第 3-2 课：如何优雅地使用 MyBatis XML 配置版

MyBatis 是现如今最流行的 ORM 框架之一，我们先来了解一下什么是 ORM 框架。

ORM 框架

对象关系映射（Object Relational Mapping，ORM）模式是一种为了解决面向对象与关系数据库存在的互不匹配的现象的技术。简单的说，ORM 是通过使用描述对象和数据库之间映射的元数据，将程序中的对象自动持久化到关系数据库中。

为什么需要 ORM?

当你开发一个应用程序的时候（不使用 O/R Mapping），可能会写不少数据访问层代码，用来从数据库保存、删除、读取对象信息等；在 DAL 中写了很多的方法来读取对象数据、改变状态对象等任务，而这些代码写起来总是重复的。针对这些问题 ORM 提供了解决方案，简化了将程序中的对象持久化到关系数据库中的操作。

ORM 框架的本质是简化编程中操作数据库的编码，在 Java 领域发展到现在基本上就剩两家最为流行，一个是宣称可以不用写一句 SQL 的 Hibernate，一个是以动态 SQL 见长的 MyBatis，两者各有特点。在企业级系统开发中可以根据需求灵活使用，会发现一个有趣的现象：传统企业大都喜欢使用 Hibernate，而互联网行业通常使用 MyBatis。

MyBatis 介绍

MyBatis 是一款标准的 ORM 框架，被广泛的应用于各企业开发中。MyBatis 最早是 Apache 的一个开源项目 iBatis，2010 年这个项目由 Apache Software Foundation 迁移到了 Google Code，并且改名为 MyBatis，2013 年 11 月又迁移到 Github。从 MyBatis 的迁移史，也可以看出源码托管平台的发展史，GitHub 目前已经成为世界上最大的开源软件托管平台，建议大家多多关注这个全球最大的同性社交网站。

MyBatis 支持普通的 SQL 查询，存储过程和高级映射的优秀持久层框架。MyBatis 消除了几乎所有的 JDBC 代码和参数的手工设置以及对结果集的检索封装。MaBatis 可以使用简单的 XML 或注解用于配置和原始映射，将接口和 Java 的 POJO（Plain Old Java Objects，普通的 Java 对象）映射成数据库中的记录。

作为一款使用广泛的开源软件，它的特点有哪些呢?

优点

- SQL 被统一提取出来，便于统一管理和优化
- SQL 和代码解耦，将业务逻辑和数据访问逻辑分离，使系统的设计更清晰、更易维护、更易单元测试
- 提供映射标签，支持对象与数据库的 ORM 字段关系映射
- 提供对象关系映射标签，支持对象关系组件维护
- 灵活书写动态 SQL，支持各种条件来动态生成不同的 SQL

缺点

- 编写 SQL 语句时工作量很大，尤其是字段多、关联表多时，更是如此
- SQL 语句依赖于数据库，导致数据库移植性差

MyBatis 几个重要的概念

Mapper 配置可以使用基于 XML 的 Mapper 配置文件来实现，也可以使用基于 Java 注解的 MyBatis 注解来实现，甚至可以直接使用 MyBatis 提供的 API 来实现。

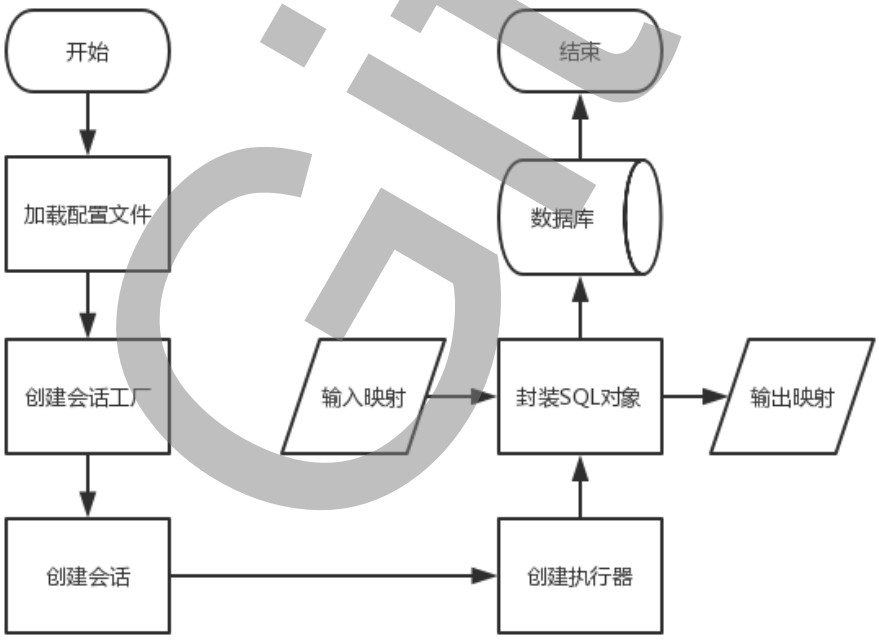
Mapper 接口是指自行定义的一个数据操作接口，类似于通常所说的 DAO 接口。早期的 Mapper 接口需要自定义去实现，现在 MyBatis 会自动为 Mapper 接口创建动态代理对象。Mapper 接口的方法通常与 Mapper 配置文件中的 select、insert、update、delete 等 XML 结点存在一一对应关系。

