第三阶段 MyBatis-03-整合

第三阶段 MyBatis-03-整合
回顾：
今天任务
教学目标
一. Spring整合MyBatis
1、mybatis-spring.jar
2、SqlSessionFactoryBean
2.1、SqlSessionFactoryBean的初始化
2.2、获取SqlSessionFactoryBean实例
3、MapperFactoryBean
4、MapperScannerConfigurer
5、SqlSessionTemplate
二、Spring整合MyBatis应用
1、功能描述
2、创建工程
3、数据库表结构及数据记录
4、实例对象
5、配置文件
6、测试执行，输出结果
课前默写
作业
面试题

回顾：

1. 掌握MyBatis动态SQL
2. 掌握MyBatis关联映射
3. 掌握MyBatis延迟加载
4. 掌握MyBatis缓存

今天任务

1. Spring整合MyBatis框架的配置详解
2. Spring整合MyBatis框架的具体应用

教学目标

1. 掌握Spring整合MyBatis框架的配置详解
2. 掌握Spring整合MyBatis框架的具体应用

一. Spring整合MyBatis

Spring在整合MyBatis时，需要一些jar包以及API，接下来介绍常用的jar包以及API

1、mybatis-spring.jar

整合spring与mybatis框架的工具包

2、SqlSessionFactoryBean

通过分析整合示例中的配置文件，我们可以知道配置的bean其实是成树状结构的，而在树的最顶层是类型为org.mybatis.spring.SqlSessionFactoryBean的bean，它将其他相关bean组装在了一起，那么，我们的分析就从此类开始。

SqlSessionFactoryBean这个类实现了三个接口，一个是InitializingBean，另一个是FactoryBean，还有就是ApplicationListener接口。下面说明一下实现了这三个接口的有什么作用

1. InitializingBean接口：实现了这个接口，那么当bean初始化的时候，spring就会调用该接口的实现类的afterPropertiesSet方法，去实现当spring初始化该Bean的时候所需要的逻辑。
2. FactoryBean接口：实现了该接口的类，在调用getBean的时候会返回该工厂返回的实例对象，也就是再调一次getObject方法返回工厂的实例。
3. ApplicationListener接口：实现了该接口，如果注册了该监听的话，那么就可以了监听到Spring的一些事件，然后做相应的处理

2.1、SqlSessionFactoryBean的初始化

```
public void afterPropertiesSet() throws Exception {
  notNull(dataSource, "Property 'dataSource' is required");
  notNull(sqlSessionFactoryBuilder, "Property 'sqlSessionFactoryBuilder' is
required");
  this.sqlSessionFactory = buildSqlSessionFactory();
}
```

从中我们可以看到，sqlSessionFactory的实例化便在这个方法里面实例化，buildSqlSessionFactory()方法会对我们的sqlSessionFactory做定制的初始化，初始化sqlSessionFactory有两种方式，一种是我们直接通过property直接注入到该实例中，另一种是通过解析xml的方式，就是我们在configuration.xml里面的配置，根据这些配置做了相应的初始化操作，里面也是一些标签的解析属性的获取，操作，和Spring的默认标签解析有点类似。

从buildSqlSessionFactory函数中可以看到，尽管我们还是习惯于将MyBatis的配置与Spring的配置独立出来，但是，这并不代表Spring中的配置不支持直接配置。也就是说，你完全可以取消配置中的configLocation属性，而把其中的属性直接写在SqlSessionFactoryBean中。配置文件还可以支持其他多种属性的配置，如configLocation、objectFactory、objectWrapperFactory、typeAliasesPackage、typeAliases、typeHandlersPackage、plugins、typeHandlers、transactionFactory、databaseIdProvider、mapperLocations。但是，为了体现Spring更强大的兼容性，Spring还整合了MyBatis中其他属性的注入，并通过实例configuration来承载每一步所获取的信息并最终使用sqlSessionFactoryBuilder实例根据解析到的configuration创建SqlSessionFactory实例。

