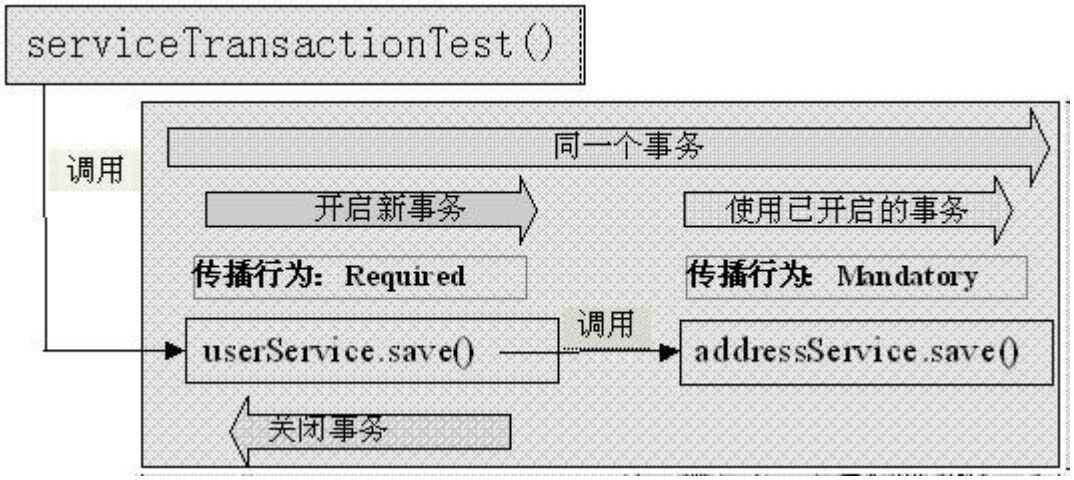


图9-10 Required+Mandatory传播行为

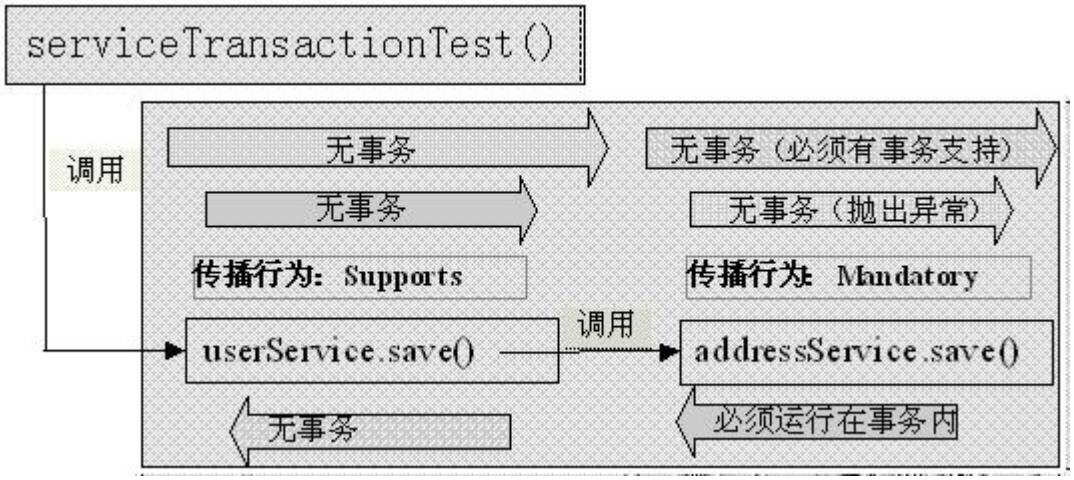


图9-11 Supports+Mandatory传播行为

Never：不支持事务，如果当前存在是事务则抛出异常，使用PROPAGATION_NEVER指定，即以非事务方式执行，如果当前存在事务，则抛出异常（IllegalTransactionStateException），如图9-12和9-13所示：

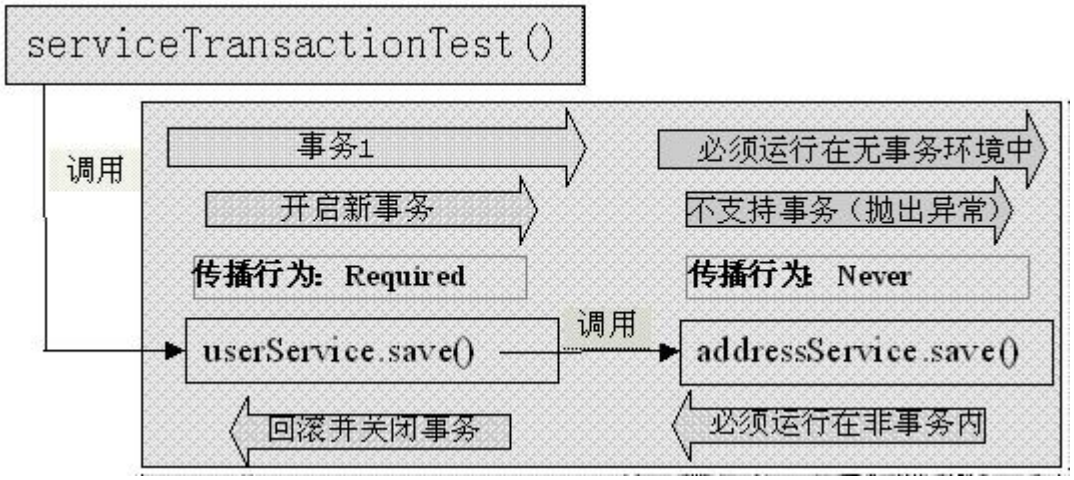


图9-12 Required+Never传播行为

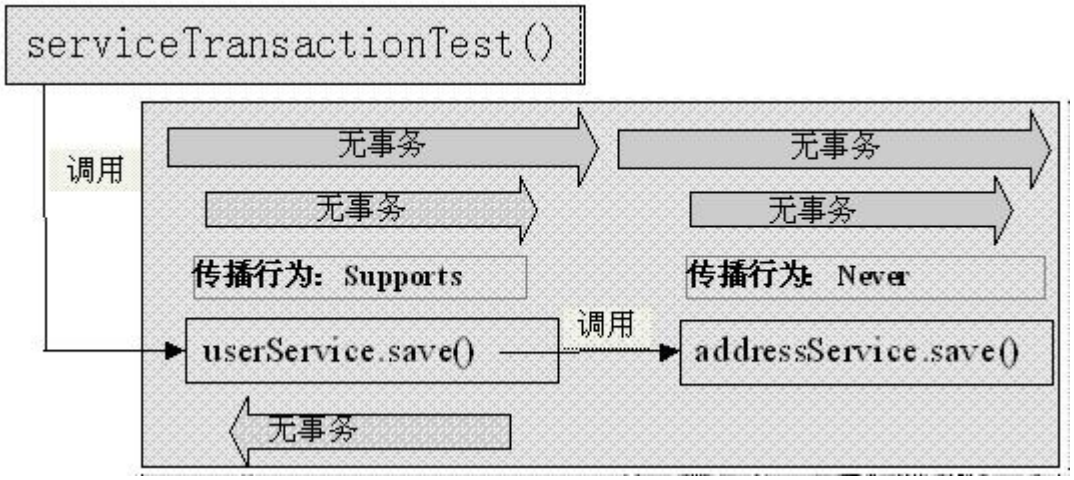


图9-13 Supports+Never传播行为

Nested：嵌套事务支持，使用PROPAGATION_NESTED指定，如果当前存在事务，则在嵌套事务内执行，如果当前不存在事务，则创建一个新的事务，嵌套事务使用数据库中的保存点来实现，即嵌套事务回滚不影响外部事务，但外部事务回滚将导致嵌套事务回滚，如图9-14和9-15所示：

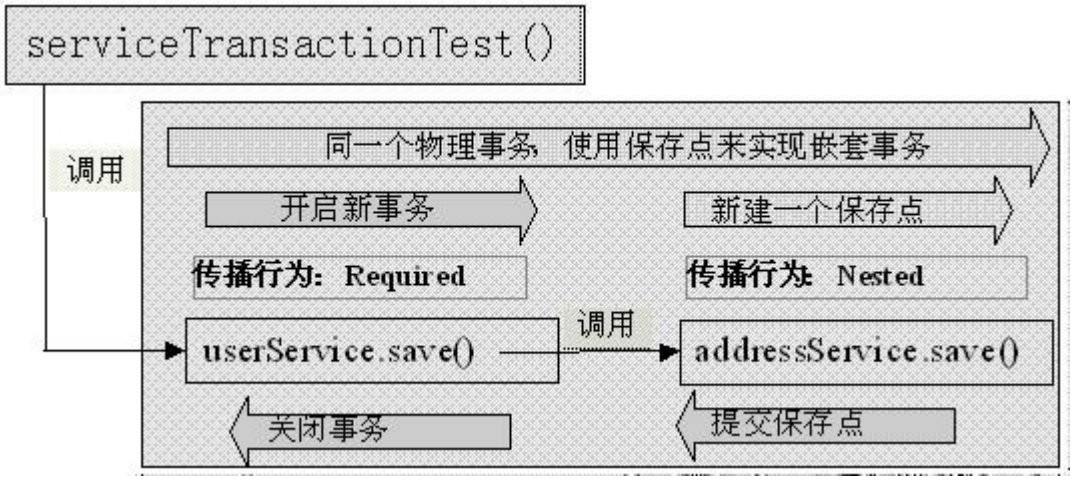


图9-14 Required+Nested传播行为

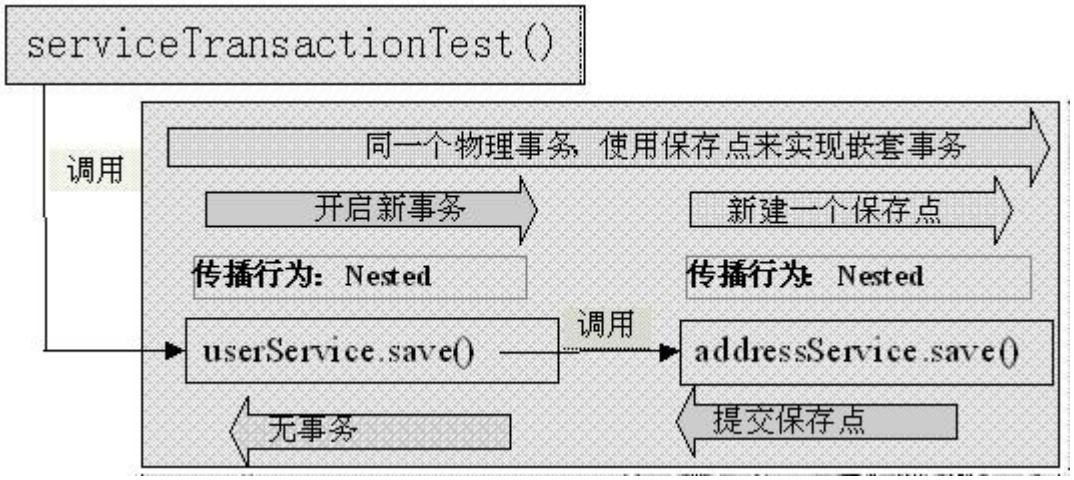


图9-15 Nested+Nested传播行为

Nested和RequiresNew的区别：

1、 RequiresNew每次都创建新的独立的物理事务，而Nested只有一个物理事务；

2、 Nested嵌套事务回滚或提交不会导致外部事务回滚或提交，但外部事务回滚将导致嵌套事务回滚，而RequiresNew由于都是全新的事务，所以之间是无关联的；

3、 Nested使用JDBC 3的保存点实现，即如果使用低版本驱动将导致不支持嵌套事务。

使用嵌套事务，必须确保具体事务管理器实现的nestedTransactionAllowed属性为true，否则不支持嵌套事务，如DataSourceTransactionManager默认支持，而HibernateTransactionManager默认不支持，需要我们来开启。

对于事务传播行为我们只演示了Required和RequiresNew，其他传播行为类似，如果对这些事务传播行为不太会使用，请参考chapter9包下的TransactionTest测试类中的testPropagation方法，方法内有详细示例。

- **事务超时**：设置事务的超时时间，单位为秒，默认为-1表示使用底层事务的超时时间；

使用如setTimeout(100)来设置超时时间，如果事务超时将抛出org.springframework.transaction.TransactionTimedOutException异常并将当前事务标记为应该回滚，即超时后事务被自动回滚；

可以使用具体事务管理器实现的defaultTimeout属性设置默认的事务超时时间，如DataSourceTransactionManager. setDefaultTimeout(10)。

- **事务只读**：将事务标识为只读，只读事务不修改任何数据；

对于JDBC只是简单的将连接设置为只读模式，对于更新将抛出异常；

而对于一些其他ORM框架有一些优化作用，如在Hibernate中，Spring事务管理器将执行“session.setFlushMode(FlushMode.MANUAL)”即指定Hibernate会话在只读事务模式下不用尝试检测和同步持久对象的状态的更新。

如果使用设置具体事务管理的validateExistingTransaction属性为true（默认false），将确保整个事务传播链都是只读或都不是只读，如图9-16是正确的事务只读设置，而图9-17是错误的事务只读设置：

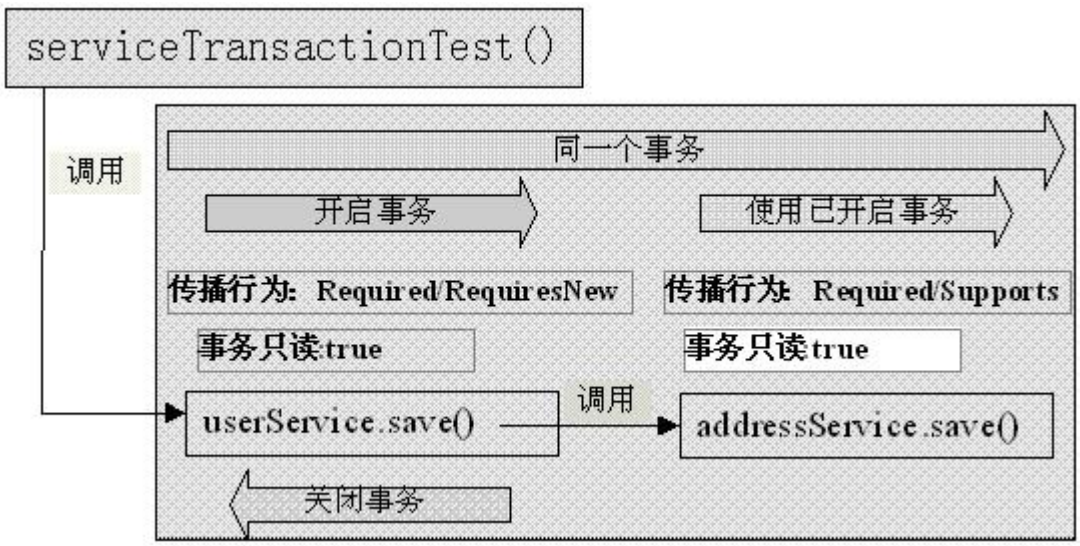


图9-16 正确的事务只读设置

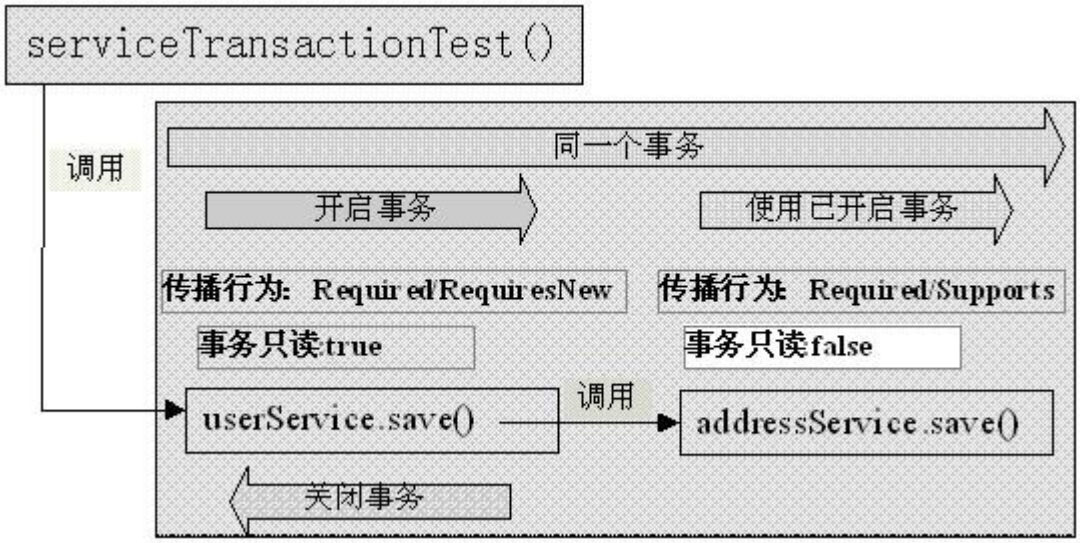


图9-17 错误的事务只读设置

如图10-17，对于错误的事务只读设置将抛出IllegalTransactionStateException异常，并伴随“Participating transaction with definition [……] is not marked as read-only……”信息，表示参与的事务只读属性设置错误。

大家有没有感觉到编程式实现事务管理是不是很繁琐冗长，重复，而且是侵入式的，因此发展到这Spring决定使用配置方式实现事务管理。

9.3.6 配置方式实现事务管理

在Spring2.x之前为了解决编程式事务管理的各种不好问题，Spring提出使用配置方式实现事务管理，配置方式利用代理机制实现，即使有TransactionProxyFactoryBean类来为目标类代理事务管理。

接下来演示一下具体使用吧：

1、重新定义业务类实现，在业务类中无需显示的事务管理代码：

java代码：

```
package cn.javass.spring.chapter9.service.impl;
//省略import
public class ConfigAddressServiceImpl implements IAddressService {
    private IAddressDao addressDao;
    public void setAddressDao(IAddressDao addressDao) {
        this.addressDao = addressDao;
    }
    @Override
    public void save(final AddressModel address) {
        addressDao.save(address);
    }
    //countAll方法实现不变
}
```

java代码：

```
package cn.javass.spring.chapter9.service.impl;
//省略import
public class ConfigUserServiceImpl implements IUserService {
    private IUserDao userDao;
    private IAddressService addressService;
    public void setUserDao(IUserDao userDao) {
        this.userDao = userDao;
    }
    public void setAddressService(IAddressService addressService) {
        this.addressService = addressService;
    }
    @Override
    public void save(final UserModel user) {
        userDao.save(user);
        user.getAddress().setUserId(user.getId());
        addressService.save(user.getAddress());
```

```
    }
    //countAll方法实现不变
}
```

从以上业务类中可以看出，没有事务管理的代码，即没有侵入式的代码。

2、在chapter9/service/applicationContext-service.xml配置文件中添加如下配置：

2.1、首先添加目标类定义：

java代码：

```
<bean id="targetUserService" class="cn.javass.spring.chapter9.service.impl.ConfigUserService
    <property name="userDao" ref="userDao"/>
    <property name="addressService" ref="targetAddressService"/>
</bean>
<bean id="targetAddressService" class="cn.javass.spring.chapter9.service.impl.ConfigAddressS
    <property name="addressDao" ref="addressDao"/>
</bean>
```

2.2、配置TransactionProxyFactoryBean类：

java代码：

```
<bean id="transactionProxyParent" class="org.springframework.transaction.interceptor.Transac
    <property name="transactionManager" ref="txManager"/>
    <property name="transactionAttributes">
```

```
    <props>
            <prop key="save*">
                    PROPAGATION_REQUIRED,
                    ISOLATION_READ_COMMITTED,
                    timeout_10,
                    -Exception,
                    +NoRollBackException
            </prop>
            <prop key="*">
                    PROPAGATION_REQUIRED,
                    ISOLATION_READ_COMMITTED,
                    readOnly
            </prop>
        </props>
</property>
</bean>
```

- **TransactionProxyFactoryBean**：用于为目标业务类创建代理的Bean；
- **abstract="true"**：表示该Bean是抽象的，用于去除重复配置；
- **transactionManager**：事务管理器定义；
- **transactionAttributes**：表示事务属性定义：
- **PROPAGATION_REQUIRED,ISOLATION_READ_COMMITTED,timeout_10,-Exception,+NoRollBackException**：事务属性定义，Required传播行为，提交读隔离级别，事务超时时间为10秒，将对所有Exception异常回滚，而对于抛出NoRollBackException异常将不发生回滚而是提交；
- **PROPAGATION_REQUIRED,ISOLATION_READ_COMMITTED,readOnly**：事务属性定义，Required传播行为，提交读隔离级别，事务是只读的，且只对默认的RuntimeException异常回滚；
- **<prop key="save*">**：表示将代理以save开头的方法，即当执行到该方法时会为该方法根据事务属性配置来开启/关闭事务；
- **<prop key="*">**：表示将代理其他所有方法，但需要注意代理方式，默认是JDK代理，只有public方法能代理；

注：事务属性的传播行为和隔离级别使用TransactionDefinition静态变量名指定；事务超时使用“timeout_超时时间”指定，事务只读使用“readOnly”指定，需要回滚的异常使用“-异常”指定，不需要回滚的异常使用“+异常”指定，默认只对RuntimeException异常回滚。

需要特别注意“-异常”和“+异常”中“异常”只是真实异常的部分名，内部使用如下方式判断：

java代码：

```
//真实抛出的异常.name.indexOf(配置中指定的需要回滚/不回滚的异常名)
exceptionClass.getName().indexOf(this.exceptionName)
```

因此异常定义时需要特别注意，配置中定义的异常只是真实异常的部分名。

2.3、定义代理Bean：

java代码：

```
<bean id="proxyUserService" parent="transactionProxyParent">
    <property name="target" ref="targetUserService"/>
</bean>
<bean id="proxyAddressService" parent="transactionProxyParent">
    <property name="target" ref="targetAddressService"/>
</bean>
```

代理Bean通过集成抽象Bean“transactionProxyParent”，并通过target属性设置目标Bean，在实际使用中应该使用该代理Bean。

3、修改测试方法并测试该配置方式是否好用：

将TransactionTest 类的testServiceTransaction测试方法拷贝一份命名为testConfigTransaction：

并在testConfigTransaction测试方法内将：

java代码：

```
IUserService userService =
ctx2.getBean("userService", IUserService.class);
IAddressService addressService =
ctx2.getBean("addressService", IAddressService.class);
```

替换为：

java代码：

```
IUserService userService =
ctx2.getBean("proxyUserService ", IUserService.class);
IAddressService addressService =
ctx2.getBean("proxyAddressService ", IAddressService.class);
```

4、执行测试，测试正常通过，说明该方式能正常工作，当调用save方法时将匹配到“**<prop key="save*">**”定义，而countAll将匹配到“**<prop key="save*">**”定义，底层代理会应用相应定义中的事务属性来创建或关闭事务。

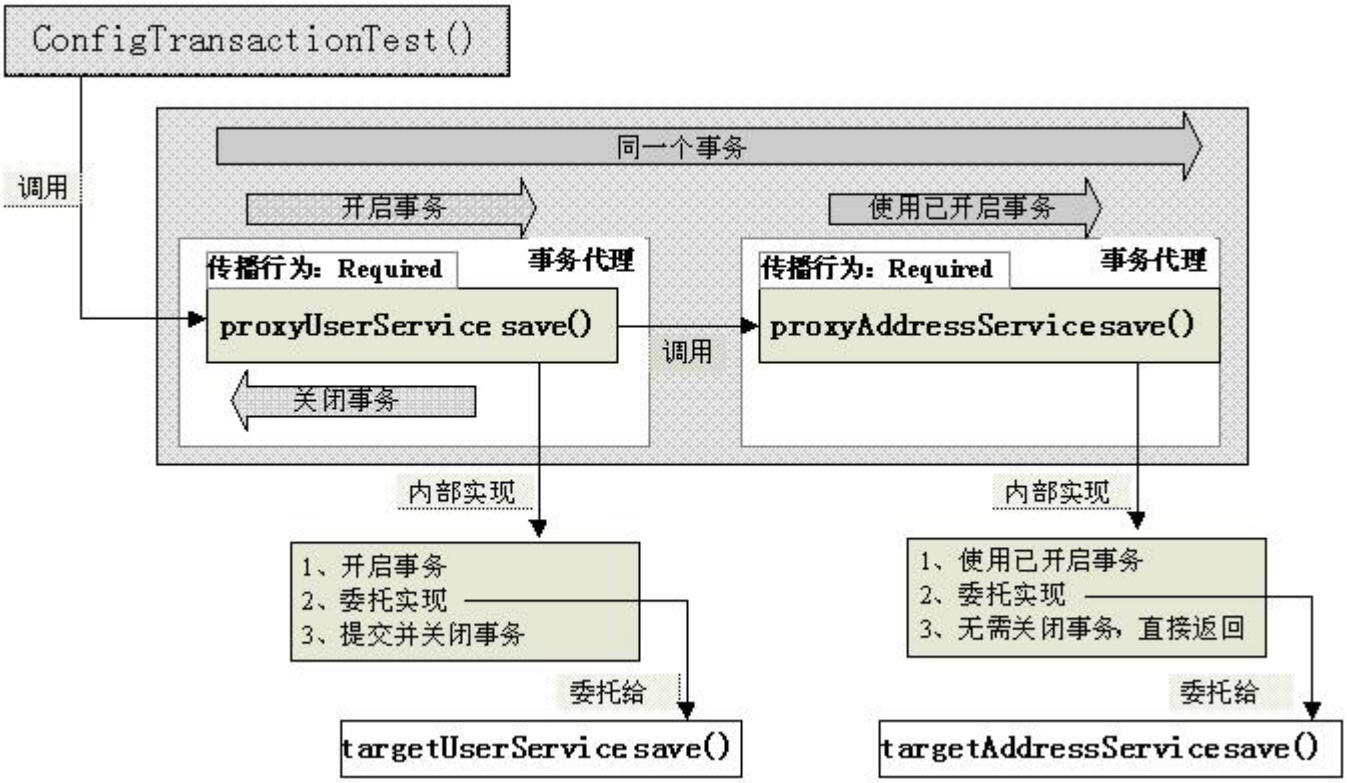


图9-18 代理方式实现事务管理

如图9-18，代理方式实现事务管理只是将硬编码的事务管理代码转移到代理中去由代理实现，在代理中实现事务管理。

注：**在代理模式下，默认只有通过代理对象调用的方法才能应用相应的事务属性，而在目标方法内的“自我调用”是不会应用相应的事务属性的，即被调用方法不会应用相应的事务属性，而是使用调用方法的事务属性。**

如图9-19所示，在目标对象targetUserService的save方法内调用事务方法“this.otherTransactionMethod()”将不会应用配置的传播行为RequriesNew，开启新事务，而是使用save方法的已开启事务，如果非要这样使用如下方式实现：

1、 修改TransactionProxyFactoryBean配置定义，添加exposeProxy属性为true；

2、 在业务方法内通过代理对象调用相应的事务方放，如“**((IUserService)AopContext.currentProxy()).otherTransactionMethod()**”即可应用配置的事务属性。

3、 使用这种方式属于侵入式，不推荐使用，除非必要。

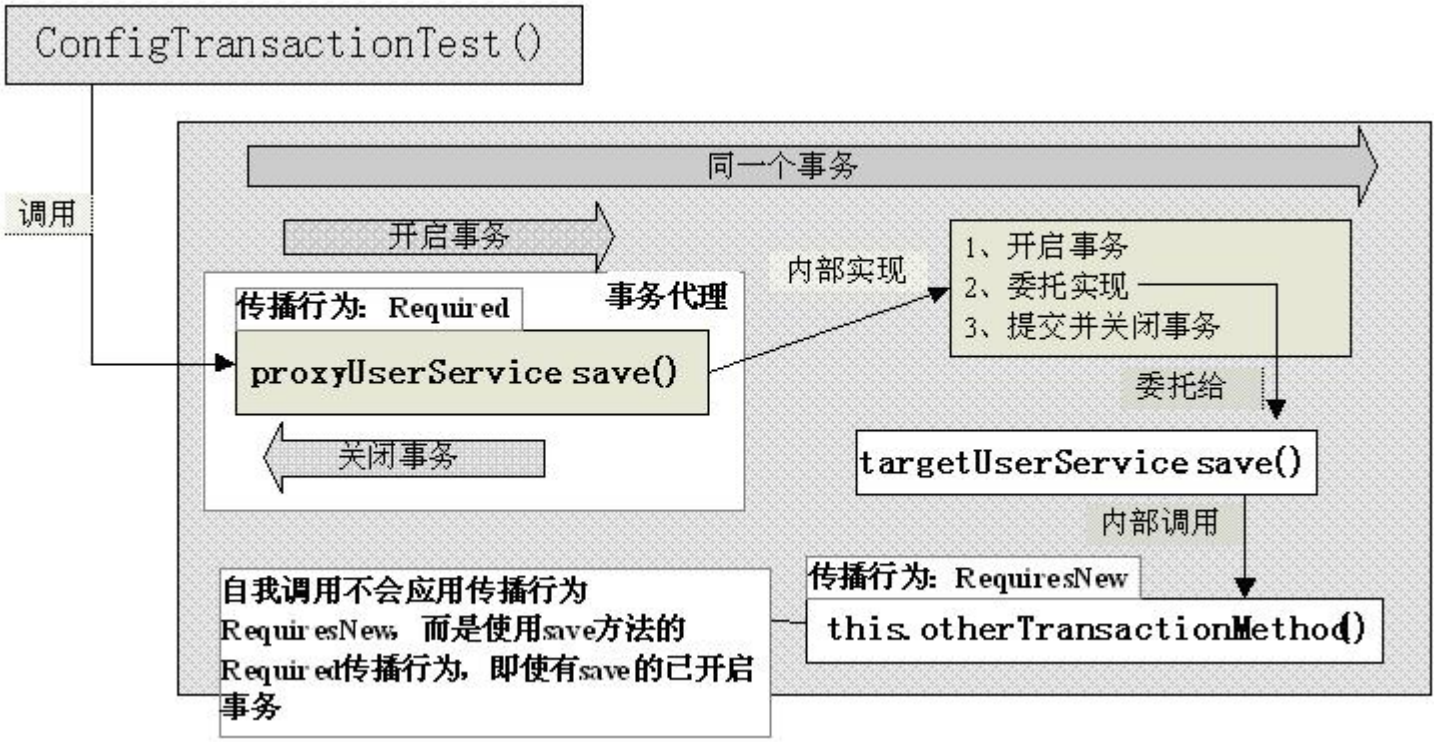


图9-19 代理方式下的自我调用

配置方式也好麻烦啊，每个业务实现都需要配置一个事务代理，发展到这，Spring想出更好的解决方案，Spring2.0及之后版本提出使用新的“<tx:tags/>”方式配置事务，从而无需为每个业务实现配置一个代理。

1.9 【第九章】 Spring的事务 之 9.4 声明式事务 ——跟我学spring3

发表时间: 2012-03-07 关键字: spring

9.4 声明式事务

9.4.1 声明式事务概述

从上节编程式实现事务管理可以深刻体会到编程式事务的痛苦，即使通过代理配置方式也是不小的工作量。

本节将介绍声明式事务支持，使用该方式后最大的获益是简单，事务管理不再是令人痛苦的，而且此方式属于无侵入式，对业务逻辑实现无影响。

接下来先来看看声明式事务如何实现吧。

9.4.2 声明式实现事务管理

1、定义业务逻辑实现，此处使用ConfigUserServiceImpl和ConfigAddressServiceImpl：

2、定义配置文件（chapter9/service/ applicationContext-service-declare.xml）：

2.1、XML命名空间定义，定义用于事务支持的tx命名空间和AOP支持的aop命名空间：

java代码：

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
      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/tx
       http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
          http://www.springframework.org/schema/aop
          http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
```

2.2、业务实现配置，非常简单，使用以前定义的非侵入式业务实现：

java代码：

```
<bean id="userService" class="cn.javass.spring.chapter9.service.impl.ConfigUserServiceImpl">
    <property name="userDao" ref="userDao"/>
    <property name="addressService" ref="addressService"/>
</bean>
<bean id="addressService" class="cn.javass.spring.chapter9.service.impl.ConfigAddressService
    <property name="addressDao" ref="addressDao"/>
</bean>
```

2.3、事务相关配置：

java代码：

```
<tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
        <tx:method name="save*" propagation="REQUIRED" isolation="READ_COMMITTED"/>
        <tx:method name="*" propagation="REQUIRED" isolation="READ_COMMITTED" read-only="tru
    </tx:attributes>
</tx:advice>
```

java代码：

```
<aop:config>
    <aop:pointcut id="serviceMethod" expression="execution(* cn..chapter9.service..*.*(..))"
```

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

- <tx:advice>：事务通知定义，用于指定事务属性，其中“transaction-manager”属性指定事务管理器，并通过< tx:attributes >指定具体需要拦截的方法；
- <tx:method name="save*">：表示将拦截以save开头的方法，被拦截的方法将应用配置的事务属性：propagation="REQUIRED"表示传播行为是Required，isolation="READ_COMMITTED"表示隔离级别是提交读；
- <tx:method name="*">：表示将拦截其他所有方法，被拦截的方法将应用配置的事务属性：propagation="REQUIRED"表示传播行为是Required，isolation="READ_COMMITTED"表示隔离级别是提交读，read-only="true"表示事务只读；
- <aop:config>：AOP相关配置：
- <aop:pointcut/>：切入点定义，定义名为"serviceMethod"的aspectj切入点，切入点表达式为"execution(* cn..chapter9.service..*.*(..))"表示拦截cn包及子包下的chapter9. service包及子包下的任何类的任何方法；
- <aop:advisor>：Advisor定义，其中切入点为serviceMethod，通知为txAdvice。

从配置中可以看出，将对cn包及子包下的chapter9. service包及子包下的任何类的任何方法应用“txAdvice”通知指定的事务属性。

3、修改测试方法并测试该配置方式是否好用：

将TransactionTest 类的testServiceTransaction测试方法拷贝一份命名为testDeclareTransaction：

并在testDeclareTransaction测试方法内将：

java代码：

```
classpath:chapter9/service/applicationContext-service.xml"
```

替换为：

java代码：

```
classpath:chapter9/service/applicationContext-service-declare.xml"
```

4、执行测试，测试正常通过，说明该方式能正常工作，当调用save方法时将匹配到事务通知中定义的“<tx:method name="save*">”中指定的事务属性，而调用countAll方法时将匹配到事务通知中定义的“<tx:method name="*">”中指定的事务属性。

声明式事务是如何实现事务管理的呢？还记不记得TransactionProxyFactoryBean实现配置式事务管理，配置式事务管理是通过代理方式实现，而声明式事务管理同样是通过AOP代理方式实现。

声明式事务通过AOP代理方式实现事务管理，利用环绕通知TransactionInterceptor实现事务的开启及关闭，而TransactionProxyFactoryBean内部也是通过该环绕通知实现的，因此可以认为是<tx:tags/>帮你定义了TransactionProxyFactoryBean，从而简化事务管理。

了解了实现方式后，接下来详细学习一下配置吧：

9.4.4 <tx:advice/>配置详解

声明式事务管理通过配置<tx:advice/>来定义事务属性，配置方式如下所示：

java代码：

```
<tx:advice id="……" transaction-manager="……">
<tx:attributes>
        <tx:method name="……"
                            propagation=" REQUIRED"
                            isolation="READ_COMMITTED"
                            timeout="-1"
                            read-only="false"
                            no-rollback-for=""
                            rollback-for=""/>
        ……
    </tx:attributes>
</tx:advice>
```

- **<tx:advice>：**id用于指定此通知的名字， transaction-manager用于指定事务管理器，默认的事务管理器名字为“transactionManager” ；
- <tx:method>：用于定义事务属性即相关联的方法名；

name：定义与事务属性相关联的方法名，将对匹配的方法应用定义的事务属性，可以使用“*”通配符来匹配一组或所有方法，如“save*”将匹配以save开头的方法，而“*”将匹配所有方法；

propagation：事务传播行为定义，默认为“REQUIRED” ，表示Required，其值可以通过TransactionDefinition的静态传播行为变量的“PROPAGATION_”后边部分指定，如“TransactionDefinition.PROPAGATION_REQUIRED”可以使用“REQUIRED”指定；

isolation：事务隔离级别定义；默认为“DEFAULT” ，其值可以通过TransactionDefinition的静态隔离级别变量的“ISOLATION_”后边部分指定，如“TransactionDefinition. ISOLATION_DEFAULT”可以使用“DEFAULT”指定：

timeout：事务超时时间设置，单位为秒，默认-1，表示事务超时将依赖于底层事务系统；

read-only：事务只读设置，默认为false，表示不是只读；

rollback-for：需要触发回滚的异常定义，以“，”分割，默认任何RuntimeException 将导致事务回滚，而任何Checked Exception 将不导致事务回滚；异常名字定义和TransactionProxyFactoryBean中含义一样

no-rollback-for：不被触发进行回滚的 Exception(s)；以“，”分割；异常名字定义和TransactionProxyFactoryBean中含义一样；

记不记得在配置方式中为了解决“自我调用”而导致的不能设置正确的事务属性问题，使用“**((IUserService)AopContext.currentProxy()).otherTransactionMethod()**”方式解决，在声明式事务要得到支持需要使用<aop:config expose-proxy="true">来开启。

9.4.5 多事务语义配置及最佳实践

什么是多事务语义？说白了就是为不同的Bean配置不同的事务属性，因为我们项目中不可能就几个Bean，而可能很多，这可能需要为Bean分组，为不同组的Bean配置不同的事务语义。在Spring中，可以通过配置多切入点和多事务通知并通过不同方式组合使用即可。

1、首先看下声明式事务配置的最佳实践吧：

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="get*" propagation="SUPPORTS" read-only="true" />
            <tx:method name="count*" propagation="SUPPORTS" read-only="true" />
          <tx:method name="find*" propagation="SUPPORTS" read-only="true" />
          <tx:method name="list*" propagation="SUPPORTS" read-only="true" />
          <tx:method name="*" propagation="SUPPORTS" read-only="true" />
```

```
        </tx:attributes>
</tx:advice>
<aop:config>
        <aop:pointcut id="txPointcut" expression="execution(* cn.javass..service.*.*(..))" />
        <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut" />
</aop:config>
```

该声明式事务配置可以应付常见的CRUD接口定义，并实现事务管理，我们只需修改切入点表达式来拦截我们的业务实现从而对其应用事务属性就可以了，如果还有更复杂的事务属性直接添加即可，即

如果我们有一个batchSaveOrUpdate方法需要“REQUIRES_NEW”事务传播行为，则直接添加如下配置即可：

java代码：

```
<tx:method name="batchSaveOrUpdate" propagation="REQUIRES_NEW" />
```

2、接下来看一下多事务语义配置吧，声明式事务最佳实践中已经配置了通用事务属性，因此可以针对需要其他事务属性的业务方法进行特例化配置：

java代码：

```
<tx:advice id="noTxAdvice" transaction-manager="txManager">
    <tx:attributes>
            <tx:method name="*" propagation="NEVER" />
    </tx:attributes>
</tx:advice>
<aop:config>
        <aop:pointcut id="noTxPointcut" expression="execution(* cn.javass..util.*.*())" />
```

```
        <aop:advisor advice-ref="noTxPointcut" pointcut-ref="noTxAdvice" />
</aop:config>
```

该声明将对切入点匹配的方法所在事务应用“Never”传播行为。

多事务语义配置时，切入点一定不要叠加，否则将应用两次事务属性，造成不必要的错误及麻烦。

9.4.6 @Transactional实现事务管理

对声明式事务管理，Spring提供基于@Transactional注解方式来实现，但需要Java 5+。

注解方式是最简单的事务配置方式，可以直接在Java源代码中声明事务属性，且对于每一个业务类或方法如果需要事务都必须使用此注解。

接下来学习一下注解事务的使用吧：

1、定义业务逻辑实现：

java代码：

```
package cn.javass.spring.chapter9.service.impl;
//省略import
public class AnnotationUserServiceImpl implements IUserService {
    private IUserDao userDao;
    private IAddressService addressService;
```

```
    public void setUserDao(IUserDao userDao) {
        this.userDao = userDao;
    }
    public void setAddressService(IAddressService addressService) {
        this.addressService = addressService;
    }
    @Transactional(propagation=Propagation.REQUIRED, isolation=Isolation.READ_COMMITTED)
    @Override
    public void save(final UserModel user) {
        userDao.save(user);
        user.getAddress().setUserId(user.getId());
        addressService.save(user.getAddress());
    }
    @Transactional(propagation=Propagation.REQUIRED, isolation=Isolation.READ_COMMITTED, rea
    @Override
    public int countAll() {
        return userDao.countAll();
    }
}
```

2、定义配置文件（chapter9/service/ applicationContext-service-annotation.xml）：

2.1、XML命名空间定义，定义用于事务支持的tx命名空间和AOP支持的aop命名空间：

java代码：

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

```
    xmlns:tx="http://www.springframework.org/schema/tx"
      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/tx
       http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
          http://www.springframework.org/schema/aop
          http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
```

2.2、业务实现配置，非常简单，使用以前定义的非侵入式业务实现：

java代码：

```
<bean id="userService" class="cn.javass.spring.chapter9.service.impl.ConfigUserServiceImpl">
    <property name="userDao" ref="userDao"/>
    <property name="addressService" ref="addressService"/>
</bean>
<bean id="addressService" class="cn.javass.spring.chapter9.service.impl.ConfigAddressService
    <property name="addressDao" ref="addressDao"/>
</bean>
```

2.3、事务相关配置：

java代码：

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

使用如上配置已支持声明式事务。

3、修改测试方法并测试该配置方式是否好用：

将TransactionTest 类的testServiceTransaction测试方法拷贝一份命名为testAnntationTransactionTest：

将测试代码片段：

java代码：

```
classpath:chapter9/service/applicationContext-service.xml"
```

替换为：

java代码：

```
classpath:chapter9/service/applicationContext-service-annotation.xml"
```

将测试代码段

java代码：

```
userService.save(user);
```

替换为：

java代码：

```
try {
    userService.save(user);
    Assert.fail();
} catch (RuntimeException e) {
}
Assert.assertEquals(0, userService.countAll());
Assert.assertEquals(0, addressService.countAll());
```

4、执行测试，测试正常通过，说明该方式能正常工作，因为在AnnotationAddressServiceImpl类的save方法中抛出异常，因此事务需要回滚，所以两个countAll操作都返回0。

9.4.7 @Transactional配置详解

Spring提供的<tx:annotation-driven/>用于开启对注解事务管理的支持，从而能识别Bean类上的@Transactional注解元数据，其具有以下属性：

