

百知教育 — Spring系列课程 — 持久层整合

第一章、持久层整合

1.Spring框架为什么要与持久层技术进行整合

- 1 1. JavaEE开发需要持久层进行数据库的访问操作。
- 2 2. JDBC Hibernate MyBatis进行持久开发过程存在大量的代码冗余
- 3 3. Spring基于模板设计模式对于上述的持久层技术进行了封装

2. Spring可以与那些持久层技术进行整合？

- 1 1. JDBC
- 2 |- JDBCTemplate
- 3 2. Hibernate (JPA)
- 4 |- HibernateTemplate
- 5 3. MyBatis
- 6 |- SqlSessionFactoryBean MapperScannerConfigure

第二章、Spring与MyBatis整合

1. MyBatis开发步骤的回顾

- 1 1. 实体
- 2 2. 实体别名
- 3 3. 表
- 4 4. 创建DAO接口
- 5 5. 实现Mapper文件
- 6 6. 注册Mapper文件
- 7 7. MybatisAPI调用

2. Mybatis在开发过程中存在问题

- 1 配置繁琐 代码冗余
- 2
- 3 1. 实体
- 4 2. 实体别名 配置繁琐
- 5 3. 表
- 6 4. 创建DAO接口
- 7 5. 实现Mapper文件
- 8 6. 注册Mapper文件 配置繁琐
- 9 7. MybatisAPI调用 代码冗余

3. Spring与Mybatis整合思路分析

mybatis-config.xml

1. dataSource
2. typeAliases
3. mapper文件注册

```

InputStream inputStream = Resources.getResourceAsStream("mybatis-config.xml");
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

SqlSession session = sqlSessionFactory.openSession();
UserDAO userDAO = session.getMapper(UserDAO.class);
ProductDAO class
        
```

SqlSessionFactoryBean

作用：用于封装SqlSessionFactory创建的代码
注意：mybatis-config.xml 省略 不要

```

<bean id="dataSource" class="" />
<bean id="ssfb" class="SqlSessionFactoryBean">
  <property name="dataSource" ref="dataSource">
    <property name="typeAliasesPackage">
      com.baizhiedu.entity
      指定 实体类所在的包 Spring自动创建别名 User Product
    <property name="mapperLocations">
      通配的设置
      *Mapper.xml --> UserDAOMapper.xml ProductDAOMapper.xml
    </bean>
        
```

MapperScannerConfigure

```

<bean id="scanner" class="MapperScannerConfigure">
  <property name="sqlSessionFactoryBeanName" value="ssfb">
  <property name="basePackage" --> 设置DAO接口所在的包
    com.baizhiedu.dao
  </bean>
  最终 创建dao对象
  注意：MapperScannerConfigure所创建的DAO对象 它的id值 是接口 首单词首字母小写
  UserDAO --> userDAO
  ProductDAO --> productDAO
  ctx.getBean("id ")
        
```

4. Spring与Mybatis整合的开发步骤

- 配置文件 (ApplicationContext.xml) 进行相关配置

```

1  #配置 是需要配置一次
2  <bean id="dataSource" class="" />
3
4  <!--创建SqlSessionFactory-->
5  <bean id="ssfb" class="SqlSessionFactoryBean">
6    <property name="dataSource" ref="" />
7    <property name="typeAliasesPackage">
8      指定 实体类所在的包  com.baizhiedu.entity  User
9      Product
10   </property>
11   <property name="mapperLocations">
12     指定 配置文件(映射文件)的路径 还有通用配置
13     com.baizhiedu.mapper/*Mapper.xml
14   </property>
15 </bean>
16
17 <!--DAO接口的实现类
18   session --> session.getMapper() --- xxxDAO实现类对象
19   XXXDAO --> xxxDAO
20 -->
21 <bean id="scanner" class="MapperScannerConfigure">
22   <property name="sqlSessionFactoryBeanName" value="ssfb" />
23   <property name="basePacakge">
24     指定 DAO接口放置的包  com.baizhiedu.dao
25   </property>
26 </bean>
        
```