2.2、获取SqlSessionFactoryBean实例

```
public SqlSessionFactory getObject() throws Exception {
  if (this.sqlSessionFactory == null) {
    afterPropertiesSet();
  }
  return this.sqlSessionFactory;
}
```

在给dao注入sqlSessionFactory的时候，依赖填写SqlSessionFactoryBean的实例就可以了。

3、MapperFactoryBean

首先，我们需要从Mybatis官网上下载Mybatis-Spring的jar包添加到我们项目的类路径下，当然也需要添加Mybatis的相关jar包和Spring的相关jar包。我们知道在Mybatis的所有操作都是基于一个SqlSession的，而SqlSession是由SqlSessionFactory来产生的，SqlSessionFactory又是由SqlSessionFactoryBuilder来生成的。但是Mybatis-Spring是基于SqlSessionFactoryBean的。在使用Mybatis-Spring的时候，我们也需要SqlSession，而且这个SqlSession是内嵌在程序中的，一般不需要我们直接访问。SqlSession也是由SqlSessionFactory来产生的，但是Mybatis-Spring给我们封装了一个SqlSessionFactoryBean，在这个bean里面还是通过SqlSessionFactoryBuilder来建立对应的SqlSessionFactory，进而获取到对应的SqlSession。通过SqlSessionFactoryBean我们可以通过对其指定一些属性来提供Mybatis的一些配置信息。所以接下来我们需要在Spring的applicationContext配置文件中定义一个SqlSessionFactoryBean。

Xml代码：

```
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
<property name="dataSource" ref="dataSource" />
       <property name="mapperLocations"
              value="classpath:com/qf/mybatis/mappers/*Mapper.xml" />
       <property name="typeAliasesPackage" value="com.qf.model" />
</bean>
```

在定义SqlSessionFactoryBean的时候，dataSource属性是必须指定的，它表示用于连接数据库的数据源。当然，我们也可以指定一些其他的属性，下面简单列举几个：

- mapperLocations：它表示我们的Mapper文件存放的位置，当我们的Mapper文件跟对应的Mapper接口处于同一位置的时候可以不用指定该属性的值。
- configLocation：用于指定Mybatis的配置文件位置。如果指定了该属性，那么会以该配置文件的内容作为配置信息构建对应的SqlSessionFactoryBuilder，但是后续属性指定的内容会覆盖该配置文件里面指定的对应内容。
- typeAliasesPackage：它一般对应我们的实体类所在的包，这个时候会自动取对应包中不包括包名的简单类名作为包括包名的别名。多个package之间可以用逗号或者分号等来进行分隔。
- typeAliases：数组类型，用来指定别名的。指定了这个属性后，Mybatis会把这个类型的短名称作为这个类型的别名，前提是该类上没有标注@Alias注解，否则将使用该注解对应的值作为此种类型的别名。

Xml代码

```
<property name="typeAliases">
     <array>
         <value>com.qf.mybatis.model.Blog</value>
         <value>com.qf.mybatis.model.Comment</value>
     </array>
 </property>
```

- plugins：数组类型，用来指定Mybatis的Interceptor。
- typeHandlersPackage：用来指定TypeHandler所在的包，如果指定了该属性，SqlSessionFactoryBean会自动把该包下面的类注册为对应的TypeHandler。多个package之间可以用逗号或者分号等来进行分隔。
- typeHandlers：数组类型，表示TypeHandler。

接下来就是在Spring的applicationContext文件中定义我们想要的Mapper对象对应的MapperFactoryBean了。通过MapperFactoryBean可以获取到我们想要的Mapper对象。MapperFactoryBean实现了Spring的FactoryBean接口，所以MapperFactoryBean是通过FactoryBean接口中定义的getObject方法来获取对应的Mapper对象的。在定义一个MapperFactoryBean的时候有两个属性需要我们注入，一个是Mybatis-Spring用来生成实现了SqlSession接口的SqlSessionTemplate对象的sqlSessionFactory；另一个就是我们所要返回的对应的Mapper接口了。

