Spring和Hibernate整合

事务传播特性

1. PROPAGATION_REQUIRED: 如果存在一个事务，则支持当前事务。如果没有事务则开启
2. PROPAGATION_SUPPORTS: 如果存在一个事务，支持当前事务。如果没有事务，则非事务的执行
3. PROPAGATION_MANDATORY: 如果已经存在一个事务，支持当前事务。如果没有一个活动的事务，则抛出异常。
4. PROPAGATION_REQUIRES_NEW: 总是开启一个新的事务。如果一个事务已经存在，则将这个存在的事务挂起。
5. PROPAGATION_NOT_SUPPORTED: 总是非事务地执行，并挂起任何存在的事务。
6. PROPAGATION_NEVER: 总是非事务地执行，如果存在一个活动事务，则抛出异常
7. PROPAGATION_NESTED：如果一个活动的事务存在，则运行在一个嵌套的事务中； 如果没有活动事 务则按 TransactionDefinition.PROPAGATION_REQUIRED 属性执行

Spring事务的隔离级别

1. ISOLATION_DEFAULT： 这是一个PlatfromTransactionManager默认的隔离级别，使用数据库默认的事务隔离级别.

另外四个与JDBC的隔离级别相对应

2. ISOLATION_READ_UNCOMMITTED： 这是事务最低的隔离级别，它充许令外一个事务可以看到这个事务未提交的数据。 这种隔离级别会产生脏读，不可重复读和幻像读。
3. ISOLATION_READ_COMMITTED： 保证一个事务修改的数据提交后才能被另外一个事务读取。另外一个事务不能读取该事务未提交的数据
4. ISOLATION_REPEATABLE_READ： 这种事务隔离级别可以防止脏读，不可重复读。但是可能出现幻像读。它除了保证一个事务不能读取另一个事务未提交的数据外，还保证了避免下面的情况产生(不可重复读)。
5. ISOLATION_SERIALIZABLE 这是花费最高代价但是最可靠的事务隔离级别。事务被处理为顺序执行。 除了防止脏读，不可重复读外，还避免了幻像读。

脏读，不可重复读，幻觉读

1. **脏读：**脏读就是指当一个事务正在访问数据，并且对数据进行了修改，而这种修改还没有提交到数据库中，这时，另外一个事务也访问这个数据，然后使用了这个数据。
2. **不可重复读：**是指在一个事务内，多次读同一数据。在这个事务还没有结束时，另外一个事务也访问该同一数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改，那么第一个事务两次读到的的数据可能是不一样的。这样就发生了在一个事务内两次读到的数据是不一样的，因此称为是不可重复读。例如，一个编辑人员两次读取同一文档，但在两次读取之间，作者重写了该文档。当编辑人员第二次读取文档时，文档已更改。原始读取不可重复。如果只有在作者全部完成编写后编辑人员才可以读取文档，则可以避免该问题。
3. **幻读：**是指当事务不是独立执行时发生的一种现象，例如第一个事务对一个表中的数据进行了修改，这种修改涉及到表中的全部数据行。同时，第二个事务也修改这个表中的数据，这种修改是向表中插入一行新数据。那么，以后就会发生操作第一个事务的用户发现表中还有没有修改的数据行，就好象发生了幻觉一样。

声明式事务

- ✓ 声明式事务处理的优点
  - ➢ 代码中无须关注事务逻辑，由Spring声明式事务管理负责事务逻辑。
  - ➢ 无须与具体的事务逻辑耦合，可以方便地在不同事务逻辑之间切换。
- ✓ 声明式事务可以由两种方式实现
  - ➢ 使用XML配置声明性事务
  - ➢ 使用注解配置声明性事务

使用XML配置声明性事务——session工厂和事务管理器

```
<!-- 通过查找hibernate的配置文件创建出Session工厂对象 -->
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="configLocations">
        <!-- 通过classpath类搜索路径找到hibernate配置文件 -->
        <value>classpath:hibernate.cfg.xml</value>
    </property>
</bean>
<!-- 配置事务管理器 -->
<bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <!-- 注入Session工厂对象 -->
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

使用XML配置声明性事务——事务的传播特性

```
<!-- 配置事务的传播特性 -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <!-- 配置哪些具体方法的事务传播特性为REQUIRED，隔离级别为DEFAULT默认
    Spring默认只有发生运行时异常才自动回退，rollback-for设置发生所有异常都自动回退
    需要注意的是，用这种方式，方法的命名规则需要统一-->
    <tx:attributes>
        <tx:method name="add*" propagation="REQUIRED" rollback-for="Exception"/>
        <tx:method name="update*" propagation="REQUIRED"  rollback-for="Exception"/>
        <tx:method name="delete*" propagation="REQUIRED" rollback-for="Exception"/>
        <!--配置除了以上方法之外的方法的事务传播特性，read-only表示只读事务，仅
    查询-->
        <tx:method name="*" read-only="true"/>
    </tx:attributes>
</tx:advice>
```

使用XML配置声明性事务——配置切点

```
<!-- 配置哪些类的哪些方法参与事务 -->
<aop:config>
    <!-- 配置切点是com.spring.dao包下的所有类的所有方法，实际上一般都是在service层横切入事务 -->
    <aop:pointcut id="allServiceMethod" expression="execution(* com.spring.dao.*.*(..))"/>
    <!-- 配置增强，advice-ref引用上面配置好的事务传播特性作为增强-->
    <aop:advisor pointcut-ref="allServiceMethod" advice-ref="txAdvice"/>
</aop:config>
```

配置Bean注入SessionFactory