Executor，MyBatis 中所有的 Mapper 语句的执行都是通过 Executor 进行的，Executor 是 MyBatis 的一个核心接口。

SqlSession，是 MyBatis 的关键对象，是执行持久化操作的独享，类似于 JDBC 中的 Connection，SqlSession 对象完全包含以数据库为背景的所有执行 SQL 操作的方法，它的底层封装了 JDBC 连接，可以用 SqlSession 实例来直接执行被映射的 SQL 语句。

SqlSessionFactory，是 MyBatis 的关键对象，它是单个数据库映射关系经过编译后的内存镜像。SqlSessionFactory 对象的实例可以通过 SqlSessionFactoryBuilder 对象类获得，而 SqlSessionFactoryBuilder 则可以从 XML 配置文件或一个预先定制的 Configuration 的实例构建出。

MyBatis 的工作流程如下：



- 首先加载 Mapper 配置的 SQL 映射文件，或者是注解的相关 SQL 内容。
- 创建会话工厂，MyBatis 通过读取配置文件的信息来构造出会话工厂（SqlSessionFactory）。
- 创建会话。根据会话工厂，MyBatis 就可以通过它来创建会话对象（SqlSession），会话对象是一个接口，该接口中包含了对数据库操作的增、删、改、查方法。
- 创建执行器。因为会话对象本身不能直接操作数据库，所以它使用了一个叫做数据库执行器（Executor）的接口来帮它执行操作。
- 封装 SQL 对象。在这一步，执行器将待处理的 SQL 信息封装到一个对象中（MappedStatement），该对象包括 SQL 语句、输入参数映射信息（Java 简单类型、HashMap 或 POJO）和输出结果映射信息（Java 简单类型、HashMap 或 POJO）。
- 操作数据库。拥有了执行器和 SQL 信息封装对象就使用它们访问数据库了，最后再返回操作结果，结束流程。

在我们具体的使用过程中，就是按照上述的流程来执行。

什么是 MyBatis-Spring-Boot-Starter

mybatis-spring-boot-starter 是 MyBatis 帮助我们快速集成 Spring Boot 提供的一个组件包，使用这个组件可以做到以下几点：

- 构建独立的应用
- 几乎可以零配置
- 需要很少的 XML 配置

> mybatis-spring-boot-starter 依赖于 MyBatis-Spring 和 Spring Boot，最新版 1.3.2 需要 MyBatis-Spring 1.3 以上，Spring Boot 版本 1.5 以上。

注意 mybatis-spring-boot-starter 是 MyBatis 官方开发的 Starter，而不是 Spring Boot 官方开发的启动包，其实是 MyBatis 看 Spring Boot 市场使用度非常高，因此主动开发出 Starter 包进行集成，但这一集成确实解决了很多问题，使用起来比以前简单很多。mybatis-spring-boot-starter 主要提供了两种解决方案，一种是简化后的 XML 配置版，一种是使用注解解决一切问题。

MyBatis 以前只有 XML 配置这种使用的形式，到了后来注解使用特别广泛，MyBatis 也顺应潮流提供了注解的支持，从这里可以看出 MyBatis 一直都跟随着主流技术的变化来完善自己。接下来给大家介绍一下如何使用 XML 版本。