- transaction-manager：指定事务管理器名字，默认为transactionManager，当使用其他名字时需要明确指定；
- proxy-target-class：表示将使用的代码机制，默认false表示使用JDK代理，如果为true将使用CGLIB代理
- order：定义事务通知顺序，默认Ordered.LOWEST_PRECEDENCE，表示将顺序决定权交给AOP来处理。

Spring使用@Transaction来指定事务属性，可以在接口、类或方法上指定，如果类和方法上都指定了@Transaction，则方法上的事务属性被优先使用，具体属性如下：

- value：指定事务管理器名字，默认使用<tx:annotation-driven/>指定的事务管理器，用于支持多事务管理器环境；
- **propagation**：指定事务传播行为，默认为Required，使用Propagation.REQUIRED指定；
- **isolation**：指定事务隔离级别，默认为“DEFAULT”，使用Isolation.DEFAULT指定；
- **readOnly**：指定事务是否只读，默认false表示事务非只读；
- **timeout**：指定事务超时时间，以秒为单位，默认-1表示事务超时将依赖于底层事务系统；
- **rollbackFor**：指定一组异常类，遇到该类异常将回滚事务；
- **rollbackForClassname**：指定一组异常类名字，其含义与<tx:method>中的rollback-for属性语义完全一样；

- **noRollbackFor**：指定一组异常类，即使遇到该类异常也将提交事务，即不回滚事务；
- **noRollbackForClassname**：指定一组异常类名字，其含义与<tx:method>中的no-rollback-for属性语义完全一样；

Spring提供的@Transaction注解事务管理内部同样利用环绕通知TransactionInterceptor实现事务的开启及关闭。

使用@Transactional注解事务管理需要特别注意以下几点：

- 如果在接口、实现类或方法上都指定了@Transactional 注解，则优先级顺序为方法>实现类>接口；
- 建议只在实现类或实现类的方法上使用@Transactional，而不要在接口上使用，这是因为如果使用JDK代理机制是没问题，因为其使用基于接口的代理；而使用使用CGLIB代理机制时就会遇到问题，因为其使用基于类的代理而不是接口，这是**因为接口上的@Transactional注解是“不能继承的”**；

  具体请参考**基于JDK动态代理和CGLIB动态代理的实现Spring注解管理事务（@Trasactional）到底有什么区别。**

- 在Spring代理机制下(不管是JDK动态代理还是CGLIB代理)，“自我调用”同样不会应用相应的事务属性，其语义和<tx:tags>中一样；
- 默认只对RuntimeException异常回滚；
- 在使用Spring代理时，默认只有在public可见度的方法的@Transactional 注解才是有效的，其它可见度（protected、private、包可见）的方法上即使有@Transactional 注解也不会应用这些事务属性的，Spring也不会报错，如果你非要使用非公共方法注解事务管理的话，可考虑使用AspectJ。

9.4.9 与其他AOP通知协作

Spring声明式事务实现其实就是Spring AOP+线程绑定实现，利用AOP实现开启和关闭事务，利用线程绑定（ThreadLocal）实现跨越多个方法实现事务传播。

由于我们不可能只使用一个事务通知，可能还有其他类型事务通知，而且如果这些通知中需要事务支持怎么办？这就牵扯到通知执行顺序的问题上了，因此如果可能与其他AOP通知协作的话，而且这些通知中需要使用声明式事务管理支持，事务通知应该具有最高优先级。

9.4.10 声明式or编程式

编程式事务时不推荐的，即使有很少事务操作，Spring发展到现在，没有理由使用编程式事务，只有在为了深入理解Spring事务管理才需要学习编程式事务使用。

推荐使用声明式事务，而且强烈推荐使用<tx:tags>方式的声明式事务，因为其是无侵入代码的，可以配置模板化的事务属性并运用到多个项目中。

而@Transaction注解事务，可以使用，不过作者更倾向于使用<tx:tags>声明式事务。

能保证项目正常工作的事务配置就是最好的。

9.4.11 混合事务管理

所谓混合事务管理就是混合多种数据访问技术使用，如混合使用Spring JDBC + Hibernate，接下来让我们学习一下常见混合事务管理：

1、 Hibernate + Spring JDBC/iBATIS：使用HibernateTransactionManager即可支持；

2、 JPA + Spring JDBC/iBATIS：使用JpaTransactionManager即可支持；

3、 JDO + Spring JDBC/iBATIS：使用JtaTransactionManager即可支持；

混合事务管理最大问题在于如果我们使用第三方ORM框架，如Hibernate，会遇到一级及二级缓存问题，尤其是二级缓存可能造成如使用Spring JDBC和Hibernate查询出来的数据不一致等。

因此不建议使用这种混合使用和混合事务管理。

1.10 【第十章】集成其它Web框架 之 10.1 概述 ——跟我学spring3

发表时间: 2012-03-09 关键字: spring

10.1 概述

10.1.1 Spring和Web框架

Spring框架不仅提供了一套自己的Web框架实现，还支持集成第三方Web框架（如Struts1x、Struts2x）。

Spring实现的SpringMVC Web框架将在第十八章详细介绍。

由于现在有很大部分公司在使用第三方Web框架，对于并不熟悉SpringMVC Web框架的公司，为了充分利用开发人员已掌握的技术并相使用Spring的功能，想集成所使用的Web框架；由于Spring框架的高度可配置和可选择性，因此集成这些第三方Web框架是非常简单的。

之所以想把这些第三方Web框架集成到Spring中，最核心的价值是享受Spring的某些强大功能，如一致的数据访问，事务管理，IOC，AOP等等。

Spring为所有Web框架提供一致的通用配置，从而不管使用什么Web框架都使用该通用配置。

10.1.2 通用配置

Spring对所有Web框架抽象出通用配置，以减少重复配置，其中主要有以下配置：

1、Web环境准备：

1.1、在spring项目下创建如图10-1目录结构：

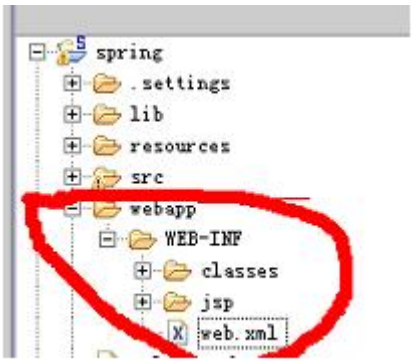


图10-1 web目录结构

1.2、右击spring项目选择【Propeties】，然后选择【Java Build Path】中的【Source】选项卡，将类输出路径修改为“spring/webapp/WEB-INF/classes”，如图10-2所示：

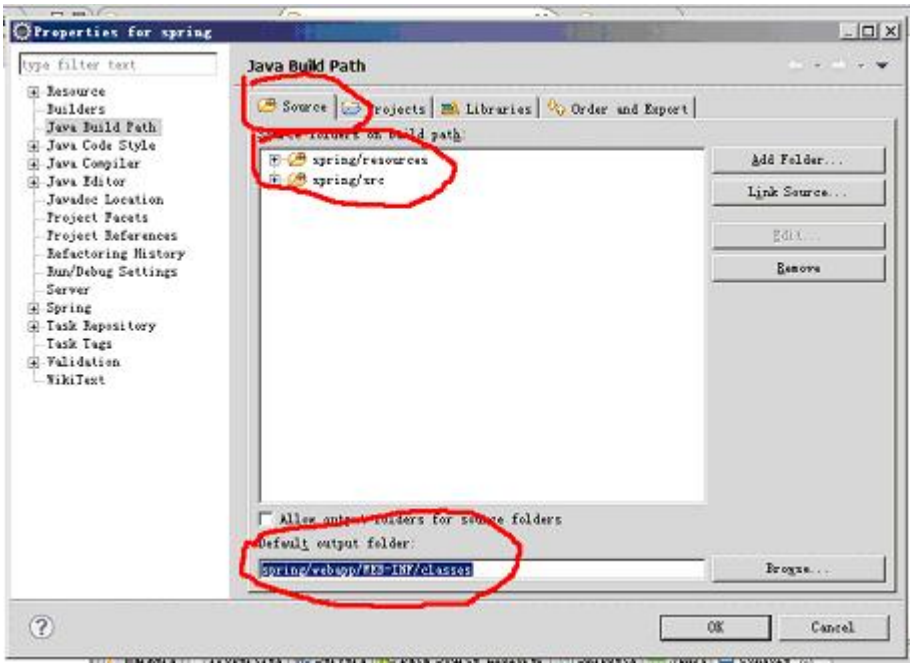


图10-2 修改类输出路径

1.3、web.xml初始内容如下：

java代码：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
```

```
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
</web-app>
```

<web-app version="2.4">表示采用Servlet 2.4规范的Web程序部署描述格式

2、 指定Web应用上下文实现：在Web环境中，Spring提供WebApplicationContext（继承ApplicationContext）接口用于配置Web应用，该接口应该被实现为在Web应用程序运行时只读，即在初始化完毕后不能修改Spring Web容器（WebApplicationContext），但可能支持重载。

Spring提供XmlWebApplicationContext实现，并在Web应用程序中默认使用该实现，可以通过在web.xml配置文件中使用如下方式指定：

java代码：

```
<context-param>
    <param-name>contextClass</param-name>
    <param-value>
        org.springframework.web.context.support.XmlWebApplicationContext
    </param-value>
</context-param>
```

如上指定是可选的，只有当使用其他实现时才需要显示指定。

3、 指定加载文件位置：

前边已经指定了Spring Web容器实现，那从什么地方加载配置文件呢？

默认情况下将加载/WEB-INF/applicationContext.xml配置文件，当然也可以使用如下形式在web.xml中定义要加载自定义的配置文件，多个配置文件用“,”分割：

java代码：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:chapter10/applicationContext-message.xml
    </param-value>
</context-param>
```

通用Spring配置文件（resources/chapter10/applicationContext-message.xml）内容如下所示：

java代码：

```
<bean id="message" class="java.lang.String">
    <constructor-arg index="0" value="Hello Spring"/>
</bean>
```

4、 加载和关闭Spring Web容器：

我们已经指定了Spring Web容器实现和配置文件，那如何才能让Spring使用相应的Spring Web容器实现加载配置文件呢？

Spring使用ContextLoaderListener监听器来加载和关闭Spring Web容器，即使用如下方式在web.xml中指定：

java代码：

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

ContextLoaderListener监听器将在Web应用启动时使用指定的配置文件初始化Spring Web容器，在Web应用关闭时销毁Spring Web容器。

注：监听器是从Servlet 2.3才开始支持的，因此如果Web应用所运行的环境是Servlet 2.2版本则可以使用ContextLoaderServlet来完成，但从Spring3.x版本之后ContextLoaderServlet被移除了。

5、 在Web环境中获取Spring Web容器:

既然已经定义了Spring Web容器，那如何在Web中访问呢？Spring提供如下方式来支持获取Spring Web容器（WebApplicationContext）：

java代码：

```
WebApplicationContextUtils.getWebApplicationContext(servletContext);
或
WebApplicationContextUtils.getRequiredWebApplicationContext(servletContext);
```

如果当前Web应用中的ServletContext 中没有相应的Spring Web容器，对于getWebApplicationContext()方法将返回null,而getRequiredWebApplicationContext()方法将抛出异常，建议使用第二种方式，因为缺失Spring Web容器而又想获取它，很明显是错误的，应该抛出异常。

6、 通用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.web-3.0.5.RELEASE.jar

此jar包为所有Web框架所共有，提供WebApplicationContext及实现等。

7、Web服务器选择及测试：

目前比较流行的支持Servlet规范的开源Web服务器包括Tomcat、Resin、Jetty等，Web服务器有独立运行和嵌入式运行之分，嵌入式Web服务器可以在测试用例中运行不依赖于外部环境，因此我们使用嵌入式Web服务器。

Jetty是一个非常轻量级的Web服务器，并且提供嵌入式运行支持，在此我们选用Jetty作为测试使用的Web服务器。

7.1、准备Jetty嵌入式Web服务器运行需要的jar包：

到http://dist.codehaus.org/jetty/网站下载jetty-6.1.24，在下载的jetty-6.1.24.zip包中拷贝如下jar包

7.2、在单元测试中启动Web服务器：

java代码：

```
package cn.javass.spring.chapter10;
import org.junit.Test;
import org.mortbay.jetty.Server;
import org.mortbay.jetty.webapp.WebAppContext;
public class WebFrameWorkIntegrateTest {
    @Test
    public void testWebFrameWork() throws Exception {
        Server server = new Server(8080);
        WebAppContext webapp = new WebAppContext();
        webapp.setResourceBase("webapp");
        //webapp.setDescriptor("webapp/WEB-INF/web.xml");
        webapp.setContextPath("/");
        webapp.setClassLoader(Thread.currentThread().getContextClassLoader());
        server.setHandler(webapp);
        server.start();
        server.join();
       //server.stop();
    }
}
```

- **创建内嵌式Web服务器：**使用new Server(8080)新建一个Jetty服务器，监听端口为8080；
- **创建一个Web应用：**使用new WebAppContext()新建一个Web应用对象，一个Web应用可以认为就是一个WebAppContext对象；
- **指定Web应用的目录：**使用webapp.setResourceBase("webapp")指定Web应用位于项目根目录下的“webapp”目录下；
- **指定部署描述符：**使用webapp.setDescriptor("webapp/WEB-INF/web.xml")；此处指定部署描述符为项目根目录下的“webapp/WEB-INF/web.xml”，该步骤是可选的，如果web.xml位于Web应用的WEB-INF下。
- **指定Web应用请求上下文：**使用webapp.setContextPath("/")指定请求上下文为“/”，从而访问该Web应用可以使用如“http://localhost:8080/hello.do”形式访问；
- 指定类装载器：因为Jetty自带的ClassLoader在内嵌环境中对中文路径处理有问题，因此我们使用Eclispe的ClassLoader，即通过“webapp.setClassLoader(Thread.currentThread().getContextClassLoader())”指定；
- **启动Web服务器：**使用“server.start()”启动并使用“server.join()”保证Web服务器一直运行；
- **关闭Web服务器：**可以通过某种方式执行“server.stop()”来关闭Web服务器；另一种方式是通过【Console】控制台面板的【Terminate】终止按钮关闭，如图10-3所示：

图10-3 点击红色按钮关闭Web服务器

1.11 【第十章】集成其它Web框架 之 10.2 集成Struts1.x ——跟我学spring3

发表时间: 2012-03-12 关键字: spring, struts1

先进行通用配置， 【第十章】集成其它Web框架 之 10.1 概述

10.2 集成Struts1.x

10.2.1 概述

Struts1.x是最早实现MVC（模型-视图-控制器）模式的Web框架之一，其使用非常广泛，虽然目前已经有Struts2.x等其他Web框架，但仍有很多公司使用Struts1.x框架。

集成Struts1.x也非常简单，除了通用配置外，有两种方式可以将Struts1.x集成到Spring中：

- 最简单集成：使用Spring提供的WebApplicationContextUtils工具类中的获取Spring Web容器，然后通过Spring Web容器获取Spring管理的Bean；
- Struts1.x插件集成：利用Struts1.x中的插件ContextLoaderPlugin来将Struts1.x集成到Spring中。

接下来让我们首先让我们准备Struts1x所需要的jar包：

1.1、从下载的spring-framework-3.0.5.RELEASE-with-docs.zip中dist目录查找如下jar包，该jar包用于提供集成struts1.x所需要的插件实现等：

org.springframework.web.struts-3.0.5.RELEASE.jar

1.2、从下载的spring-framework-3.0.5.RELEASE-dependencies.zip中查找如下依赖jar包，该组jar是struts1.x需要的jar包：

com.springsource.org.apache.struts-1.2.9.jar //struts1.2.9实现包

com.springsource.org.apache.commons.digester-1.8.1.jar //用于解析struts配置文件

com.springsource.org.apache.commons.beanutils-1.8.0.jar //用于请求参数绑定

com.springsource.javax.servlet-2.5.0.jar //Servlet 2.5 API

antlr.jar //语法分析包（已有）

commons-logging.jar //日志记录组件包（已有）

servlet-api.jar //Servlet API包（已有）

jsp-api.jar //JSP API包（已有，可选）

commons-validator.jar **//验证包（可选）**

commons-fileupload.jar **//文件上传包（可选）**

10.2.2 最简单集成

只使用通用配置，利用WebApplicationContextUtils提供的获取Spring Web容器方法获取Spring Web容器，然后从Spring Web容器获取Spring管理的Bean。

1、 第一个Action实现：

java代码：

```
package cn.javass.spring.chapter10.struts1x.action;
import org.apache.struts.action.Action;
//省略部分import
public class HelloWorldAction1 extends Action {
    public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest

        WebApplicationContext ctx = WebApplicationContextUtils.
            getRequiredWebApplicationContext(getServlet().getServletContext());
```

```
        String message = ctx.getBean("message", String.class);
        request.setAttribute("message",  message);
        return mapping.findForward("hello");
    }
}
```

此Action实现非常简单，首先通过WebApplicationContextUtils获取Spring Web容器，然后从Spring Web容器中获取“message”Bean并将其放到request里，最后转到“hello”所代表的jsp页面。

2、JSP页面定义（webapp/WEB-INF/jsp/hello.jsp）：

java代码：

```
<%@ page language="java" pageEncoding="UTF-8"
contentType="text/html; charset=UTF-8" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
    <title>Hello World</title>
</head>
<body>
    ${message}
</body>
</html>
```

3、配置文件定义：

3.1、Spring配置文件定义（resources/chapter10/applicationContext-message.xml）：

在此配置文件中定义我们使用的“message”Bean；

java代码：

```
<bean id="message" class="java.lang.String">
    <constructor-arg index="0" value="Hello Spring"/>
</bean>
```

3.2、struts配置文件定义（resources/chapter10/struts1x/struts-config.xml）：

java代码：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
          "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
<struts-config>
  <action-mappings>
    <action path="/hello" type="cn.javass.spring.chapter10.struts1x.action.HelloWorldAction1
       <forward name="hello" path="/WEB-INF/jsp/hello.jsp"/>
    </action>
  </action-mappings>
</struts-config>
```

3.3、web.xml部署描述符文件定义（webapp/WEB-INF/web.xml）添加如下内容：

java代码：

```
<!-- Struts1.x前端控制器配置开始    -->
       <servlet>
        <servlet-name>hello</servlet-name>
```

```
        <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>

        <init-param>
            <param-name>config</param-name>
            <param-value>
                      /WEB-INF/classes/chapter10/struts1x/struts-config.xml
             </param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
       </servlet>
       <servlet-mapping>
        <servlet-name>hello</servlet-name>
        <url-pattern>*.do</url-pattern>
      </servlet-mapping>
<!-- Struts1.x前端控制器配置结束    -->
```

Struts1.x前端控制器配置了ActionServlet前端控制器，其拦截以.do开头的请求，Strut配置文件通过初始化参数“config”来指定，如果不知道“config”参数则默认加载的配置文件为“/WEB-INF/ struts-config.xml”。

4、执行测试：在Web浏览器中输入http://localhost:8080/hello.do可以看到“Hello Spring”信息说明测试正常。

有朋友想问，我不想使用这种方式，我想在独立环境内测试，没关系，您只需将spring/lib目录拷贝到spring/webapp/WEB-INF/下，然后将webapp拷贝到如tomcat中即可运行，尝试一下吧。

Spring还提供ActionSupport类来简化获取WebApplicationContext，Spring为所有标准Action类及子类提供如下支持类，即在相应Action类后边加上Support后缀：

- ActionSupport
- DispatchActionSupport
- LookupDispatchActionSupport
- MappingDispatchActionSupport

具体使用方式如下：

1、Action定义

java代码：

```
package cn.javass.spring.chapter10.struts1x.action;
//省略import
public class HelloWorldAction2 extends ActionSupport {
public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest requ
        WebApplicationContext ctx = getWebApplicationContext();
        String message = ctx.getBean("message", String.class);
        request.setAttribute("message", message);
        return mapping.findForward("hello");
    }
}
```

和第一个示例唯一不同的是直接调用**getWebApplicationContext()**即可获得Spring Web容器。

2、修改Struts配置文件（resources/chapter10/struts1x/struts-config.xml）添加如下Action定义：

java代码：

```
<action path="/hello2" type="cn.javass.spring.chapter10.struts1x.action.HelloWorldAction2">
    <forward name="hello" path="/WEB-INF/jsp/hello.jsp"/>
</action>
```

3、启动嵌入式Web服务器并在Web浏览器中输入http://localhost:8080/hello2.do可以看到“Hello Spring”信息说明Struts1集成成功。

这种集成方式好吗？而且这种方式算是集成吗？直接获取Spring Web容器然后从该Spring Web容器中获取Bean，暂且看作是集成吧，这种集成对于简单操作可以接受，但更复杂的注入呢？接下来让我们学习使用Struts插件进行集成。

10.2.2 Struts1.x插件集成

Struts插件集成使用ContextLoaderPlugin类，该类用于为ActionServlet加载Spring配置文件。

1、在Struts配置文件（resources/chapter10/struts1x/struts-config.xml）中配置插件：

java代码：

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
    <set-property property="contextClass" value="org.springframework.web.context.support.Xml
    <set-property property="contextConfigLocation" value="/WEB-INF/hello-servlet.xml"/>
    <set-property property="namespace" value="hello"/>
</plug-in>
```

- **contextClass：**可选，用于指定WebApplicationContext实现类，默认是XmlWebApplicationContext；
- **contextConfigLocation：**指定Spring配置文件位置，如果我们的ActionServlet 在 web.xml 里面通过<servlet-name>hello</servlet-name>指定名字为“hello”，且没有指定contextConfigLocation，则默认Spring配置文件是/WEB-INF/hello-servlet.xml；
- **namespace：**因为默认使用ActionServlet在web.xml定义中的Servlet的名字，因此如果想要使用其他名字可以使用该变量指定，如指定“hello”，将加载的Spring配置文件为/WEB-INF/hello-servlet.xml；

由于我们的ActionServlet在web.xml中的名字为hello，而我们的配置文件在/WEB-INF/hello-servlet.xml，因此contextConfigLocation和namespace可以不指定，因此最简单配置如下：

java代码：

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn"/>
```

通用配置的Spring Web容器将作为ContextLoaderPlugin中创建的Spring Web容器的父容器存在，然而可以省略通用配置而直接在struts配置文件中通过ContextLoaderPlugin插件指定所有配置文件。

插件已经配置了，那如何定义Action、配置Action、配置Spring管理Bean呢，即如何真正集成Spring+Struts1x呢？使用插件方式时Action将在Spring中配置而不是在Struts中配置了，Spring目前提供以下两种方式：

- 将Struts配置文件中的<action>的type属性指定为DelegatingActionProxy，然后在Spring中配置同名的Spring管理的Action Bean；
- 使用Spring提供的DelegatingRequestProcessor重载 Struts 默认的 RequestProcessor来从Spring容器中查找同名的Spring管理的Action Bean。

看懂了吗？好像没怎么看懂，那就直接上代码，有代码有真相。

2、定义Action实现，由于Action将在Spring中配置，因此message可以使用依赖注入方式了：

java代码：

```
package cn.javass.spring.chapter10.struts1x.action;
//省略
public class HelloWorldAction3 extends Action {
    private String message;
    public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest

        request.setAttribute("message", message);
        return mapping.findForward("hello");
    }
    public void setMessage(String message) {//有setter方法，大家是否想到setter注入
        this.message = message;
```

```
    }
}
```

3、DelegatingActionProxy方式与Spring集成配置：

3．1、在Struts配置文件（resources/chapter10/struts1x/struts-config.xml）中进行Action定义：

java代码：

```
<action path="/hello3" type="org.springframework.web.struts.DelegatingActionProxy">
    <forward name="hello" path="/WEB-INF/jsp/hello.jsp"/>
</action>
```

3.2、在Spring配置文件（webapp/WEB-INF/hello-servlet.xml）中定义Action对应的Bean：

java代码：

```
<bean name="/hello3" class="cn.javass.spring.chapter10.struts1x.action.HelloWorldAction3">
    <property name="message" ref="message"/>
</bean>
```

3.3、启动嵌入式Web服务器并在Web浏览器中输入http://localhost:8080/hello3.do可以看到“Hello Spring”信息说明测试正常。

从以上配置中可以看出：

- Struts配置文件中<action>标签的path属性和Spring配置文件的name属性应该完全一样，否则错误；

- Struts通过**DelegatingActionProxy**去到Spring Web容器中查找同名的Action Bean；

很简单吧，DelegatingActionProxy是个代理Action，其实现了Action类，其内部帮我们查找相应的Spring管理Action Bean并把请求转发给这个真实的Action。

4、DelegatingRequestProcessor方式与Spring集成：

4.1、首先要替换掉Struts默认的RequestProcessor，在Struts配置文件（resources/chapter10/struts1x/struts-config.xml）中添加如下配置：

java代码：

```
<controller>
    <set-property property="processorClass"
         value="org.springframework.web.struts.DelegatingRequestProcessor"/>
</controller>
```

4．2、在Struts配置文件（resources/chapter10/struts1x/struts-config.xml）中进行Action定义：

java代码：

```
<action path="/hello4" type=" cn.javass.spring.chapter10.struts1x.action.HelloWorldAction3">
    <forward name="hello" path="/WEB-INF/jsp/hello.jsp"/>
</action>
```

或更简单形式：

java代码：

```
<action path="/hello4">
    <forward name="hello" path="/WEB-INF/jsp/hello.jsp"/>
</action>
```

4.3、在Spring配置文件（webapp/WEB-INF/hello-servlet.xml）中定义Action对应的Bean：

java代码：

```
<bean name="/hello4" class="cn.javass.spring.chapter10.struts1x.action.HelloWorldAction3">
    <property name="message" ref="message"/>
</bean>
```

4.4、启动嵌入式Web服务器并在Web浏览器中输入http://localhost:8080/hello4.do可以看到“Hello Spring”信息说明Struts1集成成功。

从以上配置中可以看出：

- Struts配置文件中<action>标签的path属性和Spring配置文件的name属性应该完全一样，否则错误；
- Struts通过**DelegatingRequestProcessor**去到Spring Web容器中查找同名的Action Bean；

很简单吧，只是由**DelegatingRequestProcessor**去帮我们查找相应的Action Bean，但没有代理Action了，所以推荐使用该方式。

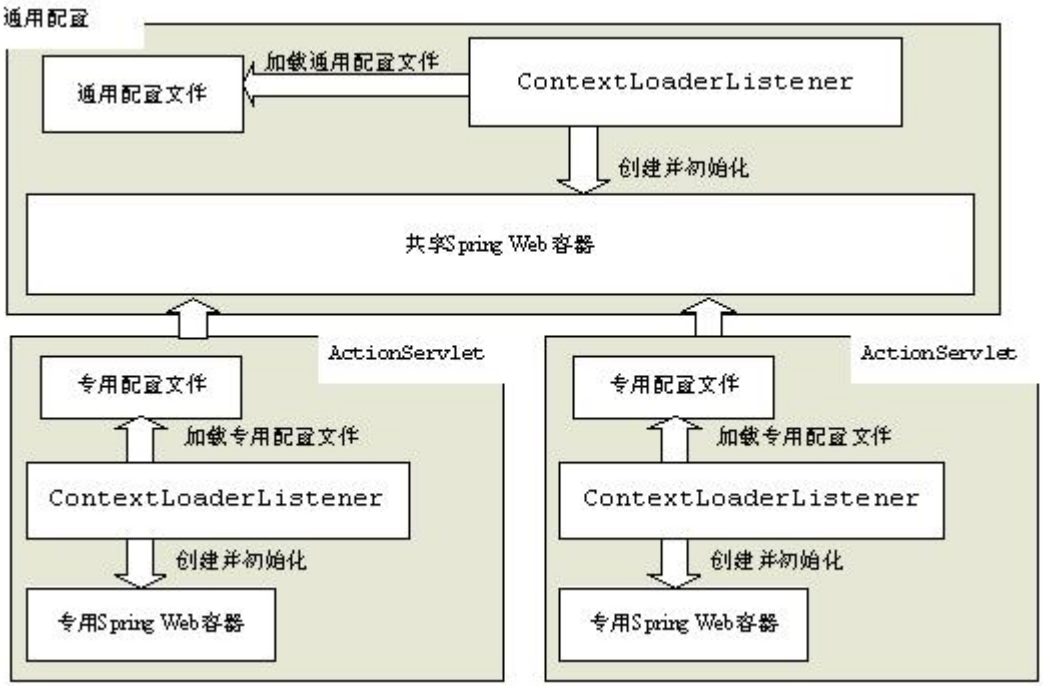


图10-4 共享及专用Spring Web容器

Struts1x与Spring集成到此就完成了，在集成时需要注意一下几点：

- 推荐使用ContextLoaderPlugin+DelegatingRequestProcessor方式集成；
- 当有多个Struts模块时建议在通用配置部分配置通用部分，因为通用配置在正在Web容器中是可共享的，而在各个Struts模块配置文件中配置是不可共享的，因此不推荐直接使用ContextLoaderPlugin中为每个模块都指定所有配置，因为**ContextLoaderPlugin加载的Spring容器只对当前的ActionServlet有效对其他ActionServlet无效**，如图10-4所示。

1.12 【第十章】集成其它Web框架 之 10.3 集成Struts2.x ——跟我学spring3

发表时间: 2012-03-12 关键字: spring, struts2

先进行通用配置， 【第十章】集成其它Web框架 之 10.1 概述

10.3 集成Struts2.x

10.3.1 概述

Struts2前身是WebWork，核心并没有改变，其实就是把WebWork改名为struts2，与Struts1一点关系没有。

Struts2中通过ObjectFactory接口实现创建及获取Action实例，类似于Spring的IoC容器，所以Action实例可以由ObjectFactory实现来管理，因此集成Spring的关键点就是如何创建ObjectFactory实现来从Spring容器中获取相应的Action Bean。

Struts2提供一个默认的ObjectFactory接口实现StrutsSpringObjectFactory，该类用于根据Struts2配置文件中相应Bean信息从Spring 容器中获取相应的Action。

因此Struts2.x与Spring集成需要使用StrutsSpringObjectFactory类作为中介者。

接下来让我们首先让我们准备Struts2x所需要的jar包

准备Struts2.x需要的jar包，到Struts官网http://struts.apache.org/下载struts-2.2.1.1版本，拷贝如下jar包到项目的lib目录下并添加到类路径：

lib\struts2-core-2.2.1.1.jar //核心struts2包

lib\xwork-core-2.2.1.1.jar //命令框架包，独立于Web环境，为Struts2

//提供核心功能的支持包

lib\freemarker-2.3.16.jar //提供模板化UI标签及视图技术支持

lib\ognl-3.0.jar //对象图导航工具包，类似于SpEL

lib\ struts2-spring-plugin-2.2.1.1.jar //集成Spring的插件包

lib\commons-logging-1.0.4.jar //日志记录组件包（已有）

lib\commons-fileupload-1.2.1.jar //用于支持文件上传的包

10.3.2 使用ObjectFactory集成

1、Struts2.x的Action实现：

java代码：

```
package cn.javass.spring.chapter10.struts2x.action;
import org.apache.struts2.ServletActionContext;
import com.opensymphony.xwork2.ActionSupport;
public class HelloWorldAction extends ActionSupport {
    private String message;
    @Override
    public String execute() throws Exception {
        ServletActionContext.getRequest().setAttribute("message", message);
        return "hello";
    }
    public void setMessage(String message) {//setter注入
        this.message = message;
    }
}
```

2、JSP页面定义，使用Struts1x中定义的JSP页面“webapp/WEB-INF/jsp/hello.jsp”；

3、Spring一般配置文件定义（resources/chapter10/applicationContext-message.xml）：

在此配置文件中定义我们使用的“message”Bean；

java代码：

```
<bean id="message" class="java.lang.String">
    <constructor-arg index="0" value="Hello Spring"/>
</bean>
```

4、Spring Action 配置文件定义（resources/chapter10/hello-servlet.xml）：

java代码：

```
<bean name="helloAction" class="cn.javass.spring.chapter10.struts2x.action.HelloWorldAction"
    <property name="message" ref="message"/>
</bean>
```

Struts2的Action在Spring中配置，而且应该是prototype，因为Struts2的Action是有状态的，定义在Spring中，那Struts如何找到该Action呢？

5、struts2配置文件定义（resources/chapter10/struts2x/struts.xml）：

java代码：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <constant name="struts.objectFactory" value="org.apache.struts2.spring.StrutsSpringObjec
    <constant name="struts.devMode" value="true"/>
    <package name="default" extends="struts-default">
        <action name="hello" class="helloAction">
            <result name="hello" >/WEB-INF/jsp/hello.jsp</result>
        </action>
    </package>
</struts>
```

- **struts.objectFactory**：通过在Struts配置文件中使用常量属性**struts.objectFactory**来定义Struts将要使用的ObjectFactory实现，此处因为需要从Spring容器中获取Action对象，因此需要使用StrutsSpringObjectFactory来集成Spring；
- <action name="hello" **class="helloAction"**>：StrutsSpringObjectFactory对象工厂将根据<action>标签的class属性去Spring容器中查找同名的Action Bean；即本例中将到Spring容器中查找名为helloAction的Bean。

6、web.xml部署描述符文件定义（webapp/WEB-INF/web.xml）：

6.1、由于Struts2只能使用通用配置，因此需要在通用配置中加入Spring Action配置文件（chapter10/struts2x/struts2x-servlet.xml）：

java代码：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
```

```
        classpath:chapter10/applicationContext-message.xml,
        classpath:chapter10/struts2x/struts2x-servlet.xml
    </param-value>
</context-param>
```

Struts2只能在通用配置中指定所有Spring配置文件，并没有如Struts1自己指定Spring配置文件的实现。

6.2、Strut2前端控制器定义，在web.xml中添加如下配置：

java代码：

```
<!-- Struts2.x前端控制器配置开始    -->
<filter>
    <filter-name>struts2x</filter-name>
    <filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
    <init-param>
              <param-name>config</param-name>
              <param-value>
                     struts-default.xml,struts-plugin.xml,chapter10/struts2x/struts.xml
               </param-value>
       </init-param>
</filter>
<filter-mapping>
    <filter-name>struts2x</filter-name>
    <url-pattern>*.action</url-pattern>
</filter-mapping>
<!-- Struts2.x前端控制器配置结束    -->
```

- **FilterDispatcher**：Struts2前端控制器为FilterDispatcher，是Filter实现，不是Servlet；
- **config**：通过初始化参数config指定配置文件为struts-default.xml，struts-plugin.xml，chapter10/struts2x/struts.xml；如果不知道该参数则默认加载struts-default.xml，struts-plugin.xml，struts.xml(位于webapp/WEB-INF/classes下)；显示指定时需要将struts-default.xml，struts-plugin.xml也添加上。
- ***.action**：将拦截以“.action”结尾的HTTP请求；

- **struts2x**：FilterDispatcher前端控制器的名字为struts2x，因此相应的Spring配置文件名为struts2x-servlet.xml。

7、执行测试，在Web浏览器中输入http://localhost:8080/hello.action可以看到“Hello Spring”信息说明Struts2集成成功。

集成Strut2也是非常简单，在此我们总结一下吧：

- **配置文件位置：**

Struts配置文件默认加载“struts-default.xml,struts-plugin.xml, struts.xml”，其中struts-default.xml和struts-plugin.xml是Struts自带的，而struts.xml是我们指定的，默认位于webapp/WEB-INF/classes下；

如果需要将配置文件放到其他位置，需要在web.xml的<filter>标签下，使用初始化参数config指定，如“struts-default.xml，struts-plugin.xml，chapter10/struts2x/struts.xml”，其中“struts-default.xml和struts-plugin.xml”是不可省略的，默认相对路径是类路径。

- **集成关键ObjectFactory**：在Struts配置文件或属性文件中使用如下配置知道使用StrutsSpringObjectFactory来获取Action实例：

在struts.xml中指定：

java代码：

```
<constant name="struts.objectFactory" value="org.apache.struts2.spring.StrutsSpringObjectFac
```

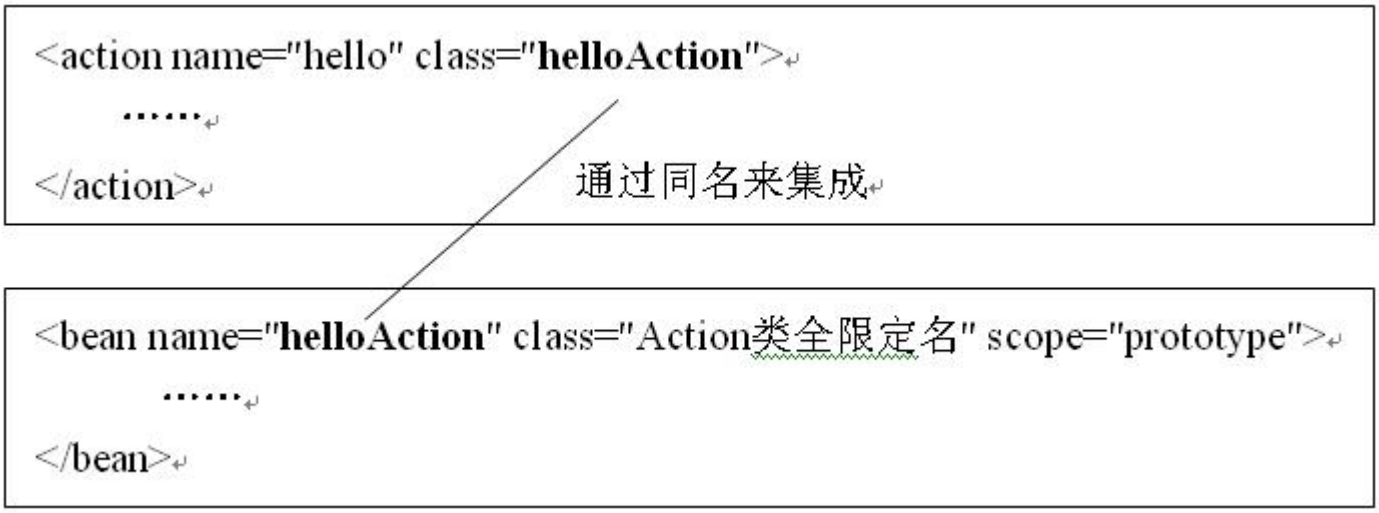
或在struts.properties文件（webapp/WEB-INF/classes/）中：

java代码：

```
struts.objectFactory=org.apache.struts2.spring.StrutsSpringObjectFactory
```

- **集成关键Action定义：**

StrutsSpringObjectFactory将根据Struts2配置文件中的<action class=”” >标签的classes属性名字去到Spring配置文件中查找同名的Bean定义，这也是集成的关键。



- **Spring配置文件中Action定义**：由于Struts2的Action是有状态的，因此应该将Bean定义为prototype。

如图10-5，Sturt2与Spring集成的关键就是StrutsSpringObjectFactory，注意图只是说明Struts与Spring如何通过中介者StrutsSpringObjectFactory来实现集成，不能代表实际的类交互。

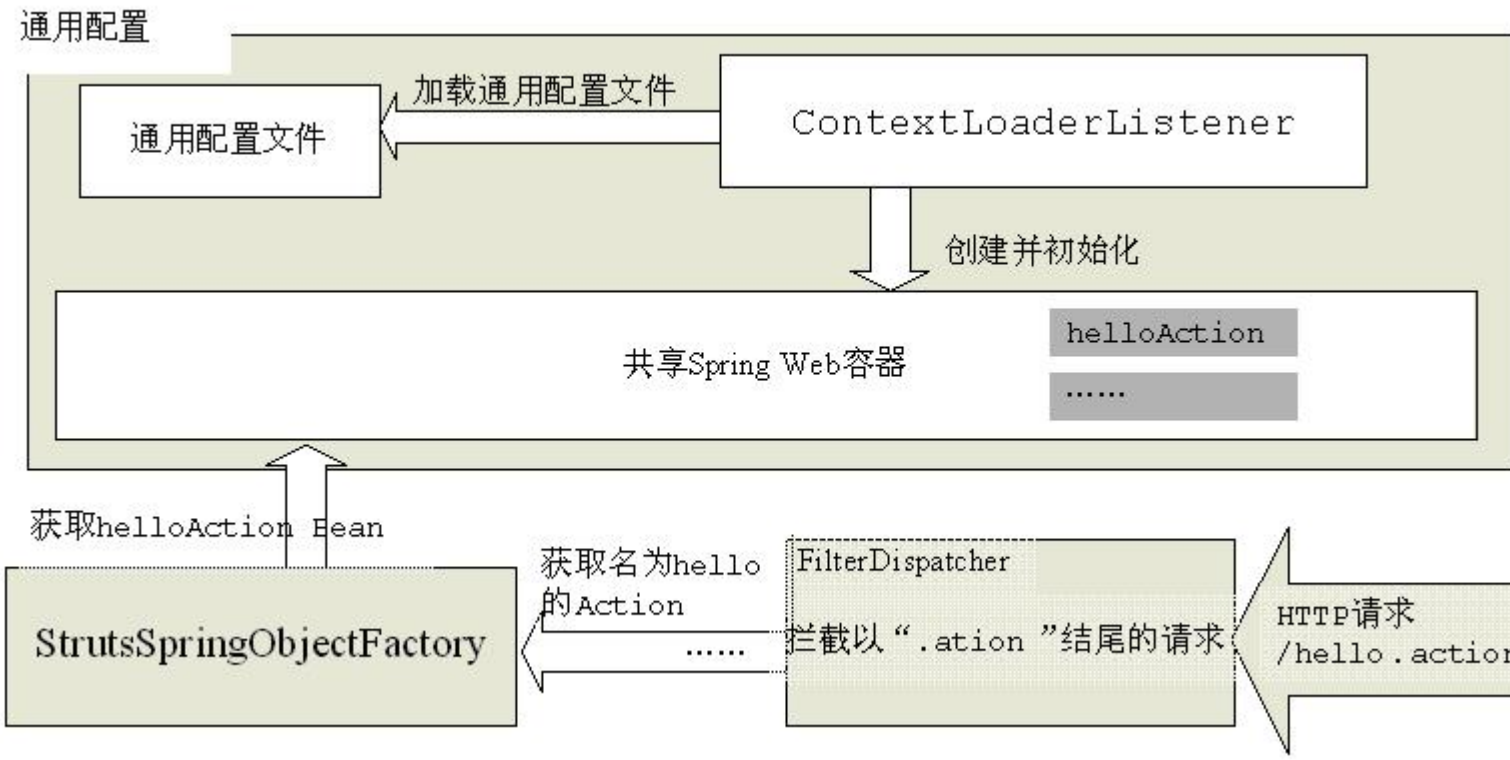


图10-5 Strut2与Spring集成

1.13 【第十章】集成其它Web框架 之 10.4 集成JSF ——跟我学spring3

发表时间: 2012-03-13 关键字: spring, jsf

先进行通用配置， 【第十章】集成其它Web框架 之 10.1 概述

10.4 集成JSF

10.4.1 概述

JSF（JavaServer Faces）框架是Java EE标准之一，是一个基于组件及事件驱动的Web框架，JSF只是一个标准（规范），目前有很多厂家实现，如Oracle的默认标准实现Mojarra、Apache的MyFaces、Jboss的RichFaces 等。

本示例将使用Oracle标准实现Mojarra，请到官网http://javaserverfaces.java.net/下载最新的JSF实现。

JSF目前有JSF1.1、JSF1.2、JSF2版本实现。

Spring集成JSF有三种种方式：

- **最简单集成：**使用FacesContextUtils工具类的getWebApplicationContext方法，类似于Struts1x中的最简单实现；
- **VariableResolver实现：**Spring提供javax.faces.el.VariableResolver的两种实现DelegatingVariableResolver和SpringBeanVariableResolver，此方式适用于JSF1.1、JSF1.2及JSF2，但在JSF1.2和JSF2中不推荐使用该方式，而是使用第三种集成方式；
- **ELResolver实现：**Spring提供javax.el.ELResolver（Unified EL）实现SpringBeanFacesELResolver用于集成JSF1.2和JSF2。

接下来让我们首先让我们准备JSF所需要的jar包：

首先准备JSF所依赖的包：

commons-digester.jar //必须，已有

commons-collections.jar //必须，已有

commons-beanutils.jar //必须，已有

jsp-api.jar //必须，已有

servlet-api.jar //必须，已有

jstl.jar //可选

standard.jar //可选

准备JSF包，到http://javaserverfaces.java.net/下载相应版本的Mojarra实现，如下载JSF1.2实现mojarra-1.2_15-b01-FCS-binary.zip，拷贝如下jar包到类路径：

lib\jsf-api.jar //JSF规范接口包

lib\jsf-impl.jar //JSF规范实现包

10.4.2 最简单集成

类似于Struts1x中的最简单集成，Spring集成JSF也提供类似的工具类FacesContextUtils，使用如下方式获取WebApplicationContext：

java代码：