- 编码

```
1  # 实战经常根据需求 写的代码
2  1. 实体
3  2. 表
4  3. 创建DAO接口
5  4. 实现Mapper文件
```

5. Spring与Mybatis整合编码

- 搭建开发环境(jar)

```
1  <dependency>
2    <groupId>org.springframework</groupId>
3    <artifactId>spring-jdbc</artifactId>
4    <version>5.1.14.RELEASE</version>
5  </dependency>
6
7  <dependency>
8    <groupId>org.mybatis</groupId>
9    <artifactId>mybatis-spring</artifactId>
10   <version>2.0.2</version>
11 </dependency>
12
13 <dependency>
14   <groupId>com.alibaba</groupId>
15   <artifactId>druid</artifactId>
16   <version>1.1.18</version>
17 </dependency>
18
19 <dependency>
20   <groupId>mysql</groupId>
21   <artifactId>mysql-connector-java</artifactId>
22   <version>5.1.48</version>
23 </dependency>
24
25 <dependency>
26   <groupId>org.mybatis</groupId>
27   <artifactId>mybatis</artifactId>
28   <version>3.4.6</version>
29 </dependency>
```

- Spring配置文件的配置

```
1  <!--连接池-->
2  <bean id="dataSource"
3    class="com.alibaba.druid.pool.DruidDataSource">
4    <property name="driverClassName" value="com.mysql.jdbc.Driver">
5    </property>
6    <property name="url" value="jdbc:mysql://localhost:3306/suns?
7    useSSL=false"></property>
8    <property name="username" value="root"></property>
9    <property name="password" value="123456"></property>
10  </bean>
11
12  <!--创建SqlSessionFactory SqlSessionFactoryBean-->
```

```

10 <bean id="sqlSessionFactoryBean"
    class="org.mybatis.spring.SqlSessionFactoryBean">
11     <property name="dataSource" ref="dataSource"></property>
12     <property name="typeAliasesPackage"
    value="com.baizhiedu.entity"></property>
13     <property name="mapperLocations">
14         <list>
15             <value>classpath:com.baizhiedu.mapper/*Mapper.xml</value>
16         </list>
17     </property>
18 </bean>
19
20 <!--创建DAO对象 MapperScannerConfigure-->
21
22 <bean id="scanner"
    class="org.mybatis.spring.mapper.MapperScannerConfigurer">
23     <property name="sqlSessionFactoryBeanName"
    value="sqlSessionFactoryBean"></property>
24     <property name="basePackage" value="com.baizhiedu.dao">
    </property>
25 </bean>

```

- 编码

```

1  1. 实体
2  2. 表
3  3. DAO接口
4  4. Mapper文件配置

```

6. Spring与Mybatis整合细节

- 问题：Spring与Mybatis整合后，为什么DAO不提交事务，但是数据能够插入数据库中？

```

1  Connection --> tx
2  Mybatis(Connection)
3
4  本质上控制连接对象(Connection) ----> 连接池(DataSource)
5  1. Mybatis提供的连接池对象 ----> 创建Connection
6      Connection.setAutoCommit(false) 手工的控制了事务，操作完成后，手工提交
7  2. Druid (C3P0 DBCP) 作为连接池 ----> 创建Connection
8      Connection.setAutoCommit(true) true默认值 保持自动控制事务，一条sql 自动提交
9  答案：因为Spring与Mybatis整合时，引入了外部连接池对象，保持自动的事务提交这个机制(Connection.setAutoCommit(true))，不需要手工进行事务的操作，也能进行事务的提交
10
11  注意：未来实战中，还会手工控制事务(多条sql一起成功，一起失败)，后续Spring通过事务控制解决这个问题。

```

第三章、Spring的事务处理

1. 什么是事务？

```
1  保证业务操作完整性的一种数据库机制
2
3  事务的4特点： A C I D
4  1. A 原子性
5  2. C 一致性
6  3. I 隔离性
7  4. D 持久性
```

2. 如何控制事务

```
1  JDBC:
2      Connection.setAutoCommit(false);
3      Connection.commit();
4      Connection.rollback();
5  Mybatis:
6      Mybatis自动开启事务
7
8      sqlSession(Connection).commit();
9      sqlSession(Connection).rollback();
10
11  结论：控制事务的底层 都是Connection对象完成的。
```