定义好相应Mapper接口对应的MapperFactoryBean之后，我们就可以把我们对应的Mapper接口注入到由Spring管理的bean对象中了，比如Service bean对象。这样当我们需要使用到相应的Mapper接口时，MapperFactoryBean会从它的getObject方法中获取对应的Mapper接口，而getObject内部还是通过我们注入的属性调用SqlSession接口的getMapper(Mapper接口)方法来返回对应的Mapper接口的。这样就通过把SqlSessionFactory和相应的Mapper接口交给Spring管理实现了Mybatis跟Spring的整合。

Spring的applicationContext.xml配置文件：

Xml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:context="http://www.springframework.org/schema/context"
     xmlns:mvc="http://www.springframework.org/schema/mvc"
     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
      http://www.springframework.org/schema/mvc
      http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">
     <context:component-scan base-package="com.tiantian.mybatis"/>
     <context:property-placeholder location="classpath:config/jdbc.properties"/>
     <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
       <property name="driverClassName" value="${jdbc.driver}" />
        <property name="url" value="${jdbc.url}" />
```

```
        <property name="username" value="${jdbc.username}" />
        <property name="password" value="${jdbc.password}" />
    </bean>

    <bean id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="mapperLocations"
value="classpath:com/tiantian/mybatis/mapper/*.xml"/>
        <property name="typeAliasesPackage" value="com.qf.mybatis.model" />
    </bean>

    <bean id="blogMapper" class="org.mybatis.spring.mapper.MapperFactoryBean">
        <property name="mapperInterface"
            value="com.tiantian.mybatis.mapper.BlogMapper" />
        <property name="sqlSessionFactory" ref="sqlSessionFactory" />
    </bean>
</beans>
```

BlogMapper.xml文件：

Xml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="com.qf.mybatis.mapper.BlogMapper">
<!--  新增记录  -->
    <insert id="insertBlog" parameterType="Blog" useGeneratedKeys="true"
keyProperty="id">
        insert into t_blog(title,content,owner) values(#{title},#{content},#
{owner})
    </insert>
<!--  查询单条记录  -->
    <select id="selectBlog" parameterType="int" resultMap="BlogResult">
        select * from t_blog where id = #{id}
    </select>
<!--  修改记录  -->
    <update id="updateBlog" parameterType="Blog">
   update t_blog set title = #{title},content = #{content},owner = #{owner} where
id = #{id}
    </update>
<!--  查询所有记录  -->
    <select id="selectAll" resultType="Blog">
        select * from t_blog
    </select>
<!--  删除记录  -->
    <delete id="deleteBlog" parameterType="int">
```

```
        delete from t_blog where id = #{id}
    </delete>
</mapper>
```

BlogMapper.java：

Java代码：

```
publicinterface BlogMapper {
    public Blog selectBlog(int id);
     publicvoid insertBlog(Blog blog);
     publicvoid updateBlog(Blog blog);
     publicvoid deleteBlog(int id);
     public List<Blog> selectAll();
}
```

BlogServiceImpl.java：

Java代码

```
@Service
publicclass BlogServiceImpl implements BlogService {

    private BlogMapper blogMapper;

    publicvoid deleteBlog(int id) {
        blogMapper.deleteBlog(id);
    }

   public Blog find(int id) {
   returnblogMapper.selectBlog(id);
    }

    public List<Blog> find() {
        returnblogMapper.selectAll();
    }

    publicvoid insertBlog(Blog blog) {
        blogMapper.insertBlog(blog);
    }

    publicvoid updateBlog(Blog blog) {
        blogMapper.updateBlog(blog);
    }

    public BlogMapper getBlogMapper() {
        returnblogMapper;
```

```
    }

    @Resource
    publicvoid setBlogMapper(BlogMapper blogMapper) {
       this.blogMapper = blogMapper;
    }

}
```