```
WebApplicationContext ctx = FacesContextUtils.getWebApplicationContext(FacesContext.getCurre
```

当然我们不推荐这种方式，而是推荐使用接下来介绍的另外两种方式。

10.4.2 使用VariableResolver实现集成

Spring提供javax.faces.el.VariableResolver的两种实现DelegatingVariableResolver和SpringBeanVariableResolver，其都是Spring与JSF集成的中介者，此方式适用于JSF1.1、JSF1.2及JSF2：

- DelegatingVariableResolver：首先委托给JSF默认VariableResolver实现去查找JSF管理Bean，如果找不到再委托给Spring容器去查找Spring管理Bean；
- SpringBeanVariableResolver：其与DelegatingVariableResolver查找正好相反，首先委托给Spring容器去查找Spring管理Bean，如果找不到再委托给JSF默认VariableResolver实现去查找JSF管理Bean。

接下来看一下如何在JSF中集成Spring吧（本示例使用JSF1.2，其他版本的直接替换jar包即可）：

1、JSF管理Bean（Managed Bean）实现：

java代码：

```
package cn.javass.spring.chapter10.jsf;
public class HelloBean {
    private String message;
    public void setMessage(String message) {
        this.message = message;
    }
    public String getMessage() {
        return message;
    }
```

```
}
```

2、JSF配置文件定义（resources/chapter10/jsf/faces-config.xml）：

java代码：

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config version="1.2" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-

    <application>
        <variable-resolver>
            org.springframework.web.jsf.DelegatingVariableResolver
        </variable-resolver>
    </application>

    <managed-bean>
        <managed-bean-name>helloBean</managed-bean-name>
        <managed-bean-class>
            cn.javass.spring.chapter10.jsf.HelloBean
        </managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>
        <managed-property>
            <property-name>message</property-name>
            <value>#{message}</value>
        </managed-property>
    </managed-bean>
</faces-config>
```

- **与Spring集成：**通过<variable-resolver>标签来指定集成Spring的中介者DelegatingVariableResolver；

- **注入Spring管理Bean**：通过<managed-property>标签的**<value>#{message}</value>**注入Spring管理Bean“message”。

4、JSP页面定义（webapp/hello-jsf.jsp）：

java代码：

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/l
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>
<html>
<head>
    <title>Hello World</title>
</head>
<body>
    <h:outputText value="#{helloBean.message}"/>
</body>
</html>
</f:view>
```

5、JSF前端控制器定义，在web.xml中添加如下配置：

指定JSF配置文件位置，通过javax.faces.CONFIG_FILES上下文初始化参数指定JSF配置文件位置，多个可用“，”分割，如果不指定该参数则默认加载的配置文件为“/WEB-INF/ faces-config.xml”：

java代码：

```
<!-- JSF配置文件开始 -->
<context-param>
    <param-name>javax.faces.CONFIG_FILES</param-name>
    <param-value>
        /WEB-INF/classes/chapter10/jsf/faces-config-jsf1x.xml
    </param-value>
</context-param>
<!-- JSF配置文件结束 -->
```

前端控制器定义：使用FacesServlet作为JSF的前端控制器，其拦截以“.jsf”结尾的HTTP请求：

java代码：

```
<!-- jsf前端控制器配置开始   -->
<servlet>
    <servlet-name>jsf</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>jsf</servlet-name>
    <url-pattern>*.jsf</url-pattern>
</servlet-mapping>
<!-- jsf前端控制器配置结束   -->
```

7、执行测试，在Web浏览器中输入http://localhost:8080/hello-jsf.jsp可以看到“Hello Spring”信息说明JSF集成成功。

自此，JSF集成Spring已经成功，在此可以把DelegatingVariableResolver替换为SpringBeanVariableResolver，其只有在查找相应依赖时顺序是正好相反的，其他完全一样。

如果您的项目使用JSF1.2或JSF2，推荐使用SpringBeanFacesELResolver，因为其实标准的Unified EL实现，而且VariableResolver接口已经被注释为@Deprecated，表示可能在以后的版本中去掉该接口。

10.4.2 使用ELResolver实现集成

JSF1.2之前，JSP和JSF各个使用自己的一套表达式语言（EL Language），即如JSF使用VariableResolver实现来解析JSF EL表达式，而从JSF1.2和JSP2.1开始使用Unified EL，从而统一了表达式语言。

因此集成JSF1.2+可以通过实现Unified EL来完成集成，即Spring提供ELResolver接口实现SpringBeanFacesELResolver用于集成使用。

类似于VariableResolver实现，通过SpringBeanFacesELResolver集成首先将从Spring容器中查找相应的Spring管理Bean，如果没找到再通过默认的JSF ELResolver实现查找JSF管理Bean。

接下来看一下示例一下吧：

1、 添加Unified EL所需要的jar包：

el-api.jar //Unified EL规范接口包

由于在Jetty中已经包含了该api，因此该步骤可选。

2、 修改JSF配置文件（resources/chapter10/jsf/faces-config.xml）：

将如下配置

java代码：

```
<variable-resolver>
    org.springframework.web.jsf.DelegatingVariableResolver
</variable-resolver>
```

修改为：

java代码：

```
<el-resolver>
org.springframework.web.jsf.el.SpringBeanFacesELResolver
</el-resolver>
```

3、执行测试，在Web浏览器中输入http://localhost:8080/hello-jsf.jsp可以看到“Hello Spring”信息说明JSF集成成功。

自此JSF与Spring集成就算结束了，是不是也很简单。

1.14 【第十一章】 SSH集成开发积分商城 之 11.1 概述 ——跟我学spring3

发表时间: 2012-03-13 关键字: spring, ssh

11.1 概述

11.1.1 功能概述

本节将通过介绍一个积分商城系统来演示如何使用SSH集成进行开发。

积分商城一般是购物网站的子模块，提供一些礼品或商品用于奖励老用户或使用积分来折换成现金，如图11-1所示。

图11-1 购物网站与积分商城

积分商城功能点：

- **后台管理**

交易管理模块：用于查看积分交易历史；

商品管理模块：用于CRUD积分兑换商品；

日报或月报：用于发送给运营人员每日积分兑换情况，一般通过email发送；

.....

- **前台展示**

商品展示：展示给用户可以使用积分兑换的商品；

支付模块：用户成功兑换商品后扣除用户相应积分

添加积分模块：提供接口用于其他产品赠送积分使用，如每天在线10小时赠送50积分，购买相应商品增加相应积分；

订单管理模块：订单管理模块可以使用现有购物平台的订单管理。

购物平台、用户系统及积分商城交互如图11-2所示，其中用户系统负责用户登录，购物平台是购物网站核心，积分商城用于用户使用积分购买商品。

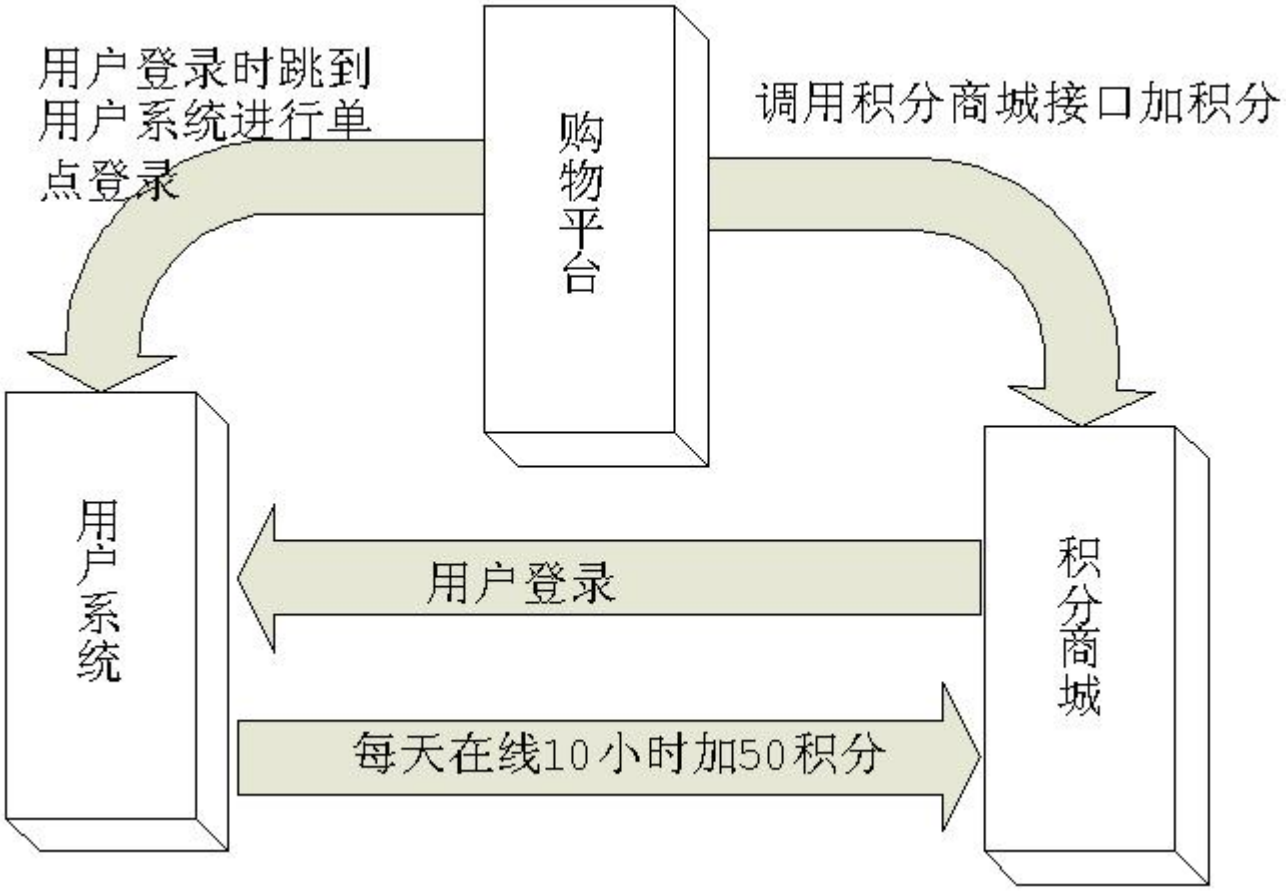


图11-2 购物平台、用户系统及积分商城交互

由于积分商城也是很复杂，由于篇幅原因不打算完全介绍，只介绍其中一个模块——商品（兑换码）管理及购买，该模块主要提供给用户使用积分兑换一些优惠券或虚拟物品（如移动充值卡）等等。

11.1.2技术选型

由于本节是关于SSH集成的，因此选用技术如下：

- 平台：Java EE；
- 运行环境：Windows XP，JDK1.6；
- 编辑器：Eclipse3.6 + SpringSource Tool Suite；
- Web容器：tomcat6.0.20；
- 数据库：mysql5.4.3；
- 框架：Struts2.0.14、Spring3.0.5、Hibernate3.6.0.Final；
- 日志记录：log4j1.2.15；
- 数据库连接池：proxool0.9.1；
- 视图技术：JSP 2.0。

技术选定了，应该考虑平台架构了，这关系到项目的成功与否。

11.1.3系统架构

积分商城系统架构也将采用经典的三层架构，如图11-3所示：

图11-3 三层架构

分层的目的是约束层次边界，每层的职责和目标应明确和单一，每层专注自己的事情，不要跨越分层边界，具体每层功能如下：

- **数据访问层：**封装底层数据库或文件系统访问细节，从而对业务逻辑层提供一致的接口，使业务逻辑层不关心底层细节；
- **业务逻辑层：**专注于业务逻辑实现，不关心底层如何访问，并在该层实现如声明式事务管理，组装分页对象；
- **表现层：**应该非常轻量级及非常“薄（功能非常少，几乎全是委托）”，拦截用户请求并响应，表现层数据验证，负责根据请求委托给业务逻辑层进行业务处理，本层不实现任何业务逻辑，且提供用户交互界面；
- **数据模型层：**数据模型定义，提供给各层使用，不应该算作三层架构中的某一层，因为数据模型可使用其他对象（如Map）代替之。

系统架构已选定，在此我们进行优化一下，因为在进行基于SSH的三层架构进行开发时通常会有一些通用功能、如通用DAO、通用Service、通用Action、通用翻页等等，因此我们再进行开发时都是基于通用功能进行的，能节省不少开发时间，从而可以使用这些节约的时间干自己想干的事情，如图10-4所示。

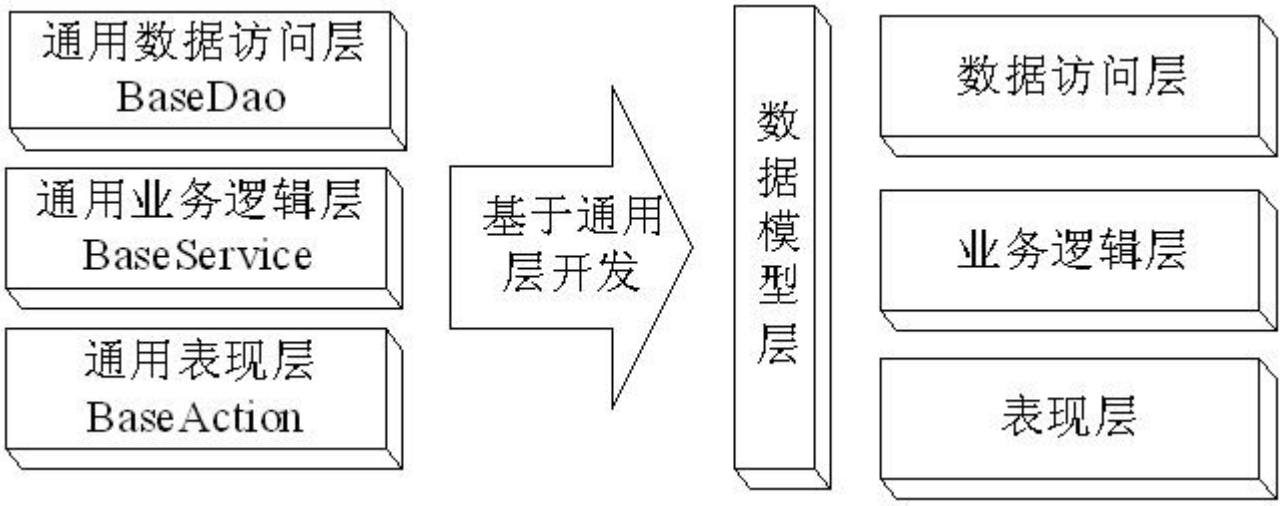


图10-4 基于通用层的三层架构

11.1.4项目搭建

1、创建动态web工程：

通过【File】>【New】>【other】>【Web】>【Dynamic Web Project】创建一个Web工程，如图11-5所示；

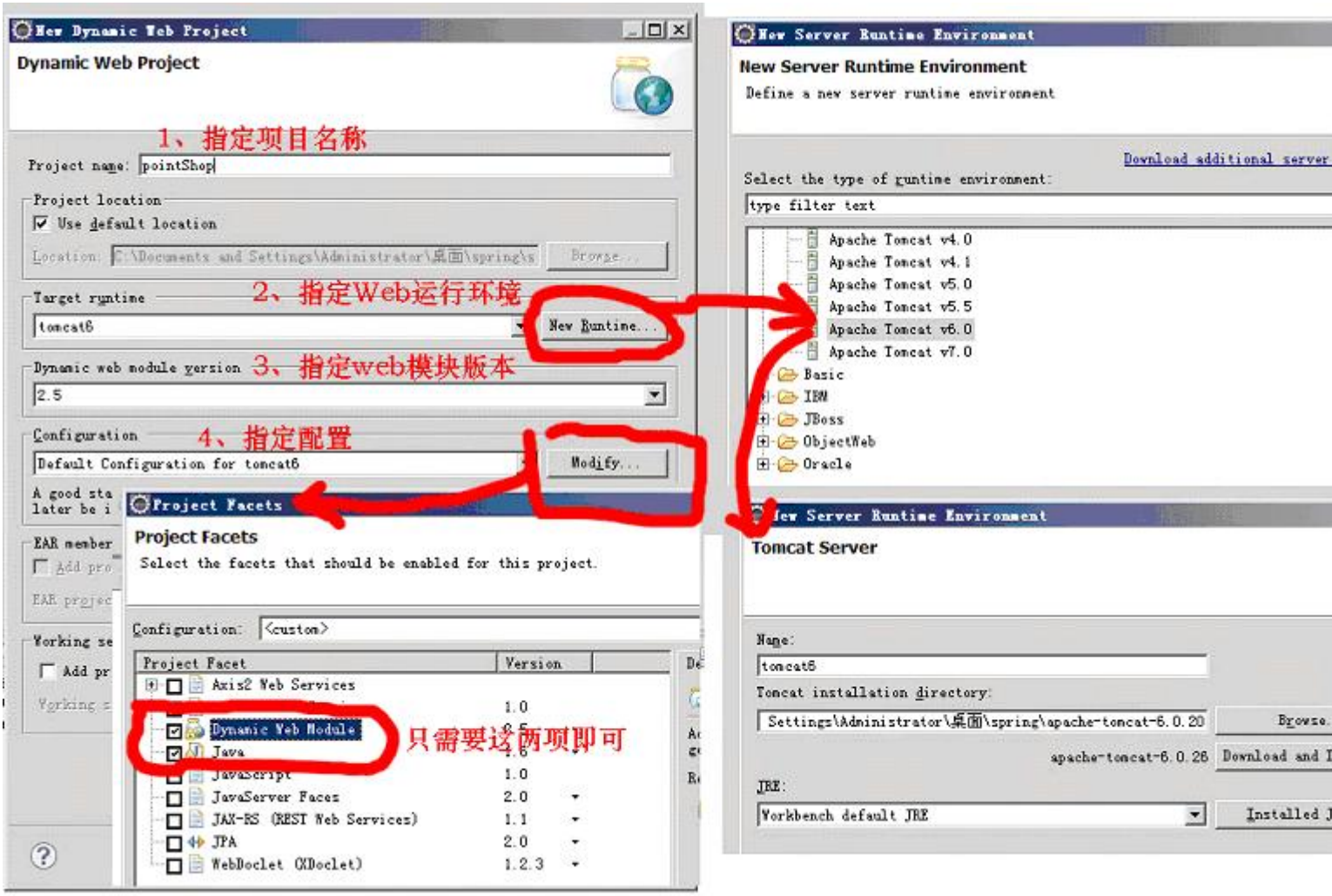


图11-5 Web工程配置

1、 项目结构，如图11-6所示：

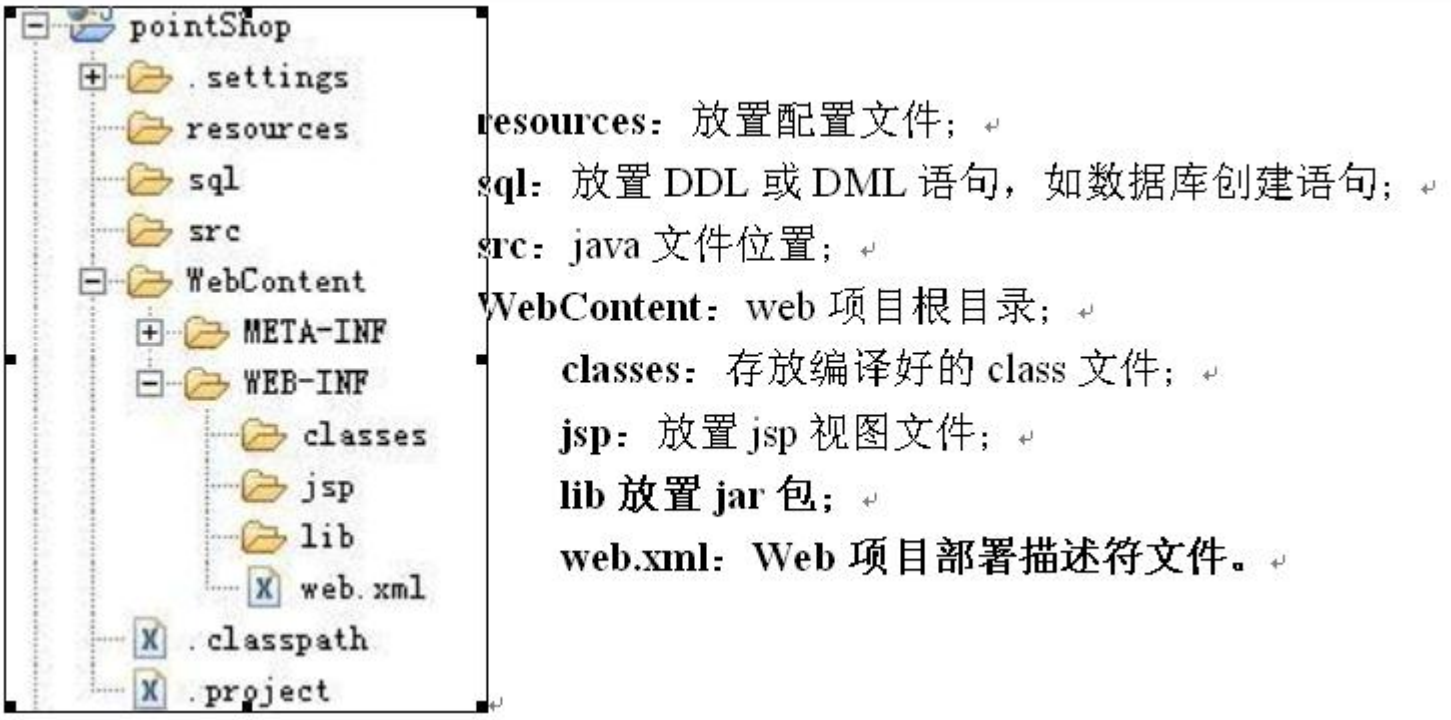


图11-6 项目结构

3、项目属性修改：

3.1、字符编码修改，如图11-7所示，在实际项目中一定要统一字符编码：

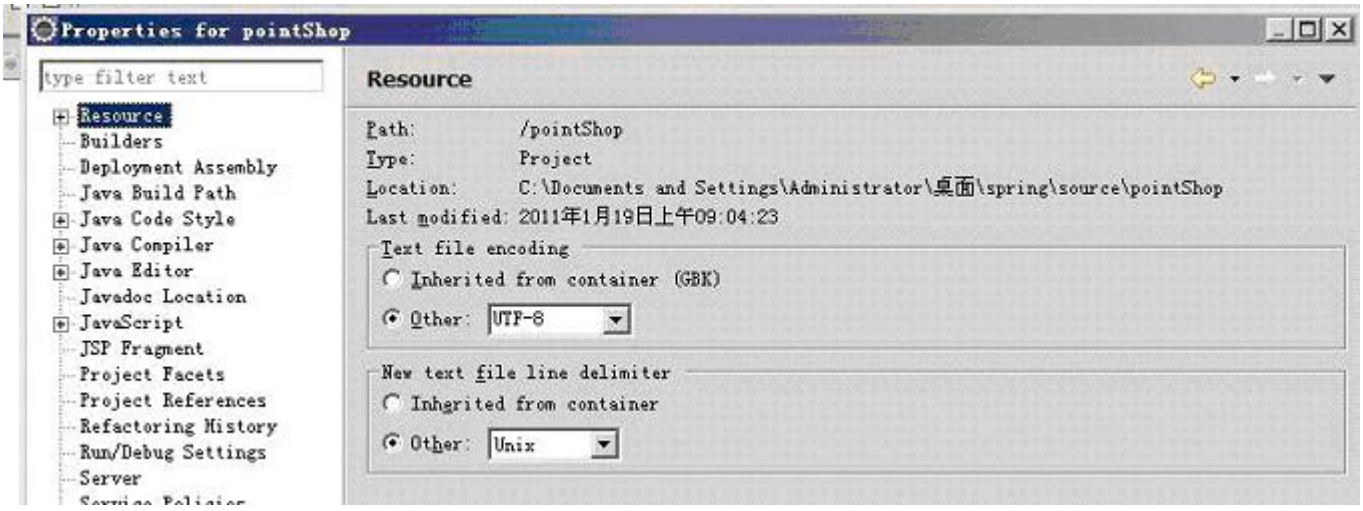


图11-7 修改项目字符编码

3.2、类路径输出修改，如图11-8，将类路径输出改为/WEB-INF/classes下：

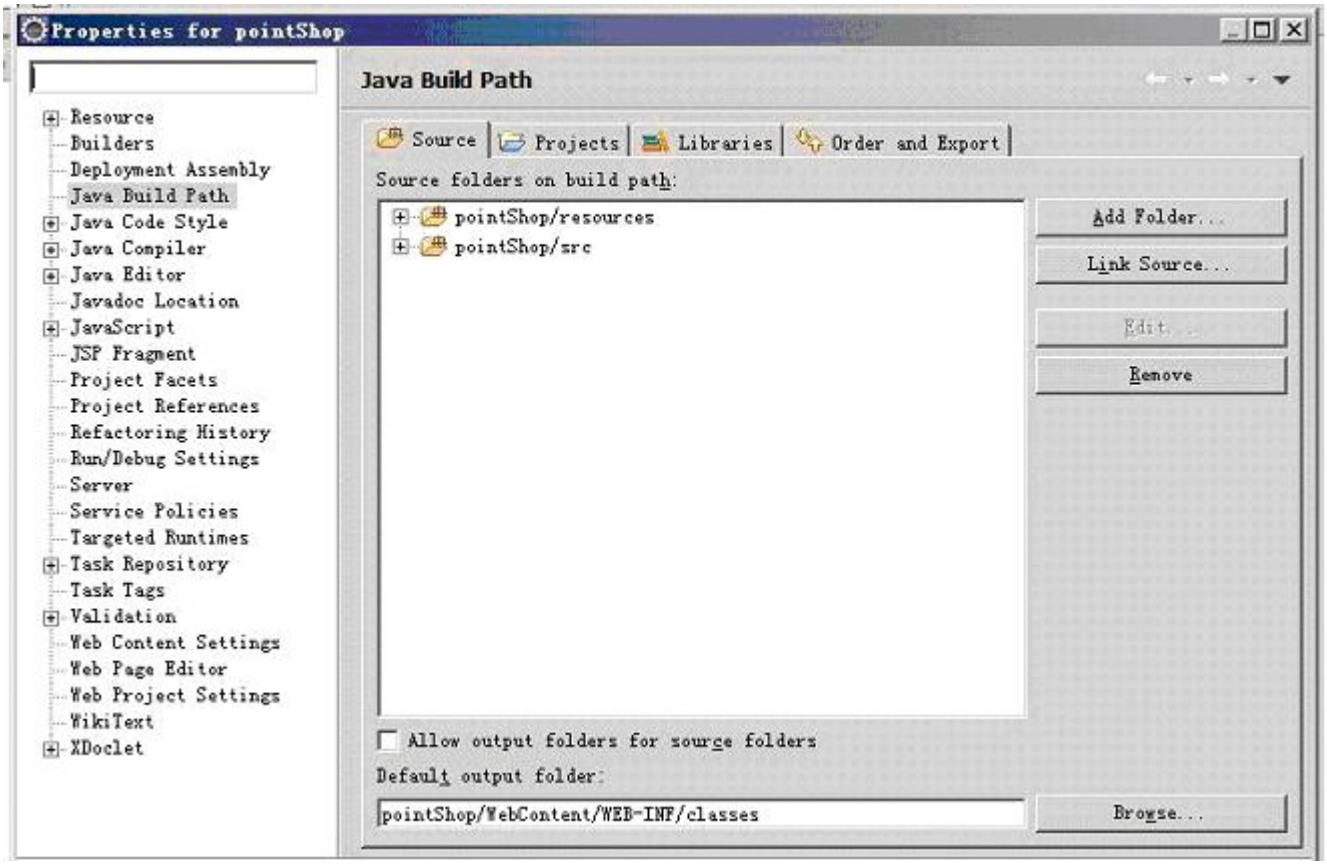


图11-8 类路径修改

4、准备jar包：

4.1、Spring项目依赖包，到下载的spring-framework-3.0.5.RELEASE-with-docs.zip中拷贝如下jar包：

dist\org.springframework.aop-3.0.5.RELEASE.jar

dist\org.springframework.asm-3.0.5.RELEASE.jar

dist\org.springframework.beans-3.0.5.RELEASE.jar

dist\org.springframework.context-3.0.5.RELEASE.jar

dist\org.springframework.core-3.0.5.RELEASE.jar

dist\org.springframework.expression-3.0.5.RELEASE.jar

dist\org.springframework.jdbc-3.0.5.RELEASE.jar

dist\org.springframework.orm-3.0.5.RELEASE.jar

dist\org.springframework.transaction-3.0.5.RELEASE.jar

dist\org.springframework.web-3.0.5.RELEASE.jar

4.2、Spring及其他项目依赖包，到spring-framework-3.0.5.RELEASE-dependencies.zip中拷贝如下jar吧：

com.springsource.net.sf.cglib-2.2.0.jar

com.springsource.org.aopalliance-1.0.0.jar

com.springsource.org.apache.commons.beanutils-1.8.0.jar

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

com.springsource.org.apache.commons.digester-1.8.1.jar

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

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

com.springsource.org.apache.taglibs.standard-1.1.2.jar

com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar

4.3、Hibernate依赖包，到hibernate-distribution-3.6.0.Final.zip中拷贝如下jar包：

hibernate3.jar

lib\jpa\hibernate-jpa-2.0-api-1.0.0.Final.jar

lib\required\dom4j-1.6.1.jar

lib\required\javassist-3.12.0.GA.jar

lib\required\jta-1.1.jar

lib\ required \slf4j-api-1.6.1.jar

lib\required\antlr-2.7.6.jar

4.4、数据库连接池依赖包，到proxool-0.9.1.zip中拷贝如下jar包：

lib\proxool-0.9.1.jar

lib\proxool-cglib.jar

4.5、准备mysql JDBC连接依赖包：

mysql-connector-java-5.1.10.jar

4.6、slf4j依赖包准备，到下载的slf4j-1.6.1.zip包中拷贝如下jar包：

slf4j-log4j12-1.6.1.jar

4.7、Strut2依赖包，到struts-2.2.1.1.zip中拷贝如下jar包：

lib\struts2-core-2.2.1.1.jar

lib\xwork-core-2.2.1.1.jar

lib\freemarker-2.3.16.jar

lib\ognl-3.0.jar

lib\struts2-spring-plugin-2.2.1.1.jar

lib\commons-fileupload-1.2.1.jar

jar包终于准备完了，是不是很头疼啊，在此推荐使用maven进行依赖管理，无需拷贝这么多jar包，而是通过配置方式来指定使用的依赖，具体maven知识请到官方网站http://maven.apache.org/了解。

1.15 【第十一章】 SSH集成开发积分商城 之 11.2 实现通用层 ——跟我学spring3

发表时间: 2012-03-14 关键字: spring

11.2 实现通用层

11.2.1 功能概述

通过抽象通用的功能，从而复用，减少重复工作：

- 对于一些通用的常量使用一个专门的常量类进行定义；
- 对于视图分页，也应该抽象出来，如JSP做出JSP标签；
- 通用的数据层代码，如通用的CRUD，减少重复劳动，节约时间；
- 通用的业务逻辑层代码，如通用的CRUD，减少重复劳动，节约时间；
- 通用的表现层代码，同样用于减少重复，并提供更好的代码结构规范。

11.2.2 通用的常量定义

目标：在一个常量类中定义通用的常量的好处是如果需要修改这些常量值只需在一个地方修改即可，变的地方只有一处而不是多处。

如默认分页大小如果在多处硬编码定义为10，突然发生变故需要将默认分页大小10为5，怎么办？如果当初我们提取出来放在一个通用的常量类中是不是只有一处变动。

java代码：

```
package cn.javass.commons;
public class Constants {
    public static final int DEFAULT_PAGE_SIZE = 5;  //默认分页大小
    public static final String DEFAULT_PAGE_NAME = "page";
    public static final String CONTEXT_PATH = "ctx";
}
```

如上代码定义了通用的常量，如默认分页大小。

11.2.2通用分页功能

分页功能是项目中必不可少的功能，因此通用分页功能十分有必要，有了通用的分页功能，即有了规范，从而保证视图页面的干净并节约了开发时间。

1、 分页对象定义，用于存放是否有上一页或下一页、当前页记录、当前页码、分页上下文，该对象是分页中必不可少对象，一般在业务逻辑层组装Page对象，然后传送到表现层展示，然后通用的分页标签使用该对象来决定如何显示分页：

java代码：

```
package cn.javass.commons.pagination;
import java.util.Collections;
import java.util.List;
public class Page<E> {/** 表示分页中的一页。*/
    private boolean hasPre; //是否有上一页
    private boolean hasNext; //是否有下一页
    private List<E> items; //当前页包含的记录列表
    private int index; //当前页页码(起始为1)
    //省略setter
    public int getIndex() {
        return this.index;
    }
    public boolean isHasPre() {
        return this.hasPre;
    }
    public boolean isHasNext() {
        return this.hasNext;
    }
    public List<E> getItems() {
        return this.items == null ? Collections.<E>emptyList() : this.items;
    }
```

```
}
```

2、 **分页标签实现**，将使用Page对象数据决定如何展示分页，如图11-9和11-10所示：

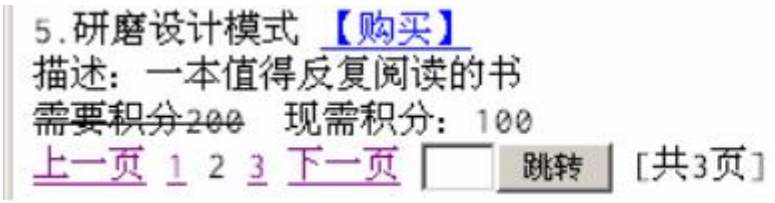


图11-9 11-10 通用分页标签实现

图11-9和11-10展示了两种分页展示策略，由于分页标签和集成SSH没多大关系且不是必须的并由于篇幅问题不再列出分页标签源代码，有兴趣的朋友请参考cn.javass.commons.pagination.NavigationTag类文件。代码下载地址

11.2.3 通用数据访问层

目标：通过抽象实现最基本的CURD操作，提高代码复用，可变部分按需实现。

1、通用数据访问层接口定义

java代码：

```
package cn.javass.commons.dao;
import java.io.Serializable;
import java.util.List;
public interface IBaseDao<M extends Serializable, PK extends Serializable> {
    public void save(M model);// 保存模型对象
    public void saveOrUpdate(M model);// 保存或更新模型对象
```

```
    public void update(M model);// 更新模型对象
    public void merge(M model);// 合并模型对象状态到底层会话
    public void delete(PK id);// 根据主键删除模型对象
    public M get(PK id);// 根据主键获取模型对象
    public int countAll();//统计模型对象对应数据库表中的记录数
    public List<M> listAll();//查询所有模型对象
    public List<M> listAll(int pn, int pageSize);// 分页获取所有模型对象
}
```

通用DAO接口定义了如CRUD通用操作，而可变的（如查询所有已发布的接口，即有条件查询等）需要在相应DAO接口中定义，并通过泛型“M”指定数据模型类型和“PK”指定数据模型主键类型。

2、通用数据访问层DAO实现

此处使用Hibernate实现，即实现是可变的，对业务逻辑层只提供面向接口编程，从而隐藏数据访问层实现细节。

实现时首先通过反射获取数据模型类型信息，并根据这些信息获取Hibernate对应的数据模型的实体名，再根据实体名组装出通用的查询和统计记录的HQL，从而达到同样目的。

注意我们为什么把实现生成HQL时放到init方法中而不是构造器中呢？因为SessionFactory是通过setter注入，setter注入晚于构造器注入，因此在构造器中使用SessionFactory会是null，因此放到init方法中，并在Spring配置文件中指定初始化方法为init来完成生成HQL。

java代码：

```
package cn.javass.commons.dao.hibernate;
//为节省篇幅省略import
public abstract class BaseHibernateDao<M extends Serializable, PK extends Serializable> extends
    private Class<M> entityClass;
```

```
    private String HQL_LIST_ALL;
    private String HQL_COUNT_ALL;
    @SuppressWarnings("unchecked")
    public void init() {//通过初始化方法在依赖注入完毕时生成HQL
        //1、通过反射获取注解“M”（即模型对象）的类类型
        this.entityClass = (Class<M>) ((ParameterizedType) getClass().getGenericSuperclass()).
        //2、得到模型对象的实体名
        String entityName = getSessionFactory().getClassMetadata(this.entityClass).getEntityNa
        //3、根据实体名生成HQL
        HQL_LIST_ALL = "from " + entityName;
        HQL_COUNT_ALL = " select count(*) from " + entityName;
    }
    protected String getListAllHql() {//获取查询所有记录的HQL
        return HQL_LIST_ALL;
    }
    protected String getCountAllHql() {//获取统计所有记录的HQL
        return HQL_COUNT_ALL;
    }
    public void save(M model) {
        getHibernateTemplate().save(model);
    }
    public void saveOrUpdate(M model) {
        getHibernateTemplate().saveOrUpdate(model);
    }
    public void update(M model) {
        getHibernateTemplate().update(model);
    }

    public void merge(M model) {
        getHibernateTemplate().merge(model);
    }
    public void delete(PK id) {
        getHibernateTemplate().delete(this.get(id));
    }
    public M get(PK id) {
        return getHibernateTemplate().get(this.entityClass, id);
```

```
 }
 public int countAll() {
      Number total = unique(getCountAllHql());
      return total.intValue();
}
 public List<M> listAll() {
      return list(getListAllHql());
 }
 public List<M> listAll(int pn, int pageSize) {
      return list(getListAllHql(), pn, pageSize);
 }
 protected <T> List<T> list(final String hql, final Object... paramlist) {
     return list(hql, -1, -1, paramlist);//查询所有记录
 }
  /** 通用列表查询,当pn<=-1 且 pageSize<=-1表示查询所有记录
    * @param <T> 模型类型
    * @param hql Hibernate查询语句
    * @param pn  页码 从1开始，
    * @param pageSize 每页记录数
    * @param paramlist 可变参数列表
    * @return 模型对象列表
    */
  @SuppressWarnings("unchecked")
 protected <T> List<T> list(final String hql, final int pn, final int pageSize, final Object
  return getHibernateTemplate().
      executeFind(new HibernateCallback<List<T>>() {
          public List<T> doInHibernate(Session session)  throws HibernateException, SQLExcep
              Query query = session.createQuery(hql);
              if (paramlist != null) {
                  for (int i = 0; i < paramlist.length; i++) {
                      query.setParameter(i, paramlist[i]);//设置占位符参数
                  }
              }
              if (pn > -1 && pageSize > -1) {//分页处理
                  query.setMaxResults(pageSize);//设置将获取的最大记录数
                  int start = PageUtil.getPageStart(pn, pageSize);
```

```
                    if(start != 0) {
                        query.setFirstResult(start);//设置记录开始位置
                    }
                }
                return query.list();
            }
        });
    }
   /**  根据查询条件返回唯一一条记录
     * @param <T> 返回类型
     * @param hql Hibernate查询语句
     * @param paramlist 参数列表
     * @return 返回唯一记录 */
    @SuppressWarnings("unchecked")
    protected <T> T unique(final String hql, final Object... paramlist) {
        return getHibernateTemplate().execute(new HibernateCallback<T>() {
            public T doInHibernate(Session session) throws HibernateException, SQLException {
                Query query = session.createQuery(hql);
                if (paramlist != null) {
                    for (int i = 0; i < paramlist.length; i++) {
                        query.setParameter(i, paramlist[i]);
                    }
                }
                return (T) query.setMaxResults(1).uniqueResult();
            }
        });
   }
   //省略部分可选的便利方法，想了解更多请参考源代码
}
```

通用DAO实现代码相当长，但麻烦一次，以后有了这套通用代码将会让工作很轻松，该通用DAO还有其他便利方法因为本示例不需要且由于篇幅原因没加上，请参考源代码。

11.2.4 通用业务逻辑层

目标：实现通用的业务逻辑操作，将常用操作封装提高复用，可变部分同样按需实现。

1、通用业务逻辑层接口定义

java代码：