XML 版本保持映射文件的方式，最新版的使用主要体现在不需要实现 Dao 的实现层，系统会自动根据方法名在映射文件中找到对应的 SQL。

初始化脚本

为了方便项目演示，需要在 test 仓库创建 users 表，脚本如下：

```
DROP TABLE IF EXISTS `users`;
CREATE TABLE `users` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT '主键id',
  `userName` varchar(32) DEFAULT NULL COMMENT '用户名',
  `passWord` varchar(32) DEFAULT NULL COMMENT '密码',
  `user_sex` varchar(32) DEFAULT NULL,
  `nick_name` varchar(32) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

关键依赖包

当然任何模式都需要首先引入 mybatis-spring-boot-starter 的 pom 文件，现在最新版本是 1.3.2。

```
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.3.2</version>
</dependency>
```

application 配置

application.properties 添加相关配置：

```
mybatis.config-location=classpath:mybatis/mybatis-config.xml
mybatis.mapper-locations=classpath:mybatis/mapper/*.xml
mybatis.type-aliases-package=com.neo.model

spring.datasource.url=jdbc:mysql://localhost:3306/test?serverTimezone=UTC&useUnico
de=true&characterEncoding=utf-8&useSSL=true
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

其中：

- mybatis.config-location，配置 mybatis-config.xml 路径，mybatis-config.xml 中配置 MyBatis 基础属性；
- mybatis.mapper-locations，配置 Mapper 对应的 XML 文件路径；
- mybatis.type-aliases-package，配置项目中实体类包路径；
- spring.datasource.*，数据源配置。

Spring Boot 启动时数据源会自动注入到 SqlSessionFactory 中，使用 SqlSessionFactory 构建 SqlSessionFactory，再自动注入到 Mapper 中，最后我们直接使用 Mapper 即可。

启动类

在启动类中添加对 Mapper 包扫描 @MapperScan，Spring Boot 启动的时候会自动加载包路径下的 Mapper。

```
@Spring BootApplication
@MapperScan("com.neo.mapper")
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

或者直接在 Mapper 类上面添加注解 @Mapper，建议使用上面那种，不然每个 mapper 加个注解也挺麻烦的。

示例演示

MyBatis 公共属性

mybatis-config.xml 主要配置常用的 typeAliases，设置类型别名，为 Java 类型设置一个短的名字。它只和 XML 配置有关，存在的意义仅在于用来减少类完全限定名的冗余。

```
<configuration>
    <typeAliases>
        <typeAlias alias="Integer" type="java.lang.Integer" />
        <typeAlias alias="Long" type="java.lang.Long" />
        <typeAlias alias="HashMap" type="java.util.HashMap" />
        <typeAlias alias="LinkedHashMap" type="java.util.LinkedHashMap" />
        <typeAlias alias="ArrayList" type="java.util.ArrayList" />
        <typeAlias alias="LinkedList" type="java.util.LinkedList" />
    </typeAliases>
</configuration>
```

这样我们在使用 Mapper.xml 的时候，需要引入可以直接这样写：

```
resultType="Integer"
//或者
parameterType="Long"
```

添加 User 的映射文件

第一步，指明对应文件的 Mapper 类地址：

```
<mapper namespace="com.neo.mapper.UserMapper" >
```

第二部，配置表结构和类的对应关系：

```
<resultMap id="BaseResultMap" type="com.neo.model.User" >
    <id column="id" property="id" jdbcType="BIGINT" />
    <result column="userName" property="userName" jdbcType="VARCHAR" />
    <result column="passWord" property="passWord" jdbcType="VARCHAR" />
    <result column="user_sex" property="userSex" javaType="com.neo.enums.UserSexEn
um"/>
    <result column="nick_name" property="nickName" jdbcType="VARCHAR" />
</resultMap>
```

这里为了更好的贴近工作情况，将类的两个字段和数据库字段设置为不一致，其中一个使用了枚举。使用枚举有一个非常大的优点，插入此属性的数据会自动进行校验，如果不是枚举的内容会报错。

第三步，写具体的 SQL 语句，比如这样：