4、MapperScannerConfigurer

利用上面的方法进行整合的时候，我们有一个Mapper就需要定义一个对应的MapperFactoryBean，当我们的Mapper比较少的时候，这样做也还可以，但是当我们的Mapper相当多时我们再这样定义一个个Mapper对应的MapperFactoryBean就显得速度比较慢了。为此Mybatis-Spring为我们提供了一个叫做MapperScannerConfigurer的类，通过这个类Mybatis-Spring会自动为我们注册Mapper对应的MapperFactoryBean对象。

如果我们需要使用MapperScannerConfigurer来帮我们自动扫描和注册Mapper接口的话我们需要在Spring的applicationContext配置文件中定义一个MapperScannerConfigurer对应的bean。对于MapperScannerConfigurer而言有一个属性是我们必须指定的，那就是basePackage。basePackage是用来指定Mapper接口文件所在的基包的，在这个基包或其所有子包下面的Mapper接口都将被搜索到。多个基包之间可以使用逗号或者分号进行分隔。最简单的MapperScannerConfigurer定义就是只指定一个basePackage属性，如：

Xml代码

```
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
   <property name="basePackage" value="com.tiantian.mybatis.mapper" />
</bean>
```

这样MapperScannerConfigurer就会扫描指定基包下面的所有接口，并把它们注册为一个个MapperFactoryBean对象。当使用MapperScannerConfigurer加basePackage属性的时候，我们上面例子的applicationContext配置文件将变为这样：

Xml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:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    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
     http://www.springframework.org/schema/mvc
     http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">
```

```
    <context:component-scan base-package="com.qf.mybatis" />
    <context:property-placeholder location="classpath:config/jdbc.properties" />

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
       destroy-method="close">
       <property name="driverClassName" value="${jdbc.driver}" />
       <property name="url" value="${jdbc.url}" />
       <property name="username" value="${jdbc.username}" />
       <property name="password" value="${jdbc.password}" />
    </bean>
    <bean id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
       <property name="dataSource" ref="dataSource" />
     <property name="mapperLocations"
value="classpath:com/qf/mybatis/mapper/*.xml" />
       <property name="typeAliasesPackage" value="com.qf.mybatis.model" />
    </bean>
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
       <property name="basePackage" value="com.qf.mybatis.mapper" />
    </bean>
 </beans>
```

有时候我们指定的基包下面的并不全是我们定义的Mapper接口，为此MapperScannerConfigurer还为我们提供了另外两个可以缩小搜索和注册范围的属性。一个是annotationClass，另一个是markerInterface。

- annotationClass：当指定了annotationClass的时候，MapperScannerConfigurer将只注册使用了annotationClass注解标记的接口。
- markerInterface：markerInterface是用于指定一个接口的，当指定了markerInterface之后，MapperScannerConfigurer将只注册继承自markerInterface的接口。

如果上述两个属性都指定了的话，那么MapperScannerConfigurer将取它们的并集，而不是交集。即使用了annotationClass进行标记或者继承自markerInterface的接口都将被注册为一个MapperFactoryBean。

现在假设我们的Mapper接口都继承了一个SuperMapper接口，那么我们就可以这样来定义我们的MapperScannerConfigurer。

Xml代码

```
 <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="basePackage" value="com.qf.mybatis.mapper" />
    <property name="markerInterface" value="com.qf.mybatis.mapper.SuperMapper"/>
 </bean>
```

如果是都使用了注解MybatisMapper标记的话，那么我们就可以这样来定义我们的MapperScannerConfigurer。

Xml代码