```
package cn.javass.commons.service;
//由于篇幅问题省略import
public interface IBaseService<M extends Serializable, PK extends Serializable> {
    public M save(M model); //保存模型对象
    public void saveOrUpdate(M model);// 保存或更新模型对象
    public void update(M model);// 更新模型对象
    public void merge(M model);// 合并模型对象状态
    public void delete(PK id);// 删除模型对象
    public M get(PK id);// 根据主键获取模型对象
    public int countAll();//统计模型对象对应数据库表中的记录数
    public List<M> listAll();//获取所有模型对象
    public Page<M> listAll(int pn);// 分页获取默认分页大小的所有模型对象
    public Page<M> listAll(int pn, int pageSize);// 分页获取所有模型对象
}
```

3、 通用业务逻辑层接口实现

通用业务逻辑层通过将通用的持久化操作委托给DAO层来实现通用的数据模型CRUD等操作。

通过通用的setDao方法注入通用DAO实现，在各Service实现时可以通过强制转型获取各转型后的DAO。

java代码：

```
package cn.javass.commons.service.impl;
//由于篇幅问题省略import
public abstract class BaseServiceImpl<M extends Serializable, PK extends Serializable> implemer
    protected IBaseDao<M, PK> dao;
    public void setDao(IBaseDao<M, PK> dao) {//需要依赖注入
        this.dao = dao;
    }
    public IBaseDao<M, PK> getDao() {
        return this.dao;
    }
    public M save(M model) {
        getDao().save(model);
        return model;
    }
    public void merge(M model) {
        getDao().merge(model);
    }
    public void saveOrUpdate(M model) {
        getDao().saveOrUpdate(model);
    }
    public void update(M model) {
        getDao().update(model);
    }
    public void delete(PK id) {
        getDao().delete(id);
    }
    public void deleteObject(M model) {
        getDao().deleteObject(model);
    }
    public M get(PK id) {
        return getDao().get(id);
    }
    public int countAll() {
        return getDao().countAll();
    }
```

```
    public List<M> listAll() {
        return getDao().listAll();
    }
    public Page<M> listAll(int pn) {
        return this.listAll(pn, Constants.DEFAULT_PAGE_SIZE);
    }
    public Page<M> listAll(int pn, int pageSize) {
        Integer count = countAll();
        List<M> items = getDao().listAll(pn, pageSize);
        return PageUtil.getPage(count, pn, items, pageSize);
    }
}
```

11.2.6通用表现层

目标：规约化常见请求和响应操作，将常见的CURD规约化，采用规约编程提供开发效率，减少重复劳动。

Struts2常见规约编程：

- **通用字段驱动注入：**如分页字段一般使用“pn”或“page”来指定当前分页页码参数名，通过Struts2的字段驱动注入实现分页页码获取的通用化；
- **通用Result：**对于CURD操作完全可以提取公共的Result名字，然后在Strust2配置文件中进行规约配置；
- **数据模型属性名：**在页面展示中，如新增和修改需要向值栈或请求中设置数据模型，在此我们定义统一的数据模型名如“model”，这样在项目组中形成约定，大家只要按照约定来能提高开发效率；
- **分页对象属性名：**与数据模型属性名同理，在此我们指定为“page”；
- **便利方法：**如获取值栈、请求等可以提供公司内部需要的便利方法。

1、通用表现层Action实现：

java代码：

```
package cn.javass.commons.web.action;
import cn.javass.commons.Constants;
//省略import
public class BaseAction extends ActionSupport {
    /** 通用Result */
    public static final String LIST = "list";
    public static final String REDIRECT = "redirect";
    public static final String ADD = "add";
    /** 模型对象属性名*/
    public static final String MODEL = "model";
    /** 列表模型对象属性名*/
    public static final String PAGE = Constants.DEFAULT_PAGE_NAME;
    public static final int DEFAULT_PAGE_SIZE = Constants.DEFAULT_PAGE_SIZE;
    private int pn = 1;  /** 页码，默认为1 */
    //省略pn的getter和setter，自己补上
    public ActionContext getActionContext() {
        return ActionContext.getContext();
    }
    public ValueStack getValueStack() {//获取值栈的便利方法
        return getActionContext().getValueStack();
    }
}
```

2、通用表现层JSP视图实现：

将视图展示的通用部分抽象出来减少页面设计的工作量。

2.1、通用JSP页头文件（WEB-INF/jsp/common/inc/header.jsp）：

此处实现比较简单，实际中可能包含如菜单等信息，对于可变部分使用请求参数来获取，从而保证了可变与不可变分离，如标题使用“${param.title}”来获取。

java代码：

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loos
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>${param.title}</title>
</head>
<body>
```

2.2、通用JSP页尾文件（WEB-INF/jsp/common/inc/footer.jsp）：

此处比较简单，实际中可能包含公司版权等信息。

java代码：

```
</body>
</html>
```

2.3、通用JSP标签定义文件（WEB-INF/jsp/common/inc/tld.jsp）：

在一处定义所有标签，避免标签定义使代码变得凌乱，且如果有多个页面需要新增或删除标签即费事又费力。

java代码：

```
<%@taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<%@taglib prefix="s" uri="/struts-tags" %
```

2.4、通用错误JSP文件（WEB-INF/jsp/common/error.jsp）：

当系统遇到错误或异常时应该跳到该页面来显示统一的错误信息并可能在该页保存异常信息。

java代码：

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
<jsp:include page="inc/header.jsp"/>
失败或遇到异常！
<jsp:include page="inc/footer.jsp"/>
```

2.5、通用正确JSP文件（WEB-INF/jsp/common/success.jsp）：

对于执行成功的操作可以使用通用的页面表示，可变部分同样可以使用可变的请求参数传入。

java代码：

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
<jsp:include page="inc/header.jsp"/>
成功！
<jsp:include page="inc/footer.jsp"/>
```

3、通用设置web环境上下文路径拦截器：

用于设置当前web项目的上下文路径，即可以在JSP页面使用“${ctx}”获取当前上下文路径。

java代码：

```
package cn.javass.commons.web.filter;
//省略import
/** 用户设置当前web环境上下文，用于方便如JSP页面使用 */
public class ContextPathFilter implements Filter {
    @Override
    public void init(FilterConfig config) throws ServletException {
    }
    @Override
public void doFilter(
ServletRequest request, ServletResponse response, FilterChain chain)
throws IOException, ServletException {
        String contextPath = ((HttpServletRequest) request).getContextPath();
        request.setAttribute(Constants.CONTEXT_PATH, contextPath);
        chain.doFilter(request, response);
    }
    @Override
    public void destroy() {
    }
}
```

11.2.7通用配置文件

目标：通用化某些常用且不可变的配置文件，同样目标是提高复用，减少工作量。

1、Spring资源配置文件（resources/applicationContext-resources.xml）：

定义如配置元数据替换Bean、数据源Bean等通用的Bean。

java代码：

```
<bean class=
"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
<property name="locations">
            <list>
                <value>classpath:resources.properties</value>
            </list>
        </property>
    </bean>
    <bean id="dataSource" class="org.springframework.jdbc.datasource.LazyConnectionDataSourcePr
        <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.simultaneousBuildTh
                <property name="trace" value="${proxool.trace}" />
            </bean>
        </property>
    </bean>
</beans>
```

通过通用化如数据源来提高复用，对可变的如数据库驱动、URL、用户名等采用替换配置元数据形式进行配置，具体配置含义请参考【7.5集成Spring JDBC及最佳实践】。

2、替换配置元数据的资源文件（resources/resources.properties）：

定义替换配置元数据键值对用于替换Spring配置文件中可变的配置元数据。

java代码：

```
#数据库连接池属性
proxool.maxConnCount=10
proxool.minConnCount=5
proxool.statistics=1m,15m,1h,1d
proxool.simultaneousBuildThrottle=30
proxool.trace=false
db.driver.class=com.mysql.jdbc.Driver
db.url=jdbc:mysql://localhost:3306/point_shop?useUnicode=true&characterEncoding=utf8
db.username=root
db.password=
```

3、通用Struts2配置文件（WEB-INF/struts.xml）：

由于是要集成Spring，因此需要使用StrutsSpringObjectFactory，我们需要在action名字中出现“/”因此定义struts.enable.SlashesInActionNames=true。

在此还定义了“custom-default”包继承struts-default包，且是抽象的，在包里定义了如全局结果集全局异常映射。

java代码：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
```

```
    "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <constant name="struts.objectFactory" value="org.apache.struts2.spring.StrutsSpringObjectF
    <!-- 允许action的名字中出现"/" -->
    <constant name="struts.enable.SlashesInActionNames" value="true"/>
    <package name="custom-default" extends="struts-default" abstract="true">
        <global-results>
            <result name="success">/WEB-INF/jsp/common/success.jsp</result>
            <result name="error">/WEB-INF/jsp/common/error.jsp</result>
            <result name="exception">/WEB-INF/jsp/common/error.jsp</result>
        </global-results>
        <global-exception-mappings>
           <exception-mapping result="exception" exception="java.lang.Exception"/>
        </global-exception-mappings>
    </package>
</struts>
```

4、通用log4j日志记录配置文件（resources/log4j.xml）：

可以配置基本的log4j配置文件然后在其他地方通过拷贝来定制需要的日志记录配置。

java代码：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
       <!-- Appenders -->
       <appender name="console" class="org.apache.log4j.ConsoleAppender">
              <param name="Target" value="System.out" />
              <layout class="org.apache.log4j.PatternLayout">
                     <param name="ConversionPattern" value="%-5p: %c - %m%n" />
              </layout>
```

```
        </appender>
        <!-- Root Logger -->
        <root>
                <priority value="DEBUG" />
                <appender-ref ref="console" />
        </root>
</log4j:configuration>
```

4、通用web.xml配置文件定义（WEB-INF/web.xml）：

定义如通用的集成配置、设置web环境上下文过滤器、字符过滤器（防止乱码）、通用的Web框架拦截器（如Struts2的）等等，从而可以通过拷贝复用。

java代码：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.5" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xm
    <!-- 通用配置开始   -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            classpath:applicationContext-resources.xml
        </param-value>
    </context-param>
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    <!-- 通用配置结束   -->
```

```
<!-- 设置web环境上下文（方便JSP页面获取）开始    -->
<filter>
    <filter-name>Set Context Path</filter-name>
    <filter-class>cn.javass.commons.web.filter.ContextPathFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>Set Context Path</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
<!-- 设置web环境上下文（方便JSP页面获取）结束    -->
<!-- 字符编码过滤器（防止乱码）开始    -->
<filter>
    <filter-name>Set Character Encoding</filter-name>
    <filter-class>
        org.springframework.web.filter.CharacterEncodingFilter
    </filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>UTF-8</param-value>
    </init-param>
    <init-param>
        <param-name>forceEncoding</param-name>
        <param-value>true</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>Set Character Encoding</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
<!-- 字符编码过滤器（防止乱码）结束    -->
<!-- Struts2.x前端控制器配置开始    -->
<filter>
    <filter-name>struts2Filter</filter-name>
    <filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
</filter>
```

```
    <filter-mapping>
        <filter-name>struts2Filter</filter-name>
        <url-pattern>*.action</url-pattern>
    </filter-mapping>
    <!-- Struts2.x前端控制器配置结束    -->
</web-app>
```

1.16 【第十一章】 SSH集成开发积分商城 之 11.3 实现积分商城层 ——跟我学spring3

发表时间: 2012-03-16 关键字: spring, ssh

11.3 实现积分商城层

11.3.1 概述

积分商城是基于通用层之上进行开发，这样我们能减少很多重复的劳动，加快项目开发进度。

11.3.2 实现数据模型层

1、商品表，定义了如商品名称、简介、原需积分、现需积分等，其中是否发布表示只有发布（true）了的商品才会在前台删除，是否已删除表示不会物理删除，商品不应该物理删除，而是逻辑删除，版本属性用于防止并发更新。

java代码：

```
package cn.javass.point.model;
/** 商品表 */
@Entity
@Table(name = "tb_goods")
public class GoodsModel implements java.io.Serializable {
     /** 主键 */
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", length = 10)
    private int id;
    /** 商品名称 */
    @Column(name = "name", nullable = false, length = 100)
    private String name;
    /** 商品简介 */
    @Column(name = "description", nullable = false, length = 100)
    private String description;
    /** 原需积分 */
    @Column(name = "original_point", nullable = false, length = 10)
    private int originalPoint;
    /** 现需积分 */
```

```
    @Column(name = "now_point", nullable = false, length = 10)
    private int nowPoint;
    /** 是否发布，只有发布的在前台显示 */
    @Column(name = "published", nullable = false)
    private boolean published;
    /** 是否删除，商品不会被物理删除的 */
    @Column(name = "is_delete", nullable = false)
    private boolean deleted;
    /** 版本 */
    @Version  @Column(name = "version", nullable = false, length = 10)
    private int version;
    //省略getter和setter、hashCode及equals，实现请参考源代码
}
```

2、商品兑换码表，定义了兑换码、兑换码所属商品（兑换码和商品直接是多对一关系）、购买人、购买时间、是否已经购买（防止一个兑换码多个用户兑换）、版本。

java代码：

```
package cn.javass.point.model;
import java.util.Date;
//省略部分import
/** 商品兑换码表 */
@Entity
@Table(name = "tb_goods_code")
public class GoodsCodeModel implements java.io.Serializable {
     /** 主键 */
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", length = 10)
    private int id;
```

```
    /** 所属商品 */
    @ManyToOne
    private GoodsModel goods;
    /** 兑换码*/
    @Column(name = "code", nullable = false, length = 100)
    private String code;
    /** 兑换人,实际环境中应该和用户表进行对应*/
    @Column(name = "username", nullable = true, length = 100)
    private String username;

    /** 兑换时间*/
    @Column(name = "exchange_time")
    private Date exchangeTime;
    /** 是否已经兑换*/
    @Column(name = "exchanged")
    private boolean exchanged = false;
    /** 版本 */
    @Version
    @Column(name = "version", nullable = false, length = 10)
    private int version;
    //省略getter和setter、hashCode及equals，实现请参考源代码
}
```

3、商品表及商品兑换码表之间关系，即一个商品有多个兑换码，如图11-10所示：

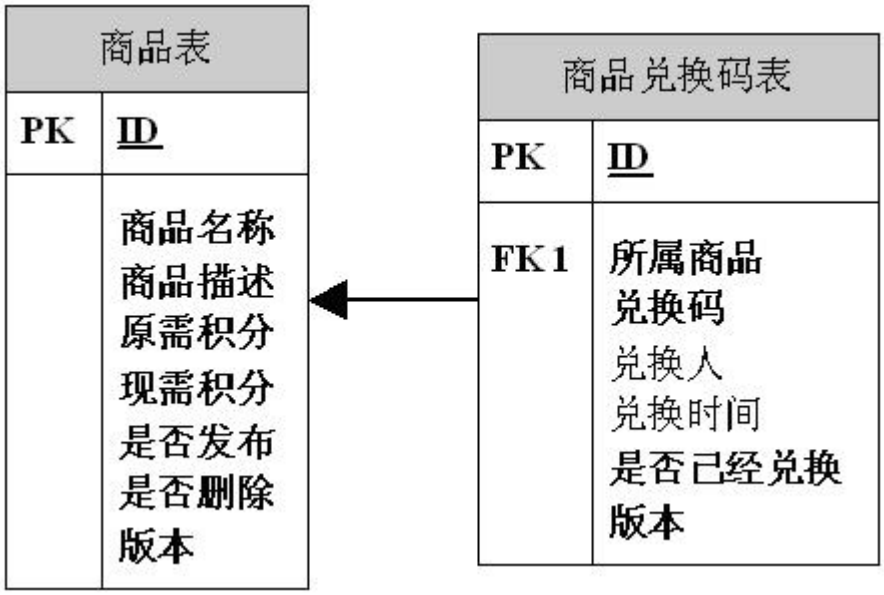


图11-10商品表及商品兑换码表之间关系

4、 创建数据库及表结构的SQL语句文件（sql/ pointShop_schema.sql）：

java代码：

```
CREATE DATABASE IF NOT EXISTS 'point_shop'
DEFAULT CHARACTER SET 'utf8';
USE 'point_shop';
DROP TABLE IF EXISTS 'tb_goods_code';
DROP TABLE IF EXISTS 'tb_goods';
-- ----------------------------
-- Table structure for 商品表
-- ----------------------------
CREATE TABLE 'tb_goods' (
  'id' int(10) unsigned NOT NULL AUTO_INCREMENT COMMENT '商品id',
  'name' varchar(100) NOT NULL COMMENT '商品名称',
  'description' varchar(100) NOT NULL COMMENT '商品简介',

  'original_point' int(10) unsigned NOT NULL  COMMENT '原需积分',
  'now_point' int(10) unsigned NOT NULL  COMMENT '现需积分',
  'published' bool NOT NULL  COMMENT '是否发布',
  'is_delete' bool NOT NULL DEFAULT false COMMENT '是否删除',
  'version' int(10) unsigned NOT NULL DEFAULT 0 COMMENT '版本',
```

```
   PRIMARY KEY ('id'),
   INDEX('name'),
   INDEX('published')
  )ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='商品表';
-- ----------------------------
-- Table structure for 商品兑换码表
-- ----------------------------
CREATE TABLE 'tb_goods_code' (
  'id' int(10) unsigned NOT NULL AUTO_INCREMENT COMMENT '主键id',
  'username' varchar(100) COMMENT '兑换用户',
  'goods_id' int(10) unsigned NOT NULL COMMENT '所属商品id',
  'code' varchar(100)  NOT NULL COMMENT '积分',
  'exchange_time' datetime  COMMENT '购买时间',
  'exchanged' bool DEFAULT false COMMENT '是否已经兑换',
  'version' int(10) unsigned NOT NULL DEFAULT 0 COMMENT '版本',
   PRIMARY KEY ('id'),
   FOREIGN KEY ('goods_id') REFERENCES 'tb_goods' ('id')  ON DELETE CASCADE
  )ENGINE=InnoDB AUTO_INCREMENT=1000000 DEFAULT CHARSET=utf8 COMMENT='商品兑换码表';
```

Mysql数据库引擎应该使用InnoDB，如果使用MyISM将不支持事务。

11.3.3 实现数据访问层

数据访问层只涉及与底层数据库或文件系统等打交道，不会涉及业务逻辑，一定注意层次边界，不要在数据访问层实现业务逻辑。

商品模块的应该实现如下功能：

- 继承通用数据访问层的CRUD功能；
- 分页查询所有已发布的商品
- 统计所有已发布的商品；

商品兑换码模块的应该实现如下功能：

- 继承通用数据访问层的CRUD功能；
- 根据商品ID分页查询该商品的兑换码
- 根据商品ID统计该商品的兑换码记录数；
- 根据商品ID获取一个还没有兑换的商品兑换码

1、商品及商品兑换码DAO接口定义：

商品及商品兑换码DAO接口定义直接继承IBaseDao，无需在这些接口中定义重复的CRUD方法了，并通过泛型指定数据模型类及主键类型。

java代码：

```
package cn.javass.point.dao;
//省略import
/** 商品模型对象的DAO接口 */
public interface IGoodsDao extends IBaseDao<GoodsModel, Integer> {
    /** 分页查询所有已发布的商品*/
    List<GoodsModel> listAllPublished(int pn);
    /** 统计所有已发布的商品记录数*/
    int countAllPublished();
}
```

java代码：

```
package cn.javass.point.dao;
//省略import
```

```
/** 商品兑换码模型对象的DAO接口 */
public interface IGoodsCodeDao extends IBaseDao<GoodsCodeModel, Integer> {
    /** 根据商品ID统计该商品的兑换码记录数*/
    public int countAllByGoods(int goodsId);
    /** 根据商品ID查询该商品的兑换码列表*/
    public List<GoodsCodeModel> listAllByGoods(int pn, int goodsId);
    /** 根据商品ID获取一个还没有兑换的商品兑换码 */
    public GoodsCodeModel getOneNotExchanged(int goodsId);
}
```

2、 商品及商品兑换码DAO接口实现定义：

DAO接口实现定义都非常简单，对于CRUD实现直接从BaseHibernateDao继承即可，无需再定义重复的CRUD实现了，并通过泛型指定数据模型类及主键类型。

java代码：

```
package cn.javass.point.dao.hibernate;
//省略import
public class GoodsHibernateDao extends BaseHibernateDao<GoodsModel, Integer> implements IGoodsD
    @Override  //覆盖掉父类的delete方法，不进行物理删除
    public void delete(Integer id) {
        GoodsModel goods = get(id);
        goods.setDeleted(true);
        update(goods);
    }
    @Override //覆盖掉父类的getCountAllHql方法，查询不包括逻辑删除的记录
    protected String getCountAllHql() {
        return super.getCountAllHql() + " where deleted=false";
    }
```

```
    @Override //覆盖掉父类的getListAllHql方法，查询不包括逻辑删除的记录
    protected String getListAllHql() {
        return super.getListAllHql() + " where deleted=false";
    }
    @Override //统计没有被逻辑删除的且发布的商品数量
    public int countAllPublished() {
        String hql = getCountAllHql() + " and published=true";
        Number result = unique(hql);
            return result.intValue();
    }
    @Override //查询没有被逻辑删除的且发布的商品
    public List<GoodsModel> listAllPublished(int pn) {
        String hql = getListAllHql() + " and published=true";
        return list(hql, pn, Constants.DEFAULT_PAGE_SIZE);
    }
}
```

java代码：

```
package cn.javass.point.dao.hibernate;
//省略import
public class GoodsCodeHibernateDao extends
BaseHibernateDao<GoodsCodeModel, Integer> implements IGoodsCodeDao {
@Override  //根据商品ID查询该商品的兑换码
    public List<GoodsCodeModel> listAllByGoods(int pn, int goodsId) {
        final String hql = getListAllHql() + " where goods.id = ?";
        return list(hql, pn, Constants.DEFAULT_PAGE_SIZE , goodsId);
    }
@Override //根据商品ID统计该商品的兑换码数量
    public int countAllByGoods(int goodsId) {
        final String hql = getCountAllHql() + " where goods.id = ?";
        Number result = unique(hql, goodsId);
```

```
        return result.intValue();
    }
```

3、Spring DAO层配置文件（resources/cn/javass/point/dao/ applicationContext-hibernate.xml）：

DAO配置文件中定义Hibernate的SessionFactory、事务管理器和DAO实现。

java代码：

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.annotation.AnnotationSessi
    <property name="dataSource" ref="dataSource"/><!-- 1、指定数据源 -->
    <property name="annotatedClasses">           <!-- 2、指定注解类 -->
        <list>
            <value>cn.javass.point.model.GoodsModel</value>
            <value>cn.javass.point.model.GoodsCodeModel</value>
        </list>
    </property>
    <property name="hibernateProperties"><!-- 3、指定Hibernate属性 -->
       <props>
           <prop key="hibernate.dialect">${hibernate.dialect}</prop>
           <prop key="hibernate.show_sql">${hibernate.show_sql}</prop>
           <prop key="hibernate.format_sql">${hibernate.format_sql}</prop>
           <prop key="hibernate.hbm2ddl.auto">${hibernate.hbm2ddl.auto}</prop>
        </props>
    </property>
</bean>
<bean id="txManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
      <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

java代码：

```
<bean id="abstractDao" abstract="true" init-method="init">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
<bean id="goodsDao" class="cn.javass.point.dao.hibernate.GoodsHibernateDao" parent="abstractDa
<bean id="goodsCodeDao" class="cn.javass.point.dao.hibernate.GoodsCodeHibernateDao" parent="ab
```

4、修改替换配置元数据的资源文件（resources/resources.properties），添加Hibernate属性相关：

java代码：

```
#Hibernate属性
hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect
hibernate.hbm2ddl.auto=none
hibernate.show_sql=false
hibernate.format_sql=true
```

11.3.4 实现业务逻辑层

业务逻辑层实现业务逻辑，即系统中最复杂、最核心的功能，不应该在业务逻辑层出现如数据库访问等底层代码，对于这些操作应委托给数据访问层实现，从而保证业务逻辑层的独立性和可复用性，并应该在业务逻辑层组装分页对象。

商品模块应该实现如下功能：

- CURD操作，直接委托给通用业务逻辑层；
- 根据页码查询所有已发布的商品的分页对象，即查询指定页的记录，这是和数据访问层不同的；

商品兑换码模块应该实现如下功能：

- CURD操作，直接委托给通用业务逻辑层；
- 根据页码和商品Id查询查询所有商品兑换码分页对象，即查询指定页的记录；
- 新增指定商品的兑换码，用于对指定商品添加兑换码；
- 购买指定商品兑换码操作，用户根据商品购买该商品的兑换码，如果指定商品的兑换码没有了将抛出没有兑换码异常NotCodeException；

1、商品及商品兑换码Service接口定义：

接口定义时，对于CRUD直接继承IBaseService即可，无需再在这些接口中定义重复的CRUD方法了，并通过泛型指定数据模型类及数据模型的主键。

java代码：

```
package cn.javass.point.service;
//省略import
public interface IGoodsService extends IBaseService<GoodsModel, Integer> {
    /**根据页码查询所有已发布的商品的分页对象*/
    Page<GoodsModel> listAllPublished(int pn);
}
```

java代码：

```
package cn.javass.point.service;
//省略import
public interface IGoodsCodeService extends IBaseService<GoodsCodeModel, Integer> {
    /** 根据页码和商品Id查询查询所有商品兑换码分页对象*/
    public Page<GoodsCodeModel> listAllByGoods(int pn, int goodsId);
    /** 新增指定商品的兑换码*/
    public void save(int goodsId, String[] codes);
    /** 购买指定商品兑换码 */
    GoodsCodeModel buy(String username, int goodsId) throws NotCodeException ;
}
```

2、NotCodeException异常定义，表示指定商品的兑换码已经全部被兑换了，没有剩余的兑换码了：

java代码：

```
package cn.javass.point.exception;
/** 购买失败异常,表示没有足够的兑换码 */
public class NotCodeException extends RuntimeException {
}
```

NotCodeException异常类实现RuntimeException，当需要更多信息时可以在异常中定义，异常比硬编码错误代码（如-1表示没有足够的兑换码）更好理解。

3、商品及商品兑换码Service接口实现定义：

接口实现时，CRUD实现直接从BaseServcice继承即可，无需再在这些专有实现中定义重复的CRUD实现了，并通过泛型指定数据模型类及数据模型的主键。

java代码：

```
package cn.javass.point.service.impl;
//省略import
public class GoodsServiceImpl extends BaseServiceImpl<GoodsModel, Integer> implements IGoodsSe
    @Override
    public Page<GoodsModel> listAllPublished(int pn) {
        int count = getGoodsDao().countAllPublished();
        List<GoodsModel> items = getGoodsDao().listAllPublished(pn);
        return PageUtil.getPage(count, pn, items, Constants.DEFAULT_PAGE_SIZE);
    }
    IGoodsDao getGoodsDao() {//将通用DAO转型
        return (IGoodsDao) getDao();
    }
}
```

java代码：

```
package cn.javass.point.service.impl;
//省略import
public class GoodsCodeServiceImpl extends BaseServiceImpl<GoodsCodeModel, Integer> implements I
    private IGoodsService goodsService;
    public void setGoodsService(IGoodsService goodsService) {//注入IGoodsService
        this.goodsService = goodsService;
    }
    private IGoodsCodeDao getGoodsCodeDao() {//将注入的通用DAO转型
        return (IGoodsCodeDao) getDao();
    }
    @Override
    public Page<GoodsCodeModel> listAllByGoods(int pn, int goodsId) {
        Integer count = getGoodsCodeDao().countAllByGoods(goodsId);
```

```
        List<GoodsCodeModel> items = getGoodsCodeDao().listAllByGoods(pn, goodsId);
        return PageUtil.getPage(count, pn, items, Constants.DEFAULT_PAGE_SIZE);
    }
    @Override
    public void save(int goodsId, String[] codes) {
        GoodsModel goods = goodsService.get(goodsId);
        for(String code : codes) {
            if(StringUtils.hasText(code)) {
                GoodsCodeModel goodsCode = new GoodsCodeModel();
                goodsCode.setCode(code);
                goodsCode.setGoods(goods);
            save(goodsCode);
            }
        }
    }
    @Override
    public GoodsCodeModel buy(String username, int goodsId)  throws NotCodeException {
        //1、实际实现时要验证用户积分是否充足
        //2、其他逻辑判断
        //3、实际实现时要记录交易记录开始
       GoodsCodeModel goodsCode = getGoodsCodeDao().getOneNotExchanged(goodsId);

       if(goodsCode == null) {
            //3、实际实现时要记录交易记录失败
            throw new NotCodeException();
                        //目前只抛出一个异常，还可能比如并发购买情况
       }
       goodsCode.setExchanged(true);
       goodsCode.setExchangeTime(new Date());
       goodsCode.setUsername(username);
       save(goodsCode);
       //3、实际实现时要记录交易记录成功
       return goodsCode;
    }
}
```

save方法和buy方法实现并不是最优的，save方法中如果兑换码有上千个怎么办？这时就需要批处理了，通过批处理比如20条一提交数据库来提高性能。buy方法就要考虑多个用户同时购买同一个兑换码如何处理？

交易历史一定要记录，从交易开始到交易结束（不管成功与否）一定要记录用于当客户投诉时查询相应数据。

4、Spring Service层配置文件（resources/cn/javass/point/service/ applicationContext-service.xml）：

Service层配置文件定义了事务和Service实现。

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="del*" propagation="REQUIRED" />
        <tx:method name="remove*" propagation="REQUIRED" />
        <tx:method name="buy*" propagation="REQUIRED" />
        <tx:method name="count*" propagation="SUPPORTS" read-only="true" />
        <tx:method name="find*" propagation="SUPPORTS" read-only="true" />
        <tx:method name="list*" propagation="SUPPORTS" read-only="true" />
        <tx:method name="*" propagation="SUPPORTS" read-only="true" />
    </tx:attributes>
</tx:advice>
```

java代码：

```
<aop:config>
    <aop:pointcut id="txPointcut" expression="execution(* cn.javass.point.service.*.*(..))" />
    <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut" />
</aop:config>
<bean id="goodsService" class="cn.javass.point.service.impl.GoodsServiceImpl">
    <property name="dao" ref="goodsDao"/>
</bean>
<bean id="goodsCodeService" class="cn.javass.point.service.impl.GoodsCodeServiceImpl">
    <property name="dao" ref="goodsCodeDao"/>
    <property name="goodsService" ref="goodsService"/>
</bean>
```

11.3.5 实现表现层

表现层显示页面展示和交互，应该支持多种视图技术（如JSP、Velocity），表现层实现不应该实现诸如业务逻辑层功能，只负责调用业务逻辑层查找数据模型并委托给相应的视图进行展示数据模型。

积分商城分为前台和后台，前台负责与客户进行交互，如购买商品；后台是负责商品及商品兑换码维护的，只应该管理员有权限操作。

后台模块：

- 商品管理模块：负责商品的维护，包括列表、新增、修改、删除、查询所有商品兑换码功能；
- 商品兑换码管理模块：包括列表、新增、删除所有兑换码操作；

前台模块：只有已发布商品展示，用户购买指定商品时，如果购买成功则给用户发送兑换码，购买失败给用户错误提示。

表现层Action实现时一般使用如下规约编程：

- **Action方法定义：**使用如list方法表示展示列表，doAdd方法表示去新增页面，add方法表示提交新增页面的结果并委托给Service层进行处理；
- **结果定义：**如使用“list”结果表示到展示列表页面，“add”结果去新增页面等等；
- **参数设置：**一般使用如“model”表示数据模型，使用“page”表示分页对象。

1、集成Struts2和Spring配置：

1.1、Spring Action配置文件：即Action将从Spring容器中获取，前台和后台配置文件应该分开以便好管理；

- 后台Action配置文件resources/cn/javass/web/pointShop-admin-servlet.xml；
- 前台Action配置文件resources/cn/javass/web/pointShop-front-servlet.xml；

1.2、Struts配置文件定义（resources/struts.xml）：

为了提高开发效率和采用规约编程，我们将使用模式匹配通配符来定义action。对于管理后台和前台应该分开，URL模式将类似于/{module}/{action}/{method}.action：

- module即模块名如admin，action即action前缀名，如后台的“GoodsAction”可以使用“goods”，method即Action中的方法名如“list”。
- 可以在Struts配置文件中使用{1}访问第一个通配符匹配的结果，以此类推；
- Reuslt也采用规约编程，即只有符合规律的放置jsp文件才会匹配到，如Result为“/WEB-INF/jsp/admin/{1}/list.jsp”，而URL为/goods/list.action 结果将为“/WEB-INF/jsp/admin/goods/list.jsp”。

java代码：

```
<package name="admin" extends="custom-default" namespace="/admin">
        <action name="*/*" class="/admin/{1}Action" method="{2}">
           <result name="redirect" type="redirect">/admin/{1}/list.action</result>
           <result name="list">/WEB-INF/jsp/admin/{1}/list.jsp</result>
           <result name="add">/WEB-INF/jsp/admin/{1}/add.jsp</result>
        </action>
    </package>
```

在此我们继承了“**custom-default**”包来支持action名字中允许“/”。

如“/admin/goods/list.action”将调用cn.javass.point.web.admin.action.GoodsAction的list方法。

java代码：

```
<package name="front" extends="custom-default">
        <action name="*/*" class="/front/{1}Action" method="{2}">
           <result name="redirect" type="redirect">/{1}/list.action</result>
           <result name="list">/WEB-INF/jsp/front/{1}/list.jsp</result>
           <result name="add">/WEB-INF/jsp/front/{1}/add.jsp</result>
           <result name="buyResult">/WEB-INF/jsp/front/{1}/buyResult.jsp</result>
        </action>
    </package>
```

如“/goods/list.action”将调用cn.javass.point.web.front.action.GoodsAction的list方法。

1.3、web.xml配置：将Spring配置文件加上；

java代码：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:applicationContext-resources.xml,
        classpath:cn/javass/point/dao/applicationContext-hibernate.xml,
        classpath:cn/javass/point/service/applicationContext-service.xml,
        classpath:cn/javass/point/web/pointShop-admin-servlet.xml,
        classpath:cn/javass/point/web/pointShop-front-servlet.xml
```

```
    </param-value>
</context-param>
```

2、后台商品管理模块

商品管理模块实现商品的CRUD，本示例只演示新增，删除和更新由于篇幅问题留作练习。

2.1、Action实现

java代码：

```
package cn.javass.point.web.admin.action;
//省略import
public class GoodsAction extends BaseAction {
    public String list() {//列表、展示所有商品（包括未发布的）
        getValueStack().set(PAGE, goodsService.listAll(getPn()));
        return LIST;
    }
    public String doAdd() {//到新增页面
        goods = new GoodsModel();
        getValueStack().set(MODEL, goods);
        return ADD;
    }
    public String add() {//保存新增模型对象
        goodsService.save(goods);
        return REDIRECT;
    }
    //字段驱动数据填充
```

```
    private int id = -1;   //前台提交的商品ID
    private GoodsModel goods; //前台提交的商品模型对象
    //省略字段驱动数据的getter和setter
    //依赖注入Service
    private IGoodsService goodsService;
    //省略依赖注入的getter和setter
}
```

2.2、Spring配置文件定义（resources/cn/javass/web/pointShop-admin-servlet.xml）：

java代码：

```
<bean name="/admin/goodsAction" class="cn.javass.point.web.admin.action.GoodsAction" scope="pr
    <property name="goodsService" ref="goodsService"/>
</bean>
```

2.3、JSP实现商品列表页面（WEB-INF/jsp/admin/goods/list.jsp）

查询所有商品，通过迭代“page.items”（Page对象的items属性中存放着分页列表数据）来显示商品列表，在最后应该有分页标签（请参考源代码，示例无），如类似于“<my:page url="${ctx}/admin/goods/list.action"/>”来定义分页元素。

java代码：

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
<%@ include file="../../common/inc/tld.jsp"%>
<jsp:include page="../../common/inc/header.jsp">
```

```
    <jsp:param name="title" value="商品管理-商品列表"/>
</jsp:include>
<a href="${ctx}/admin/goods/doAdd.action">新增</a><br/>
<table border="1">
    <tr>
        <th>ID</th>
        <th>商品名称</th>
        <th>商品描述</th>
        <th>原需积分</th>
        <th>现需积分</th>
        <th>是否已发布</th>
        <th></th>
        <th></th>
        <th></th>
</tr>
  <s:iterator value="page.items">
<tr>
        <td><a href="${ctx}/admin/goods/toUpdate.action?id=<s:property value='id'/>"><s:proper
        <td><s:property value="name"/></td>
        <td><s:property value="description"/></td>
        <td><s:property value="originalPoint"/></td>
        <td><s:property value="nowPoint"/></td>
        <td><s:property value="published"/></td>
        <td>更新</td> <td>删除</td>
        <td><a href="${ctx}/admin/goodsCode/list.action?goodsId=<s:property value='id'/>">查看C
    </tr>
    </s:iterator>
</table>
<jsp:include page="../../common/inc/footer.jsp"/>
```

右击“pointShop”项目选择【Run As】>【Run On Server】启动Tomcat服务器，在浏览器中输入“http://localhost:8080/pointShop/admin/goods/list.action”将显示图11-11界面。

图11-11 后台商品列表页面

2.4、JSP实现商品新增页面（WEB-INF/jsp/admin/goods/add.jsp）

表单提交到/admin/goods/add.action即cn.javass.point.web.admin.action.GoodsAction的add方法。并将参数绑定到goods属性上，在此我们没有进行数据验证，在实际项目中页面中和Action中都要进行数据验证。

java代码：

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html; charset=UTF-8" %>
<%@ include file="../../common/inc/tld.jsp"%>
<jsp:include page="../../common/inc/header.jsp">
    <jsp:param name="title" value="商品管理-新增"/>
</jsp:include>
<s:fielderror cssStyle="color:red"/>
<s:form action="/admin/goods/add.action" method="POST" acceptcharset="UTF-8" >
<s:token/>
<table border="1">
    <s:hidden name="goods.id" value="%{model.id}"/>
    <s:hidden name="goods.version" value="%{model.version}"/>
    <tr>
        <s:textfield label="商品名称" name="goods.name" value="%{model.name}" required="true"/>
    </tr>

    <tr>
        <s:textarea label="商品简介" name="goods.description" value="%{model.description}" requ
    </tr>
```

```
    <tr>
        <s:textfield label="原需积分" name="goods.originalPoint" value="%{model.originalPoint}"
    </tr>
    <tr>
        <s:textfield label="现需积分" name="goods.nowPoint" value="%{model.nowPoint}" required=
    </tr>
    <tr>
        <s:radio label="是否发布" name="goods.published" list="#{true:'发布',false:'不发布'}" va
    </tr>
    <tr>
       <td><input name="submit" type="submit" value="新增"/></td>
    <td>
        <input name="cancel" type="button" onclick="javascript:window.location.href='${ctx}/adr
    </td>
    </tr>
</table>
</s:form>
<jsp:include page="../../common/inc/footer.jsp"/>
```

右击“pointShop”选择【Run As】>【Run On Server】启动Tomcat服务器，在商品列表页面单间【新增】按钮将显示图11-11界面。

图11-12 后台商品新增页面

3、后台兑换码管理

提供根据商品ID查询兑换码列表及新增兑换码操作，兑换码通过文本框输入多个，使用换行分割。

3.1、Action实现

java代码：

```
package cn.javass.point.web.admin.action;
//省略import
public class GoodsCodeAction extends BaseAction {
    public String list() {
        getValueStack().set(MODEL, goodsService.get(goodsId));
        getValueStack().set(PAGE,
       goodsCodeService.listAllByGoods(getPn(), goodsId));
        return LIST;
    }
    public String doAdd() {
        getValueStack().set(MODEL, goodsService.get(goodsId));
        return ADD;
    }
    public String add() {
        String[] codes = splitCodes();
        goodsCodeService.save(goodsId, codes);
        return list();
    }
    private String[] splitCodes() {//将根据换行分割code码
        if(codes == null) {
```

```
            return new String[0];
        }
        return codes.split("\r"); //简单起见不考虑“\n”
    }
    //字段驱动数据填充
    private int id = -1;      //前台提交的商品兑换码ID
    private int goodsId = -1; //前台提交的商品ID
    private String codes;//前台提交的兑换码，由换行分割
    private GoodsCodeModel goodsCode; //前台提交的商品兑换码模型对象
    //省略字段驱动数据的getter和setter
    //依赖注入Service
    private IGoodsCodeService goodsCodeService;
    private IGoodsService goodsService;
    //省略依赖注入的getter和setter
}
```

3.2、Spring配置文件定义（resources/cn/javass/web/pointShop-admin-servlet.xml）：

java代码：

```
<bean name="/admin/goodsCodeAction"
class="cn.javass.point.web.admin.action.GoodsCodeAction" scope="prototype">
<property name="goodsService" ref="goodsService"/>
    <property name="goodsCodeService" ref="goodsCodeService"/>
</bean>
```

3.3、JSP实现商品兑换码列表页面（WEB-INF/jsp/admin/goodsCode/list.jsp）

商品兑换码列表页面时将展示相应商品的兑换码。

java代码：

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
<%@ include file="../../common/inc/tld.jsp"%>
<jsp:include page="../../common/inc/header.jsp">
    <jsp:param name="title" value="商品管理-商品Code码列表"/>
</jsp:include>
<a href="${ctx}/admin/goodsCode/doAdd.action?goodsId=${model.id}">新增</a>|
<a href="${ctx}/admin/goods/list.action">返回商品列表</a><br/>
<table border="1">
    <tr>
        <th>ID</th>
        <th>所属商品</th>
        <th>兑换码</th>
        <th>购买人</th>
        <th>兑换时间</th>
        <th>是否已经兑换</th>
        <th></th>
    </tr>
    <s:iterator value="page.items">
    <tr>
        <td><s:property value="id"/></td>
        <td><s:property value="goods.name"/></td>
        <td><s:property value="code"/></td>
        <td><s:property value="username"/></td>
        <td><s:date name="exchangeTime" format="yyyy-MM-dd"/></td>
        <td><s:property value="exchanged"/></td>
        <td>删除</td>
    </tr>
    </s:iterator>
</table>
<jsp:include page="../../common/inc/footer.jsp"/>
```

右击“pointShop”选择【Run As】>【Run On Server】启动Web服务器，在浏览器中输入“http://localhost:8080/pointShop/admin/goods/list.action”，然后在指定商品后边点击【查看兑换码】将显示图11-15界面。

新增 | 返回商品列表

ID	所属商品	兑换码	购买人	兑换时间	是否已经兑换	
1	研磨设计模式	12234443232			false	删除
2	研磨设计模式	342342342342			false	删除
3	研磨设计模式	3423424234234			false	删除