```
<select id="getAll" resultMap="BaseResultMap"  >
   SELECT
   *
   FROM users
</select>
```

MyBatis XML 有一个特点是可以复用 XML，比如我们公用的一些 XML 片段可以提取出来，在其他 SQL 中去引用。例如：

```
<sql id="Base_Column_List" >
    id, userName, passWord, user_sex, nick_name
</sql>

<select id="getAll" resultMap="BaseResultMap"  >
   SELECT
   <include refid="Base_Column_List" />
   FROM users
</select>
```

这个例子就是，上面定义了需要查询的表字段，下面 SQL 使用 include 引入，避免了写太多重复的配置内容。

下面是常用的增、删、改、查的例子：

```
<select id="getOne" parameterType="Long" resultMap="BaseResultMap" >
    SELECT
   <include refid="Base_Column_List" />
   FROM users
   WHERE id = #{id}
</select>

<insert id="insert" parameterType="com.neo.model.User" >
   INSERT INTO
        users
        (userName,passWord,user_sex)
    VALUES
        (#{userName}, #{passWord}, #{userSex})
</insert>

<update id="update" parameterType="com.neo.model.User" >
   UPDATE
        users
   SET
    <if test="userName != null">userName = #{userName},</if>
    <if test="passWord != null">passWord = #{passWord},</if>
    nick_name = #{nickName}
   WHERE
        id = #{id}
</update>

<delete id="delete" parameterType="Long" >
   DELETE FROM
         users
   WHERE
         id =#{id}
</delete>
```

上面 update 的 SQL 使用了 if 标签，可以根据不同的条件生产动态 SQL，这就是 MyBatis 最大的特点。

编写 Dao 层的代码

```
public interface UserMapper {

    List<UserEntity> getAll();

    UserEntity getOne(Long id);

    void insert(UserEntity user);

    void update(UserEntity user);

    void delete(Long id);
}
```

> 注意：这里的方法名需要和 XML 配置中的 id 属性一致，不然会找不到方法去对应执行的 SQL。

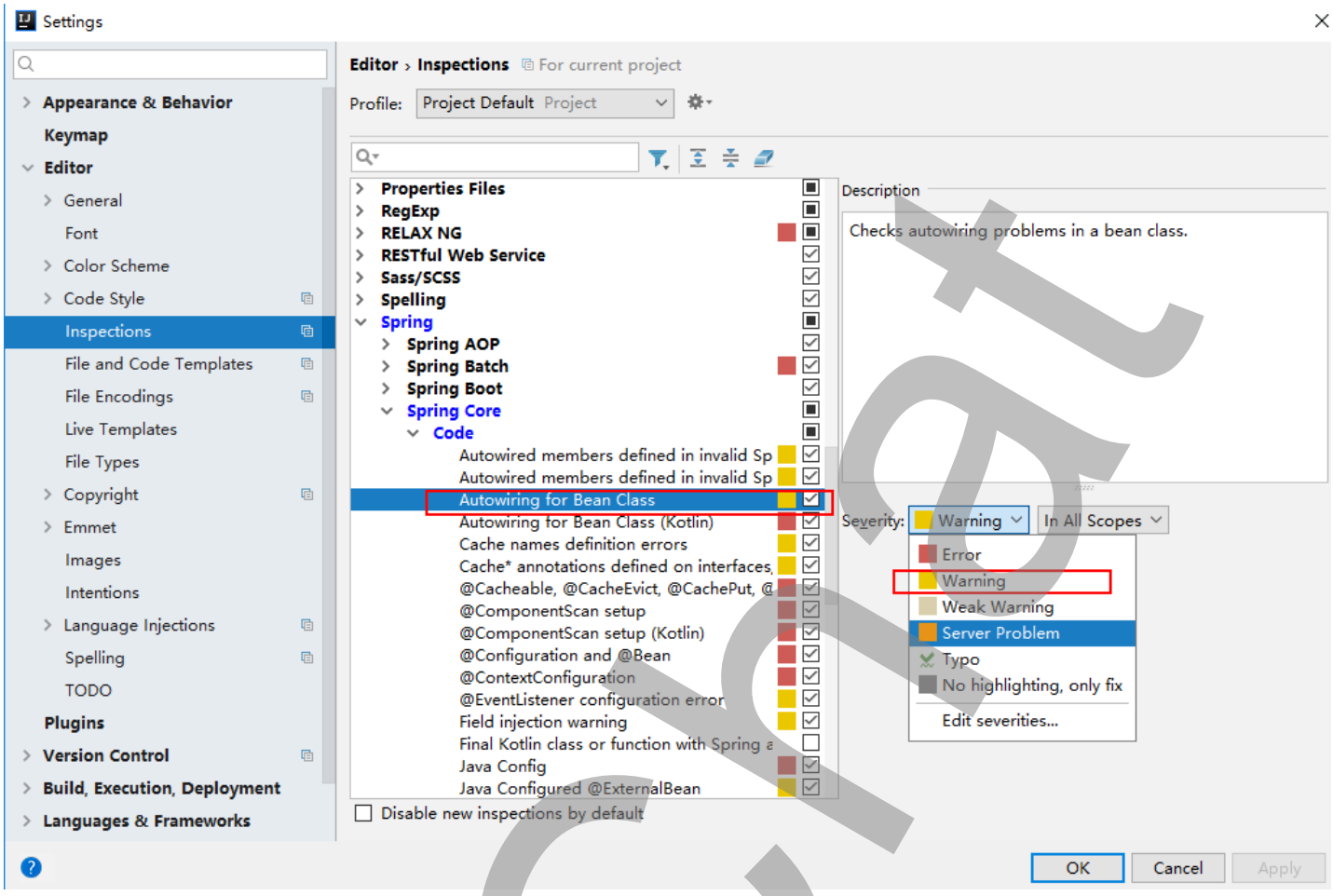
测试使用

按照 Spring 一贯使用形式，直接将对应的 Mapper 注入即可。