```
 <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="basePackage" value="com.qfmybatis.mapper" />
    <property name="annotationClass"
value="com.qfmybatis.annotation.MybatisMapper"/>
 </bean>
```

除了用于缩小注册Mapper接口范围的属性之外，我们还可以指定一些其他属性，如：

- sqlSessionFactory：**这个属性已经废弃**。当我们使用了多个数据源的时候我们就需要通过sqlSessionFactory来指定在注册MapperFactoryBean的时候需要使用的SqlSessionFactory，因为在没有指定sqlSessionFactory的时候，会以Autowired的方式自动注入一个。换言之当我们只使用一个数据源的时候，即只定义了一个SqlSessionFactory的时候我们就可以不给MapperScannerConfigurer指定SqlSessionFactory。
- sqlSessionFactoryBeanName：它的功能跟sqlSessionFactory是一样的，只是它指定的是定义好的SqlSessionFactory对应的bean名称。
- sqlSessionTemplate：**这个属性已经废弃**。它的功能也是相当于sqlSessionFactory的，因为就像前面说的那样，MapperFactoryBean最终还是使用的SqlSession的getMapper方法取的对应的Mapper对象。当定义有多个SqlSessionTemplate的时候才需要指定它。对于一个MapperFactoryBean来说SqlSessionFactory和SqlSessionTemplate只需要其中一个就可以了，当两者都指定了的时候，SqlSessionFactory会被忽略。
- sqlSessionTemplateBeanName：指定需要使用的sqlSessionTemplate对应的bean名称。

注意：由于使用sqlSessionFactory和sqlSessionTemplate属性时会使一些内容在PropertyPlaceholderConfigurer之前加载，导致在配置文件中使用到的外部属性信息无法被及时替换而出错，因此官方现在新的Mybatis-Spring中已经把sqlSessionFactory和sqlSessionTemplate属性废弃了，推荐大家使用sqlSessionFactoryBeanName属性和sqlSessionTemplateBeanName属性。

Xml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:context="http://www.springframework.org/schema/context"
     xmlns:mvc="http://www.springframework.org/schema/mvc"
     xmlns:mybatis="http://www.mybatis.org/schema/mybatis"
     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
      http://www.mybatis.org/schema/mybatis
     http://www.mybatis.org/schema/mybatis/mybatis-spring.xsd
     http://www.springframework.org/schema/mvc
      http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

     <context:component-scan base-package="com.qf.mybatis" />
```

```
    <context:property-placeholder location="classpath:config/jdbc.properties" />

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
       destroy-method="close">
       <property name="driverClassName" value="${jdbc.driver}" />
       <property name="url" value="${jdbc.url}" />
       <property name="username" value="${jdbc.username}" />
       <property name="password" value="${jdbc.password}" />
    </bean>
    <bean id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
       <property name="dataSource" ref="dataSource" />
     <property name="mapperLocations"
value="classpath:com/qf/mybatis/mapper/*.xml" />
       <property name="typeAliasesPackage" value="com.qf.mybatis.model" />
    </bean>
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
       <property name="basePackage" value="com.qf.mybatis.mapper" />
       <property name="markerInterface"
value="com.qf.mybatis.mapper.SuperMapper"/>
       <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"/>
    </bean>
 </beans>
```

5、SqlSessionTemplate

除了上述整合之后直接使用Mapper接口之外，Mybatis-Spring还为我们提供了一种直接使用SqlSession的方式。Mybatis-Spring为我们提供了一个实现了SqlSession接口的SqlSessionTemplate类，它是线程安全的，可以被多个Dao同时使用。同时它还跟Spring的事务进行了关联，确保当前被使用的SqlSession是一个已经和Spring的事务进行绑定了的。而且它还可以自己管理Session的提交和关闭。当使用了Spring的事务管理机制后，SqlSession还可以跟着Spring的事务一起提交和回滚。

使用SqlSessionTemplate时我们可以在Spring的applicationContext配置文件中如下定义：

```
<bean id=*"sqlSession"* class=*"org.mybatis.spring.SqlSessionTemplate"*>
      <constructor-arg index=*"0"* ref=*"sqlSessionFactory"* />
   </bean>
```