图11-15 商品兑换码列表

3.4、JSP实现商品兑换码新增页面（WEB-INF/jsp/admin/goodsCode/add.jsp）

用于新增指定商品的兑换码。

java代码：

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
<%@ include file="../../common/inc/tld.jsp"%>
<jsp:include page="../../common/inc/header.jsp">
    <jsp:param name="title" value="用户管理-新增"/>
</jsp:include>
<s:fielderror cssStyle="color:red"/>
<s:form action="/admin/goodsCode/add.action" method="POST" acceptcharset="UTF-8">
<s:token/>
<s:hidden name="goodsId" value="%{model.id}" />
<table border="1">
```

```
    <tr>
        <s:textfield label="所属商品" name="model.name" readonly="true"/>
    </tr>
    <tr>
        <s:textarea label="code码" name="codes" cols="20" rows="3"/>
    </tr>
    <tr>
       <td><input name="submit" type="submit" value="新增"/></td>
       <td><input name="cancel" type="button" onclick="javascript:window.location.href='${ctx}/
    </tr>
</table>
</s:form>
<jsp:include page="../../common/inc/footer.jsp"/>
```

右击“pointShop”选择【Run As】>【Run On Server】启动Tomcat服务器，在商品兑换码列表中单击【新增】按钮将显示图11-16界面。

图11-16 兑换码新增页面

4、前台商品展示及购买模块：

前台商品展示提供商品展示及购买页面，购买时应考虑是否有足够兑换码等，此处错误消息使用硬编码，应该考虑使用国际化支持，请参考学习国际化。

4.1、Action实现

java代码：

```
package cn.javass.point.web.front.action;
//省略import
public class GoodsAction extends BaseAction {
    private static final String BUY_RESULT = "buyResult";
    public String list() {
        getValueStack().set(PAGE, goodsService.listAllPublished(getPn()));
        return LIST;
    }
    public String buy() {
        String username = "test";
        GoodsCodeModel goodsCode = null;
        try {
            goodsCode = goodsCodeService.buy(username, goodsId);
        } catch (NotCodeException e) {
            this.addActionError("没有足够的兑换码了");
            return BUY_RESULT;
        } catch (Exception e) {
            e.printStackTrace();
            this.addActionError("未知错误");
            return BUY_RESULT;
        }
        this.addActionMessage("购买成功，您的兑换码为 :"+ goodsCode.getCode());
        getValueStack().set(MODEL, goodsCode);
        return BUY_RESULT;
    }
    //字段驱动数据填充
    private int goodsId;
```

```
    //省略字段驱动数据的getter和setter
    //依赖注入Service
    IGoodsService goodsService;
    IGoodsCodeService goodsCodeService;
    //省略依赖注入的getter和setter
}
```

4.2、**Spring配置文件定义（resources/cn/javass/web/pointShop-front-servlet.xml）：**

java代码：

```
<bean name="/front/goodsAction" class="cn.javass.point.web.front.action.GoodsAction" scope="pr
    <property name="goodsService" ref="goodsService"/>
    <property name="goodsCodeService" ref="goodsCodeService"/>
</bean>
```

4.3、JSP实现前台商品展示及购买页面（WEB-INF/jsp/ goods/list.jsp）

java代码：

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
<%@ include file="../../common/inc/tld.jsp"%>
<jsp:include page="../../common/inc/header.jsp">
    <jsp:param name="title" value="积分商城-商品列表"/>
</jsp:include>
<s:iterator value="page.items" status="status">
       <s:property value="#status.index + 1"/>.<s:property value="name"/>
```

```
        <a href="${ctx}/goods/buy.action?goodsId=<s:property value='id'/>">【购买】</a><br/>
       描述：<s:property value="description"/><br/>
        <s>需要积分<s:property value="originalPoint"/></s>  现需积分：<b><s:property va
</s:iterator>
<jsp:include page="../../common/inc/footer.jsp"/>
```

右击“pointShop”选择【Run As】>【Run On Server】启动Web服务器，在浏览器中输入**http://localhost:8080/pointShop/goods/list.action**将显示图11-17界面。

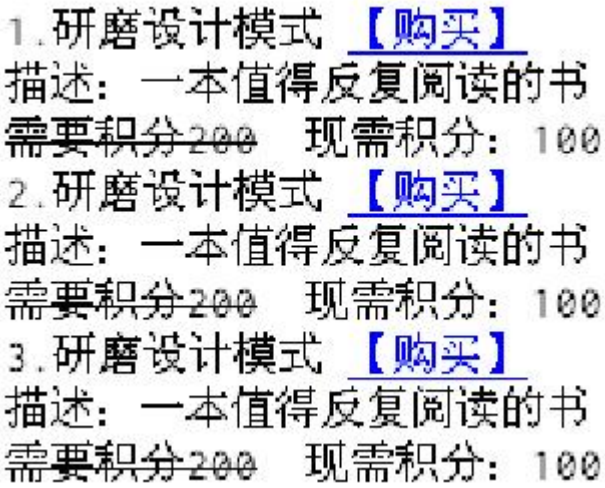


图11-17 前台商品展示即购买页面

在前台商品展示即购买页面中点击购买，如果库存中还有兑换码，将购买成功，否则购买失败。

4.3、商品购买结果页面（WEB-INF/jsp/admin/goods/buyResult.jsp）

购买成功将通过“<s:actionmessage/>”标签显示成功信息并将兑换码显示给用户，购买失败将通过“<s:actionerror/>”标签提示如积分不足或兑换码没有了等错误信息。

java代码：

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"%>
<%@ include file="../../common/inc/tld.jsp"%>
<jsp:include page="../../common/inc/header.jsp">
    <jsp:param name="title" value="积分商城-购买结果"/>
</jsp:include>
<s:actionerror/>
<s:actionmessage/>
<jsp:include page="../../common/inc/footer.jsp"/>
```

在商品展示及购买列表购买成功或失败将显示图11-18或图11-19界面。

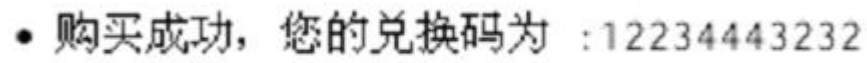


图11-18 购买成功页面

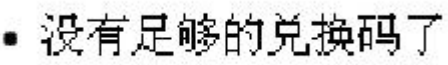


图11-19 购买失败页面

到此SSH集成已经结束，集成SSH是非常简单的，但开发流程及开发思想是关键。

我们整个开发过程是首先抽象和提取通用的模块和代码，这样可以复用减少开发时间，其次是基于通用层开发不可预测部分（即可变部分），因为每个项目的功能是不一样的。在开发过程中还集中将重复内容提取到一处这样方便以后修改。

1.17 【第十二章】零配置 之 12.1 概述 ——跟我学spring3

发表时间: 2012-03-19 关键字: spring, 零配置

12.1 概述

12.1.1 什么是零配置

在SSH集成一章中大家注意到项目结构和包结构是不是很有规律，类库放到WEB-INF/lib文件夹下，jsp文件放到WEB-INF/jsp文件夹下，web.xml需要放到WEB-INF文件夹下等等，为什么要这么放呢？不这样放可以吗？

所谓零配置，并不是说一点配置都没有了，而是配置很少而已。通过约定来减少需要配置的数量，提高开发效率。

因此SSH集成时的项目结构和包结构完全是任意的，可以通过配置方式来指定位置，因此如web.xml完全可以不放在WEB-INF下边而通过如tomcat配置文件中新指定web.xml位置。

还有在SSH集成中还记得使用在Struts2配置文件中使用模式匹配通配符来定义action，只要我们的URL模式将类似于/{module}/{action}/{method}.action即可自动映射到相应的Action类的方法上，但如果你的URL不对肯定是映射不到的，这就是规约。

零配置并不是没有配置，而是通过约定来减少配置。那如何实现零配置呢？

12.1.2 零配置的实现方式

零配置实现主要有以下两种方式：

- **惯例优先原则：**也称为约定大于配置或规约大于配置（convention over configuration），即通过约定代码结构或命名规范来减少配置数量，同样不会减少配置文件；即通过约定好默认规范来提高开发效率；如Struts2配置文件使用模式匹配通配符来定义action就是惯例优先原则。
- **基于注解的规约配置：**通过在指定类上指定注解，通过注解约定其含义来减少配置数量，从而提高开发效率；如事务注解@Transaction是不是基于注解的规约，只有在指定的类或方法上使用该注解就表示其需要事务。

对惯例优先原则支持的有项目管理工具Maven，它约定了一套非常好的项目结构和一套合理的默认值来简化日常开发，作者比较喜欢使用Maven构建和管理项目；另外还有Strtus2的convention-plugin也提供了零配置支持等等。

大家还记得【7.5 集成Spring JDBC及最佳实践】时的80/20法则吗？零配置是不是同样很好的体现了这个法则，在日常开发中同样80%时间使用默认配置，而20%时间可能需要特定配置。

12.1.3 Spring3的零配置

Spring3中零配置的支持主要体现在Spring Web MVC框架的惯例优先原则和基于注解配置。

Spring Web MVC框架的惯例优先原则采用默认的命名规范来减少配置。

Spring基于注解的配置采用约定注解含义来减少配置，包括注解实现Bean配置、注解实现Bean定义和Java类替换配置文件三部分：

- **注解实现Bean依赖注入：**通过注解方式替代基于XML配置中的依赖注入，如使用@Autowired注解来完成依赖注入。
- **注解实现Bean定义：**通过注解方式进行Bean配置元数据定义，从而完全将Bean配置元数据从配置文件中移除。
- **Java类替换配置文件：**使用Java类来定义所有的Spring配置，完全消除XML配置文件。

1.18 【第十二章】零配置 之 12.2 注解实现Bean依赖注入 ——跟我学spring3

发表时间: 2012-03-19 关键字: spring, 零配置

12.2 注解实现Bean依赖注入

12.2.1 概述

注解实现Bean配置主要用来进行如依赖注入、生命周期回调方法定义等，不能消除XML文件中的Bean元数据定义，**且基于XML配置中的依赖注入的数据将覆盖基于注解配置中的依赖注入的数据。**

Spring3的基于注解实现Bean依赖注入支持如下三种注解：

- **Spring自带依赖注入注解：** Spring自带的一套依赖注入注解；
- **JSR-250注解：**Java平台的公共注解，是Java EE 5规范之一，在JDK6中默认包含这些注解，从Spring2.5开始支持。
- **JSR-330注解**：Java 依赖注入标准，Java EE 6规范之一，可能在加入到未来JDK版本，从Spring3开始支持；
- **JPA注解：**用于注入持久化上下文和尸体管理器。

这三种类型的注解在Spring3中都支持，类似于注解事务支持，想要使用这些注解需要在Spring容器中开启注解驱动支持，即使用如下配置方式开启：

java代码：

```
<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-3.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <context:annotation-config/>
```

```
</beans>
```

这样就能使用注解驱动依赖注入了，该配置文件位于“resources/ chapter12/ dependecyInjectWithAnnotation.xml”。

12.2.2 Spring自带依赖注入注解

一、@Required：依赖检查；

对应于基于XML配置中的依赖检查，但XML配置的依赖检查将检查所有setter方法，详见【3.3.4 依赖检查】；

基于@Required的依赖检查表示注解的setter方法必须，即必须通过在XML配置中配置setter注入，如果没有配置在容器启动时会抛出异常从而保证在运行时不会遇到空指针异常，@Required只能放置在setter方法上，且通过XML配置的setter注入，可以使用如下方式来指定：

java代码：

```
@Requried
setter方法
```

1、准备测试Bean

java代码：

```
package cn.javass.spring.chapter12;
public class TestBean {
    private String message;
    @Required
```

```
    public void setMessage(String message) {
        this.message = message;
    }
    public String getMessage() {
        return message;
    }
}
```

2、在Spring配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下Bean配置：

java代码：

```
<bean id="testBean" class="cn.javass.spring.chapter12.TestBean">
<property name="message" ref="message"/>
</bean>
<bean id="message" class="java.lang.String">
    <constructor-arg index="0" value="hello"/>
</bean>
```

3、测试类和测试方法如下：

java代码：

```
package cn.javass.spring.chapter12;
//省略import
public class DependencyInjectWithAnnotationTest {
    private static String configLocation = "classpath:chapter12/dependecyInjectWithAnnotation.
    private static ApplicationContext ctx = new ClassPathXmlApplicationContext(configLocation)
```

```
    //1、Spring自带依赖注入注解
     @Test
    public void testRequiredForXmlSetterInject() {
        TestBean testBean = ctx.getBean("testBean", TestBean.class);
        Assert.assertEquals("hello", testBean.getMessage());
    }
}
```

在XML配置文件中必须指定setter注入，否则在Spring容器启动时将抛出如下异常：

java代码：

```
org.springframework.beans.factory.BeanCreationException:
Error creating bean with name 'testBean' defined in class path resource [chapter12/dependecyIn
nested exception is org.springframework.beans.factory.BeanInitializationException: Property 'me
```

二、@Autowired：自动装配

自动装配，用于替代基于XML配置的自动装配，详见【3.3.3 自动装配】。

基于@Autowired的自动装配，默认是根据类型注入，可以用于构造器、字段、方法注入，使用方式如下：

java代码：

```
@Autowired(required=true)
构造器、字段、方法
```

@Autowired默认是根据参数类型进行自动装配，且必须有一个Bean候选者注入，如果允许出现0个Bean候选者需要设置属性“required=false”，“required”属性含义和@Required一样，只是@Required只适用于基于XML配置的setter注入方式。

（1）、构造器注入：通过将@Autowired注解放在构造器上来完成构造器注入，默认构造器参数通过类型自动装配，如下所示：

1、准备测试Bean，在构造器上添加@AutoWired注解：

java代码：

```
package cn.javass.spring.chapter12;
import org.springframework.beans.factory.annotation.Autowired;
public class TestBean11 {
    private String message;
    @Autowired //构造器注入
    private TestBean11(String message) {
        this.message = message;
    }
    //省略message的getter和setter
}
```

2、在Spring配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下Bean配置：

java代码：

```
<bean id="testBean11" class="cn.javass.spring.chapter12.TestBean11"/>
```

3、测试类如下：

java代码：

```
@Test
public void testAutowiredForConstructor() {
    TestBean11 testBean11 = ctx.getBean("testBean11", TestBean11.class);
    Assert.assertEquals("hello", testBean11.getMessage());
}
```

在Spring配置文件中没有对“testBean11”进行构造器注入和setter注入配置，而是通过在构造器上添加@Autowired来完成根据参数类型完成构造器注入。

（2）、字段注入：通过将@Autowired注解放在构造器上来完成字段注入。

1、准备测试Bean，在字段上添加@AutoWired注解：

java代码：

```
package cn.javass.spring.chapter12;
import org.springframework.beans.factory.annotation.Autowired;
public class TestBean12 {
    @Autowired //字段注入
```

```
    private String message;
    //省略getter和setter
}
```

2、在Spring配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下Bean配置：

java代码：

```
<bean id="testBean12" class="cn.javass.spring.chapter12.TestBean12"/>
```

3、测试方法如下：

java代码：

```
@Test
public void testAutowiredForField() {
    TestBean12 testBean12 = ctx.getBean("testBean12", TestBean12.class);
    Assert.assertEquals("hello", testBean12.getMessage());
}
```

字段注入在基于XML配置中无相应概念，字段注入不支持静态类型字段的注入。

（3）、方法参数注入：通过将@Autowired注解放在方法上来完成方法参数注入。

1、准备测试Bean，在方法上添加@AutoWired注解：

java代码：

```
package cn.javass.spring.chapter12;
import org.springframework.beans.factory.annotation.Autowired;
public class TestBean13 {
    private String message;
    @Autowired //setter方法注入
    public void setMessage(String message) {
        this.message = message;
    }
    public String getMessage() {
        return message;
    }
}
```

java代码：

```
package cn.javass.spring.chapter12;
//省略import
public class TestBean14 {
    private String message;
    private List<String> list;
    @Autowired(required = true) //任意一个或多个参数方法注入
    private void initMessage(String message, ArrayList<String> list) {
        this.message = message;
        this.list = list;
    }
    //省略getter和setter
}
```

2、在Spring配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下Bean配置：

java代码：

```
<bean id="testBean13" class="cn.javass.spring.chapter12.TestBean13"/>
<bean id="testBean14" class="cn.javass.spring.chapter12.TestBean14"/>
<bean id="list" class="java.util.ArrayList">
    <constructor-arg index="0">
        <list>
            <ref bean="message"/>
            <ref bean="message"/>
        </list>
   </constructor-arg>
</bean>
```

3、测试方法如下：

java代码：

```
@Test
public void testAutowiredForMethod() {
    TestBean13 testBean13 = ctx.getBean("testBean13", TestBean13.class);
    Assert.assertEquals("hello", testBean13.getMessage());

    TestBean14 testBean14 = ctx.getBean("testBean14", TestBean14.class);
    Assert.assertEquals("hello", testBean14.getMessage());
    Assert.assertEquals(ctx.getBean("list", List.class), testBean14.getList());
}
```

方法参数注入除了支持setter方法注入，还支持1个或多个参数的普通方法注入，在基于XML配置中不支持1个或多个参数的普通方法注入，方法注入不支持静态类型方法的注入。

注意“**initMessage(String message, ArrayList<String> list)**”方法签名中为什么使用ArrayList而不是List呢？具体参考【3.3.3 自动装配】一节中的集合类型注入区别。

三、@Value：注入SpEL表达式；

用于注入SpEL表达式，可以放置在字段方法或参数上，使用方式如下：

java代码：

```
@Value(value = "SpEL表达式")
字段、方法、参数
```

1、可以在类字段上使用该注解：

java代码：

```
@Value(value = "#{message}")
private String message;
```

2、可以放置在带@Autowired注解的方法的参数上：

java代码：

```
@Autowired
public void initMessage(@Value(value = "#{message}#{message}") String message) {
    this.message = message;
}
```

3、还可以放置在带@Autowired注解的构造器的参数上：

java代码：

```
@Autowired
private TestBean43(@Value(value = "#{message}#{message}") String message) {
    this.message = message;
}
```

具体测试详见DependencyInjectWithAnnotationTest 类的testValueInject测试方法。

四、@Qualifier：限定描述符，用于细粒度选择候选者；

@Autowired默认是根据类型进行注入的，因此如果有多个类型一样的Bean候选者，则需要限定其中一个候选者，否则将抛出异常，详见【3.3.3 自动装配】中的根据类型进行注入；

@Qualifier限定描述符除了能根据名字进行注入，但能进行更细粒度的控制如何选择候选者，具体使用方式如下：

java代码：

```
@Qualifier(value = "限定标识符")
字段、方法、参数
```

（1）、根据基于XML配置中的<qualifier>标签指定的名字进行注入，使用如下方式指定名称：

java代码：

```
<qualifier  type="org.springframework.beans.factory.annotation.Qualifier"  value="限定标识符"/>
```

其中type属性可选，指定类型，默认就是Qualifier注解类，name就是给Bean候选者指定限定标识符，一个Bean定义中只允许指定类型不同的<qualifier>，如果有多个相同type后面指定的将覆盖前面的。

1、准备测试Bean：

java代码：

```
package cn.javass.spring.chapter12;
import javax.sql.DataSource;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;

public class TestBean31 {
```

```
    private DataSource dataSource;
    @Autowired
    //根据<qualifier>标签指定Bean限定标识符
    public void initDataSource(@Qualifier("mysqlDataSource") DataSource dataSource) {
        this.dataSource = dataSource;
    }
    public DataSource getDataSource() {
        return dataSource;
    }
}
```

2、在Spring配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下Bean配置：

java代码：

```
<bean id="testBean31" class="cn.javass.spring.chapter12.TestBean31"/>
```

我们使用@Qualifier("mysqlDataSource")来指定候选Bean的限定标识符，我们需要在配置文件中使用<qualifier>标签来指定候选Bean的限定标识符“mysqlDataSource”：

java代码：

```
<bean id="mysqlDataSourceBean" class="org.springframework.jdbc.datasource.DriverManagerDataSou
     <qualifier value="mysqlDataSource"/>
</bean>
```

3、测试方法如下：

java代码：

```
@Test
public void testQualifierInject1() {
    TestBean31 testBean31 = ctx.getBean("testBean31", TestBean31.class);
    try {
        //使用<qualifier>指定的标识符只能被@Qualifier使用
        ctx.getBean("mysqlDataSource");
        Assert.fail();
    } catch (Exception e) {
        //找不到该Bean
        Assert.assertTrue(e instanceof NoSuchBeanDefinitionException);
    }
     Assert.assertEquals(ctx.getBean("mysqlDataSourceBean"), testBean31.getDataSource());
}
```

从测试可以看出使用<qualifier>标签指定的限定标识符只能被@Qualifier使用，不能作为Bean的标识符，如“ctx.getBean("mysqlDataSource")”是获取不到Bean的。

（2）、缺省的根据Bean名字注入：最基本方式，是在Bean上没有指定<qualifier>标签时一种容错机制，即缺省情况下使用Bean标识符注入，但如果你指定了<qualifier>标签将不会发生容错。

1、准备测试Bean：

java代码：

```
package cn.javass.spring.chapter12;
//省略import
public class TestBean32 {
    private DataSource dataSource;
    @Autowired
    @Qualifier(value = "mysqlDataSource2") //指定Bean限定标识符
```

```
    //@Qualifier(value = "mysqlDataSourceBean")
    //是错误的注入，不会发生回退容错，因为你指定了<qualifier>
    public void initDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
    public DataSource getDataSource() {
        return dataSource;
    }
}
```

2、在Spring配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下Bean配置：

java代码：

```
<bean id="testBean32" class="cn.javass.spring.chapter12.TestBean32"/>
<bean id="oracleDataSource"  class="org.springframework.jdbc.datasource.DriverManagerDataSourc
```

3、测试方法如下：

java代码：

```
@Test
public void testQualifierInject2() {
    TestBean32 testBean32 = ctx.getBean("testBean32", TestBean32.class);
    Assert.assertEquals(ctx.getBean("oracleDataSource"), testBean32.getDataSource());
}
```

默认情况下（没指定<qualifier>标签）@Qualifier的value属性将匹配Bean 标识符。

（3）、扩展@Qualifier限定描述符注解：对@Qualifier的扩展来提供细粒度选择候选者；

具体使用方式就是自定义一个注解并使用@Qualifier注解其即可使用。

首先让我们考虑这样一个问题，如果我们有两个数据源，分别为Mysql和Oracle，因此注入两者相关资源时就牵扯到数据库相关，如在DAO层注入SessionFactory时，当然可以采用前边介绍的方式，但为了简单和直观我们希望采用自定义注解方式。

1、扩展@Qualifier限定描述符注解来分别表示Mysql和Oracle数据源

java代码：

```
package cn.javass.spring.chapter12.qualifier;
import org.springframework.beans.factory.annotation.Qualifier;
/** 表示注入Mysql相关 */
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Mysql {
}
```

java代码：

```
package cn.javass.spring.chapter12.qualifier;
import org.springframework.beans.factory.annotation.Qualifier;
/** 表示注入Oracle相关 */
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Oracle {
}
```

2、准备测试Bean：

java代码：

```
package cn.javass.spring.chapter12;
//省略import
public class TestBean33 {
    private DataSource mysqlDataSource;
    private DataSource oracleDataSource;
    @Autowired
    public void initDataSource(@Mysql DataSource mysqlDataSource, @Oracle DataSource oracleDat
        this.mysqlDataSource = mysqlDataSource;
        this.oracleDataSource = oracleDataSource;
    }
    public DataSource getMysqlDataSource() {
        return mysqlDataSource;
    }
    public DataSource getOracleDataSource() {
        return oracleDataSource;
    }
}
```

3、在Spring配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下Bean配置：

java代码：

```
<bean id="testBean33" class="cn.javass.spring.chapter12.TestBean33"/>
```

4、在Spring修改定义的两个数据源：

java代码：

```
<bean id="mysqlDataSourceBean" class="org.springframework.jdbc.datasource.DriverManagerDataSou
     <qualifier value="mysqlDataSource"/>
     <qualifier type="cn.javass.spring.chapter12.qualifier.Mysql"/>
</bean>
<bean id="oracleDataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource
      <qualifier type="cn.javass.spring.chapter12.qualifier.Oracle"/>
</bean>
```

5、测试方法如下：

java代码：

```
@Test
public void testQualifierInject3() {
    TestBean33 testBean33 = ctx.getBean("testBean33", TestBean33.class);
    Assert.assertEquals(ctx.getBean("mysqlDataSourceBean"), testBean33.getMysqlDataSoruce());
    Assert.assertEquals(ctx.getBean("oracleDataSource"), testBean33.getOracleDataSoruce());
}
```

测试也通过了，说明我们扩展的@Qualifier限定描述符注解也能很好工作。

前边演示了不带属性的注解，接下来演示一下带参数的注解：

1、首先定义数据库类型：

java代码：

```
package cn.javass.spring.chapter12.qualifier;
public enum DataBase {
    ORACLE, MYSQL;
}
```

2、其次扩展@Qualifier限定描述符注解

java代码：

```
package cn.javass.spring.chapter12.qualifier;
//省略import
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface DataSourceType {
    String ip();      //指定ip,用于多数据源情况
    DataBase database();//指定数据库类型
}
```

3、准备测试Bean：

java代码：

```
package cn.javass.spring.chapter12;
import javax.sql.DataSource;
import org.springframework.beans.factory.annotation.Autowired;
import cn.javass.spring.chapter12.qualifier.DataBase;
import cn.javass.spring.chapter12.qualifier.DataSourceType;
public class TestBean34 {
    private DataSource mysqlDataSource;
    private DataSource oracleDataSource;
    @Autowired
    public void initDataSource(
            @DataSourceType(ip="localhost", database=DataBase.MYSQL)
            DataSource mysqlDataSource,
            @DataSourceType(ip="localhost", database=DataBase.ORACLE)
            DataSource oracleDataSource) {
        this.mysqlDataSource = mysqlDataSource;
        this.oracleDataSource = oracleDataSource;
    }
      //省略getter方法
}
```

4、在Spring配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下Bean配置：

java代码：

```
<bean id="testBean34" class="cn.javass.spring.chapter12.TestBean34"/>
```

5、在Spring修改定义的两个数据源：

java代码：

```
<bean id="mysqlDataSourceBean" class="org.springframework.jdbc.datasource.DriverManagerDataSou
    <qualifier value="mysqlDataSource"/>
    <qualifier type="cn.javass.spring.chapter12.qualifier.Mysql"/>
    <qualifier type="cn.javass.spring.chapter12.qualifier.DataSourceType">
        <attribute key="ip" value="localhost"/>
        <attribute key="database" value="MYSQL"/>
    </qualifier>
</bean>
<bean id="oracleDataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource
    <qualifier type="cn.javass.spring.chapter12.qualifier.Oracle"/>
    <qualifier type="cn.javass.spring.chapter12.qualifier.DataSourceType">
        <attribute key="ip" value="localhost"/>
        <attribute key="database" value="ORACLE"/>
    </qualifier>
</bean>
```

6、测试方法如下：

java代码：

```
@Test
public void testQualifierInject3() {
    TestBean34 testBean34 = ctx.getBean("testBean34", TestBean34.class);
    Assert.assertEquals(ctx.getBean("mysqlDataSourceBean"), testBean34.getMysqlDataSource());
    Assert.assertEquals(ctx.getBean("oracleDataSource"), testBean34.getOracleDataSoruce());
}
```

测试也通过了，说明我们扩展的@Qualifier限定描述符注解也能很好工作。

四、自定义注解限定描述符：完全不使用@Qualifier，而是自己定义一个独立的限定注解；

1、首先使用如下方式定义一个自定义注解限定描述符：

java代码：

```
package cn.javass.spring.chapter12.qualifier;
//省略import
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface CustomQualifier {
    String value();
}
```

2、准备测试Bean：

java代码：

```
package cn.javass.spring.chapter12;
//省略import
public class TestBean35 {
    private DataSource dataSoruce;
    @Autowired
    public TestBean35(@CustomQualifier("oracleDataSource") DataSource dataSource) {
        this.dataSoruce = dataSource;
```

```
    }
    public DataSource getDataSoruce() {
        return dataSoruce;
    }
}
```

3、在Spring配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下Bean配置：

java代码：

```
<bean id="testBean35" class="cn.javass.spring.chapter12.TestBean35"/>
```

4、然后在Spring配置文件中注册CustomQualifier自定义注解限定描述符，只有注册了Spring才能识别：

java代码：

```
<bean id="customAutowireConfigurer" class="org.springframework.beans.factory.annotation.Custom
    <property name="customQualifierTypes">
        <set>
            <value>cn.javass.spring.chapter12.qualifier.CustomQualifier</value>
        </set>
   </property>
</bean>
```

5、测试方法如下：

java代码：

```
@Test
public void testQualifierInject5() {
    TestBean35 testBean35 = ctx.getBean("testBean35", TestBean35.class);
    Assert.assertEquals(ctx.getBean("oracleDataSource"), testBean35.getDataSource());
}
```

从测试中可看出，自定义的和Spring自带的没什么区别，因此如果没有足够的理由请使用Spring自带的Qualifier注解。

到此限定描述符介绍完毕，在此一定要注意以下几点：

- 限定标识符和Bean的描述符是不一样的；
- 多个Bean定义中可以使用相同的限定标识符；
- 对于集合、数组、字典类型的限定描述符注入，将注入多个具有相同限定标识符的Bean。

12.2.3 JSR-250注解

一、@Resource：自动装配，默认根据类型装配，如果指定name属性将根据名字装配，可以使用如下方式来指定：

java代码：

```
@Resource(name = "标识符")
字段或setter方法
```

1、准备测试Bean：

java代码：

```
package cn.javass.spring.chapter12;
import javax.annotation.Resource;
public class TestBean41 {
    @Resource(name = "message")
    private String message;
    //省略getter和setter
}
```

2、在Spring配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下Bean配置：

java代码：

```
<bean id="testBean41" class="cn.javass.spring.chapter12.TestBean41"/>
```

3、测试方法如下：

java代码：

```
@Test
public void testResourceInject1() {
    TestBean41 testBean41 = ctx.getBean("testBean41", TestBean41.class);
    Assert.assertEquals("hello", testBean41.getMessage());
}
```

使用非常简单，和@Autowired不同的是可以指定name来根据名字注入。

使用@Resource需要注意以下几点：

- @Resource注解应该只用于setter方法注入，不能提供如@Autowired多参数方法注入；
- @Resource在没有指定name属性的情况下首先将根据setter方法对于的字段名查找资源，如果找不到再根据类型查找；
- @Resource首先将从JNDI环境中查找资源，如果没找到默认再到Spring容器中查找，因此如果JNDI环境中有和Spring容器同名的资源时需要注意。

二、@PostConstruct和PreDestroy：通过注解指定初始化和销毁方法定义；

1、在测试类TestBean41中添加如下代码：

java代码：

```
@PostConstruct
public void init() {
    System.out.println("==========init");
}
@PreDestroy
public void destroy() {
    System.out.println("==========destroy");
}
```

2、修改测试方法如下：

java代码：

```
@Test
public void resourceInjectTest1() {
    ((ClassPathXmlApplicationContext) ctx).registerShutdownHook();
```

```
    TestBean41 testBean41 = ctx.getBean("testBean41", TestBean41.class);
    Assert.assertEquals("hello", testBean41.getMessage());
}
```

类似于通过<bean>标签的init-method和destroy-method属性指定的初始化和销毁方法，但具有更高优先级，即注解方式的初始化和销毁方法将先执行。

12.2.4 JSR-330注解

在测试之前需要准备JSR-330注解所需要的jar包，到spring-framework-3.0.5.RELEASE-dependencies.zip中拷贝如下jar包到类路径：

com.springsource.javax.inject-1.0.0.jar

一、@Inject：等价于默认的@Autowired，只是没有required属性；

二、@Named：指定Bean名字，对应于Spring自带@Qualifier中的缺省的根据Bean名字注入情况；

三、@Qualifier：只对应于Spring自带@Qualifier中的扩展@Qualifier限定描述符注解，即只能扩展使用，没有value属性。

1、首先扩展@Qualifier限定描述符注解来表示Mysql数据源

java代码：

```
package cn.javass.spring.chapter12.qualifier;
//省略部分import
import javax.inject.Qualifier;
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface JSR330Mysql {
}
```

2、准备测试Bean：

java代码：

```
package cn.javass.spring.chapter12;
import javax.inject.Inject;
import javax.inject.Named;
import javax.sql.DataSource;
import cn.javass.spring.chapter12.qualifier.JSR330Mysql;
public class TestBean51 {
    private DataSource mysqlDataSource;
    private DataSource oracleDataSource;
    @Inject
    public void initDataSoruce(
            @JSR330Mysql  DataSource mysqlDataSource,
            @Named("oracleDataSource") DataSource oracleDataSource) {
        this.mysqlDataSource = mysqlDataSource;
        this.oracleDataSource = oracleDataSource;

    }
```

```
        //省略getter
}
```

3、在Spring配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下Bean配置：

java代码：

```
<bean id="testBean51" class="cn.javass.spring.chapter12.TestBean51"/>
```

4、在Spring修改定义的mysqlDataSourceBean数据源：

java代码：

```
<bean id="mysqlDataSourceBean" class="org.springframework.jdbc.datasource.DriverManagerDataSou
         <qualifier value="mysqlDataSource"/>
         <qualifier type="cn.javass.spring.chapter12.qualifier.Mysql"/>
         <qualifier type="cn.javass.spring.chapter12.qualifier.DataSourceType">
             <attribute key="ip" value="localhost"/>
             <attribute key="database" value="MYSQL"/>
         </qualifier>
         <qualifier type="cn.javass.spring.chapter12.qualifier.JSR330Mysql"/>
</bean>
```

5、测试方法如下：

java代码：

```
@Test
public void testInject() {
    TestBean51 testBean51 = ctx.getBean("testBean51", TestBean51.class);
    Assert.assertEquals(ctx.getBean("mysqlDataSourceBean"), testBean51.getMysqlDataSource());
    Assert.assertEquals(ctx.getBean("oracleDataSource"), testBean51.getOracleDataSource());
}
```

测试也通过了，说明JSR-330注解也能很好工作。

从测试中可以看出JSR-330注解和Spring自带注解依赖注入时主要有以下特点：

- Spring自带的@Autowired的缺省情况等价于JSR-330的@Inject注解；
- Spring自带的@Qualifier的缺省的根据Bean名字注入情况等价于JSR-330的@Named注解；
- Spring自带的@Qualifier的扩展@Qualifier限定描述符注解情况等价于JSR-330的@Qualifier注解。

12.2.5 JPA注解

用于注入EntityManagerFactory和EntityManager。

1、准备测试Bean：

java代码：

```
package cn.javass.spring.chapter12;
//省略import
```

```
public class TestBean61 {
    @PersistenceContext(unitName = "entityManagerFactory")
    private EntityManager entityManager;

    @PersistenceUnit(unitName = "entityManagerFactory")
    private EntityManagerFactory entityManagerFactory;

    public EntityManager getEntityManager() {
        return entityManager;
    }
    public EntityManagerFactory getEntityManagerFactory() {
        return entityManagerFactory;
    }
}
```

2、在Spring配置文件（chapter12/dependecyInjectWithAnnotation.xml）添加如下Bean配置：

java代码：

```
<import resource="classpath:chapter7/applicationContext-resources.xml"/>
<import resource="classpath:chapter8/applicationContext-jpa.xml"/>
<bean id="testBean61" class="cn.javass.spring.chapter12.TestBean61"/>
```

此处需要引用第七章和八章的配置文件，细节内容请参考七八两章。

3、测试方法如下：

java代码：

```
@Test
public void testJpaInject() {
    TestBean61 testBean61 = ctx.getBean("testBean61", TestBean61.class);
    Assert.assertNotNull(testBean61.getEntityManager());
    Assert.assertNotNull(testBean61.getEntityManagerFactory());
}
```

测试也通过了，说明JPA注解也能很好工作。

JPA注解类似于@Resource注解同样是先根据unitName属性去JNDI环境中查找，如果没找到在到Spring容器中查找。

1.19 【第十二章】零配置 之 12.3 注解实现Bean定义 ——跟我学spring3

发表时间: 2012-03-22 关键字: spring, 零配置

12.3 注解实现Bean定义

12.3.1 概述

前边介绍的Bean定义全是基于XML方式定义配置元数据，且在【12.2注解实现Bean依赖注入】一节中介绍了通过注解来减少配置数量，但并没有完全消除在XML配置文件中的Bean定义，因此有没有方式完全消除XML配置Bean定义呢？

Spring提供通过扫描类路径中的特殊注解类来自动注册Bean定义。同注解驱动事务一样需要开启自动扫描并注册Bean定义支持，使用方式如下（resources/chapter12/ componentDefinitionWithAnnotation.xml）：

java代码：

```
<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"
    xmlns:context="http://www.springframework.org/schema/context"
    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
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <aop:aspectj-autoproxy />

    <context:component-scan base-package="cn.javass.spring.chapter12"/>

</beans>
```

使用<context:component-scan>标签来表示需要要自动注册Bean定义，而通过base-package属性指定扫描的类路径位置。

<context:component-scan>标签将自动开启“**注解实现Bean依赖注入**”支持。

此处我们还通过<aop:aspectj-autoproxy/>用于开启Spring对@AspectJ风格切面的支持。

Spring基于注解实现Bean定义支持如下三种注解：

- **Spring自带的@Component注解及扩展@Repository、@Service、@Controller**，如图12-1所示；
- **JSR-250 1.1版本中中定义的@ManagedBean注解**，是Java EE 6标准规范之一，不包括在JDK中，需要在应用服务器环境使用（如Jboss），如图12-2所示；
- **JSR-330的@Named注解**，如图12-3所示。

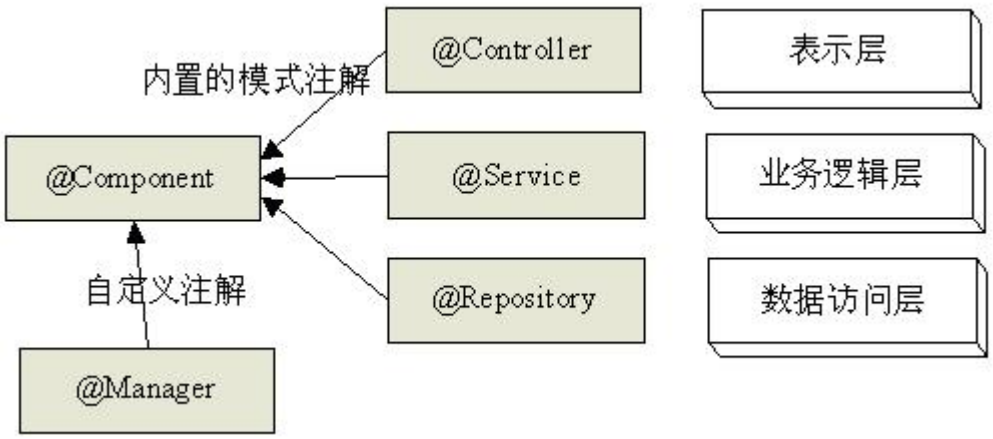


图12-1 Spring自带的@Component注解及扩展

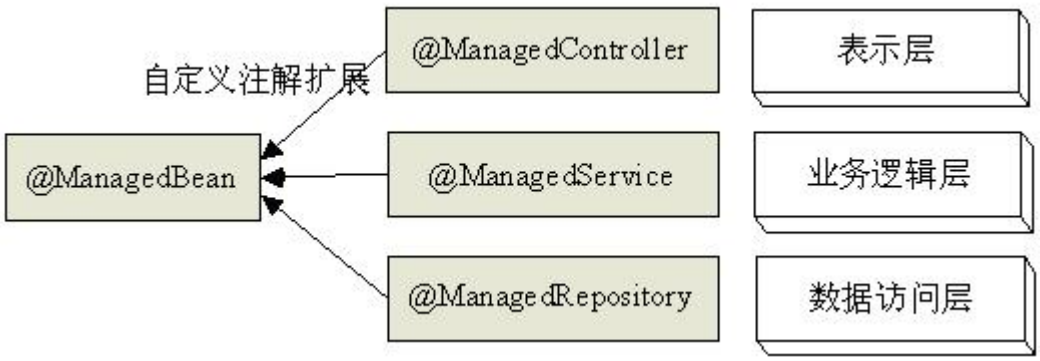


图12-2 JSR-250中定义的@ManagedBean注解及自定义扩展

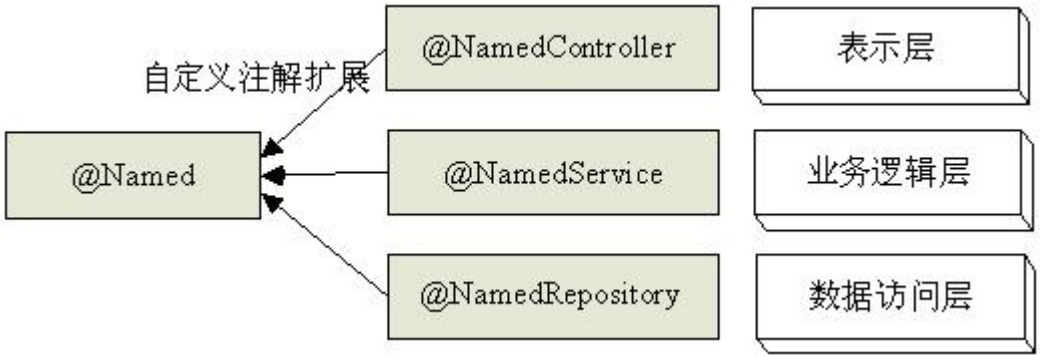


图12-3 JSR-330的@Named注解及自定义扩展

图12-2和图12-3中的自定义扩展部分是为了配合Spring自带的模式注解扩展自定义的，并不包含在Java EE 6规范中，在Java EE 6中相应的服务层、DAO层功能由EJB来完成。

在Java EE中有些注解运行放置在多个地方，如@Named允许放置在类型、字段、方法参数上等，因此一般情况下放置在类型上表示定义，放置在参数、方法等上边一般代表使用（如依赖注入等等）。

12.3.2 Spring自带的@Component注解及扩展

一、@Component：定义Spring管理Bean，使用方式如下：

java代码：

```
@Component("标识符")
POJO类
```

在类上使用@Component注解，表示该类定义为Spring管理Bean，使用默认value（可选）属性表示Bean标识符。

1、定义测试Bean类:

java代码：

```
package cn.javass.spring.chapter12;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.stereotype.Component;
@Component("component")
public class TestCompoment {
    @Autowired
    private ApplicationContext ctx;
    public ApplicationContext getCtx() {
        return ctx;
    }
}
```

2、Spring配置文件使用chapter12/ componentDefinitionWithAnnotation.xml即可且无需修改；

3、定义测试类和测试方法：

java代码：