```
@Resource
private UserMapper userMapper;
```

如果使用的是 Idea，这块的注解经常会报“could not autowire”，Eclipse 却没有问题，其实代码是正确的，这是 Idea 的误报。可以选择降低 Autowired 检测的级别，不要提示就好。

在 File | Settings | Editor | Inspections 选项中使用搜索功能找到 Autowiring for Bean Class，将 Severity 的级别由之前的 error 改成 warning 即可。



接下来直接使用 userMapper 进行数据库操作即可。

```
@Test
public void testUser()  {
    //增加
    userMapper.insert(new User("aa", "a123456", UserSexEnum.MAN));
    //删除
    int count=userMapper.delete(2l);
    User user = userMapper.getOne(1l);
    user.setNickName("smile");
    //修改
    userMapper.update(user);
    //查询
    List<User> users = userMapper.getAll();
}
```

在示例代码中，写了两份的使用示例，一个是 Test，一个在 Controller 层，方便大家下载查看。

分页查询

多条件分页查询是实际工作中最常使用的功能之一，MyBatis 特别擅长处理这类的问题。在实际工作中，会对分页进行简单的封装，方便前端使用。另外在 Web 开发规范使用中，Web 层的参数会以 param 为后缀的

对象进行传参，以 result 结尾的实体类封装返回的数据。

下面给大家以 User 多条件分页查询为例进行讲解。

先定义一个分页的基础类：

```
public class PageParam {
    private int beginLine;       //起始行
    private Integer pageSize = 3;
    private Integer currentPage=0;     // 当前页
    //getter setter省略
    public int getBeginLine() {
        return pageSize*currentPage;//自动计算起始行
    }
}
```

默认每页 3 条记录，可以根据前端传参进行修改。

user 的查询条件参数类继承分页基础类：

```
public class UserParam extends PageParam{
    private String userName;
    private String userSex;
    //getter setter省略
}
```

接下来配置具体的 SQL，先将查询条件提取出来。

```
<sql id="Base_Where_List">
    <if test="userName != null  and userName != ''">
        and userName = #{userName}
    </if>
    <if test="userSex != null and userSex != ''">
        and user_sex = #{userSex}
    </if>
</sql>
```

从对象 UserParam 中获取分页信息和查询条件，最后进行组合。

```
<select id="getList" resultMap="BaseResultMap" parameterType="com.neo.param.UserPa
ram">
    select
    <include refid="Base_Column_List" />
    from users
    where 1=1
    <include refid="Base_Where_List" />
    order by id desc
    limit #{beginLine} , #{pageSize}
</select>
```

前端需要展示总共的页码，因此需要统计出查询结果的总数。