这样我们就可以通过Spring的依赖注入在Dao中直接使用SqlSessionTemplate来编程了，这个时候我们的Dao可能是这个样子：

Java代码

```
@Repository
publicclass BlogDaoImpl implements BlogDao {

    private SqlSessionTemplate sqlSessionTemplate;
```

```
    publicvoid deleteBlog(int id) {
        sqlSessionTemplate.delete("com.qf.mybatis.mapper.BlogMapper.deleteBlog",
id);
    }

    public Blog find(int id) {
 returnsqlSessionTemplate.selectOne("com.qf.mybatis.mapper.BlogMapper.selectBlog",
id);
    }

   public List<Blog> find() {

 returnthis.sqlSessionTemplate.selectList("com.qf.mybatis.mapper.BlogMapper.select
All");
    }

    publicvoid insertBlog(Blog blog) {

 this.sqlSessionTemplate.insert("com.qf.mybatis.mapper.BlogMapper.insertBlog",
blog);
    }

    publicvoid updateBlog(Blog blog) {

 this.sqlSessionTemplate.update("com.qf.mybatis.mapper.BlogMapper.updateBlog",
blog);
   }

    public SqlSessionTemplate getSqlSessionTemplate() {
        returnsqlSessionTemplate;
    }

    @Resource
  publicvoid setSqlSessionTemplate(SqlSessionTemplate sqlSessionTemplate) {
        this.sqlSessionTemplate = sqlSessionTemplate;
    }

 }
```

二、Spring整合MyBatis应用

前面讲到有关 mybatis 连接数据库，然后进行进行数据增删改查，以及多表联合查询的例子，但很多的项目中，通常会用 spring 这个粘合剂来管理 datasource 等。充分利用 spring 基于接口的编程，以及aop ,ioc 带来的方便。用 spring 来管理 mybatis 与管理 hibernate 有很多类似的地方。在这一节中，我们重点介绍数据源管理以及 bean 的配置。

整个Mybatis与Spring集成示例要完成的步骤如下：

- 功能描述
- 创建工程
- 数据库表结构及数据记录
- 实例对象
- 配置文件
- 测试执行，输出结果

1、功能描述

需要完成这样的一个简单功能，即，指定一个用户（ID=1），查询出这个用户的基本信息，并关联查询这个用户的所有订单。

2、创建工程

首先创建一个工程的名称为：mybatis07-spring，在 src 源代码目录下建立文件夹 config，并将原来的 mybatis 配置文件 Configuration.xml 移动到这个文件夹中, 并在 config 文家夹中建立 Spring 配置文件：applicationContext.xml。

3、数据库表结构及数据记录

在本示例中，用到两个表：用户表和订单表，其结构和数据记录如下：

```
CREATE TABLE `user` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `username` varchar(64) NOT NULL DEFAULT '',
  `mobile` varchar(16) NOT NULL DEFAULT '',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;

-- ----------------------------
-- Records of user
-- ----------------------------
INSERT INTO `user` VALUES ('1', 'qf', '13838009988');
INSERT INTO `user` VALUES ('2', 'saya', '13838009988');
```

订单表结构和数据如下：

```
CREATE TABLE `order` (
  `order_id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `user_id` int(10) unsigned NOT NULL DEFAULT '0',
  `order_no` varchar(16) NOT NULL DEFAULT '',
  `money` float(10,2) unsigned DEFAULT '0.00',
  PRIMARY KEY (`order_id`)
) ENGINE=InnoDB AUTO_INCREMENT=17 DEFAULT CHARSET=utf8;

-- ----------------------------
-- Records of order
-- ----------------------------
INSERT INTO `order` VALUES ('1', '1', '1509289090', '99.90');
```

```
INSERT INTO `order` VALUES ('2', '1', '1519289091', '290.80');
INSERT INTO `order` VALUES ('3', '1', '1509294321', '919.90');
INSERT INTO `order` VALUES ('4', '1', '1601232190', '329.90');
INSERT INTO `order` VALUES ('5', '1', '1503457384', '321.00');
INSERT INTO `order` VALUES ('6', '1', '1598572382', '342.00');
INSERT INTO `order` VALUES ('7', '1', '1500845727', '458.00');
INSERT INTO `order` VALUES ('8', '1', '1508458923', '1200.00');
INSERT INTO `order` VALUES ('9', '1', '1504538293', '2109.00');
INSERT INTO `order` VALUES ('10', '1', '1932428723', '5888.00');
INSERT INTO `order` VALUES ('11', '1', '2390423712', '3219.00');
INSERT INTO `order` VALUES ('12', '1', '4587923992', '123.00');
INSERT INTO `order` VALUES ('13', '1', '4095378812', '421.00');
INSERT INTO `order` VALUES ('14', '1', '9423890127', '678.00');
INSERT INTO `order` VALUES ('15', '1', '7859213249', '7689.00');
INSERT INTO `order` VALUES ('16', '1', '4598450230', '909.20');
```

