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. JavaEE开发需要持久层进行数据库的访问操作。
2. JDBC Hibernate MyBatis进行持久开发过程存在大量的代码冗余
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. JDBC
     |-  JDBCTemplate
2. Hibernate (JPA)
     |-  HibernateTemplate
3. MyBatis
     |-  SqlSessionFactoryBean MapperScannerConfigure
```

第二章、Spring与MyBatis整合

1. MyBatis开发步骤的回顾

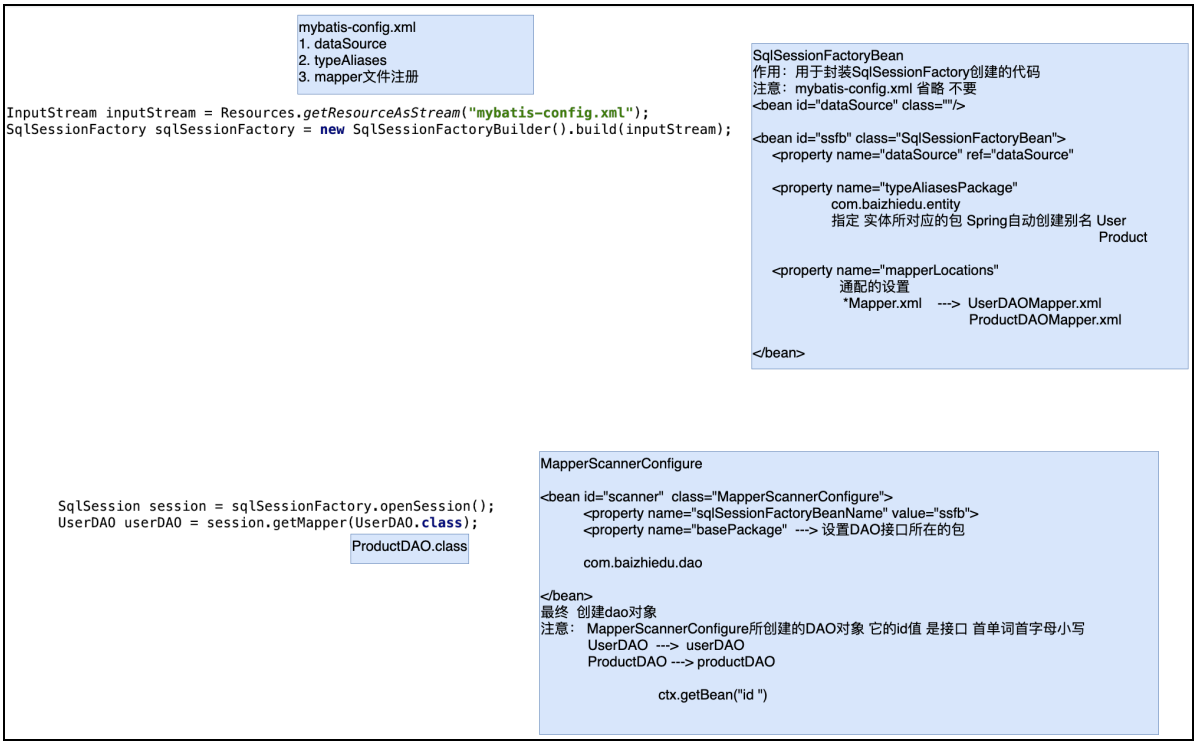
```
1. 实体
2. 实体别名
3. 表
4. 创建DAO接口
5. 实现Mapper文件
6. 注册Mapper文件
7. MybatisAPI调用
```

2. Mybatis在开发过程中存在问题

```
配置繁琐  代码冗余

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

3. Spring与Mybatis整合思路分析



4. Spring与Mybatis整合的开发步骤

- 配置文件（ApplicationContext.xml) 进行相关配置

```
#配置 是需要配置一次
<bean id="dataSource" class=""/>

<!--创建SqlSessionFactory-->
<bean id="ssfb" class="SqlSessionFactoryBean">
    <property name="dataSource" ref=""/>
    <property name="typeAliasesPackage">
         指定 实体类所在的包  com.baizhiedu.entity  User
                                                  Product
    </property>
    <property name="mapperLocations">
          指定 配置文件(映射文件)的路径 还有通用配置
          com.baizhiedu.mapper/*Mapper.xml
    </property>
</bean>

<!--DAO接口的实现类
    session ---> session.getMapper() --- xxxDAO实现类对象
    XXXDAO  ---> xXXDAO
-->
<bean id="scanner" class="MapperScannerConfigure">
    <property name="sqlSessionFactoryBeanName" value="ssfb"/>
    <property name="basePacakge">
        指定 DAO接口放置的包  com.baizhiedu.dao
    </property>
</bean>
```