```
<select id="getCount" resultType="Integer" parameterType="com.neo.param.UserParam"
>
    select
    count(1)
    from users
    where 1=1
    <include refid="Base_Where_List" />
</select>
```

Mapper 中定义的两个方法和配置文件相互对应。

```
public interface UserMapper {
    List<UserEntity> getList(UserParam userParam);
    int getCount(UserParam userParam);
}
```

具体使用：

```
@Test
public void testPage() {
    UserParam userParam=new UserParam();
    userParam.setUserSex("WOMAN");
    userParam.setCurrentPage(1);
    List<UserEntity> users=userMapper.getList(userParam);
    long count=userMapper.getCount(userParam);
    Page page = new Page(userParam,count,users);
    System.out.println(page);
}
```

在实际使用中，只需要传入 CurrentPage 参数即可，默认 0 就是第一页，传 1 就是第二页的内容，最后将结果封装为 Page 返回给前端。

```
public class Page<E> implements Serializable {
    private int currentPage = 0; //当前页数
    private long totalPage;       //总页数
    private long totalNumber;    //总记录数
    private List<E> list;        //数据集
}
```

Page 将分页信息和数据信息进行封装，方便前端显示第几页、总条数和数据，这样分页功能就完成了。

多数据源处理

接下来为大家介绍如何使用 MyBatis 配置多数据源使用。

配置文件

首先我们需要配置两个不同的数据源：

```
mybatis.config-location=classpath:mybatis/mybatis-config.xml

spring.datasource.one.jdbc-url=jdbc:mysql://localhost:3306/test1?serverTimezone=UT
C&useUnicode=true&characterEncoding=utf-8&useSSL=true
spring.datasource.one.username=root
spring.datasource.one.password=root
spring.datasource.one.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.two.jdbc-url=jdbc:mysql://localhost:3306/test2?serverTimezone=UT
C&useUnicode=true&characterEncoding=utf-8&useSSL=true
spring.datasource.two.username=root
spring.datasource.two.password=root
spring.datasource.two.driver-class-name=com.mysql.cj.jdbc.Driver
```

> 注意，需要提前在 test1 和 test2 库中创建好 User 表结构。

第一个数据源以 spring.datasource.one.* 为前缀连接数据库 test1，第二个数据源以 spring.datasource.two.* 为前缀连接数据库 test2。

同时需要将上述的 UserMapper.xml 文件复制两份到 resources/mybatis/mapper/one 和 resources/mybatis/mapper/two 目录下各一份。

数据源配置

为两个数据源创建不同的 Mapper 包路径，将以前的 UserMapper 复制到包 com.neo.mapper.one 和 com.neo.mapper.two 路径下，并且分别重命名为：User1Mapper、User2Mapper。

配置第一个数据源，新建 DataSource1Config。

首先加载配置的数据源：

```
@Bean(name = "oneDataSource")
@ConfigurationProperties(prefix = "spring.datasource.one")
@Primary
public DataSource testDataSource() {
    return DataSourceBuilder.create().build();
}
```

> 注意，在多数据源中只能指定一个 @Primary 作为默认的数据源使用。

根据创建的数据源，构建对应的 SqlSessionFactory。

```
@Bean(name = "oneSqlSessionFactory")
@Primary
public SqlSessionFactory testSqlSessionFactory(@Qualifier("oneDataSource") DataSou
rce dataSource) throws Exception {
    SqlSessionFactoryBean bean = new SqlSessionFactoryBean();
    bean.setDataSource(dataSource);
    bean.setMapperLocations(new PathMatchingResourcePatternResolver().getResources
("classpath:mybatis/mapper/one/*.xml"));
    return bean.getObject();
}
```

代码中需要指明需要加载的 Mapper xml 文件。

同时将数据源添加到事务中。

```
@Bean(name = "oneTransactionManager")
@Primary
public DataSourceTransactionManager testTransactionManager(@Qualifier("oneDataSour
ce") DataSource dataSource) {
    return new DataSourceTransactionManager(dataSource);
}
```

接下来将上面创建的 SqlSessionFactory 注入，创建我们在 Mapper 中需要使用的 SqlSessionTemplate。