```
package cn.javass.spring.chapter12;
//省略import
public class ComponentDefinitionWithAnnotationTest {
    private static String configLocation = "classpath:chapter12/componentDefinitionWithAnnotat
    private static ApplicationContext ctx = new ClassPathXmlApplicationContext(configLocation)
    @Test
    public void testComponent() {
        TestCompoment component = ctx.getBean("component", TestCompoment.class);
        Assert.assertNotNull(component.getCtx());
```

```
    }
}
```

测试成功说明被@Component注解的POJO类将自动被Spring识别并注册到Spring容器中，且自动支持自动装配。

@AspectJ风格的切面可以通过@Compenent注解标识其为Spring管理Bean，而@Aspect注解不能被Spring自动识别并注册为Bean，必须通过@Component注解来完成，示例如下：

java代码：

```
package cn.javass.spring.chapter12.aop;
//省略import
@Component
@Aspect
public class TestAspect {
    @Pointcut(value="execution(* *(..))")
    private void pointcut() {}
    @Before(value="pointcut()")
    public void before() {
        System.out.println("=======before");
    }
}
```

通过@Component将切面定义为Spring管理Bean。

二、@Repository：@Component扩展，被@Repository注解的POJO类表示DAO层实现，从而见到该注解就想到DAO层实现，使用方式和@Component相同；

1、定义测试Bean类:

java代码：

```
package cn.javass.spring.chapter12.dao.hibernate;
import org.springframework.stereotype.Repository;
@Repository("testHibernateDao")
public class TestHibernateDaoImpl {

}
```

2、Spring配置文件使用chapter12/ componentDefinitionWithAnnotation.xml即可且无需修改；

3、定义测试方法：

java代码：

```
@Test
public void testDao() {
TestHibernateDaoImpl dao =
ctx.getBean("testHibernateDao", TestHibernateDaoImpl.class);
Assert.assertNotNull(dao);
}
```

测试成功说明被@Repository注解的POJO类将自动被Spring识别并注册到Spring容器中，且自动支持自动装配，并且被@Repository注解的类表示DAO层实现。

三、@Service：@Component扩展，被@Service注解的POJO类表示Service层实现，从而见到该注解就想到Service层实现，使用方式和@Component相同；

1、定义测试Bean类:

java代码：

```
package cn.javass.spring.chapter12.service.impl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Service;
import cn.javass.spring.chapter12.dao.hibernate.TestHibernateDaoImpl;
@Service("testService")
public class TestServiceImpl {
    @Autowired
    @Qualifier("testHibernateDao")
    private TestHibernateDaoImpl dao;
    public TestHibernateDaoImpl getDao() {
        return dao;
    }
}
```

2、Spring配置文件使用chapter12/ componentDefinitionWithAnnotation.xml即可且无需修改；

3、定义测试方法：

java代码：

```
@Test
public void testService() {
    TestServiceImpl service = ctx.getBean("testService", TestServiceImpl.class);
    Assert.assertNotNull(service.getDao());
}
```

测试成功说明被@Service注解的POJO类将自动被Spring识别并注册到Spring容器中，且自动支持自动装配，并且被@Service注解的类表示Service层实现。

四、@Controller：@Component扩展，被@Controller注解的类表示Web层实现，从而见到该注解就想到Web层实现，使用方式和@Component相同；

1、定义测试Bean类:

java代码：

```
package cn.javass.spring.chapter12.action;
//省略import
@Controller
public class TestAction {
    @Autowired
    private TestServiceImpl testService;

    public void list() {
        //调用业务逻辑层方法
    }
}
```

2、Spring配置文件使用chapter12/ componentDefinitionWithAnnotation.xml即可且无需修改；

3、定义测试方法：

java代码：

```
@Test
public void testWeb() {
    TestAction action = ctx.getBean("testAction", TestAction.class);
    Assert.assertNotNull(action);
}
```

测试成功说明被@Controller注解的类将自动被Spring识别并注册到Spring容器中，且自动支持自动装配，并且被@Controller注解的类表示Web层实现。

大家是否注意到@Controller中并没有定义Bean的标识符，那么默认Bean的名字将是以小写开头的类名（不包括包名），即如“TestAction”类的Bean标识符为“testAction”。

六、自定义扩展：Spring内置了三种通用的扩展注解@Repository、@Service、@Controller ，大多数情况下没必要定义自己的扩展，在此我们演示下如何扩展@Component注解来满足某些特殊规约的需要；

在此我们可能需要一个缓存层用于定义缓存Bean，因此我们需要自定义一个@Cache的注解来表示缓存类。

1、扩展@Component：

java代码：

```
package cn.javass.spring.chapter12.stereotype;
//省略import
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Cache{
	String value() default "";
}
```

扩展十分简单，只需要在扩展的注解上注解@Component即可，**@Repository、**@Service、@Controller也是通过该方式实现的，没什么特别之处

2、定义测试Bean类:

java代码：

```
package cn.javass.spring.chapter12.cache;
@Cache("cache")
public class TestCache {

}
```

2、Spring配置文件使用chapter12/ componentDefinitionWithAnnotation.xml即可且无需修改；

3、定义测试方法：

java代码：

```
@Test
public void testCache() {
    TestCache cache = ctx.getBean("cache", TestCache.class);
    Assert.assertNotNull(cache);
}
```

测试成功说明自定义的@Cache注解也能很好的工作，而且实现了我们的目的，使用@Cache来表示被注解的类是Cache层Bean。

12.3.3 JSR-250中定义的@ManagedBean注解

@javax.annotation.ManagedBean需要在实现Java EE 6规范的应用服务器上使用，虽然Spring3实现了，但@javax.annotation.ManagedBean只有在Java EE 6环境中才有定义，因此测试前需要我们定义ManagedBean类。

1、定义javax.annotation.ManagedBean注解类：

java代码：

```
package javax.annotation;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface ManagedBean {
     String value() default "";
}
```

其和@Component完全相同，唯一不同的就是名字和创建者（一个是Spring，一个是Java EE规范）。

2、定义测试Bean类:

java代码：

```
package cn.javass.spring.chapter12;
import javax.annotation.Resource;
import org.springframework.context.ApplicationContext;
@javax.annotation.ManagedBean("managedBean")
public class TestManagedBean {
    @Resource
    private ApplicationContext ctx;
    public ApplicationContext getCtx() {
        return ctx;
    }
}
```

2、Spring配置文件使用chapter12/ componentDefinitionWithAnnotation.xml即可且无需修改；

3、定义测试方法：

java代码：

```
@Test
public void testManagedBean() {
    TestManagedBean testManagedBean = ctx.getBean("managedBean", TestManagedBean.class);
    Assert.assertNotNull(testManagedBean.getCtx());
}
```

测试成功说明被@ManagedBean注解类也能正常工作。

自定义扩展就不介绍了，大家可以参考@Component来完成如图12-2所示的自定义扩展部分。

12.3.4 JSR-330的@Named注解

@Named不仅可以用于依赖注入来指定注入的Bean的标识符，还可以用于定义Bean。即注解在类型上表示定义Bean，注解在非类型上（如字段）表示指定依赖注入的Bean标识符。

1、定义测试Bean类:

java代码：

```
package cn.javass.spring.chapter12;
//省略import
@Named("namedBean")
```

```
public class TestNamedBean {
    @Inject
    private ApplicationContext ctx;
    public ApplicationContext getCtx() {
        return ctx;
    }
}
```

2、Spring配置文件使用chapter12/ componentDefinitionWithAnnotation.xml即可且无需修改；

3、定义测试方法：

java代码：

```
@Test
public void testNamedBean() {
TestNamedBean testNamedBean =
    ctx.getBean("namedBean", TestNamedBean.class);
    Assert.assertNotNull(testNamedBean.getCtx());
}
```

测试成功说明被@Named注解类也能正常工作。

自定义扩展就不介绍了，大家可以参考@Component来完成如图12-3所示的自定义扩展部分。

12.3.5 细粒度控制Bean定义扫描

在XML配置中完全消除了Bean定义，而是只有一个<context:component-scan>标签来支持注解Bean定义扫描。

前边的示例完全采用默认扫描设置，如果我们有几个组件不想被扫描并自动注册、我们想更改默认的Bean标识符生成策略该如何做呢？接下来让我们看一下如何细粒度的控制Bean定义扫描，具体定义如下：

java代码：

```
<context:component-scan
        base-package=""
        resource-pattern="**/*.class"
        name-generator="org.springframework.context.annotation.AnnotationBeanNameGenerator"
        use-default-filters="true"
        annotation-config="true">
                <context:include-filter type="aspectj" expression=""/>
                <context:exclude-filter type="regex" expression=""/>
</context:component-scan>
```

- **base-package：**表示扫描注解类的开始位置，即将在指定的包中扫描，其他包中的注解类将不被扫描，默认将扫描所有类路径；
- **resource-pattern：**表示扫描注解类的后缀匹配模式，即“base-package+resource-pattern”将组成匹配模式用于匹配类路径中的组件，默认后缀为“**/*.class”，即指定包下的所有以.class结尾的类文件；
- **name-generator：默认情况下的Bean标识符生成策略，默认是**AnnotationBeanNameGenerator，其将生成以小写开头的类名（不包括包名）；可以自定义自己的标识符生成策略；
- **use-default-filters：**默认为true表示过滤@Component、@ManagedBean、@Named注解的类，如果改为false默认将不过滤这些默认的注解来定义Bean，即这些注解类不能被过滤到，即不能通过这些注解进行Bean定义；
- **annotation-config：**表示是否自动支持注解实现Bean依赖注入，默认支持，如果设置为false，将关闭支持注解的依赖注入，需要通过<context:annotation-config/>开启。

默认情况下将自动过滤@Component、@ManagedBean、@Named注解的类并将其注册为Spring管理Bean，可以通过在<context:component-scan>标签中指定自定义过滤器将过滤到匹配条件的类注册为Spring管理Bean，具体定义方式如下：

java代码：

```
<context:include-filter type="aspectj" expression=""/>
<context:exclude-filter type="regex" expression=""/>
```

- **<context:include-filter>：**表示过滤到的类将被注册为Spring管理Bean；
- **<context:exclude-filter>：**表示过滤到的类将不被注册为Spring管理Bean，它比<context:include-filter>具有更高优先级；
- **type：**表示过滤器类型，目前支持注解类型、类类型、正则表达式、aspectj表达式过滤器，当然也可以自定义自己的过滤器，实现org.springframework.core.type.filter.TypeFilter即可；
- **expression：**表示过滤器表达式。

一般情况下没必要进行自定义过滤，如果需要请参考如下示例：

1、cn.javass.spring.chapter12.TestBean14自动注册为Spring管理Bean：

java代码：

```
<context:include-filter type="assignable" expression="cn.javass.spring.chapter12.TestBean14"/>
```

2、把所有注解为org.aspectj.lang.annotation.Aspect自动注册为Spring管理Bean：

java代码：

```
<context:include-filter type="annotation"
expression="org.aspectj.lang.annotation.Aspect"/>
```

3、将把匹配到正则表达式“cn\.javass\.spring\.chapter12\.TestBean2*”排除，不注册为Spring管理Bean：

java代码：

```
<context:exclude-filter type="regex" expression="cn\.javass\.spring\.chapter12\.TestBean2*"/>
```

4、将把匹配到aspectj表达式“cn.javass.spring.chapter12.TestBean3*”排除，不注册为Spring管理Bean：

java代码：

```
<context:exclude-filter type="aspectj" expression="cn.javass.spring.chapter12.TestBean3*"/>
```

具体使用就要看项目需要了，如果以上都不满足需要请考虑使用自定义过滤器。

12.3.6 提供更多的配置元数据

1、@Lazy：定义Bean将延迟初始化，使用方式如下：

java代码：

```
@Component("component")
@Lazy(true)
public class TestCompoment {
……
}
```

使用@Lazy注解指定Bean需要延迟初始化。

2、**@DependsOn**：定义Bean初始化及销毁时的顺序，使用方式如下：

java代码：

```
@Component("component")
@DependsOn({"managedBean"})
public class TestCompoment {
……
}
```

3、@Scope：定义Bean作用域，默认单例，使用方式如下：

java代码：

```
@Component("component")
@Scope("singleton")
public class TestCompoment {
……
}
```

4、@Qualifier：指定限定描述符，对应于基于XML配置中的<qualifier>标签，使用方式如下：

java代码：

```
@Component("component")
@Qualifier("component")
public class TestCompoment {
……
}
```

可以使用复杂的扩展，如@Mysql等等。

5、@Primary：自动装配时当出现多个Bean候选者时，被注解为@Primary的Bean将作为首选者，否则将抛出异常，使用方式如下：

java代码：

```
@Component("component")
@Primary
public class TestCompoment {
……
}
```

1.20 java私塾的spring培训PPT（欢迎下载）

发表时间: 2012-03-22 关键字: spring

java私塾的 spring培训的PPT 欢迎大家下载。

包括

IoC/DI 思想

AOP

Spring JDBC 框架 和 ORM框架集成

事务管理

SSH集成

内容摘要

使用IoC/DI容器开发需要改变的思路：

1、应用程序不主动创建对象，但要描述创建它们的方式。

2、在应用程序代码中不直接进行服务的装配，但要配置文件中描述哪一个组件需要哪一项服务。容器负责将这些装配在一起。

其原理是基于OO设计原则的The Hollywood Principle：Don‘t call us, we’ll call you（别找我，我会来找你的）。也就是说，所有的组件都是被动的（Passive），所有的组件初始化和装配都由容器负责。组件处在一个容器当中，由容器负责管理。

IoC容器功能：实例化和初始化组件，装配组件关系、生命周期管理

本质：IoC：控制权的转移，由应用程序转移到框架；

IoC/DI容器：**由**应用程序主动实例化对象**变**被动等待对象（被动实例化）；

DI： 由专门的装配器装配组件之间的关系；

IoC/DI容器：**由**应用程序主动装配对象的依赖**变**应用程序被动接受依赖

注意：IoC/DI是思想，不是技术。IoC是框架共性，只是控制权的转移，转移到框架，所以不能因为实现了IoC就叫IoC容器，而一般除了实现了IoC外，还具有DI功能的才叫IoC容器，因为容器还要负责装配组件关系，管理组件生命周期。

更多PPT下载 请到 私塾在线网站 下载

附件下载:

- Java私塾_Spring培训PPT.rar (588.4 KB)
- dl.iteye.com/topics/download/6bb3ad9e-e686-38f2-ac6e-f2599fb4889a

1.21 spring培训PPT（欢迎下载）

发表时间: 2012-03-24 关键字: spring

java私塾的 spring培训的PPT 欢迎大家下载。

包括

IoC/DI 思想

AOP

Spring JDBC 框架 和 ORM框架集成

事务管理

SSH集成

附件下载:

- Java私塾_Spring培训PPT.rar (588.4 KB)
- dl.iteye.com/topics/download/2ff4757f-37d7-359c-8ca6-d77fbda2a072

1.22 【第十二章】零配置 之 12.4 基于Java类定义Bean配置元数据 ——跟我学spring3

发表时间: 2012-03-26 关键字: spring

12.4 基于Java类定义Bean配置元数据

12.4.1 概述

基于Java类定义Bean配置元数据，其实就是通过Java类定义Spring配置元数据，且直接消除XML配置文件。

基于Java类定义Bean配置元数据中的@Configuration注解的类等价于XML配置文件，@Bean注解的方法等价于XML配置文件中的Bean定义。

基于Java类定义Bean配置元数据需要通过AnnotationConfigApplicationContext加载配置类及初始化容器，类似于XML配置文件需要使用ClassPathXmlApplicationContext加载配置文件及初始化容器。

基于Java类定义Bean配置元数据需要CGLIB的支持，因此要保证类路径中包括CGLIB的jar包。

12.4.2 Hello World

首先让我们看一下基于Java类如何定义Bean配置元数据，具体步骤如下：

1、 通过@Configuration注解需要作为配置的类，表示该类将定义Bean配置元数据；

2、 通过@Bean注解相应的方法，该方法名默认就是Bean名，该方法返回值就是Bean对象；

3、 通过AnnotationConfigApplicationContext或子类加载基于Java类的配置。

接下来让我们先来学习一下如何通过Java类定义Bean配置元数据吧：

1、定义配置元数据的Java类如下所示：

java代码：

```
package cn.javass.spring.chapter12.configuration;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Configuration
public class ApplicationContextConfig {
    @Bean
    public String message() {
        return "hello";
    }
}
```

2、定义测试类，测试一下Java配置类是否工作：

java代码：

```
package cn.javass.spring.chapter12.configuration;
//省略import
public class ConfigurationTest {
    @Test
    public void testHelloworld () {
        AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(Applic
        Assert.assertEquals("hello", ctx.getBean("message"));
    }
}
```

测试没有报错说明测试通过了，那AnnotationConfigApplicationContext是如何工作的呢，接下来让我们分析一下：

- 使用@Configuration注解配置类，该配置类定义了Bean配置元数据；
- 使用@Bean注解配置类中的方法，该方法名就是Bean的名字，该方法返回值就是Bean对象。
- 使用new AnnotationConfigApplicationContext(ApplicationContextConfig.class)创建应用上下文，构造器参数为使用@Configuration注解的配置类，读取配置类进行实例化相应的Bean。

知道如何使用了，接下来就详细介绍每个部分吧。

12.4.3 @Configuration

通过@Configuration注解的类将被作为配置类使用，表示在该类中将定义Bean配置元数据，且使用@Configuration注解的类本身也是一个Bean，使用方式如下所示：

java代码：

```
import org.springframework.context.annotation.Configuration;
@Configuration("ctxConfig")
public class ApplicationContextConfig {
    //定义Bean配置元数据
}
```

因为使用@Configuration注解的类本身也是一个Bean，因为@Configuration被@Component注解了，因此@Configuration注解可以指定value属性值，如“ctxConfig”就是该Bean的名字，如使用“ctx.getBean("ctxConfig")”将返回该Bean。

使用@Configuration注解的类不能是final的，且应该有一个默认无参构造器。

12.4.4 @Bean

通过@Bean注解配置类中的相应方法，则该方法名默认就是Bean名，该方法返回值就是Bean对象，并定义了Spring IoC容器如何实例化、自动装配、初始化Bean逻辑，具体使用方法如下：

java代码：

```
@Bean(name={},
      autowire=Autowire.NO,
      initMethod="",
      destroyMethod="")
```

- **name**：指定Bean的名字，可有多个，第一个作为Id，其他作为别名；
- **autowire**：自动装配，默认no表示不自动装配该Bean，另外还有Autowire.BY_NAME表示根据名字自动装配，Autowire.BY_TYPE表示根据类型自动装配；
- **initMethod和destroyMethod**：指定Bean的初始化和销毁方法。

示例如下所示（ApplicationContextConfig.java）

java代码：

```
@Bean
public String message() {
    return new String("hello");
}
```

如上使用方式等价于如下基于XML配置方式

java代码：

```
<bean id="message" class="java.lang.String">
    <constructor-arg index="0" value="hello"/>
</bean>
```

使用@Bean注解的方法不能是private、final或static的。

12.4.5 提供更多的配置元数据

详见【12.3.6 提供更多的配置元数据】中介绍的各种注解，这些注解同样适用于@Bean注解的方法。

12.4.6 依赖注入

基于Java类配置方式的Bean依赖注入有如下两种方式：

- 直接依赖注入，类似于基于XML配置方式中的显示依赖注入；
- 使用注解实现Bean依赖注入：如@Autowired等等。

在本示例中我们将使用【第三章 DI】中的测试Bean。

1、 直接依赖注入：包括构造器注入和setter注入。

- **构造器注入：**通过在@Bean注解的实例化方法中使用有参构造器实例化相应的Bean即可，如下所示(ApplicationContextConfig.java)：

java代码：

```
@Bean
public HelloApi helloImpl3() {
```

```
    //通过构造器注入,分别是引用注入（message()）和常量注入（1）
    return new HelloImpl3(message(), 1); //测试Bean详见【3.1.2  构造器注入】
}
```

- **setter注入**：通过在@Bean注解的实例化方法中使用无参构造器实例化后，通过相应的setter方法注入即可，如下所示(ApplicationContextConfig.java)：

java代码：

```
@Bean
public HelloApi helloImpl4() {
    HelloImpl4 helloImpl4 = new HelloImpl4();//测试Bean详见【3.1.3  setter注入】
    //通过setter注入注入引用
    helloImpl4.setMessage(message());
    //通过setter注入注入常量
    helloImpl4.setIndex(1);
    return helloImpl4;
}
```

2、使用注解实现Bean依赖注入：详见【12.2 注解实Bean依赖注入】。

具体测试方法如下(ConfigurationTest.java)：

java代码：

```
@Test
public void testDependencyInject() {
    AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(Applicatio
    ctx.getBean("helloImpl3", HelloApi.class).sayHello();
    ctx.getBean("helloImpl4", HelloApi.class).sayHello();
}
```

12.4.7 方法注入

在基于XML配置方式中，Spring支持查找方法注入和替换方法注入，但在基于Java配置方式中只支持查找方法注入，一般用于在一个单例Bean中注入一个原型Bean的情况，具体详见【3.3.5 方法注入】，如下所示（ApplicationContextConfig.java）：

java代码：

```
@Bean
@Scope("singleton")
public HelloApi helloApi2() {
    HelloImpl5 helloImpl5 = new HelloImpl5() {
        @Override
        public Printer createPrototypePrinter() {
            //方法注入，注入原型Bean
            return prototypePrinter();
        }
        @Override
        public Printer createSingletonPrinter() {
            //方法注入，注入单例Bean
            return singletonPrinter();
        }
    };
    //依赖注入,注入单例Bean
    helloImpl5.setPrinter(singletonPrinter());
```

```
    return helloImpl5;
}
```

java代码：

```
@Bean
@Scope(value="prototype")
public Printer prototypePrinter() {
    return new Printer();
 }
@Bean
@Scope(value="singleton")
public Printer singletonPrinter() {
    return new Printer();
}
```

具体测试方法如下(ConfigurationTest.java)：

java代码：

```
@Test
public void testLookupMethodInject() {
    AnnotationConfigApplicationContext ctx =
        new AnnotationConfigApplicationContext(ApplicationContextConfig.class);
    System.out.println("=======prototype sayHello======");
    HelloApi helloApi2 = ctx.getBean("helloApi2", HelloApi.class);
    helloApi2.sayHello();
    helloApi2 = ctx.getBean("helloApi2", HelloApi.class);
```

```
    helloApi2.sayHello();
}
```

如上测试等价于【3.3.5 方法注入】中的查找方法注入。

12.4.8 @Import

类似于基于XML配置中的<import/>，基于Java的配置方式提供了@Import来组合模块化的配置类，使用方式如下所示：

java代码：

```
package cn.javass.spring.chapter12.configuration;
//省略import
@Configuration("ctxConfig2")
@Import({ApplicationContextConfig.class})
public class ApplicationContextConfig2 {
    @Bean(name = {"message2"})
    public String message() {
        return "hello";
    }
}
```

具体测试方法如下(ConfigurationTest.java)：

java代码：

```
@Test
public void  importTest() {
```

```
    AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(Applicatio
    Assert.assertEquals("hello", ctx.getBean("message"));
}
```

使用非常简单，在此就不多介绍了。

12.4.9 结合基于Java和基于XML方式的配置

基于Java方式的配置方式不是为了完全替代基于XML方式的配置，两者可以结合使用，因此可以有两种结合使用方式：

- 在基于Java方式的配置类中引入基于XML方式的配置文件；
- 在基于XML方式的配置文件中中引入基于Java方式的配置。

一、在基于Java方式的配置类中引入基于XML方式的配置文件：在@Configuration注解的配置类上通过@ImportResource注解引入基于XML方式的配置文件，示例如下所示：

1、定义基于XML方式的配置文件(chapter12/configuration/importResource.xml)：

java代码：

```
<bean id="message3" class="java.lang.String">
    <constructor-arg index="0" value="test"></constructor-arg>
</bean>
```

2、修改基于Java方式的配置类ApplicationContextConfig，添加如下注解：

java代码：

```
@Configuration("ctxConfig") //1、使用@Configuration注解配置类
@ImportResource("classpath:chapter12/configuration/importResource.xml")
```

```
public class ApplicationContextConfig {
……
}
```

使用@ImportResource引入基于XML方式的配置文件，如果有多个请使用@ImportResource({"config1.xml", "config2.xml"})方式指定多个配置文件。

二、在基于XML方式的配置文件中中引入基于Java方式的配置：直接在XML配置文件中声明使用@Configuration注解的配置类即可，示例如下所示：

1、定义基于Java方式的使用@Configuration注解的配置类在此我们使用ApplicationContextConfig.java。

2、定义基于XML方式的配置文件（chapter12/configuration/xml-config.xml）：

java代码：

```
<context:annotation-config/>
<bean id="ctxConfig" class="cn.javass.spring.chapter12.configuration.ApplicationContextConfig"/
```

- <context:annotation-config/>：用于开启对注解驱动支持，详见【12.2 注解实现Bean依赖注入】；
- <bean id="ctxConfig" class="……"/>：直接将使用@Configuration注解的配置类在配置文件中进行Bean定义即可。

3、测试代码如下所示(ConfigurationTest.java)：：

java代码：

```
public void testXmlConfig() {
    String configLocations[] = {"chapter12/configuration/xml-config.xml"};
    ApplicationContext ctx = new ClassPathXmlApplicationContext(configLocations);
    Assert.assertEquals("hello", ctx.getBean("message"));
}
```

测试成功，说明通过在基于XML方式的配置文件中能获取到基于Java方式的配置文件中定义的Bean，如“message”Bean。

12.4.10 基于Java方式的容器实例化

基于Java方式的容器由AnnotationConfigApplicationContext表示，其实例化方式主要有以下几种：

一、对于只有一个@Configuration注解的配置类，可以使用如下方式初始化容器：

java代码：

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(ApplicationCon
```

二、对于有多个@Configuration注解的配置类，可以使用如下方式初始化容器：

java代码：

```
AnnotationConfigApplicationContext ctx1 = new AnnotationConfigApplicationContext(ApplicationCo
```

或者

java代码：

```
AnnotationConfigApplicationContext ctx2 = new AnnotationConfigApplicationContext();
ctx2.register(ApplicationContextConfig.class);
ctx2.register(ApplicationContextConfig2.class);
```

三、对于【12.3 注解实现Bean定义】中通过扫描类路径中的特殊注解类来自动注册Bean定义，可以使用如下方式来实现：

java代码：

```
public void testComponentScan() {
    AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
    ctx.scan("cn.javass.chapter12.confiuration");
    ctx.refresh();
    Assert.assertEquals("hello", ctx.getBean("message"));
}
```

以上配置方式等价于基于XML方式中的如下配置：

java代码：

```
<context:component-scan base-package="cn.javass.chapter12.confiuration"/>
```

四、在web环境中使用基于Java方式的配置，通过修改通用配置实现，详见**【10.1.2 通用配置】**：

1、修改通用配置中的Web应用上下文实现，在此需要使用AnnotationConfigWebApplicationContext：

java代码：

```
<context-param>
    <param-name>contextClass</param-name>
    <param-value>
        org.springframework.web.context.support.AnnotationConfigWebApplicationContext
    </param-value>
</context-param>
```

2、指定加载配置类，类似于指定加载文件位置，在基于Java方式中需要指定需要加载的配置类：

java代码：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        cn.javass.spring.chapter12.configuration.ApplicationContextConfig,
        cn.javass.spring.chapter12.configuration.ApplicationContextConfig2
    </param-value>
</context-param>
```

- **contextConfigLocation**：除了可以指定配置类，还可以指定“扫描的类路径”，其加载步骤如下：

1、首先验证指定的配置是否是类，如果是则通过注册配置类来完成Bean定义加载，即如通过ctx.register(ApplicationContextConfig.class)加载定义；

2、如果指定的配置不是类，则通过扫描类路径方式加载注解Bean定义，即将通过ctx.scan("cn.javass.chapter12.confiuration")加载Bean定义。

1.23 【第十二章】零配置 之 12.4 基于Java类定义Bean配置元数据 ——跟我学spring3

发表时间: 2012-03-26 关键字: spring

12.4 基于Java类定义Bean配置元数据

12.4.1 概述

基于Java类定义Bean配置元数据，其实就是通过Java类定义Spring配置元数据，且直接消除XML配置文件。

基于Java类定义Bean配置元数据中的@Configuration注解的类等价于XML配置文件，@Bean注解的方法等价于XML配置文件中的Bean定义。

基于Java类定义Bean配置元数据需要通过AnnotationConfigApplicationContext加载配置类及初始化容器，类似于XML配置文件需要使用ClassPathXmlApplicationContext加载配置文件及初始化容器。

基于Java类定义Bean配置元数据需要CGLIB的支持，因此要保证类路径中包括CGLIB的jar包。

12.4.2 Hello World

首先让我们看一下基于Java类如何定义Bean配置元数据，具体步骤如下：

1、 通过@Configuration注解需要作为配置的类，表示该类将定义Bean配置元数据；

2、 通过@Bean注解相应的方法，该方法名默认就是Bean名，该方法返回值就是Bean对象；

3、 通过AnnotationConfigApplicationContext或子类加载基于Java类的配置。

接下来让我们先来学习一下如何通过Java类定义Bean配置元数据吧：

1、定义配置元数据的Java类如下所示：

java代码：

```
package cn.javass.spring.chapter12.configuration;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Configuration
public class ApplicationContextConfig {
    @Bean
    public String message() {
        return "hello";
    }
}
```

2、定义测试类，测试一下Java配置类是否工作：

java代码：

```
package cn.javass.spring.chapter12.configuration;
//省略import
public class ConfigurationTest {
    @Test
    public void testHelloworld () {
        AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(Applic
        Assert.assertEquals("hello", ctx.getBean("message"));
    }
}
```

测试没有报错说明测试通过了，那AnnotationConfigApplicationContext是如何工作的呢，接下来让我们分析一下：

- 使用@Configuration注解配置类，该配置类定义了Bean配置元数据；
- 使用@Bean注解配置类中的方法，该方法名就是Bean的名字，该方法返回值就是Bean对象。
- 使用new AnnotationConfigApplicationContext(ApplicationContextConfig.class)创建应用上下文，构造器参数为使用@Configuration注解的配置类，读取配置类进行实例化相应的Bean。

知道如何使用了，接下来就详细介绍每个部分吧。

12.4.3 @Configuration

通过@Configuration注解的类将被作为配置类使用，表示在该类中将定义Bean配置元数据，且使用@Configuration注解的类本身也是一个Bean，使用方式如下所示：

java代码：

```
import org.springframework.context.annotation.Configuration;
@Configuration("ctxConfig")
public class ApplicationContextConfig {
    //定义Bean配置元数据
}
```

因为使用@Configuration注解的类本身也是一个Bean，因为@Configuration被@Component注解了，因此@Configuration注解可以指定value属性值，如“ctxConfig”就是该Bean的名字，如使用“ctx.getBean("ctxConfig")”将返回该Bean。

使用@Configuration注解的类不能是final的，且应该有一个默认无参构造器。

12.4.4 @Bean

通过@Bean注解配置类中的相应方法，则该方法名默认就是Bean名，该方法返回值就是Bean对象，并定义了Spring IoC容器如何实例化、自动装配、初始化Bean逻辑，具体使用方法如下：

java代码：

```
@Bean(name={},
      autowire=Autowire.NO,
      initMethod="",
      destroyMethod="")
```

- **name**：指定Bean的名字，可有多个，第一个作为Id，其他作为别名；
- **autowire**：自动装配，默认no表示不自动装配该Bean，另外还有Autowire.BY_NAME表示根据名字自动装配，Autowire.BY_TYPE表示根据类型自动装配；
- **initMethod和destroyMethod**：指定Bean的初始化和销毁方法。

示例如下所示（ApplicationContextConfig.java）

java代码：

```
@Bean
public String message() {
    return new String("hello");
}
```

如上使用方式等价于如下基于XML配置方式

java代码：

```
<bean id="message" class="java.lang.String">
    <constructor-arg index="0" value="hello"/>
</bean>
```

使用@Bean注解的方法不能是private、final或static的。

12.4.5 提供更多的配置元数据

详见【12.3.6 提供更多的配置元数据】中介绍的各种注解，这些注解同样适用于@Bean注解的方法。

12.4.6 依赖注入

基于Java类配置方式的Bean依赖注入有如下两种方式：

- 直接依赖注入，类似于基于XML配置方式中的显示依赖注入；
- 使用注解实现Bean依赖注入：如@Autowired等等。

在本示例中我们将使用【第三章 DI】中的测试Bean。

1、 直接依赖注入：包括构造器注入和setter注入。

- **构造器注入：**通过在@Bean注解的实例化方法中使用有参构造器实例化相应的Bean即可，如下所示(ApplicationContextConfig.java)：

java代码：

```
@Bean
public HelloApi helloImpl3() {
```

```
    //通过构造器注入,分别是引用注入（message()）和常量注入（1）
    return new HelloImpl3(message(), 1); //测试Bean详见【3.1.2  构造器注入】
}
```

- **setter注入**：通过在@Bean注解的实例化方法中使用无参构造器实例化后，通过相应的setter方法注入即可，如下所示(ApplicationContextConfig.java)：

java代码：

```
@Bean
public HelloApi helloImpl4() {
    HelloImpl4 helloImpl4 = new HelloImpl4();//测试Bean详见【3.1.3  setter注入】
    //通过setter注入注入引用
    helloImpl4.setMessage(message());
    //通过setter注入注入常量
    helloImpl4.setIndex(1);
    return helloImpl4;
}
```

2、使用注解实现Bean依赖注入：详见【12.2 注解实Bean依赖注入】。

具体测试方法如下(ConfigurationTest.java)：

java代码：

```
@Test
public void testDependencyInject() {
    AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(Applicatio
    ctx.getBean("helloImpl3", HelloApi.class).sayHello();
    ctx.getBean("helloImpl4", HelloApi.class).sayHello();
}
```

12.4.7 方法注入

在基于XML配置方式中，Spring支持查找方法注入和替换方法注入，但在基于Java配置方式中只支持查找方法注入，一般用于在一个单例Bean中注入一个原型Bean的情况，具体详见【3.3.5 方法注入】，如下所示（ApplicationContextConfig.java）：

java代码：

```
@Bean
@Scope("singleton")
public HelloApi helloApi2() {
    HelloImpl5 helloImpl5 = new HelloImpl5() {
        @Override
        public Printer createPrototypePrinter() {
            //方法注入，注入原型Bean
            return prototypePrinter();
        }
        @Override
        public Printer createSingletonPrinter() {
            //方法注入，注入单例Bean
            return singletonPrinter();
        }
    };
    //依赖注入,注入单例Bean
    helloImpl5.setPrinter(singletonPrinter());
```

```
    return helloImpl5;
}
```

java代码：

```
@Bean
@Scope(value="prototype")
public Printer prototypePrinter() {
    return new Printer();
 }
@Bean
@Scope(value="singleton")
public Printer singletonPrinter() {
    return new Printer();
}
```

具体测试方法如下(ConfigurationTest.java)：

java代码：

```
@Test
public void testLookupMethodInject() {
    AnnotationConfigApplicationContext ctx =
        new AnnotationConfigApplicationContext(ApplicationContextConfig.class);
    System.out.println("=======prototype sayHello======");
    HelloApi helloApi2 = ctx.getBean("helloApi2", HelloApi.class);
    helloApi2.sayHello();
    helloApi2 = ctx.getBean("helloApi2", HelloApi.class);
```

```
    helloApi2.sayHello();
}
```

如上测试等价于【3.3.5 方法注入】中的查找方法注入。

12.4.8 @Import

类似于基于XML配置中的<import/>，基于Java的配置方式提供了@Import来组合模块化的配置类，使用方式如下所示：

java代码：

```
package cn.javass.spring.chapter12.configuration;
//省略import
@Configuration("ctxConfig2")
@Import({ApplicationContextConfig.class})
public class ApplicationContextConfig2 {
    @Bean(name = {"message2"})
    public String message() {
        return "hello";
    }
}
```

具体测试方法如下(ConfigurationTest.java)：

java代码：

```
@Test
public void  importTest() {
```

```
    AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(Applicatio
    Assert.assertEquals("hello", ctx.getBean("message"));
}
```

使用非常简单，在此就不多介绍了。

12.4.9 结合基于Java和基于XML方式的配置

基于Java方式的配置方式不是为了完全替代基于XML方式的配置，两者可以结合使用，因此可以有两种结合使用方式：

- 在基于Java方式的配置类中引入基于XML方式的配置文件；
- 在基于XML方式的配置文件中中引入基于Java方式的配置。

一、在基于Java方式的配置类中引入基于XML方式的配置文件：在@Configuration注解的配置类上通过@ImportResource注解引入基于XML方式的配置文件，示例如下所示：

1、定义基于XML方式的配置文件(chapter12/configuration/importResource.xml)：

java代码：

```
<bean id="message3" class="java.lang.String">
    <constructor-arg index="0" value="test"></constructor-arg>
</bean>
```

2、修改基于Java方式的配置类ApplicationContextConfig，添加如下注解：

java代码：

```
@Configuration("ctxConfig") //1、使用@Configuration注解配置类
@ImportResource("classpath:chapter12/configuration/importResource.xml")
```

```
public class ApplicationContextConfig {
……
}
```

使用@ImportResource引入基于XML方式的配置文件，如果有多个请使用@ImportResource({"config1.xml", "config2.xml"})方式指定多个配置文件。

二、在基于XML方式的配置文件中中引入基于Java方式的配置：直接在XML配置文件中声明使用@Configuration注解的配置类即可，示例如下所示：

1、定义基于Java方式的使用@Configuration注解的配置类在此我们使用ApplicationContextConfig.java。

2、定义基于XML方式的配置文件（chapter12/configuration/xml-config.xml）：

java代码：

```
<context:annotation-config/>
<bean id="ctxConfig" class="cn.javass.spring.chapter12.configuration.ApplicationContextConfig"/
```

- <context:annotation-config/>：用于开启对注解驱动支持，详见【12.2 注解实现Bean依赖注入】；
- <bean id="ctxConfig" class="……"/>：直接将使用@Configuration注解的配置类在配置文件中进行Bean定义即可。

3、测试代码如下所示(ConfigurationTest.java)：：

java代码：

```
public void testXmlConfig() {
    String configLocations[] = {"chapter12/configuration/xml-config.xml"};
    ApplicationContext ctx = new ClassPathXmlApplicationContext(configLocations);
    Assert.assertEquals("hello", ctx.getBean("message"));
}
```

测试成功，说明通过在基于XML方式的配置文件中能获取到基于Java方式的配置文件中定义的Bean，如“message”Bean。

12.4.10 基于Java方式的容器实例化

基于Java方式的容器由AnnotationConfigApplicationContext表示，其实例化方式主要有以下几种：

一、对于只有一个@Configuration注解的配置类，可以使用如下方式初始化容器：

java代码：

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(ApplicationCon
```

二、对于有多个@Configuration注解的配置类，可以使用如下方式初始化容器：

java代码：

```
AnnotationConfigApplicationContext ctx1 = new AnnotationConfigApplicationContext(ApplicationCo
```

或者

java代码：

```
AnnotationConfigApplicationContext ctx2 = new AnnotationConfigApplicationContext();
ctx2.register(ApplicationContextConfig.class);
ctx2.register(ApplicationContextConfig2.class);
```

三、对于【12.3 注解实现Bean定义】中通过扫描类路径中的特殊注解类来自动注册Bean定义，可以使用如下方式来实现：

java代码：

```
public void testComponentScan() {
    AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
    ctx.scan("cn.javass.chapter12.confiuration");
    ctx.refresh();
    Assert.assertEquals("hello", ctx.getBean("message"));
}
```

以上配置方式等价于基于XML方式中的如下配置：

java代码：

```
<context:component-scan base-package="cn.javass.chapter12.confiuration"/>
```

四、在web环境中使用基于Java方式的配置，通过修改通用配置实现，详见**【10.1.2 通用配置】**：

1、修改通用配置中的Web应用上下文实现，在此需要使用AnnotationConfigWebApplicationContext：

java代码：

```
<context-param>
    <param-name>contextClass</param-name>
    <param-value>
        org.springframework.web.context.support.AnnotationConfigWebApplicationContext
    </param-value>
</context-param>
```

2、指定加载配置类，类似于指定加载文件位置，在基于Java方式中需要指定需要加载的配置类：

java代码：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        cn.javass.spring.chapter12.configuration.ApplicationContextConfig,
        cn.javass.spring.chapter12.configuration.ApplicationContextConfig2
    </param-value>
</context-param>
```

- **contextConfigLocation**：除了可以指定配置类，还可以指定“扫描的类路径”，其加载步骤如下：

1、首先验证指定的配置是否是类，如果是则通过注册配置类来完成Bean定义加载，即如通过ctx.register(ApplicationContextConfig.class)加载定义；

2、如果指定的配置不是类，则通过扫描类路径方式加载注解Bean定义，即将通过ctx.scan("cn.javass.chapter12.confiuration")加载Bean定义。

1.24 【第十二章】零配置 之 12.5 综合示例-积分商城 ——跟我学spring3

发表时间: 2012-03-27 关键字: spring

12.5 综合示例

12.5.1 概述

在第十一章中我们介绍了SSH集成，在进行SSH集成时都是通过基于XML配置文件配置每层的Bean，从而产生许多XML配置文件，本节将通过注解方式消除部分XML配置文件，实现所谓的零配置。

12.5.2 项目拷贝

1、拷贝【第十一章 SSH集成开发】中的“pointShop”项目将其命名为“pointShop2”；

2、修改“pointShop2”项目下的“.settings”文件夹下的“org.eclipse.wst.common.component”文件，将“**<property name="context-root" value="pointShop"/>**”修改为“**<property name="context-root" value="pointShop2"/>**”，即该web项目的上下文为“pointShop2”，在浏览器中可以通过http://localhost:8080/pointShop2来访问该web项目。

12.5.3 数据访问层变化

将dao层配置文件中的dao实现Bean定义删除，通过在dao实现类头上添加“@Repository”来定义dao实现Bean，并通过注解@Autowired来完成依赖注入。

1、删除DAO层配置文件(cn/javass/point/dao/applicationContext-hibernate.xml)中的如下配置：

java代码：

```
<bean id="abstractDao" abstract="true" init-method="init">
<property name="sessionFactory" ref="sessionFactory"/>
</bean>
<bean id="goodsDao" class="cn.javass.point.dao.hibernate.GoodsHibernateDao"
```

```
parent="abstractDao"/>
<bean id="goodsCodeDao" class="cn.javass.point.dao.hibernate.GoodsCodeHibernateDao"
parent="abstractDao"/>
```

2、修改通用DAO实现cn.javass.commons.dao.hibernate.BaseHibernateDao，通过注解实现依赖注入和指定初始化方法：

java代码：