- 编码

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

5. Spring与Mybatis整合编码

- 搭建开发环境(jar)

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>5.1.14.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis-spring</artifactId>
  <version>2.0.2</version>
</dependency>

<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid</artifactId>
  <version>1.1.18</version>
</dependency>

<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.48</version>
</dependency>

<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>3.4.6</version>
</dependency>
```

- Spring配置文件的配置

```
<!--连接池-->
<bean id="dataSource"
class="com.alibaba.druid.pool.DruidDataSource">
  <property name="driverClassName" value="com.mysql.jdbc.Driver">
</property>
  <property name="url" value="jdbc:mysql://localhost:3306/suns?
useSSL=false"></property>
  <property name="username" value="root"></property>
  <property name="password" value="123456"></property>
</bean>

<!--创建SqlSessionFactory SqlSessionFactoryBean-->
```

```
<bean id="sqlSessionFactoryBean"
class="org.mybatis.spring.SqlSessionFactoryBean">
  <property name="dataSource" ref="dataSource"></property>
  <property name="typeAliasesPackage"
value="com.baizhiedu.entity"></property>
  <property name="mapperLocations">
    <list>
      <value>classpath:com.baizhiedu.mapper/*Mapper.xml</value>
    </list>
  </property>
</bean>

<!--创建DAO对象 MapperScannerConfigure-->

<bean id="scanner"
class="org.mybatis.spring.mapper.MapperScannerConfigurer">
  <property name="sqlSessionFactoryBeanName"
value="sqlSessionFactoryBean"></property>
  <property name="basePackage" value="com.baizhiedu.dao">
</property>
</bean>
```

- 编码

```
1. 实体
2. 表
3. DAO接口
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?

```
Connection --> tx
Mybatis(Connection)

本质上控制连接对象(Connection) ---> 连接池(DataSource)
1. Mybatis提供的连接池对象 ---> 创建Connection
     Connection.setAutoCommit(false) 手工的控制了事务 ， 操作完成后，手工提交
2. Druid (C3P0 DBCP) 作为连接池         ---> 创建Connection
     Connection.setAutoCommit(true) true默认值 保持自动控制事务，一条sql 自动提交
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

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?

```
保证业务操作完整性的一种数据库机制

事务的4特点:  A C I D
1. A 原子性
2. C 一致性
3. I 隔离性
4. D 持久性
```

2. 如何控制事务

```
JDBC:
    Connection.setAutoCommit(false);
    Connection.commit();
    Connection.rollback();
Mybatis:
    Mybatis自动开启事务

    sqlSession(Connection).commit();
    sqlSession(Connection).rollback();

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

```
Spring是通过AOP的方式进行事务开发
```

1. 原始对象

```
public class XXXUserServiceImpl{
   private xxxDAO xxxDAO
   set get

   1. 原始对象 ---》 原始方法 ---》核心功能 (业务处理+DAO调用)
   2. DAO作为Service的成员变量, 依赖注入的方式进行赋值
}
```

2. 额外功能

```
1. org.springframework.jdbc.datasource.DataSourceTransactionManager
2. 注入DataSource
1. MethodInterceptor
   public Object invoke(MethodInvocation invocation){
      try{
        Connection.setAutoCommit(false);
        Object ret = invocation.proceed();
        Connection.commit();
      }catch(Exception e){
        Connection.rollback();
      }
        return ret;
   }
2. @Aspect
   @Around
```

3. 切入点

```
@Transactional
事务的额外功能加入给那些业务方法。

1. 类上：类中所有的方法都会加入事务
2. 方法上：这个方法会加入事务
```

4 组装切面

```
1. 切入点
2. 额外功能

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

4. Spring控制事务的编码

- 搭建开发环境 (jar)

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-tx</artifactId>
  <version>5.1.14.RELEASE</version>
</dependency>
```

- 编码

```
<bean id="userService"
class="com.baizhiedu.service.UserServiceImpl">
  <property name="userDAO" ref="userDAO"/>
</bean>

<!--DataSourceTransactionManager-->
<bean id="dataSourceTransactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionMa
nager">
  <property name="dataSource" ref="dataSource"/>
</bean>

@Transactional
public class UserServiceImpl implements UserService {
    private UserDAO userDAO;

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

- 细节

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

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

1. 什么是事务属性

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

2. 如何添加事务属性