```
<!-- 配置业务Bean，注入SessionFactory属性值 -->
<bean id="userDAO"  class="com.spring.dao.UserDAOImpl">
    <property name="sf" ref="sessionFactory"/>
    <property name="logDAO" ref="logDAO"/>
</bean>

<bean id="logDAO" class="com.spring.dao.LogDAOImpl">
    <property name="sf" ref="sessionFactory"/>
</bean>
```

使用注解配置声明性事务

加入tx:annotation-driven配置标签

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

//类加注解：类中所有方法都是默认事务传播和隔离级别
@Transactional
public class UserDAOImpl implements UserDAO{
    //方法加注解：当前方法单独设置事务传播方式
    @Transactional(propagation=Propagation.REQUIRED)
    public void addUser(User user) throws Exception {
            ……
    }
    //方法加注解：当前方法为只读事务
    @Transactional(readOnly=true)
    public User getUser(){
            ……
    }
}
```

Spring配置数据源和连接池——DBCP

```
<bean id="dataSource"
    class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName"
    value="oracle.jdbc.driver.OracleDriver">
    </property>
    <property name="url"
    value="jdbc:oracle:thin:@127.0.0.1:1521:orcl">
    </property>
    <property name="username" value="scott"></property>
    <property name="password" value="tiger"></property>
</bean>
```

Spring配置数据源和连接池——C3P0

```
<!-- 配置C3P0连接池，设定destroy-method=”close”属性， 以便Spring容器关闭时，数据源能够正常关闭-->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource" destroy-method="close">
    <!-- 指定连接数据库的驱动 -->
     <property name="driverClass" value="oracle.jdbc.driver.OracleDriver"/>
     <!-- 指定连接数据库的URL -->
     <property name="jdbcUrl" value="jdbc:oracle:thin:@localhost:1521:orcl"/>
     <!-- 指定连接数据库的用户名 -->
     <property name="user" value="spring1"/>
     <!-- 指定连接数据库的密码 -->
     <property name="password" value="spring1"/>
     <!-- 指定连接数据库连接池的最大连接数 -->
     <property name="maxPoolSize" value="20"/>
     <!-- 指定连接数据库连接池的最小连接数 -->
     <property name="minPoolSize" value="5"/>
    <!-- 指定连接数据库连接池的连接的最大空闲时间，按秒计 -->
     <property name="maxIdleTime" value="60"/>
</bean>
```

Spring配置数据源和连接池——JNDI

如果已经配置好了一个JNDI数据源，可以直接调用

```
<bean id="dataSource"
        class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName">
        <value>java:comp/env/jdbc/test</value>
    </property>
</bean>
```

Spring配置Hibernate的Session Factory

```
<!--创建Session工厂对象 -->
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <!-- 注入上面配置的数据源 -->
    <property name="dataSource" ref="dataSource"/>
    <!-- 配置Hibernate的SessionFactory属性 -->
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">
                org.hibernate.dialect.Oracle9Dialect
            </prop>
            <prop key="hibernate.show_sql">true</prop>
        </props>
    </property>
    <!-- 配置映射文件路径 -->
    <property name="mappingResources">
        <list>
            <value>com/spring/po/User.hbm.xml</value>
            <value>com/spring/po/Log.hbm.xml</value>
        </list>
    </property>
</bean>
```

HibernateDaoSupport类和HibernateTemplate

- ✓ Spring 提供了标准的Hibernate模板，对于HibernateDaoSupport类，只需要注入sessionFactory，还自动提供了HibernateTemplate的实例
- ✓ HibernateDaoSupport类本身就提供了基于AOP事务的自动处理，程序员完全可以不用理会事务的开始与提交
- ✓ 我们都在DAO实现类中继承HibernateDaoSupport类，然后通过继承的getHibernateTemplate()方法可以获得一个HibernateTemplate对象
- ✓ HibernateTemplate是一个hibernate模板对象，它封装了hibernate的数据操作，相对于hibernate的原始操作来说更简单一些

HibernateDaoSupport示例

```
public class UserDAOImpl extends HibernateDaoSupport implements UserDAO{
	//增加
	public void addUser(User user) throws Exception {
		this.getHibernateTemplate().save(user);
	}
	//返回单个
	public User getUser(Integer id) throws Exception {
		return (User) this.getHibernateTemplate().get(User.class, id);
	}
	//更新
	public void updateUser(User user) throws Exception {
		this.getHibernateTemplate().update(user);
	}
	//删除
	public void deleteUser(Integer id) throws Exception {
		User user = this.getUser(id);
		this.getHibernateTemplate().delete(user);
	}
	//返回全部
	public List<User> getAllUser() throws Exception {
		return this.getHibernateTemplate().find("from User user");
	}
}
```

HibernateCallBack接口

- ✓ Spring封装了hibernate的数据操作之后，数据操作的灵活性降低
- ✓ Spring提供了一个弥补HibernateTemplate类不足的HibernateCallBack接口，此接口只有一个doInHibernate(Session session) 的方法，实现接口覆盖此方法可以直接用Hibernate原生语句实现数据操作，然后把数据操作的结果通过此方法的返回值直接返回即可
- ✓ HibernateTemplate通过execute或者executeFind方法传入 HibernateCallBack接口的实现类对象实例实现操作

HibernateCallBack接口示例

- ✓ 示例：分页显示基本操作，返回第1条到第3条数据，采取匿名类传入HibernateCallBack接口的实现类

```
// 根据页码返回全部
public List<User> getAllUserByPage() throws Exception {
    return this.getHibernateTemplate().executeFind(new HibernateCallback() {
            public Object doInHibernate(Session session)
                                    throws HibernateException, SQLException {
                    List<User> list = session.createQuery("from User user")
                                            .setFirstResult(0).setMaxResults(3).list();
                    return list;
            }
    });
}
```