```
public abstract class BaseHibernateDao<M extends Serializable, PK extends Serializable> extends
    //省略类字段
    @Autowired @Required
    public void setSf(SessionFactory sf) {
        setSessionFactory(sf);
    }
    @PostConstruct
    @SuppressWarnings("unchecked")
    public void init() {
        //省略具体实现代码
    }
}
```

- setSf方法：通过@Autowired注解自动注入SessionFactory实现；
- init方法：通过@PostConstruct注解表示该方法是初始化方法；

3、修改cn.javass.point.dao.hibernate.GoodsHibernateDao，在该类上添加@Repository注解来进行DAO层Bean定义：

java代码：

```
@Repository
public class GoodsHibernateDao extends BaseHibernateDao<GoodsModel, Integer> implements IGoodsI
……
}
```

4、修改cn.javass.point.dao.hibernate.GoodsCodeHibernateDao，在该类上添加@Repository注解来进行DAO层Bean定义：

java代码：

```
@Repository
public class GoodsCodeHibernateDao extends BaseHibernateDao<GoodsCodeModel, Integer> implements
……
}
```

DAO层到此就修改完毕，其他地方无需修改。

12.5.4 业务逻辑层变化

将service层配置文件中的service实现Bean定义删除，通过在service实现类头上添加“@Service”来定义service实现Bean，并通过注解@Autowired来完成依赖注入。

1、删除Service层配置文件(cn/javass/point/service/applicationContext-service.xml)中的如下配置：

java代码：

```
<bean id="goodsService" class="cn.javass.point.service.impl.GoodsServiceImpl">
    <property name="dao" ref="goodsDao"/>
</bean>
```

```
<bean id="goodsCodeService" class="cn.javass.point.service.impl.GoodsCodeServiceImpl">
    <property name="dao" ref="goodsCodeDao"/>
    <property name="goodsService" ref="goodsService"/>
</bean>
```

2、修改cn.javass.point.service.impl.GoodsServiceImpl，在该类上添加@Service注解来进行Service层Bean定义：

java代码：

```
@Service
public class GoodsServiceImpl extends BaseServiceImpl<GoodsModel, Integer> implements IGoodsSer

    @Autowired @Required
    public void setGoodsDao(IGoodsDao dao) {
        setDao(dao);
    }
}
```

- **setGoodsDao方法：**用于注入IGoodsDao实现，此处直接委托给setDao方法。

3、修改cn.javass.point.service.impl.GoodsCodeServiceImpl，在该类上添加@Service注解来进行Service层Bean定义：

java代码：

```
@Service
public class GoodsCodeServiceImpl extends BaseServiceImpl<GoodsCodeModel, Integer> implements I
    @Autowired @Required
    public void setGoodsCodeDao(IGoodsCodeDao dao) {
```

```
        setDao(dao);
    }
    @Autowired @Required
    public void setGoodsService(IGoodsService goodsService) {
        this.goodsService = goodsService;
    }
}
```

- **setGoodsCodeDao方法：**用于注入IGoodsCodeDao实现，此处直接委托给setDao方法；
- **setGoodsService方法：**用于注入IGoodsService实现。

Service层到此就修改完毕，其他地方无需修改。

12.5.5 表现层变化

类似于数据访问层和业务逻辑层修改，对于表现层配置文件直接删除，通过在action实现类头上添加“@Controller”来定义action实现Bean，并通过注解@Autowired来完成依赖注入。

1、 删除表现层所有Spring配置文件(cn/javass/point/web)：

java代码：

```
cn/javass/point/web/pointShop-admin-servlet.xml
cn/javass/point/web/pointShop-front-servlet.xml
```

2、修改表现层管理模块的cn.javass.point.web.admin.action.GoodsAction，在该类上添加@Controller注解来进行表现层Bean定义，且作用域为“prototype” ：

java代码：

```
@Controller("/admin/goodsAction")
@Scope("prototype")
public class GoodsAction extends BaseAction {
    private IGoodsService goodsService;
    @Autowired @Required
    public void setGoodsService(IGoodsService goodsService) {
        this.goodsService = goodsService;
    }
}
```

- **setGoodsService方法：**用于注入IGoodsService实现。

3、修改表现层管理模块的cn.javass.point.web.admin.action.GoodsCodeAction，在该类上添加@Controller注解来进行表现层Bean定义，且作用域为“prototype”：

java代码：

```
@Controller("/admin/goodsCodeAction")
@Scope("prototype")
public class GoodsCodeAction extends BaseAction {
    @Autowired @Required
    public void setGoodsCodeService(IGoodsCodeService goodsCodeService) {
        this.goodsCodeService = goodsCodeService;
    }
    @Autowired @Required
    public void setGoodsService(IGoodsService goodsService) {
        this.goodsService = goodsService;
    }
}
```

- **setGoodsCodeService方法：**用于注入IGoodsCodeService实现；
- **setGoodsService方法：**用于注入IGoodsService实现。

3、修改表现层前台模块的cn.javass.point.web.front.action.GoodsAction，在该类上添加@Controller注解来进行表现层Bean定义，且作用域为“prototype”：

java代码：

```
@Controller("/front/goodsAction")
@Scope("prototype")
public class GoodsAction extends BaseAction {
    @Autowired @Required
    public void setGoodsService(IGoodsService goodsService) {
        this.goodsService = goodsService;
    }
    @Autowired @Required
    public void setGoodsCodeService(IGoodsCodeService goodsCodeService) {
        this.goodsCodeService = goodsCodeService;
    }
}
```

- **setGoodsCodeService方法：**用于注入IGoodsCodeService实现；
- **setGoodsService方法：**用于注入IGoodsService实现。

12.5.6 其他变化

1、定义一个基于Java方法的配置类，用于加载XML配置文件：

java代码：

```
package cn.javass.point;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.ImportResource;
@Configuration
@ImportResource(
        {"classpath:applicationContext-resources.xml",
```

```
        "classpath:cn/javass/point/dao/applicationContext-hibernate.xml",
        "classpath:cn/javass/point/service/applicationContext-service.xml"
    })
public class AppConfig {
}
```

该类用于加载零配置中一般不变的XML配置文件，如事务管理，数据源、SessionFactory，这些在几乎所有项目中都是类似的，因此可以作为通用配置。

2、修改集成其它Web框架的通用配置，将如下配置：

java代码：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:applicationContext-resources.xml,
        classpath:cn/javass/point/dao/applicationContext-hibernate.xml,
        classpath:cn/javass/point/service/applicationContext-service.xml,
        classpath:cn/javass/point/web/pointShop-admin-servlet.xml,
        classpath:cn/javass/point/web/pointShop-front-servlet.xml
    </param-value>
</context-param>
```

修改为如下配置：

java代码：

```
<context-param>
  <param-name>contextClass</param-name>
  <param-value>
       org.springframework.web.context.support.AnnotationConfigWebApplicationContext
  </param-value>
</context-param>
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>cn.javass.point</param-value>
</context-param>
```

- **contextClass**：使用notationConfigWebApplicationContext替换默认的XmlWebApplicationContext；
- **contextConfigLocation**：指定为“cn.javass.point”，表示将通过扫描该类路径“cn.javass.point”下的注解类来进行加载Bean定义。

启动pointShop2项目，在浏览器输入http://localhost:8080/pointShop2/admin/goods/list.action访问积分商城后台，如果没问题说明零配置整合成功。

到此零配置方式实现SSH集成已经整合完毕，相对于基于XML方式主要减少了配置的数量和配置文件的数量。

1.25 【第十三章】 测试 之 13.1 概述 13.2 单元测试 ——跟我学spring3

发表时间: 2012-03-28 关键字: spring, unit test

13.1 概述

13.1.1 测试

软件测试的目的首先是为了保证软件功能的正确性，其次是为了保证软件的质量，软件测试相当复杂，已经超出本书所涉及的范围，本节将只介绍软件测试流程中前两个步骤：单元测试和集成测试。

Spring提供了专门的测试模块用于简化单元测试和集成测试，单元测试和集成测试一般由程序员实现。

13.2 单元测试

13.2.1 概述

单元测试是最细粒度的测试，即具有原子性，通常测试的是某个功能（如测试类中的某个方法的功能）。

采用依赖注入后我们的代码对Spring IoC容器几乎没有任何依赖，因此在对我们代码进行测试时无需依赖Spring IoC容器，我们只需要通过简单的实例化对象、注入依赖然后测试相应方法来测试该方法是否完成我们预期的功能。

在本书中使用的传统开发流程，即先编写代码实现功能，然后再写测试来验证功能是否正确，而不是测试驱动开发，测试驱动开发是指在编写代码实现功能之前先写测试，然后再根据测试来写满足测试要求的功能代码，通过测试来驱动开发，如果对测试驱动开发感兴趣推荐阅读【测试驱动开发的艺术】。

在实际工作中，应该只对一些复杂的功能进行单元测试，对于一些简单的功能（如数据访问层的CRUD）没有必要花费时间进行单元测试。

Spring对单元测试提供如下支持：

- **Mock对象**：Spring通过Mock对象来简化一些场景的单元测试：

JNDI测试支持：在org.springframework.mock.jndi包下通过了SimpleNamingContextBuilder来来创建JNDI上下文Mock对象，从而无需依赖特定Java EE容器即可完成JNDI测试。

web测试支持：在org.springframework.mock.web包中提供了一组Servlet API的Mock对象，从而可以无需Web容器即可测试web层的类。

- **工具类**：通过通用的工具类来简化编写测试代码：

反射工具类：在org.springframework.test.util包下的ReflectionTestUtils能通过反射完成类的非public字段或setter方法的调用；

JDBC工具类：在org.springframework.test.util包下的SimpleJdbcTestUtils能读取一个sql脚本文件并执行来简化SQL的执行，还提供了如清空表、统计表中行数的简便方法来简化测试代码的编写。

接下来让我们学习一下开发过程中各层代码如何编写测试用例。

13.2.2 准备测试环境

1、Junit安装：将Junit 4包添加到“pointShop”项目中，具体方法请参照【2.2.3 Hello World】。

2、jMock安装：到jMock官网【http://www.jmock.org/】下载最新的jMock包，在本书中使用jMock2.5.1版本，将下载的“jmock-2.5.1-jars.zip ”包中的如下jar包拷贝到项目的lib目录下并添加到类路径：

objenesis-1.0.jar

jmock-script-2.5.1.jar

jmock-legacy-2.5.1.jar

jmock-junit4-2.5.1.jar

jmock-junit3-2.5.1.jar

jmock-2.5.1.jar

hamcrest-library-1.1.jar

hamcrest-core-1.1.jar

bsh-core-2.0b4.jar

注：cglib包无需添加到类路径，因为我们之前已经提供。

3、添加Spring测试支持包：将下载的spring-framework-3.0.5.RELEASE-with-docs.zip包中的如下jar包拷贝到项目的lib目录下并添加到类路径：

dist\org.springframework.test-3.0.5.RELEASE.jar

4、在“pointShop”项目下新建test文件夹并将其添加到【Java Build Path】中，该文件夹用于存放测试代码，从而分离测试代码和开发代码。

到此测试环境搭建完毕。

13.2.3 数据访问层

数据访问层单元测试，目的是测试该层定义的接口实现方法的行为是否正确，其实本质是测试是否正确与数据库交互，是否发送并执行了正确的SQL，SQL执行成功后是否正确的组装了业务逻辑层需要的数据。

数据访问层单元测试通过Mock对象与数据库交互的API来完成测试。

接下来让我们学习一下如何进行数据访问层单元测试：

1、在test文件夹下创建如下测试类：

java代码：

```
package cn.javass.point.dao.hibernate;
//省略import
public class GoodsHibernateDaoUnitTest {
    //1、Mock对象上下文，用于创建Mock对象
    private final Mockery context = new Mockery() {{
        //1.1、表示可以支持Mock非接口类，默认只支持Mock接口
        setImposteriser(ClassImposteriser.INSTANCE);
    }};
    //2、Mock HibernateTemplate类
    private final HibernateTemplate mockHibernateTemplate = context.mock(HibernateTemplate.cla
    private IGoodsDao goodsDao = null;

    @Before
    public void setUp() {
        //3、创建IGoodsDao实现
        GoodsHibernateDao goodsDaoTemp = new GoodsHibernateDao();
        //4、通过ReflectionTestUtils注入需要的非public字段数据
        ReflectionTestUtils.setField(goodsDaoTemp, "entityClass", GoodsModel.class);
        //5、注入mockHibernateTemplate对象
        goodsDaoTemp.setHibernateTemplate(mockHibernateTemplate);
        //6、赋值给我们要使用的接口
        goodsDao = goodsDaoTemp;
}
}
```

- Mockery：jMock核心类，用于创建Mock对象的，通过其mock方法来创建相应接口或类的Mock对象。
- goodsDaoTemp：需要测试的IGoodsDao实现，通过ReflectionTestUtils注入需要的非public字段数据。

2、测试支持写完后，接下来测试一下IGoodsDao的get方法是否满足需求：

java代码：

```
@Test
public void testSave () {
    //7、创建需要的Model数据
    final GoodsModel expected = new GoodsModel();
    //8、定义预期行为，并在后边来验证预期行为是否正确
    context.checking(new org.jmock.Expectations() {
        {
            //9、表示需要调用且只调用一次mockHibernateTemplate的get方法，
            //且get方法参数为(GoodsModel.class, 1)，并将返回goods
            one(mockHibernateTemplate).get(GoodsModel.class, 1);
            will(returnValue(expected));
         }
    });
    //10、调用goodsDao的get方法，在内部实现中将委托给
    //getHibernateTemplate().get(this.entityClass, id);
    //因此按照第8步定义的预期行为将返回goods
    GoodsModel actual = goodsDao.get(1);
    //11、来验证第8步定义的预期行为是否调用了
   context.assertIsSatisfied();
    //12、验证goodsDao.get(1)返回结果是否正确
    Assert.assertEquals(goods, expected);
}
```

- **context.checking()**：该方法中用于定义预期行为，其中第9步定义了需要调用一次且只调用一次mockHibernateTemplate的get方法，且get方法参数为(GoodsModel.class, 1)，并将返回goods对象。
- **goodsDao.get(1)**：调用goodsDao的get方法，在内部实现中将委托给“getHibernateTemplate().get(this.entityClass, id)”。
- **context.assertIsSatisfied()**：来验证前边定义的预期行为是否执行，且是否正确。
- **Assert.assertEquals(expected, actual)**：用于验证“goodsDao.get(1)”返回的结果是否是预期结果。

以上测试方法其实是没有必要的，对于非常简单的CRUD没有必要写单元测试，只有相当复杂的方法才有必要写单元测试。

这种通过Mock对象来测试数据访问层代码其实一点意义没有，因为这里没有与数据库交互，无法验证真实环境中与数据库交互是否正确，因此这里只是告诉你如何测试数据访问层代码，在实际工作中一般通过集成测试来完成数据访问层测试。

13.2.4 业务逻辑层

业务逻辑单元测试，目的是测试该层的业务逻辑是否正确并通过Mock 数据访问层对象来隔离与数据库交互，从而无需连接数据库即可测试业务逻辑是否正确。

接下来让我们学习一下如何进行业务逻辑层单元测试：

1、在test文件夹下创建如下测试类：

java代码：

```
package cn.javass.point.service.impl;
//省略import
public class GoodsCodeServiceImplUnitTest {
    //1、Mock对象上下文，用于创建Mock对象
    private final Mockery context = new Mockery() {{
        //1.1、表示可以支持Mock非接口类，默认只支持Mock接口
        setImposteriser(ClassImposteriser.INSTANCE);
    }};

    //2、Mock IGoodsCodeDao接口
    private IGoodsCodeDao goodsCodeDao = context.mock(IGoodsCodeDao.class);;

    private IGoodsCodeService goodsCodeService;
```

```
    @Before
    public void setUp() {
        GoodsCodeServiceImpl goodsCodeServiceTemp = new GoodsCodeServiceImpl();
        //3、依赖注入
        goodsCodeServiceTemp.setDao(goodsCodeDao);
        goodsCodeService = goodsCodeServiceTemp;
}
}
```

以上测试支持代码和数据访问层测试代码非常类似，在此不再阐述。

2、测试支持写完后，接下来测试一下购买商品Code码是否满足需求：

测试业务逻辑时需要分别测试多种场景，即如在某种场景下成功或失败等等，即测试应该全面，每个功能点都应该测试到。

2.1、测试购买失败的场景：

java代码：

```
@Test(expected = NotCodeException.class)
public void testBuyFail() {
    final int goodsId = 1;
    //4、定义预期行为，并在后边来验证预期行为是否正确
    context.checking(new org.jmock.Expectations() {
      {
          //5、表示需要调用goodsCodeDao对象的getOneNotExchanged一次且仅以此
          //且返回值为null
          one(goodsCodeDao).getOneNotExchanged(goodsId);
          will(returnValue(null));
```

```
      }
    });
    goodsCodeService.buy("test", goodsId);
    context.assertIsSatisfied();
}
```

- **context.checking()**：该方法中用于定义预期行为，其中第5步定义了需要调用一次且只调用一次goodsCodeDao的getOneNotExchanged方法，且getOneNotExchanged方法参数为(goodsId)，并将返回null。
- **goodsCodeService.buy("test", goodsId)**：调用goodsCodeService的buy方法，由于调用goodsCodeDao的getOneNotExchanged方法将返回null，因此buy方法将抛出“NotCodeException”异常，从而表示没有Code码。
- **context.assertIsSatisfied()**：来验证前边定义的预期行为是否执行，且是否正确。
- 由于我们在预期行为中调用getOneNotExchanged将返回null，因此测试将失败且抛出NotCodeException异常。

2.2、测试购买成功的场景：

java代码：

```
@Test()
public void testBuySuccess () {
    final int goodsId = 1;
    final GoodsCodeModel goodsCode = new GoodsCodeModel();
    //6、定义预期行为，并在后边来验证预期行为是否正确
    context.checking(new org.jmock.Expectations() {
      {
          //7、表示需要调用goodsCodeDao对象的getOneNotExchanged一次且仅以此
          //且返回值为null
          one(goodsCodeDao).getOneNotExchanged(goodsId);
          will(returnValue(goodsCode));
          //8、表示需要调用goodsCodeDao对象的save方法一次且仅一次
          //且参数为goodsCode
```

```
            one(goodsCodeDao).save(goodsCode);
          }
        });
    goodsCodeService.buy("test", goodsId);
    context.assertIsSatisfied();
    Assert.assertEquals(goodsCode.isExchanged(), true);
}
```

- **context.checking()**：该方法中用于定义预期行为，其中第7步定义了需要调用一次且只调用一次goodsCodeDao的getOneNotExchanged方法，且getOneNotExchanged方法参数为(goodsId)，并将返回goodsCode对象；第8步定义了需要调用goodsCodeDao对象的save一次且仅一次。
- **goodsCodeService.buy("test", goodsId)**：调用goodsCodeService的buy方法，由于调用goodsCodeDao的getOneNotExchanged方法将返回goodsCode，因此buy方法将成功执行。
- **context.assertIsSatisfied()**：来验证前边定义的预期行为是否执行，且是否正确。
- **Assert.assertEquals(goodsCode.isExchanged(), true)**：表示goodsCode已经被购买过了。
- 由于我们在预期行为中调用getOneNotExchanged将返回一个goodsCode对象，因此测试将成功，如果失败说明业务逻辑写错了。

到此业务逻辑层测试完毕，在进行业务逻辑层测试时我们只关心业务逻辑是否正确，而不关心底层数据访问层如何实现，因此测试业务逻辑层时只需Mock 数据访问层对象，然后定义一些预期行为来满足业务逻辑测试需求即可。

13.2.5 表现层

表现层测试包括如Struts2的Action单元测试、拦截器单元测试、JSP单元测试等等，在此我们只学习Struts2的Action单元测试。

Struts2的Action测试相对业务逻辑层测试相对复杂一些，因为牵扯到使用如Servlet API、ActionContext等等，这里需要通过stub（桩）实现或mock对象来模拟如HttpServletRequest等对象。

一、首先学习一些最简单的Action测试：

1、在test文件夹下创建如下测试类：

java代码：

```
package cn.javass.point.web.front;
import cn.javass.point.service.IGoodsCodeService;
import cn.javass.point.web.front.action.GoodsAction;
//省略部分import
public class GoodsActionUnitTest {
    //1、Mock对象上下文，用于创建Mock对象
    private final Mockery context = new Mockery() {{
        //1.1、表示可以支持Mock非接口类，默认只支持Mock接口
        setImposteriser(ClassImposteriser.INSTANCE);
    }};

    //2、Mock IGoodsCodeService接口
    private IGoodsCodeService goodsCodeService = context.mock(IGoodsCodeService.class);

    private GoodsAction goodsAction;

    @Before
    public void setUp() {
        goodsAction = new GoodsAction();
        //3、依赖注入
        goodsAction.setGoodsCodeService(goodsCodeService);
}
}
```

以上测试支持代码和业务逻辑层测试代码非常类似，在此不再阐述。

2、测试支持写完后，接下来测试一下前台购买商品Code码是否满足需求：

类似于测试业务逻辑时需要分别测试多种场景，测试Action同样需要分别测试多种场景。

2.1、测试购买失败的场景：

java代码：

```
@Test
public void testBuyFail() {
    final int goodsId = 1;
    //4、定义预期行为，并在后边来验证预期行为是否正确
    context.checking(new org.jmock.Expectations() {
      {
          //5、表示需要调用goodsCodeService对象的buy方法一次且仅一次
          //且抛出NotCodeException异常
          one(goodsCodeService).buy("test", goodsId);
      will(throwException(new NotCodeException()));
      }
    });
    //6、模拟Struts注入请求参数
    goodsAction.setGoodsId(goodsId);
    String actualResultCode = goodsAction.buy();
    context.assertIsSatisfied();
    Assert.assertEquals(ReflectionTestUtils.getField(goodsAction, "BUY_RESULT"), actualResultC
    Assert.assertTrue(goodsAction.getActionErrors().size() > 0);
}
```

- **context.checking()**：该方法中用于定义预期行为，其中第5步定义了需要调用goodsCodeService对象的buy方法一次且仅一次且将抛出NotCodeException异常。
- **goodsAction.setGoodsId(goodsId)**：用于模拟Struts注入请求参数，即完成数据绑定。
- **goodsAction.buy()**：调用goodsAction的buy方法，该方法将委托给IGoodsCodeService实现完成，返回值用于定位视图。
- **context.assertIsSatisfied()**：来验证前边定义的预期行为是否执行，且是否正确。
- **Assert.assertEquals(ReflectionTestUtils.getField(goodsAction, "BUY_RESULT"), actualResultCode)**：验证返回的Result是否是我们指定的。
- **Assert.assertTrue(goodsAction.getActionErrors().size() > 0)**：表示执行Action时有错误，即Action动作错误。如果条件不成立，说明我们Action功能是错误的，需要修改。

2.2、测试购买成功的场景：

java代码：

```
@Test
public void testBuySuccess() {
    final int goodsId = 1;
    final GoodsCodeModel goodsCode = new GoodsCodeModel();
    //7、定义预期行为，并在后边来验证预期行为是否正确
    context.checking(new org.jmock.Expectations() {
      {
          //8、表示需要调用goodsCodeService对象的buy方法一次且仅一次
          //且返回goodsCode对象
          one(goodsCodeService).buy("test", goodsId);
          will(returnValue(goodsCode));
      }
    });
    //9、模拟Struts注入请求参数
    goodsAction.setGoodsId(goodsId);
    String actualResultCode = goodsAction.buy();
    context.assertIsSatisfied();
    Assert.assertEquals(ReflectionTestUtils.getField(goodsAction, "BUY_RESULT"), actualResultC
    Assert.assertTrue(goodsAction.getActionErrors().size() == 0);
}
```

- **context.checking()**：该方法中用于定义预期行为，其中第5步定义了需要调用goodsCodeService对象的buy方法一次且仅一次且将返回goodsCode对象。
- **goodsAction.setGoodsId(goodsId)**：用于模拟Struts注入请求参数，即完成数据绑定。
- **goodsAction.buy()**：调用goodsAction的buy方法，该方法将委托给IGoodsCodeService实现完成，返回值用于定位视图。
- **context.assertIsSatisfied()**：来验证前边定义的预期行为是否执行，且是否正确。
- **Assert.assertEquals(ReflectionTestUtils.getField(goodsAction, "BUY_RESULT"), actualResultCode)**：验证返回的Result是否是我们指定的。

- **Assert.assertTrue(goodsAction.getActionErrors().size() == 0)**：表示执行Action时没有错误，即Action动作正确。如果条件不成立，说明我们Action功能是错误的，需要修改。

通过模拟ActionContext对象内容从而可以非常容易的测试Action中各种与http请求相关情况，无需依赖web服务器即可完成测试。但对于如果我们使用htpp请求相关对象的该如何测试？如果我们需要使用ActionContext获取值栈数据应该怎么办？这就需要Struts提供的junit插件支持了。我们会在集成测试中介绍。

对于表现层其他功能的单元测试本书不再介绍，如JSP单元测试、拦截器单元测试等等。

1.26 【第十三章】 测试 之 13.3 集成测试 ——跟我学spring3

发表时间: 2012-03-30 关键字: spring, 集成测试

13.3 集成测试

13.3.1 概述

集成测试是在单元测试之上，通常是将一个或多个已进行过单元测试的组件组合起来完成的，即集成测试中一般不会出现Mock对象，都是实实在在的真实实现。

对于单元测试，如前边在进行数据访问层单元测试时，通过Mock HibernateTemplate对象然后将其注入到相应的DAO实现，此时单元测试只测试某层的某个功能是否正确，对其他层如何提供服务采用Mock方式提供。

对于集成测试，如要进行数据访问层集成测试时，需要实实在在的HibernateTemplate对象然后将其注入到相应的DAO实现，此时集成测试将不仅测试该层功能是否正确，还将测试服务提供者提供的服务是否正确执行。

使用Spring的一个好处是能非常简单的进行集成测试，无需依赖web服务器或应用服务器即可完成测试。Spring通过提供一套TestContext框架来简化集成测试，使用TestContext测试框架能获得许多好处，如Spring IoC容器缓存、事务管理、依赖注入、Spring测试支持类等等。

13.3.2 Spring TestContext框架支持

Spring TestContext框架提供了一些通用的集成测试支持，主要提供如下支持：

一、**上下文管理及缓存：**

对于每一个测试用例（测试类）应该只有一个上下文，而不是每个测试方法都创建新的上下文，这样有助于减少启动容器的开销，提供测试效率。可通过如下方式指定要加载的上下文：

java代码：

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(
    locations={"classpath:applicationContext-resources-test.xml",
                    "classpath:cn/javass/point/dao/applicationContext-hibernate.xml"})
public class GoodsHibernateDaoIntegrationTest {
}
```

- **locations**：指定Spring配置文件位置；
- **inheritLocations**：如果设置为false，将屏蔽掉父类中使用该注解指定的配置文件位置，默认为true表示继承父类中使用该注解指定的配置文件位置。

二、**Test Fixture（测试固件）的依赖注入：**

Test Fixture可以指运行测试时需要的任何东西，一般通过@Before定义的初始化Fixture方法准备这些资源，而通过@After定义的销毁Fixture方法销毁或还原这些资源。

Test Fixture的依赖注入就是使用Spring IoC容器的注入功能准备和销毁这些资源。可通过如下方式注入Test Fixture：

java代码：

```
@Autowired
private IGoodsDao goodsDao;
@Autowired
private ApplicationContext ctx;
```

即可以通过Spring提供的注解实现Bean的依赖注入来完成Test Fixture的依赖注入。

三、事务管理：

开启测试类的事务管理支持，即使用Spring 容器的事务管理功能，从而可以独立于应用服务器完成事务相关功能的测试。为了使测试中的事务管理起作用需要通过如下方式开启测试类事务的支持：

java代码：

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(
    locations={"classpath:applicationContext-resources-test.xml",
            "classpath:cn/javass/point/dao/applicationContext-hibernate.xml"})
@TransactionConfiguration(
transactionManager = "txManager", defaultRollback=true)
public class GoodsHibernateDaoIntegrationTest {
}
```

Spring提供如下事务相关注解来支持事务管理：

- **@Transactional**：使用@Transactional注解的类或方法将得到事务支持
- **transactionManager:**指定事务管理器；
- **defaultRollback**：是否回滚事务，默认为true表示回滚事务。

Spring还通过提供如下注解来简化事务测试：

- **@Transactional**：使用@Transactional注解的类或方法表示需要事务支持；
- **@NotTransactional**：只能注解方法，使用@NotTransactional注解的方法表示不需要事务支持，即不运行在事务中，Spring 3开始已不推荐使用；
- **@BeforeTransaction和@AfterTransaction**：使用这两个注解注解的方法定义了在一个事务性测试方法之前或之后执行的行为，且被注解的方法将运行在该事务性方法的事务之外。
- **@Rollback(true)**：默认为true，用于替换@TransactionConfiguration中定义的defaultRollback指定的回滚行为。

四、常用注解支持：Spring框架提供如下注解来简化集成测试：

- **@DirtiesContext**：表示每个测试方法执行完毕需关闭当前上下文并重建一个全新的上下文，即不缓存上下文。可应用到类或方法级别，但在JUnit 3.8中只能应用到方法级别。

- **@ExpectedException：**表示被注解的方法预期将抛出一个异常，使用如@ExpectedException(NotCodeException.class)来指定异常，定义方式类似于Junit 4中的@Test(expected = NotCodeException.class)，@ExpectedException注解和@Test(expected =……)应该两者选一。
- **@Repeat：表示被注解的方法应被重复执行多少次，使用如@Repeat(2)方式指定。**
- **@Timed：**表示被注解的方法必须在多长时间内运行完毕，超时将抛出异常，使用如@Timed(millis=10)方式指定，单位为毫秒。注意此处指定的时间是如下方法执行时间之和：测试方法执行时间（或者任何测试方法重复执行时间之和）、@Before和@After注解的测试方法之前和之后执行的方法执行时间。而Junit 4中的@Test(timeout=2)指定的超时时间只是测试方法执行时间，不包括任何重复等。
- **除了支持如上注解外，还支持【第十二章 零配置】中依赖注入等注解。**

五、TestContext框架支持类：提供对测试框架的支持，如Junit、TestNG测试框架，用于集成Spring TestContext和测试框架来简化测试，TestContext框架提供如下支持类：

- **JUnit 3.8支持类：**提供对Spring TestContext框架与Junit3.8测试框架的集成：

AbstractJUnit38SpringContextTests：我们的测试类继承该类后将获取到Test Fixture的依赖注入好处。

AbstractTransactionalJUnit38SpringContextTests：我们的测试类继承该类后除了能得到Test Fixture的依赖注入好处，还额外获取到事务管理支持。

- **JUnit 4.5+支持类：**提供对Spring TestContext框架与Junit4.5+测试框架的集成：

AbstractJUnit4SpringContextTests：我们的测试类继承该类后将获取到Test Fixture的依赖注入好处。

AbstractTransactionalJUnit4SpringContextTests：我们的测试类继承该类后除了能得到Test Fixture的依赖注入好处，还额外获取到事务管理支持。

- **定制 Junit4.5+运行器：**通过定制自己的Junit4.5+运行器从而无需继承JUnit 4.5+支持类即可完成需要的功能，如Test Fixture的依赖注入、事务管理支持，

@RunWith(SpringJUnit4ClassRunner.class)：使用该注解注解到测试类上表示将集成Spring TestContext和Junit 4.5+测试框架。

@TestExecutionListeners：该注解用于指定TestContext框架的监听器用于与TestContext框架管理器发布的测试执行事件进行交互，TestContext框架提供如下三个默认的监听器：DependencyInjectionTestExecutionListener、DirtiesContextTestExecutionListener、TransactionalTestExecutionListener分别完成对Test Fixture的依赖注入、@DirtiesContext支持和事务管理支持，即在默认情况下将自动注册这三个监听器，另外还可以使用如下方式指定监听器：

java代码：

```
@RunWith(SpringJUnit4ClassRunner.class)
@TestExecutionListeners({})
public class GoodsHibernateDaoIntegrationTest {
}
```

如上配置将通过定制的Junit4.5+运行器运行，但不会完成Test Fixture的依赖注入、事务管理等等，如果只需要Test Fixture的依赖注入，可以使用@TestExecutionListeners({DependencyInjectionTestExecutionListener.class})指定。

- **TestNG支持类：**提供对Spring TestContext框架与TestNG测试框架的集成：

AbstractTestNGSpringContextTests：我们的测试类继承该类后将获取到Test Fixture的依赖注入好处。

AbstractTransactionalTestNGSpringContextTests：我们的测试类继承该类后除了能得到Test Fixture的依赖注入好处，还额外获取到事务管理支持。

到此Spring TestContext测试框架减少完毕了，接下来让我们学习一下如何进行集成测试吧。

13.3.3 准备集成测试环境

对于集成测试环境各种配置应该和开发环境或实际生产环境配置相分离，即集成测试时应该使用单独搭建一套独立的测试环境，不应使用开发环境或实际生产环境的配置，从而保证测试环境、开发、生产环境相分离。

1、拷贝一份Spring资源配置文件applicationContext-resources.xml，并命名为applicationContext-resources-test.xml表示用于集成测试使用，并修改如下内容：

java代码：

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="locations">
```

```
        <list>
            <value>classpath:resources-test.properties</value>
        </list>
  </property>
</bean>
```

2、拷贝一份替换配置元数据的资源文件（resources/resources.properties），并命名为resources-test.properties表示用于集成测试使用，并修改为以下内容：

java代码：

```
db.driver.class=org.hsqldb.jdbcDriver
db.url=jdbc:hsqldb:mem:point_shop
db.username=sa
db.password=
#Hibernate属性
hibernate.dialect=org.hibernate.dialect.HSQLDialect
hibernate.hbm2ddl.auto=create-drop
hibernate.show_sql=false
hibernate.format_sql=true
```

- **jdbc:hsqldb:mem:point_shop**：我们在集成测试时将使用HSQLDB，并采用内存数据库模式运行；
- **hibernate.hbm2ddl.auto=create-drop**：表示在创建SessionFactory时根据Hibernate映射配置创建相应Model的表结构，并在SessionFactory关闭时删除这些表结构。

到此我们测试环境修改完毕，在进行集成测试时一定要保证测试环境、开发环境、实际生产环境相分离，即对于不同的环境使用不同的配置文件。

13.3.4 数据访问层

数据访问层集成测试，同单元测试一样目的不仅测试该层定义的接口实现方法的行为是否正确，而且还要测试是否正确与数据库交互，是否发送并执行了正确的SQL，SQL执行成功后是否正确的组装了业务逻辑层需要的数据。

数据访问层集成测试不再通过Mock对象与数据库交互的API来完成测试，而是使用实实在在存在的与数据库交互的对象来完成测试。

接下来让我们学习一下如何进行数据访问层集成测试：

1、在test文件夹下创建如下测试类：

java代码：

```
package cn.javass.point.dao.hibernate;
//省略import
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(
    locations={"classpath:applicationContext-resources-test.xml",
                    "classpath:cn/javass/point/dao/applicationContext-hibernate.xml"})
@TransactionConfiguration(transactionManager = "txManager", defaultRollback=false)
public class GoodsHibernateDaoIntegrationTest {
    @Autowired
    private ApplicationContext ctx;
    @Autowired
    private IGoodsCodeDao goodsCodeDao;
}
```

- **@RunWith(SpringJUnit4ClassRunner.class)**：表示使用自己定制的Junit4.5+运行器来运行测试，即完成Spring TestContext框架与Junit集成；
- **@ContextConfiguration**：指定要加载的Spring配置文件，此处注意我们的Spring资源配置文件为“applicationContext-resources-test.xml” ；
- **@TransactionConfiguration**：开启测试类的事务管理支持配置，并指定事务管理器和默认回滚行为；
- **@Autowired**：完成Test Fixture（测试固件）的依赖注入。

2、测试支持写完后，接下来测试一下分页查询所有已发布的商品是否满足需求：

java代码：

```
@Transactional
@Rollback
@Test
public void testListAllPublishedSuccess() {
    GoodsModel goods = new GoodsModel();
    goods.setDeleted(false);
    goods.setDescription("");
    goods.setName("测试商品");
    goods.setPublished(true);
    goodsDao.save(goods);
    Assert.assertTrue(goodsDao.listAllPublished(1).size() == 1);
    Assert.assertTrue(goodsDao.listAllPublished(2).size() == 0);
}
```

- **@Transactional**：表示测试方法将允许在事务环境；
- **@Rollback**：表示替换@ContextConfiguration指定的默认事务回滚行为，即将在测试方法执行完毕时回滚事务。

数据访问层的集成测试也是非常简单，与数据访问层的单元测试类似，也应该只对复杂的数据访问层代码进行测试。

13.3.5 业务逻辑层

业务逻辑层集成测试，目的同样是测试该层的业务逻辑是否正确，对于数据访问层实现通过Spring IoC容器完成装配，即使用真实的数据访问层实现来获取相应的底层数据。

接下来让我们学习一下如何进行业务逻辑层集成测试：

1、在test文件夹下创建如下测试类：

java代码：

```
@ContextConfiguration(
locations={"classpath:applicationContext-resources-test.xml",
            "classpath:cn/javass/point/dao/applicationContext-hibernate.xml",
            "classpath:cn/javass/point/service/applicationContext-service.xml"})
@TransactionConfiguration(transactionManager = "txManager", defaultRollback=false)
public class GoodsCodeServiceImplIntegrationTest extends AbstractJUnit4SpringContextTests {
    @Autowired
    private IGoodsCodeService goodsCodeService;
    @Autowired
    private IGoodsService goodsService;
}
```

- **AbstractJUnit4SpringContextTests**：表示将Spring TestContext框架与Junit4.5+测试框架集成；
- **@ContextConfiguration**：指定要加载的Spring配置文件，此处注意我们的Spring资源配置文件为“applicationContext-resources-test.xml” ；
- **@TransactionConfiguration**：开启测试类的事务管理支持配置，并指定事务管理器和默认回滚行为；
- **@Autowired**：完成Test Fixture（测试固件）的依赖注入。

2、测试支持写完后，接下来测试一下购买商品Code码是否满足需求：

2.1、测试购买失败的场景：

java代码：

```
@Transactional
@Rollback
@ExpectedException(NotCodeException.class)
@Test
public void testBuyFail() {
    goodsCodeService.buy("test", 1);
}
```

由于我们数据库中没有相应商品的Code码，因此将抛出NotCodeException异常。

2.2、测试购买成功的场景：

java代码：

```
@Transactional
@Rollback
@Test
public void testBuySuccess() {
```

```
    //1.添加商品
    GoodsModel goods = new GoodsModel();
    goods.setDeleted(false);
    goods.setDescription("");
    goods.setName("测试商品");
    goods.setPublished(true);
    goodsService.save(goods);

    //2.添加商品Code码
    GoodsCodeModel goodsCode = new GoodsCodeModel();
    goodsCode.setGoods(goods);
    goodsCode.setCode("test");
    goodsCodeService.save(goodsCode);
    //3.测试购买商品Code码
    GoodsCodeModel resultGoodsCode = goodsCodeService.buy("test", 1);
    Assert.assertEquals(goodsCode.getId(), resultGoodsCode.getId());
}
```

由于我们添加了指定商品的Code码因此购买将成功，如果失败说明业务写错了，应该重写。

业务逻辑层的集成测试也是非常简单，与业务逻辑层的单元测试类似，也应该只对复杂的业务逻辑层代码进行测试。

13.3.5 表现层

对于表现层集成测试，同样类似于单元测试，但对于业务逻辑层都将使用真实的实现，而不再是通过Mock对象来测试，这也是集成测试和单元测试的区别。

接下来让我们学习一下如何进行表现层Action集成测试：

1、准备Struts提供的junit插件, 到struts-2.2.1.1.zip中拷贝如下jar包到类路径：

lib\struts2-junit-plugin-2.2.1.1.jar

2、测试支持类：Struts2提供StrutsSpringTestCase测试支持类，我们所有的Action测试类都需要继承该类；

3、准备Spring配置文件：由于我们的测试类继承StrutsSpringTestCase且将通过覆盖该类的getContextLocations方法来指定Spring配置文件，但由于getContextLocations方法只能返回一个配置文件，因此我们需要新建一个用于导入其他Spring配置文件的配置文件applicationContext-test.xml，具体内容如下：

java代码：

```
<import resource="classpath:applicationContext-resources-test.xml"/>
<import resource="classpath:cn/javass/point/dao/applicationContext-hibernate.xml"/>
<import resource="classpath:cn/javass/point/service/applicationContext-service.xml"/>
<import resource="classpath:cn/javass/point/web/pointShop-admin-servlet.xml"/>
<import resource="classpath:cn/javass/point/web/pointShop-front-servlet.xml"/>
```

3、在test文件夹下创建如下测试类：

java代码：

```
package cn.javass.point.web.front;
//省略import
@RunWith(SpringJUnit4ClassRunner.class)
@TestExecutionListeners({})
public class GoodsActionIntegrationTest extends StrutsSpringTestCase {
    @Override
    protected String getContextLocations() {
        return "classpath:applicationContext-test.xml";
    }
    @Before
    public void setUp() throws Exception {
        //1 指定Struts2配置文件
        //该方式等价于通过web.xml中的<init-param>方式指定参数
        Map<String, String> dispatcherInitParams = new HashMap<String, String>();
        ReflectionTestUtils.setField(this, "dispatcherInitParams", dispatcherInitParams);
        //1.1 指定Struts配置文件位置
        dispatcherInitParams.put("config", "struts-default.xml,struts-plugin.xml,struts.xml");
        super.setUp();
    }
    @After
    public void tearDown() throws Exception {
        super.tearDown();
    }
}
```

- **@RunWith(SpringJUnit4ClassRunner.class)**：表示使用自己定制的Junit4.5+运行器来运行测试，即完成Spring TestContext框架与Junit集成；
- **@TestExecutionListeners({})**：没有指定任何监听器，即不会自动完成对Test Fixture的依赖注入、@DirtiesContext支持和事务管理支持；
- **StrutsSpringTestCase**：集成测试Struts2+Spring时所有集成测试类必须继承该类；

- **setUp方法**：在每个测试方法之前都执行的初始化方法，其中dispatcherInitParams用于指定等价于在web.xml中的<init-param>方式指定的参数；必须调用super.setUp()用于初始化Struts2和Spring环境。
- **tearDown()**：在每个测试方法之前都执行的销毁方法，必须调用super.tearDown()来销毁Spring容器等。

4、测试支持写完后，接下来测试一下前台购买商品Code码是否满足需求：

4.1、测试购买失败的场景：

java代码：