4、实例对象

用户表和订单表分别对应两个实例对象，分别是：User.java 和 Order.java，它们都在 com.qf.pojo 包中。

User.java代码内容如下：

```
public class User {
    private int id;
    private String username;
    private String mobile;

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getMobile() {
        return mobile;
    }
    public void setMobile(String mobile) {
        this.mobile = mobile;
    }

}
```

Order.java代码内容如下：

```
public class Order {
    private int orderId;
    private String orderNo;
    private float money;
    private int userId;
    private User user;


    public int getUserId() {
        return userId;
    }
    public void setUserId(int userId) {
        this.userId = userId;
    }
    public int getOrderId() {
        return orderId;
    }
    public void setOrderId(int orderId) {
        this.orderId = orderId;
    }
    public User getUser() {
        return user;
    }
    public void setUser(User user) {
        this.user = user;
    }
    public String getOrderNo() {
        return orderNo;
    }
    public void setOrderNo(String orderNo) {
        this.orderNo = orderNo;
    }
    public float getMoney() {
        return money;
    }
    public void setMoney(float money) {
        this.money = money;
    }

}
```

5、配置文件

这个实例中有三个重要的配置文件，它们分别是：applicationContext.xml ， Configuration.xml 以及 UserMaper.xml。

applicationContext.xml 配置文件里最主要的配置：

```
<?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>
    <typeAliases>
        <typeAlias alias="User" type="com.qf.pojo.User" />
        <typeAlias alias="Order" type="com.qf.pojo.Order" />
    </typeAliases>
    <!-- Mybatis和Spring 集成之后,这些可以完全删除（注释掉）,数据库连接的管理交给
Spring 来管理 -->
    <!--
        <environments default="development"> <environment id="development">
        <transactionManager type="JDBC"/> <dataSource type="POOLED"> <property
        name="driver" value="com.mysql.jdbc.Driver"/> <property name="url"
        value="jdbc:mysql://127.0.0.1:3306/qf?characterEncoding=utf8" />
        <property name="username" value="root"/> <property name="password"
        value=""/> </dataSource> </environment> </environments>
    -->
    <mappers>
        <mapper resource="com/qf/maper/UserMaper.xml" />
    </mappers>
</configuration>
```

配置文件 Configuration.xml 的内容如下：

```
<?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:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
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
            http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee-3.0.xsd
            http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.0.xsd"
    default-autowire="byName" default-lazy-init="false">

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
```

```
        <property name="url"
            value="jdbc:mysql://127.0.0.1:3306/qf?characterEncoding=utf8" />
        <property name="username" value="root" />
        <property name="password" value="" />
    </bean>

    <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
        <!--dataSource属性指定要用到的连接池-->
        <property name="dataSource" ref="dataSource" />
        <!--configLocation属性指定mybatis的核心配置文件-->
        <property name="configLocation" value="config/Configuration.xml" />
    </bean>

    <bean id="userMaper" class="org.mybatis.spring.mapper.MapperFactoryBean">
        <!--sqlSessionFactory属性指定要用到的SqlSessionFactory实例-->
        <property name="sqlSessionFactory" ref="sqlSessionFactory" />
        <!--mapperInterface属性指定映射器接口，用于实现此接口并生成映射器对象-->
        <property name="mapperInterface" value="com.qf.maper.UserMaper" />
    </bean>
</beans>
```