```
@Transactional(isloation=,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

```
概念：他描述了事务解决并发问题的特征
1. 什么是并发
        多个事务(用户)在同一时间，访问操作了相同的数据

        同一时间：0.000几秒 微小前 微小后
2. 并发会产生那些问题
        1. 脏读
        2. 不可重复读
        3. 幻影读
3. 并发问题如何解决
        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

```
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
解决方案  @Transactional(isolation=Isolation.READ_COMMITTED)
```

  - 不可重复读

```
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
注意：1 不是脏读 2 一个事务中
解决方案 @Transactional(isolation=Isolation.REPEATABLE_READ)
本质：  一把行锁
```

  - 幻影读

```
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
解决方案 @Transactional(isolation=Isolation.SERIALIZABLE)
本质：表锁
```

- 总结

```
并发安全：  SERIALIZABLE>REPEATABLE_READ>READ_COMMITTED
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	✅	✅

```
Oracle不支持REPEATABLE_READ值 如何解决不可重复读
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

```
ISOLATION_DEFAULT：会调用不同数据库所设置的默认隔离属性

MySQL ：REPEATABLE_READ
Oracle：READ_COMMITTED
```

- 查看数据库默认隔离属性
  - MySQL

```
select @@tx_isolation;
```

  - Oracle

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

- 隔离属性在实战中的建议

```
推荐使用Spring指定的ISOLATION_DEFAULT
 1. MySQL   repeatable_read
 2. Oracle  read_commited

未来实战中，并发访问情况 很低

如果真遇到并发问题，乐观锁
   Hibernate(JPA)  Version
   MyBatis         通过拦截器自定义开发

```

2. 传播属性(PROPAGATION)

- 传播属性的概念

```
概念：他描述了事务解决嵌套问题的特征

什么叫做事务的嵌套：他指的是一个大的事务中，包含了若干个小的事务

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)

```
基于注解 @Transaction的事务配置回顾
<bean id="userService" class="com.baizhiedu.service.UserServiceImpl">
  <property name="userDAO" ref="userDAO"/>
</bean>

<!--DataSourceTransactionManager-->
<bean id="dataSourceTransactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManage
r">
  <property name="dataSource" ref="dataSource"/>
</bean>

@Transactional(isolation=,propagation=,...)
public class UserServiceImpl implements UserService {
    private UserDAO userDAO;

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

基于标签的事务配置
<bean id="userService" class="com.baizhiedu.service.UserServiceImpl">
  <property name="userDAO" ref="userDAO"/>
</bean>

<!--DataSourceTransactionManager-->
<bean id="dataSourceTransactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManage
r">
  <property name="dataSource" ref="dataSource"/>
</bean>

事务属性
<tx:advice id="txAdvice" transacation-
manager="dataSourceTransactionManager">
    <tx:attributes>
          <tx:method name="register" isoloation="",propagation="">
</tx:method>
          <tx:method name="login" .....></tx:method>
          等效于
          @Transactional(isolation=,propagation=,)
          public void register(){

          }

    </tx:attributes>
</tx:advice>

<aop:config>
     <aop:pointcut id="pc" expression="execution(*
com.baizhiedu.service.UserServiceImpl.register(..))"></aop:pointcut>
     <aop:advisor advice-ref="txAdvice" pointcut-ref="pc">
</aop:advisor>
</aop:config>
```

- 基于标签的事务配置在实战中的应用方式

```
<bean id="userService"
class="com.baizhiedu.service.UserServiceImpl">
  <property name="userDAO" ref="userDAO"/>
</bean>

<!--DataSourceTransactionManager-->
<bean id="dataSourceTransactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionMa
nager">
  <property name="dataSource" ref="dataSource"/>
</bean>

编程时候 service中负责进行增删改操作的方法 都以modify开头
                        查询操作 命名无所谓
<tx:advice id="txAdvice" transacation-
manager="dataSourceTransactionManager">
    <tx:attributes>
          <tx:method name="register"></tx:method>
          <tx:method name="modify*"></tx:method>
          <tx:method name="*" propagation="SUPPORTS"  read-
only="true"></tx:method>
    </tx:attributes>
</tx:advice>

应用的过程中，service放置到service包中
<aop:config>
     <aop:pointcut id="pc" expression="execution(*
com.baizhiedu.service..*.*(..))"></aop:pointcut>
     <aop:advisor advice-ref="txAdvice" pointcut-ref="pc">
</aop:advisor>
</aop:config>
```