```
@Test
public void testBuyFail() throws UnsupportedEncodingException, ServletException {
    //2 前台购买商品失败
    //2.1 首先重置hhtp相关对象，并准备准备请求参数
    initServletMockObjects();
    request.setParameter("goodsId", String.valueOf(Integer.MIN_VALUE));
    //2.2 调用前台GoodsAction的buy方法完成购买相应商品的Code码
    executeAction("/goods/buy.action");
    GoodsAction frontGoodsAction = (GoodsAction) ActionContext.getContext().getActionInvocatio
    //2.3 验证前台GoodsAction的buy方法有错误
    Assert.assertTrue(frontGoodsAction.getActionErrors().size() > 0);
}
```

- **initServletMockObjects()**：用于重置所有http相关对象，如request等；
- **request.setParameter("goodsId", String.valueOf(Integer.MIN_VALUE))**：用于准备请求参数；
- **executeAction("/goods/buy.action")**：通过模拟http请求来调用前台GoodsAction的buy方法完成商品购买
- **Assert.assertTrue(frontGoodsAction.getActionErrors().size() > 0)**：表示执行Action时有错误，即Action动作错误。如果条件不成立，说明我们Action功能是错误的，需要修改。

4.2、测试购买成功的场景：

java代码：

```
@Test
public void testBuySuccess() throws UnsupportedEncodingException, ServletException {
    //3 后台新增商品
    //3.1 准备请求参数
    request.setParameter("goods.name", "测试商品");
    request.setParameter("goods.description", "测试商品描述");
    request.setParameter("goods.originalPoint", "1");
    request.setParameter("goods.nowPoint", "2");
    request.setParameter("goods.published", "true");
    //3.2 调用后台GoodsAction的add方法完成新增
    executeAction("/admin/goods/add.action");
    //2.3 获取GoodsAction的goods属性
    GoodsModel goods = (GoodsModel) findValueAfterExecute("goods");
    //4 后台新增商品Code码
    //4.1 首先重置hhtp相关对象，并准备准备请求参数
    initServletMockObjects();
    request.setParameter("goodsId", String.valueOf(goods.getId()));
    request.setParameter("codes", "a\rb");
    //4.2 调用后台GoodsCodeAction的add方法完成新增商品Code码
    executeAction("/admin/goodsCode/add.action");
    //5 前台购买商品成功
    //5.1 首先重置hhtp相关对象，并准备准备请求参数
    initServletMockObjects();
    request.setParameter("goodsId", String.valueOf(goods.getId()));
    //5.2 调用前台GoodsAction的buy方法完成购买相应商品的Code码
    executeAction("/goods/buy.action");
    GoodsAction frontGoodsAction = (GoodsAction) ActionContext.getContext().getActionInvocatio
    //5.3 验证前台GoodsAction的buy方法没有错误
    Assert.assertTrue(frontGoodsAction.getActionErrors().size() == 0);
}
```

- **executeAction("/admin/goods/add.action")**：调用后台GoodsAction的add方法，用于新增商品；
- **executeAction("/admin/goodsCode/add.action")**：调用后台GoodCodeAction的add方法用于新增商品Code码；
- **executeAction("/goods/buy.action")**：调用前台GoodsAction的buy方法，用于购买相应商品，其中Assert.assertTrue(frontGoodsAction.getActionErrors().size() == 0)表示购买成功，即Action动作正确。

表现层Action集成测试介绍就到此为止，如何深入StrutsSpringTestCase来完成集成测试已超出本书范围，如果读者对这部分感兴趣可以到Struts2官网学习最新的测试技巧。

1.27 我对IoC/DI的理解

发表时间: 2012-03-31 关键字: IoC, DI

IoC

IoC： Inversion of Control，控制反转， 控制权从应用程序转移到框架（如IoC容器），是框架共有特性

1、为什么需要IoC容器

1.1、应用程序主动控制对象的实例化及依赖装配

```
A a = new AImpl();
B b = new BImpl();
a.setB(b);
```

本质：创建对象，主动实例化，直接获取依赖，主动装配

缺点：更换实现需要重新编译源代码

很难更换实现、难于测试

耦合实例生产者和实例消费者

```
A a = AFactory.createA();
B b = BFactory.createB();
a.setB(b);
```

本质：创建对象，被动实例化，间接获取依赖，主动装配 （简单工厂）

缺点：更换实现需要重新编译源代码

很难更换实现、难于测试

```
A a = Factory.create( “a” );
B b = Factory.create( “b” );
a.setB(b);
```

<!—配置.properties-->

```
a=AImpl
b=BImpl
```

本质：创建对象，被动实例化，间接获取依赖， 主动装配
（工厂+反射+properties配置文件、
Service Locator、注册表）
缺点：冗余的依赖装配逻辑

我想直接：
//返回装配好的a

```
A a = Factory.create( “a” );
```

1.2、可配置通用工厂：**工厂主动控制**，应用程序被动接受，控制权从应用程序转移到工厂

```
//返回装配好的a
A a = Factory.create( “a” );
```

<!—配置文件-->

```
<bean id= “a”  class= “AImpl” >
    <property name= “b”  ref= “b” />
</bean>
<bean id= “b”  class= “BImpl” />
```

本质：创建对象和装配对象，
被动实例化，被动接受依赖，被动装配
（工厂+反射+xml配置文件）

缺点：不通用

步骤：

1、读取配置文件根据配置文件通过反射

创建AImpl

2、发现A需要一个类型为B的属性b

3、到工厂中找名为b的对象，发现没有，读取

配置文件通过反射创建BImpl

4、将b对象装配到a对象的b属性上

【组件的配置与使用分离开（解耦、更改实现无需修改源代码、易于更好实现）】

1.3、 IoC(控制反转)容器：**容器主动控制**

```
//返回装配好的a
A a = ApplicationContext.getBean( “a” );
```

<!—配置文件-->

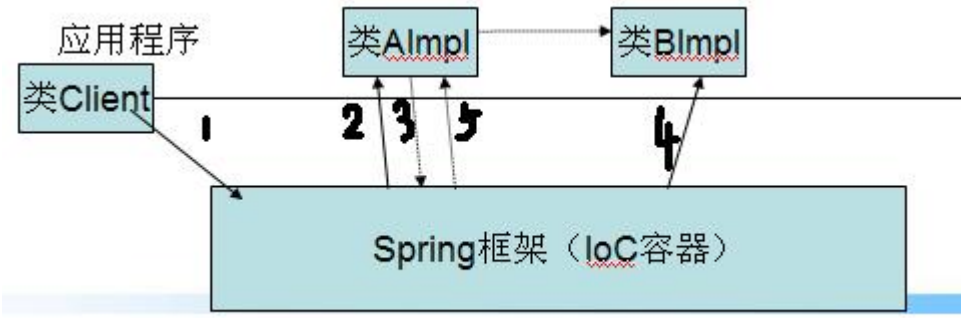
```
<bean id= “a”  class= “AImpl” >
    <property name= “b”  ref= “b” />
</bean>
<bean id= “b”  class= “BImpl” />
```

本质：创建对象和装配对象、管理对象生命周期

被动实例化，被动接受依赖，被动装配

（工厂+反射+xml配置文件）

通用



IoC容器：实现了IoC思想的容器就是IoC容器

2、IoC容器特点

【1】无需主动new对象；而是描述对象应该如何被创建即可

IoC容器帮你创建，即被动实例化；

【2】不需要主动装配对象之间的依赖关系，而是描述需要哪个服务（组件），

IoC容器会帮你装配（即负责将它们关联在一起），被动接受装配；

【3】主动变被动，好莱坞法则：别打电话给我们，我们会打给你；

【4】迪米特法则（最少知识原则）：不知道依赖的具体实现，只知道需要提供某类服务的对象（面向抽象编程），松散耦合，一个对象应当对其他对象有尽可能少的了解,不和陌生人（实现）说话

【5】IoC是一种让服务消费者不直接依赖于服务提供者的组件设计方式，是一种减少类与类之间依赖的设计原则。

3、理解IoC容器问题关键：控制的哪些方面被反转了？

1、谁控制谁？为什么叫反转？ ------ **IoC容器控制，而以前是应用程序控制，所以叫反转**

2、控制什么？ ------ **控制应用程序所需要的资源（对象、文件……）**

3、为什么控制？ ------ **解耦组件之间的关系**

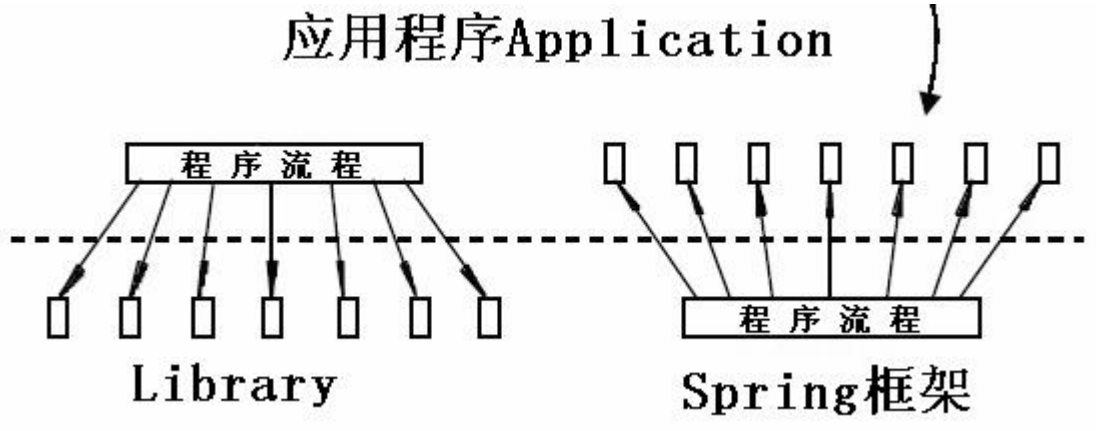
4、控制的哪些方面被反转了？ ------ **程序的控制权发生了反转：从应用程序转移到了IoC容器。**

思考：

1： IoC/DI等同于工厂吗？

2： IoC/DI跟以前的方式有什么不一样？

领会：**主从换位的思想**



4、实现了IoC思想的容器就是轻量级容器吗？

如果仅仅因为使用了控制反转就认为这些轻量级容器与众不同，就好象在说我的轿车与众不同因为它有四个轮子？

容器：提供组件运行环境，管理组件声明周期（不管组件如何创建的以及组件之间关系如何装配的）；

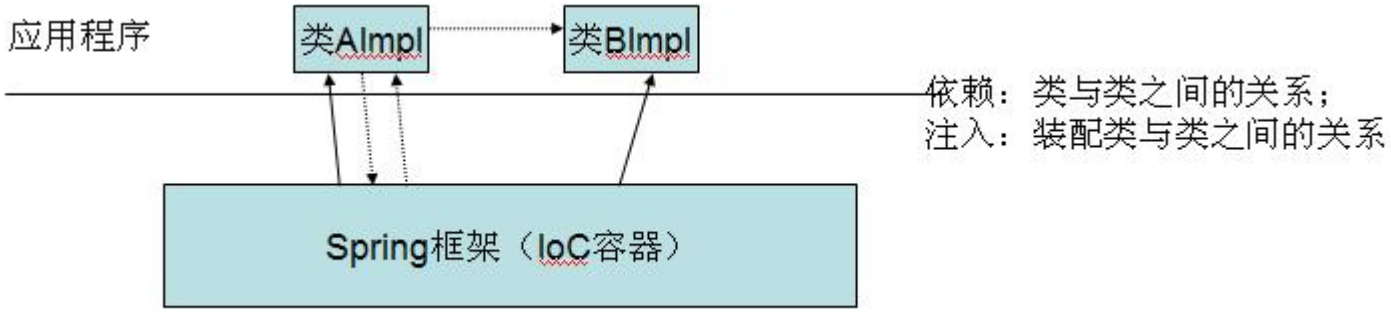
IoC容器不仅仅具有容器的功能，而且还具有一些其他特性---如依赖装配

控制反转概念太广泛，让人迷惑，后来Martin Fowler 提出依赖注入概念
Martin Fowler Inversion of Control Containers and the Dependency Injection pattern
http://martinfowler.com/articles/injection.html

DI

2、什么是DI

DI：依赖注入（Dependency Injection）：用一个单独的对象（装配器）来装配对象之间的依赖关系。



2、理解DI问题关键

谁依赖于谁？ ------- **应用程序依赖于IoC容器**
为什么需要依赖？ ------- **应用程序依赖于IoC容器装配类之间的关系**
依赖什么东西？ ------- **依赖了IoC容器的装配功能**
谁注入于谁？ ------- **IoC容器注入应用程序**
注入什么东西？ ------- **注入应用程序需要的资源（类之间的关系）**

更能描述容器其特点的名字——“依赖注入”（Dependency Injection）
IoC容器应该具有依赖注入功能，因此也可以叫DI容器

3、DI优点

【1】帮你看清组件之间的依赖关系，只需要观察依赖注入的机制（setter/构造器），就可以掌握整个依赖（类与类之间的关系）。

【2】组件之间的依赖关系由容器在运行期决定，形象的来说，即由容器动态的将某种依赖关系注入到组件之中。

【3】依赖注入的目标并非为软件系统带来更多的功能，而是为了提升组件重用的概率，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，而不用关心具体的资源来自何处、由谁实现。

使用DI限制：组件和装配器（IoC容器）之间不会有依赖关系，因此组件无法从装配器那里获得更多服务，只能获得配置信息中所提供的那些。

4、实现方式

1、构造器注入

2、setter注入

3、接口注入：在接口中定义需要注入的信息，并通过接口完成注入

```
@Autowired
public void prepare(MovieCatalog movieCatalog,
    CustomerPreferenceDao customerPreferenceDao) {
    this.movieCatalog = movieCatalog;
    this.customerPreferenceDao = customerPreferenceDao;
}
```

使用IoC/DI容器开发需要改变的思路

1、应用程序不主动创建对象，但要描述创建它们的方式。

2、在应用程序代码中不直接进行服务的装配，但要配置文件中描述哪一个组件需要哪一项服务。容器负责将这些装配在一起。

其原理是基于OO设计原则的The Hollywood Principle：Don‘t call us, we’ll call you（别找我，我会来找你的）。也就是说，所有的组件都是被动的（Passive），所有的组件初始化和装配都由容器负责。组件处在一个容器当中，由容器负责管理。

IoC容器功能：实例化、初始化组件、装配组件依赖关系、负责组件生命周期管理。

本质：

IoC：控制权的转移，由应用程序转移到框架；

IoC/DI容器：**由**应用程序主动实例化对象**变**被动等待对象（被动实例化）；

DI： 由专门的装配器装配组件之间的关系；

IoC/DI容器：**由**应用程序主动装配对象的依赖**变**应用程序被动接受依赖

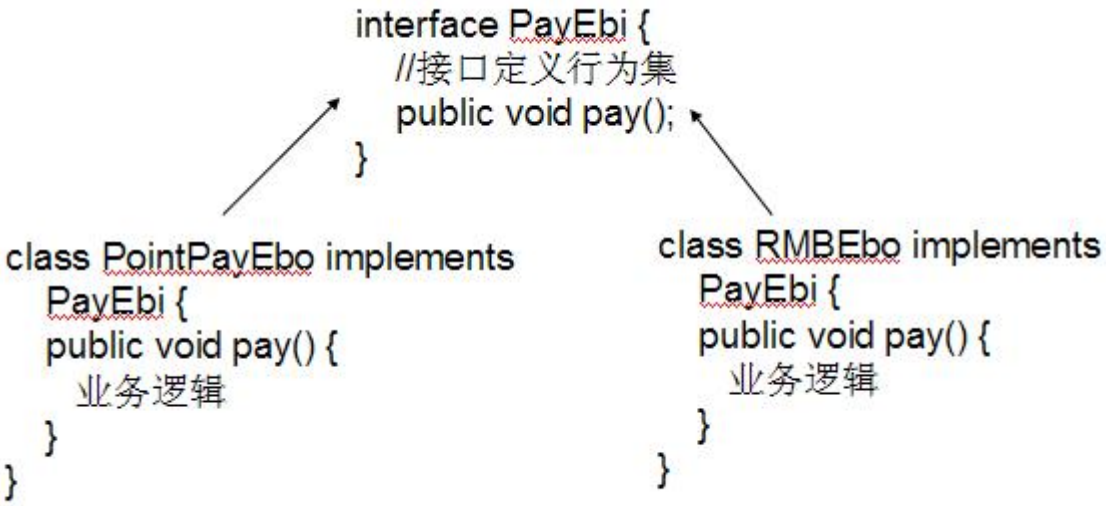
关于IoC/DI与DIP之间的关系 详见 http://www.iteye.com/topic/1122310?page=5#2335746

IoC/DI与迪米特法则 详见http://www.iteye.com/topic/1122310?page=5#2335748

1.28 我对AOP的理解

发表时间: 2012-04-05

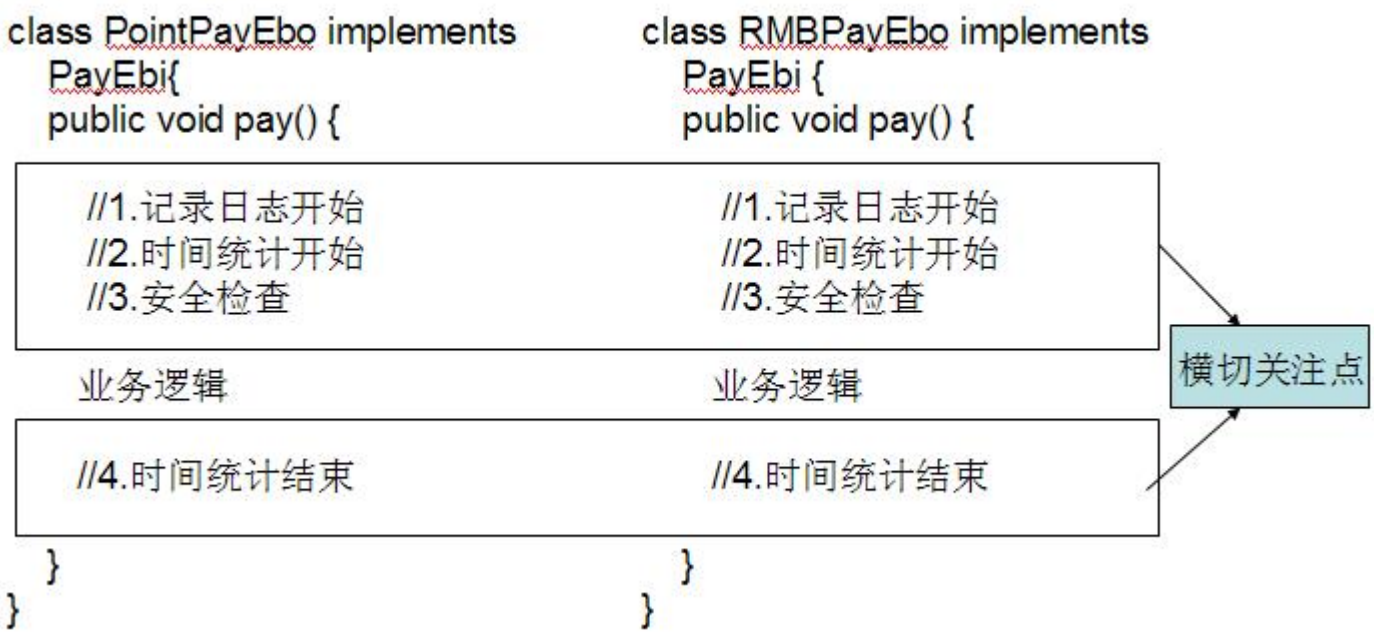
1、问题



问题：想要添加日志记录、性能监控、安全监测

2、最初解决方案

2.1、最初解决方案

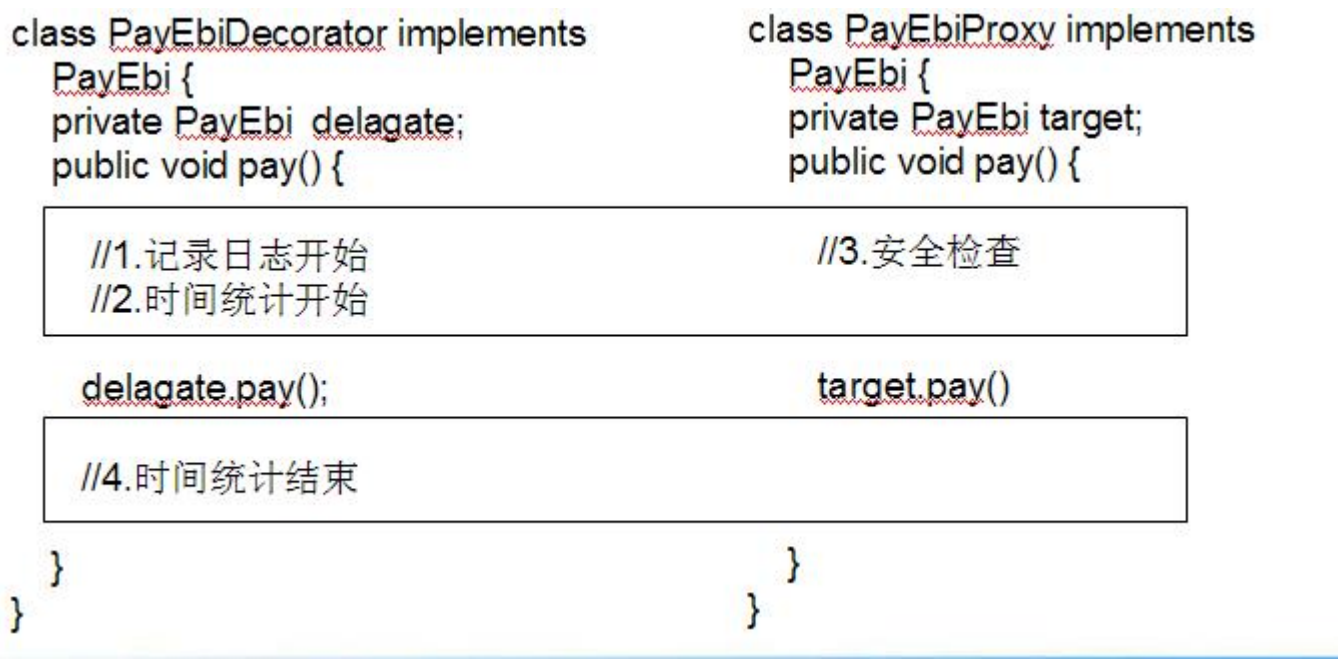


缺点：太多重复代码，且紧耦合

2.2、抽象类进行共性设计，子类进行个性设计，此处不讲解，缺点一荣俱荣，一损俱损

2.3、使用装饰器模式/代理模式改进的解决方案

装饰器模式：动态地给一个对象添加一些额外的职责。就增加功能来说， 装饰器模式相比生成子类更为灵活。
代理模式：为其他对象提供一种代理以控制对这个对象的访问。



缺点：紧耦合，每个业务逻辑需要一个装饰器实现或代理

2.4、JDK动态代理解决方案（比较通用的解决方案）

```
public class MyInvocationHandler implements InvocationHandler {
    private Object target;
    public MyInvocationHandler(Object target) {
        this.target = target;
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        //1.记录日志    2.时间统计开始      3.安全检查
        Object retVal = method.invoke(target, args);
        //4.时间统计结束
        return retVal;
    }
    public static Object proxy(Object target) {
        return Proxy.newProxyInstance(target.getClass().getClassLoader(),
                target.getClass().getInterfaces(), new MyInvocationHandler(target));
    }
}
```

编程模型

```
    //proxy      在其上调用方法的代理实例
    //method 拦截的方法
    //args       拦截的参数
    Override
     public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
         Object retVal=null;
         //预处理
         //前置条件判断
         boolean ok = true;
         if(!ok) {//不满足条件
             throw new RuntimeException("你没有权限");
         }
         else {//反射调用目标对象的某个方法
             retVal = method.invoke(target, args);
         }
         //后处理
         return retVal;
     }
```

缺点：使用麻烦，不能代理类，只能代理接口

CGLIB动态代理解决方案（比较通用的解决方案）

```
public class MyInterceptor implements MethodInterceptor  {
    private Object target;
    public MyInterceptor(Object target) {
        this.target = target;
    }
    @Override
    public Object intercept(Object proxy, Method method, Object[] args,
                                          MethodProxy invocation) throws Throwable {
        //1.记录日志 2.时间统计开始    3.安全检查
        Object retVal = invocation.invoke(target, args);
        //4.时间统计结束
        return retVal;
```

```
    }
    public static Object proxy(Object target) {
        return Enhancer.create(target.getClass(), new MyInterceptor(target));
    }
}
```

编程模型

```
 //proxy 在其上调用方法的代理实例     method拦截的方法     args   拦截的参数
 //invocation 用来去调用被代理对象方法的
@Override
public Object intercept(Object proxy, Method method, Object[] args,
                                        MethodProxy invocation) throws Throwable {
    //预处理
    //前置条件判断
    boolean ok = true;
    if(!ok) {//不满足条件
        throw new RuntimeException("出错了");
    }
    else {//调用目标对象的某个方法
        Object retVal = invocation.invoke(target, args);
    }
    //后处理
    return retVal;
}
```

优点：能代理接口和类

缺点：使用麻烦，不能代理final类

动态代理本质

本质：对目标对象增强

最终表现为类（动态创建子类），看手工生成（子类）还是自动生成（子类）

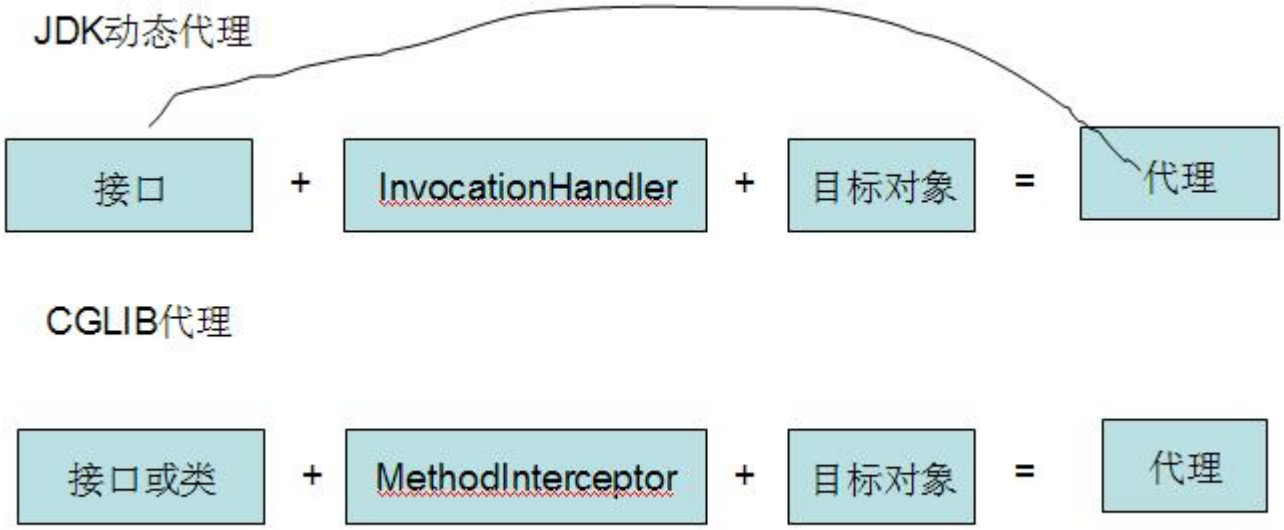
代理限制：

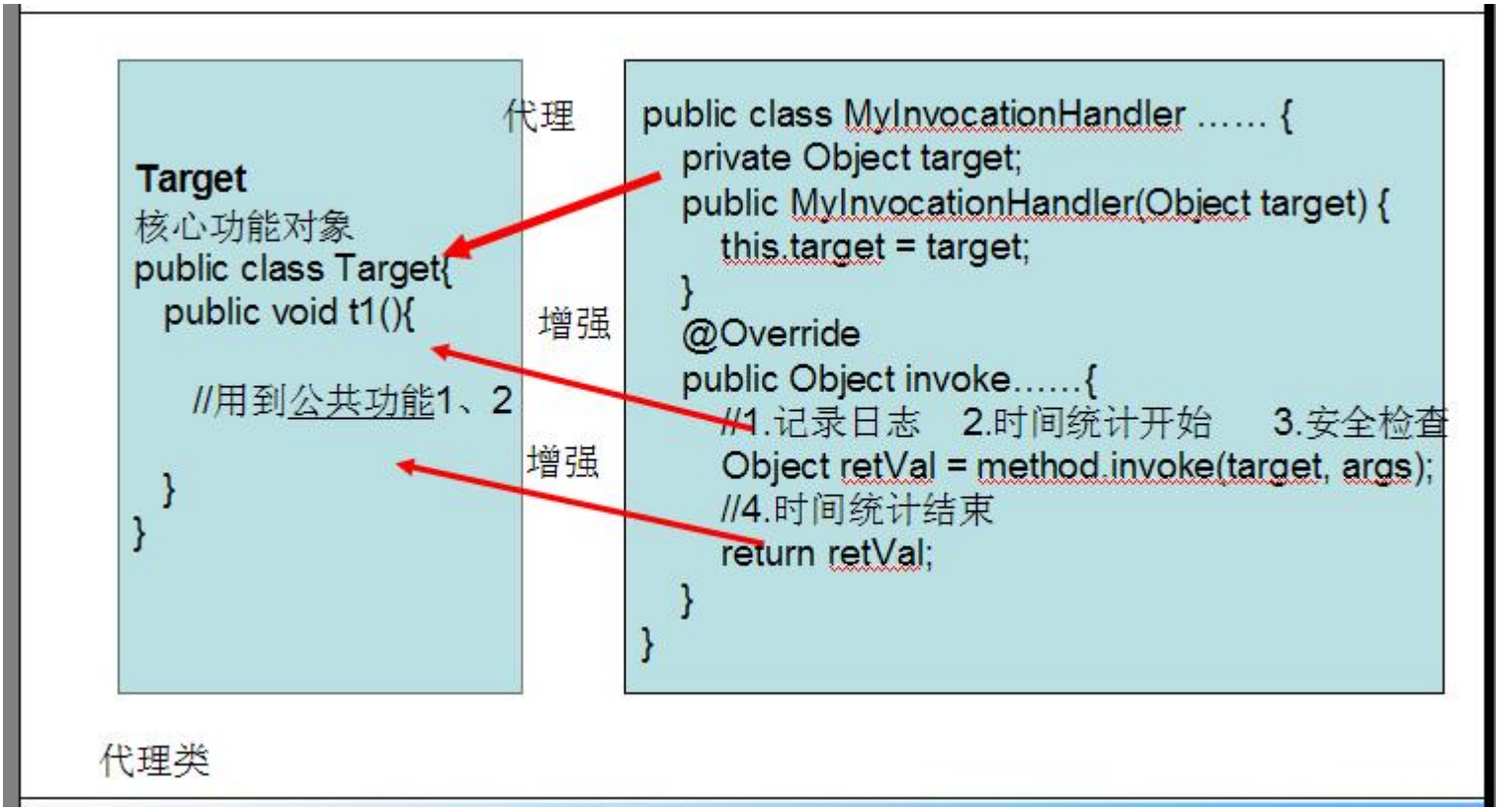
只能在父类方法被调用之前或之后进行增强（功能的修改），不能在中间进行修改，要想在方法调用中增强，需要ASM(java 字节码生成库)

其他动态代理框架

jboss：javassist（hibernate 3.3中默认为javassist）

（hibernate 3.3之前中默认为cglib）





2.5、AOP解决方案（通用且简单的解决方案）

```
@Aspect
public class PayEbiAspect {
```

```
    @Pointcut(value="execution(* pay(..))")
    public void pointcut() {}
    @Around(value="pointcut()")
    public Object around(ProceedingJoinPoint pjp) throws Throwable {
        //1.记录日志
        //2.时间统计开始
        //3.安全检查
        Object retVal = pjp.proceed();//调用目标对象的真正方法
        //4.时间统计结束
        return retVal;
    }
}
```

编程模型

```
    //2 切入点
    @Pointcut(value="execution(* *(..))")
    public void pointcut() {}
    //3 拦截器的interceptor
    @Around(value="pointcut()")
    public Object around(ProceedingJoinPoint pjp) throws Throwable {
        Object retVal=null;
        //预处理
        //前置条件判断
        boolean ok = true;
        if(!ok) {//不满足条件
           throw new RuntimeException("你没有权限");
        }
        else {//调用目标对象的某个方法
             retVal = pjp.proceed();
        }
        //后处理
        return retVal;
    }
```

缺点：依赖AOP框架

AOP入门

概念：

n**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；

n**关注点分离：**将问题细化为单独部分，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；

n**横切关注点：**会在多个模块中出现，使用现有的编程方法，横切关注点会横越多个模块，结果是使系统难以设计、理解、实现和演进，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。

nAOP是什么(Aspect Oriented Programming)

AOP是一种编程范式，提供从另一个角度来考虑程序结构以完善面向对象编程（OOP）。

AOP为开发者提供了一种描述横切关注点的机制，并能够自动将横切关注点织入到面向对象的软件系统中，从而实现了横切关注点的模块化。

AOP能够将那些与业务无关，却为业务模块所共同调用的逻辑或责任，例如事务处理、日志管理、权限控制等，封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。

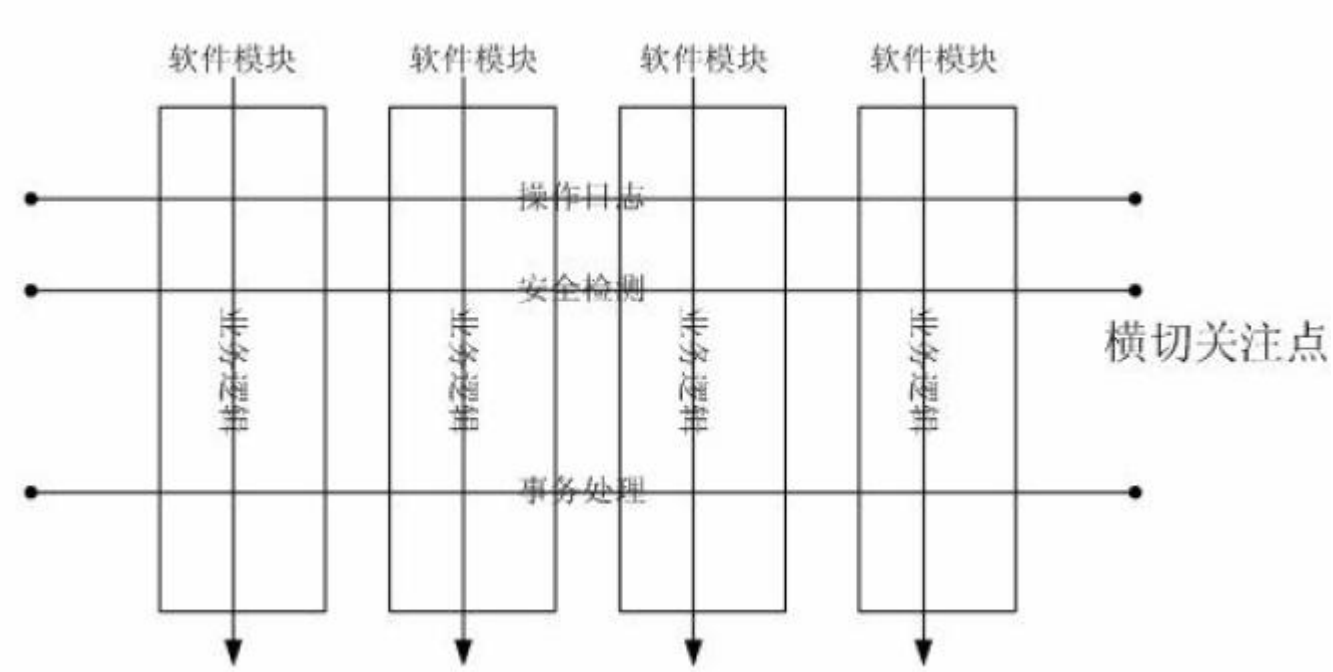
nAOP能干什么，也是AOP带来的好处

1：降低模块的耦合度

2：使系统容易扩展

3：设计决定的迟绑定：使用AOP,设计师可以推迟为将来的需求作决定，因为它
可以把这种需求作为独立的方面很容易的实现。

4：更好的代码复用性



AOP基本概念

连接点（Joinpoint）：

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“在哪里做”**；

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“在哪里做的集合”**；

增强（Advice）：或称为增强

在连接点上执行的行为，增强提供了在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；包括前置增强（before advice）、后置增强 (after advice)、环绕增强（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增强；**在AOP中表示为“做什么”；**

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。可以认为是增强、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“在哪里做和做什么集合”；**

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，需要被增强的对象，从而也可称为“被增强对象”；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“对谁做”**；

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增强（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。

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“做什么（新增什么）”**；

AOP的Advice类型

前置增强（Before advice）：

在某连接点之前执行的增强，但这个增强不能阻止连接点前的执行（除非它抛出一个异常）。

后置返回增强（After returning advice）：

在某连接点正常完成后执行的增强：例如，一个方法没有抛出任何异常，正常返回。

后置异常增强（After throwing advice）：

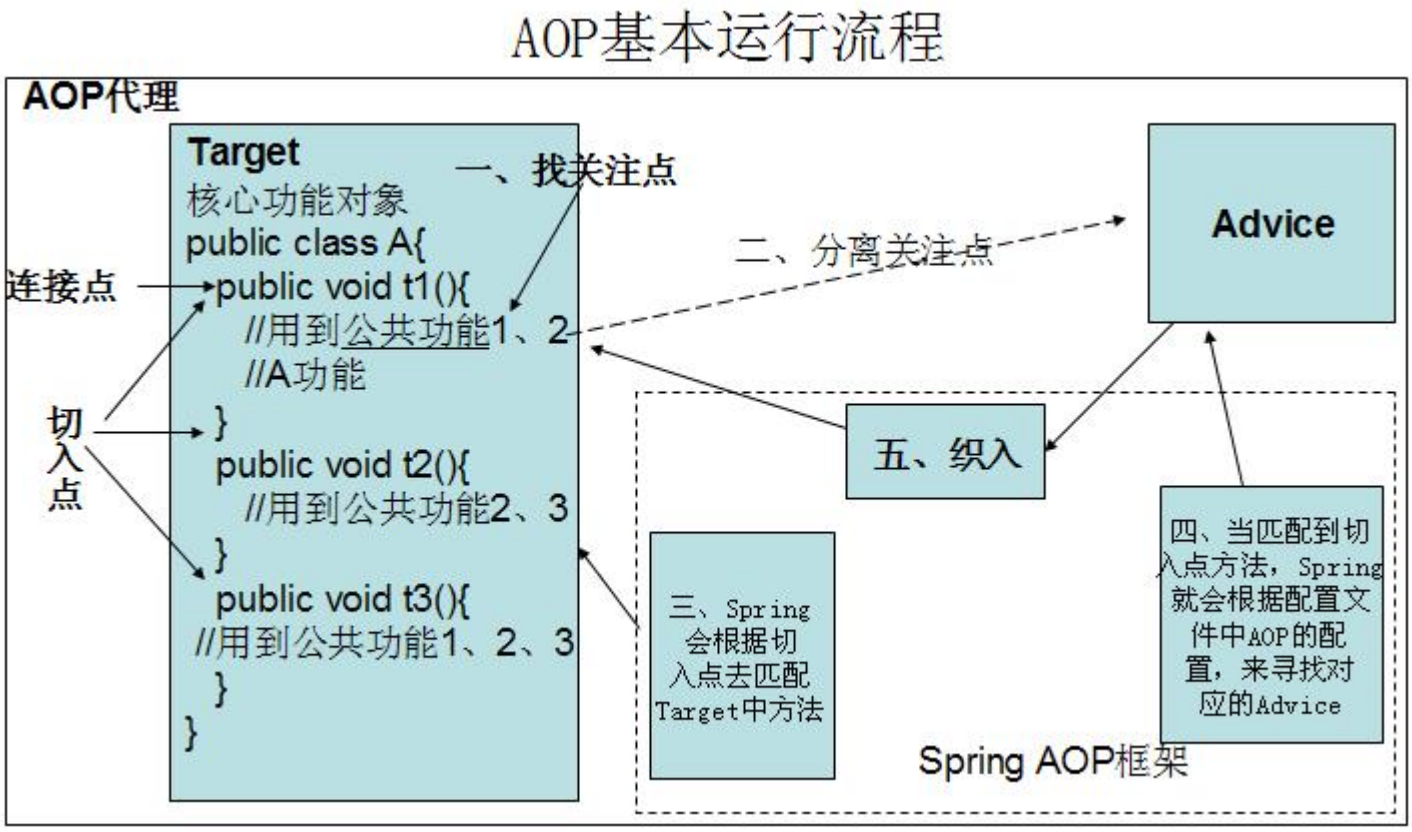
在方法抛出异常退出时执行的增强。

后置最终增强（After (finally) advice）：

当某连接点退出的时候执行的增强（不论是正常返回还是异常退出）。

环绕增强（Around Advice）：

包围一个连接点的增强，如方法调用。这是最强大的一种增强类型。 环绕增强可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点或直接返回它们自己的返回值或抛出异常来结束执行。



AOP开发步骤

类似于IoC/DI容器开发步骤，需要描述哪个连接点需要哪个通用功能（增强**）**

参考：http://www.iteye.com/topic/1122310

横切关注点的表现有：

□**代码纠结/混乱**——当一个模块或代码段同时管理多个关注点时发生这种情况。如我既要实现业务、还要实现安全和事务。即有些关注点同时被多个不同的模块实现。实现了重复的功能。

□**代码分散**——当一个关注点分布在许多模块中并且未能很好地局部化和模块化时发生这种情况 。如许多模块调用用户是否登录验证代码。调用了重复的功能。

AOP包括三个清晰的开发步骤：

1：**功能横切**：找出横切关注点。

2：**实现分离**：各自独立的实现这些横切关注点所需要完成的功能。

3：**功能回贴**：在这一步里，方面集成器通过创建一个模块单元—— 方面来指定重组的规则。重组过程——也叫织入或结合—— 则使用这些信息来构建最终系统。

推荐阅读书籍：

AspectJ in Action

AOSD中文版--基于用例的面向方面软件开发

推荐阅读的帖子：

AOP的实现机制

文章主要是为了抛砖引玉，希望有更多牛人的指点。

跟我学spring3(8-13)

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