3.Spring控制事务的开发

```
1  Spring是通过AOP的方式进行事务开发
```

1. 原始对象

```
1  public class XXXUserServiceImpl{
2      private xxxDAO xxxDAO
3      set get
4
5      1. 原始对象 ---》 原始方法 ---》 核心功能（业务处理+DAO调用）
6      2. DAO作为Service的成员变量，依赖注入的方式进行赋值
7  }
```

2. 额外功能

```
1  1. org.springframework.jdbc.datasource.DataSourceTransactionManager
2  2. 注入DataSource
3  1. MethodInterceptor
4      public Object invoke(MethodInvocation invocation){
5          try{
6              Connection.setAutoCommit(false);
7              Object ret = invocation.proceed();
8              Connection.commit();
9          }catch(Exception e){
10              Connection.rollback();
11          }
12          return ret;
13      }
14  2. @Aspect
15      @Around
```

3. 切入点

```
1  @Transactional
2  事务的额外功能加入给那些业务方法。
3
4  1. 类上：类中所有的方法都会加入事务
5  2. 方法上：这个方法会加入事务
```

4 组装切面

```
1  1. 切入点
2  2. 额外功能
3
4  <tx:annotation-driven transaction-manager="" />
```

4. Spring控制事务的编码

- 搭建开发环境 (jar)

```
1  <dependency>
2    <groupId>org.springframework</groupId>
3    <artifactId>spring-tx</artifactId>
4    <version>5.1.14.RELEASE</version>
5  </dependency>
```

- 编码

```
1  <bean id="userService"
2    class="com.baizhiedu.service.UserServiceImpl">
3    <property name="userDAO" ref="userDAO" />
4  </bean>
5
6  <!--DataSourceTransactionManager-->
7  <bean id="dataSourceTransactionManager"
8    class="org.springframework.jdbc.datasource.DataSourceTransactionMa
9    nager">
10    <property name="dataSource" ref="dataSource" />
11  </bean>
12
13  @Transactional
14  public class UserServiceImpl implements UserService {
15    private UserDAO userDAO;
16
17    <tx:annotation-driven transaction-
18      manager="dataSourceTransactionManager" />
```

- 细节

```
1  <tx:annotation-driven transaction-
2    manager="dataSourceTransactionManager" proxy-target-class="true"/>
3  进行动态代理底层实现的切换    proxy-target-class
4    默认 false JDK
5    true Cglib
```

第四章、Spring中的事务属性(Transaction Attribute)

1. 什么是事务属性

- 1 属性：描述物体特征的一系列值
- 2 性别 身高 体重 ...
- 3 事务属性：描述事务特征的一系列值
- 4 1. 隔离属性
- 5 2. 传播属性
- 6 3. 只读属性
- 7 4. 超时属性
- 8 5. 异常属性

2. 如何添加事务属性

```
1 @Transactional(isolation=,propagation=,readOnly=,timeout=,rollbackFor=,  
noRollbackFor=,)
```

3. 事务属性详解

1. 隔离属性 (ISOLATION)

- 隔离属性的概念

- 1 概念：他描述了事务解决并发问题的特征
- 2 1. 什么是并发
- 3 多个事务(用户)在同一时间，访问操作了相同的数据
- 4
- 5 同一时间：0.000几秒 微小前 微小后
- 6 2. 并发会产生那些问题
- 7 1. 脏读
- 8 2. 不可重复读
- 9 3. 幻影读
- 10 3. 并发问题如何解决
- 11 通过隔离属性解决，隔离属性中设置不同的值，解决并发处理过程中的问题。

- 事务并发产生的问题

- 脏读

- 1 一个事务，读取了另一个事务中没有提交的数据。会在本事务中产生数据不一致的问题
- 2 解决方案 @Transactional(isolation=Isolation.READ_COMMITTED)

- 不可重复读

- 1 一个事务中，多次读取相同的数据，但是读取结果不一样。会在本事务中产生数据不一致的问题
- 2 注意：1 不是脏读 2 一个事务中
- 3 解决方案 @Transactional(isolation=Isolation.REPEATABLE_READ)
- 4 本质：一把行锁

- 幻影读

- 1 一个事务中，多次对整表进行查询统计，但是结果不一样，会在本事务中产生数据不一致的问题
- 2 解决方案 @Transactional(isolation=Isolation.SERIALIZABLE)
- 3 本质：表锁