UserMaper.xml 用于定义查询和数据对象映射，其内容如下：

```
<?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="com.qf.maper.UserMaper">

        <!-- 为了返回list 类型而定义的returnMap -->
    <resultMap type="User" id="resultUser">
        <id column="id" property="id" />
        <result column="username" property="username" />
        <result column="mobile" property="mobile" />
    </resultMap>

    <!-- User 联合 Order 查询 方法的配置 (多对一的方式)  -->
    <resultMap id="resultUserOrders" type="Order">
        <id property="orderId" column="order_id" />
        <result property="orderNo" column="order_no" />
        <result property="money" column="money" />
        <result property="userId" column="user_id" />

        <association property="user" javaType="User">
            <id property="id" column="id" />
            <result property="username" column="username" />
            <result property="mobile" column="mobile" />
        </association>
    </resultMap>
```

```
    <select id="getUserOrders" parameterType="int" resultMap="resultUserOrders">
       SELECT u.*,o.* FROM `user` u, `order` o
              WHERE u.id=o.user_id AND u.id=#{id}
    </select>

    <select id="getUserById" resultMap="resultUser" parameterType="int">
        SELECT *
        FROM user
        WHERE id=#{id}
    </select>
</mapper>
```

6、测试执行，输出结果

我们创建一个测试类为：Main.java , 就在 src 目录中。其代码如下：

```
import java.util.List;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    private static ApplicationContext ctx;

    static {
        ctx = new ClassPathXmlApplicationContext(
                "config/applicationContext.xml");
    }

    public static void main(String[] args) {
        UserMaper userMaper = (UserMaper) ctx.getBean("userMaper");
        // 测试id=1的用户查询，可根据数据库中的情况修改.
        User user = userMaper.getUserById(1);
        System.out.println("获取用户 ID=1 的用户名：+user.getUsername());

        // 得到文章列表测试
        System.out.println("得到用户id为1的所有订单列表:");
        System.out.println("==============================================");
        List<Order> orders = userMaper.getUserOrders(1);

        for (Order order : orders) {
            System.out.println("订单号：+order.getOrderNo() + "，订单金额：" +
order.getMoney());
        }

    }
```

```
}
```

运行结果如下：

```
log4j:WARN No appenders could be found for logger
(org.springframework.context.support.ClassPathXmlApplicationContext).
log4j:WARN Please initialize the log4j system properly.
获取用户 ID=1 的用户名：qf
得到用户id为1的所有订单列表：
==========================================
订单号：1509289090，订单金额：99.9
订单号：1519289091，订单金额：290.8
订单号：1509294321，订单金额：919.9
订单号：1601232190，订单金额：329.9
订单号：1503457384，订单金额：321.0
订单号：1598572382，订单金额：342.0
订单号：1500845727，订单金额：458.0
订单号：1508458923，订单金额：1200.0
订单号：1504538293，订单金额：2109.0
订单号：1932428723，订单金额：5888.0
订单号：2390423712，订单金额：3219.0
订单号：4587923992，订单金额：123.0
订单号：4095378812，订单金额：421.0
订单号：9423890127，订单金额：678.0
订单号：7859213249，订单金额：7689.0
订单号：4598450230，订单金额：909.2
```

课前默写

```
1. MyBatis动态SQL的使用
2. MyBatis关联映射的配置
```

作业

```
1.  使用Spring整合MyBatis框架改写昨天的作业
```

面试题

```
1. Spring框架与MyBatis框架整合的方案
2. @MapperScan注解的作用
3. Mybatis关联映射的使用
4. ResultType和ResultMap的区别
```