```
@Bean(name = "oneSqlSessionTemplate")
@Primary
public SqlSessionTemplate testSqlSessionTemplate(@Qualifier("oneSqlSessionFactory"
) SqlSessionFactory sqlSessionFactory) throws Exception {
    return new SqlSessionTemplate(sqlSessionFactory);
}
```

最后将上面创建的 SqlSessionTemplate 注入到对应的 Mapper 包路径下，这样这个包下面的 Mapper 都会使用第一个数据源来进行数据库操作。

```
@Configuration
@MapperScan(basePackages = "com.neo.mapper.one", sqlSessionTemplateRef  = "oneSqlS
essionTemplate")
public class OneDataSourceConfig {
 ...
}
```

- basePackages 指明 Mapper 地址。
- sqlSessionTemplateRef 指定 Mapper 路径下注入的 sqlSessionTemplate。

第二个数据源配置

DataSource2Config 的配置和上面类似，方法上需要去掉 @Primary 注解，替换对应的数据源和 Mapper 路径即可。下面是 DataSource2Config 完整示例：

```
@Configuration
@MapperScan(basePackages = "com.neo.mapper.two", sqlSessionTemplateRef  = "twoSqlS
essionTemplate")
public class DataSource2Config {

    @Bean(name = "twoDataSource")
    @ConfigurationProperties(prefix = "spring.datasource.two")
    public DataSource testDataSource() {
        return DataSourceBuilder.create().build();
    }

    @Bean(name = "twoSqlSessionFactory")
    public SqlSessionFactory testSqlSessionFactory(@Qualifier("twoDataSource") Dat
aSource dataSource) throws Exception {
        SqlSessionFactoryBean bean = new SqlSessionFactoryBean();
        bean.setDataSource(dataSource);
        bean.setMapperLocations(new PathMatchingResourcePatternResolver().getResou
rces("classpath:mybatis/mapper/two/*.xml"));
        return bean.getObject();
    }

    @Bean(name = "twoTransactionManager")
    public DataSourceTransactionManager testTransactionManager(@Qualifier("twoData
Source") DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }

    @Bean(name = "twoSqlSessionTemplate")
    public SqlSessionTemplate testSqlSessionTemplate(@Qualifier("twoSqlSessionFact
ory") SqlSessionFactory sqlSessionFactory) throws Exception {
        return new SqlSessionTemplate(sqlSessionFactory);
    }

}
```

从上面的步骤我们可以总结出来，创建多数据源的过程就是：首先创建 DataSource，注入到 SqlSessionFactory 中，再创建事务，将 SqlSessionFactory 注入到创建的 SqlSessionTemplate 中，最后将 SqlSessionTemplate 注入到对应的 Mapper 包路径下。其中需要指定分库的 Mapper 包路径。

> 注意，在多数据源的情况下，我们不需要在启动类添加：@MapperScan("com.xxx.mapper") 的注解。

这样 MyBatis 多数据源的配置就完成了，如果有更多的数据源请参考第二个数据源的配置即可。

测试

配置好多数据源之后，在项目中想使用哪个数据源就把对应数据源注入到类中使用即可。

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class UserMapperTest {
    @Autowired
    private User1Mapper user1Mapper;
    @Autowired
    private User2Mapper user2Mapper;

    @Test
    public void testInsert() throws Exception {
        user1Mapper.insert(new User("aa111", "a123456", UserSexEnum.MAN));
        user1Mapper.insert(new User("bb111", "b123456", UserSexEnum.WOMAN));
        user2Mapper.insert(new User("cc222", "b123456", UserSexEnum.MAN));
    }
}
```

上面的测试类中注入了两个不同的 Mapper，对应了不同的数据源。在第一个数据源中插入了两条数据，第二个数据源中插入了一条信息，运行测试方法后查看数据库1有两条数据，数据库2有一条数据，证明多数据源测试成功。

总结

这节课介绍了 ORM 框架 和 MyBatis 框架相关概念介绍，以用户数据为例演示了 MyBatis 的增、删、改、查，以及分页查询、多数据源处理等常见场景。通过上面的示例可以发现 MyBatis 将执行 SQL 和代码做了隔离，保证代码处理和 SQL 的相对独立，层级划分比较清晰，MyBatis 对动态 SQL 支持非常友好，可以在 XML 文件中复用代码高效编写动态 SQL。

> 点击这里下载源码。