- 总结

- 1 并发安全： SERIALIZABLE>REPEATABLE_READ>READ_COMMITTED
- 2 运行效率： READ_COMMITTED>REPEATABLE_READ>SERIALIZABLE

- 数据库对于隔离属性的支持

| 隔离属性的值 | MySQL | Oracle |
|---------------------------|-------|--------|
| ISOLATION_READ_COMMITTED | ✓ | ✓ |
| IOSLATION_REPEATABLE_READ | ✓ | ✗ |
| ISOLATION_SERIALIZABLE | ✓ | ✓ |

- 1 Oracle不支持REPEATABLE_READ值 如何解决不可重复读
- 2 采用的是多版本比对的方式 解决不可重复读的问题

- 默认隔离属性

- 1 ISOLATION_DEFAULT：会调用不同数据库所设置的默认隔离属性
- 2
- 3 MySQL ： REPEATABLE_READ
- 4 Oracle： READ_COMMITTED

- 查看数据库默认隔离属性

- MySQL

```
1 select @@tx_isolation;
```

- Oracle

```
1 SELECT s.sid, s.serial#,
2     CASE BITAND(t.flag, POWER(2, 28))
3         WHEN 0 THEN 'READ COMMITTED'
4         ELSE 'SERIALIZABLE'
5     END AS isolation_level
6 FROM v$transaction t
7 JOIN v$session s ON t.addr = s.taddr
8 AND s.sid = sys_context('USERENV', 'SID');
```

- 隔离属性在实战中的建议


```
1  推荐使用Spring指定的ISOLATION_DEFAULT
2    1. MySQL    repeatable_read
3    2. Oracle   read_committed
4
5  未来实战中，并发访问情况 很低
6
7  如果真遇到并发问题，乐观锁
8    Hibernate(JPA)  Version
9    MyBatis         通过拦截器自定义开发
10
```

2. 传播属性(PROPAGATION)

- 传播属性的概念

```
1  概念：他描述事务解决嵌套问题的特征
2
3  什么叫做事务的嵌套：他指的是一个大的事务中，包含了若干小的事务
4
5  问题：大事务中融入了很多小的事务，他们彼此影响，最终就会导致外部大的事务，丧失了事务的原子性
```

- 传播属性的值及其用法

| 传播属性的值 | 外部不存在事务 | 外部存在事务 | 用法 | 备注 |
|---------------|---------|---------------|---|---------|
| REQUIRED | 开启新的事务 | 融合到外部事务中 | @Transactional(propagation = Propagation.REQUIRED) | 增删改方法 |
| SUPPORTS | 不开启事务 | 融合到外部事务中 | @Transactional(propagation = Propagation.SUPPORTS) | 查询方法 |
| REQUIRES_NEW | 开启新的事务 | 挂起外部事务，创建新的事务 | @Transactional(propagation = Propagation.REQUIRES_NEW) | 日志记录方法中 |
| NOT_SUPPORTED | 不开启事务 | 挂起外部事务 | @Transactional(propagation = Propagation.NOT_SUPPORTED) | 及其不常用 |
| NEVER | 不开启事务 | 抛出异常 | @Transactional(propagation = Propagation.NEVER) | 及其不常用 |
| MANDATORY | 抛出异常 | 融合到外部事务中 | @Transactional(propagation = Propagation.MANDATORY) | 及其不常用 |

- 默认的传播属性

```
1 REQUIRED是传播属性的默认值
```

- 推荐传播属性的使用方式

```
1 增删改 方法：直接使用默认值REQUIRED
2 查询 操作：显示指定传播属性的值为SUPPORTS
```

3. 只读属性(readOnly)

```
1 针对于只进行查询操作的业务方法，可以加入只读属性，提供运行效率
2
3 默认值： false
```

4. 超时属性(timeout)

```
1 指定了事务等待的最长时间
2
3 1. 为什么事务进行等待？
4 当前事务访问数据时，有可能访问的数据被别的事务进行加锁的处理，那么此时本事务就必须
   进行等待。
5 2. 等待时间 秒
6 3. 如何应用 @Transactional(timeout=2)
7 4. 超时属性的默认值 -1
8 最终由对应的数据库来指定
```

5. 异常属性

```
1 Spring事务处理过程中
2 默认 对于RuntimeException及其子类 采用的是回滚的策略
3 默认 对于Exception及其子类 采用的是提交的策略
4
5 rollbackFor = {java.lang.Exception,xxx,xxx}
6 noRollbackFor = {java.lang.RuntimeException,xxx,xx}
7
8 @Transactional(rollbackFor = {java.lang.Exception.class},noRollbackFor
   = {java.lang.RuntimeException.class})
9
10 建议：实战中使用RuntimeExceptin及其子类 使用事务异常属性的默认值
```

4. 事务属性常见配置总结

```
1 1. 隔离属性 默认值
2 2. 传播属性 Required(默认值) 增删改 Supports 查询操作
3 3. 只读属性 readOnly false 增删改 true 查询操作
4 4. 超时属性 默认值 -1
5 5. 异常属性 默认值
6
7 增删改操作 @Transactional
8 查询操作
   @Transactional(propagation=Propagation.SUPPORTS,readOnly=true)
```

5. 基于标签的事务配置方式(事务开发的第二种形式)

```

1  基于注解 @Transaction的事务配置回顾
2  <bean id="userService" class="com.baizhiedu.service.UserServiceImpl">
3      <property name="userDAO" ref="userDAO" />
4  </bean>
5
6  <!--DataSourceTransactionManager-->
7  <bean id="dataSourceTransactionManager"
8      class="org.springframework.jdbc.datasource.DataSourceTransactionManage
9  r">
10
11      <property name="dataSource" ref="dataSource" />
12  </bean>
13
14  @Transactional(isolation=,propagation=,...)
15  public class UserServiceImpl implements UserService {
16      private UserDAO userDAO;
17
18  <tx:annotation-driven transaction-
19  manager="dataSourceTransactionManager" />
20
21  基于标签的事务配置
22  <bean id="userService" class="com.baizhiedu.service.UserServiceImpl">
23      <property name="userDAO" ref="userDAO" />
24  </bean>
25
26  <!--DataSourceTransactionManager-->
27  <bean id="dataSourceTransactionManager"
28      class="org.springframework.jdbc.datasource.DataSourceTransactionManage
29  r">
30      <property name="dataSource" ref="dataSource" />
31  </bean>
32
33  事务属性
34  <tx:advice id="txAdvice" transaction-
35  manager="dataSourceTransactionManager">
36      <tx:attributes>
37          <tx:method name="register" isolation="",propagation="">
38          </tx:method>
39          <tx:method name="login" .....></tx:method>
40          等效于
41          @Transactional(isolation=,propagation=,)
42          public void register(){
43              }
44      </tx:attributes>
45  </tx:advice>
46
47  <aop:config>
48      <aop:pointcut id="pc" expression="execution(*
49  com.baizhiedu.service.UserServiceImpl.register(..))"></aop:pointcut>
50      <aop:advisor advice-ref="txAdvice" pointcut-ref="pc">
51      </aop:advisor>
52  </aop:config>

```

- 基于标签的事务配置在实战中的应用方式

```

1  <bean id="userService"
    class="com.baizhiedu.service.UserServiceImpl">
2      <property name="userDAO" ref="userDAO" />
3  </bean>
4
5  <!--DataSourceTransactionManager-->
6  <bean id="dataSourceTransactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionMa
    nager">
7      <property name="dataSource" ref="dataSource" />
8  </bean>
9
10  编程时候 service中负责进行增删改操作的方法 都以modify开头
11      查询操作 命名无所谓
12  <tx:advice id="txAdvice" transaction-
    manager="dataSourceTransactionManager">
13      <tx:attributes>
14          <tx:method name="register"></tx:method>
15          <tx:method name="modify*"></tx:method>
16          <tx:method name="*" propagation="SUPPORTS" read-
    only="true"></tx:method>
17      </tx:attributes>
18  </tx:advice>
19
20  应用的过程中, service放置到service包中
21  <aop:config>
22      <aop:pointcut id="pc" expression="execution(*
    com.baizhiedu.service..*.*(..))"></aop:pointcut>
23      <aop:advisor advice-ref="txAdvice" pointcut-ref="pc">
24  </aop:advisor>
    </aop:config>

```


