电商后台项目核心功能及多数据源架构实战

图灵：楼兰

前面这几节课，主要是带大家扫除一些基础的开发问题，为后面理解项目设计做铺垫。这一节课主要来理解电商管理后台。电商管理后台跟其他管理系统一样，本质上是一个纯粹的CRUD管理系统，没有什么技术难度。这一类系统通常都不是项目的核心模块，但是却是串联核心业务必不可少的辅助模块。对于后台管理系统，我们今天还是关注他的技术路线，至于业务功能，大家可以随着以后各个业务线的具体设计过程再继续深入。

今天第一节课程，我们重点只关注两个问题：

- 通过后台项目快速理解电商数据全貌。这一部分会带大家先整体梳理一下电商项目的后台表结构，快速梳理电商管理系统的后台数据管理功能。
- 如何给普通的CRUD管理系统赋能。电商后台功能虽然复杂，但是其实后端技术层面的难度并不大，相信大家就算自己实现也没有什么难度。但是在今天课程中，会带大家打开技术想象力，给这个简单的系统增加不一样的设计。带你接触一下什么是互联网思维。

一、电商后台项目需要访问的数据源说明

整个电商管理后台本质上是一个纯粹的CRUD管理系统，所以，

整个电商项目的数据库设计情况：

库名	表前缀	说明
tl_mall_goods	pms_	商品相关表
tl_mall_normal	cms_	其他辅助功能相关表
tl_mall_promotion	sms_	促销相关表
tl_mall_user	ums_	用户管理相关表
tl_mall_cart	oms_cart_	购物车相关表
tl_mall_order	oms_	订单相关表

这些不同库中的很多基础数据，除了购物车模块外，都需要由电商管理后台进行统一管理。所以，对于电商管理系统，会采用多数据源管理的方式，尽量快速的完成基础数据维护。其中，订单库由于进行了分库分表，管理比较复杂，所以电商管理后台不会直接访问订单相关的表，而是通过微服务的方式调用订单模块的相关功能来间接管理订单。

二、电商后台使用MyBatis-plus快速访问多个数据源

电商后台项目使用的MyBatis-plus框架访问数据库。对于MyBatis和MyBatis-plus框架，这里就不多做介绍了。而我们这个电商后台管理项目，与常见的一些普通的管理系统的最大区别，在于这个电商项目管理数据的方式更为直接粗暴，直接跨多个数据库管理的后台数据。这里分享三种常用的多数据源管理方案：

> 示例参见：springboot_dynamicdatasource

1、使用Spring提供的AbstractRoutingDataSource

这种方式的核心是使用Spring提供的AbstractRoutingDataSource抽象类，注入多个数据源。

```
@Component
@Primary   // 将该Bean设置为主要注入Bean
public class DynamicDataSource extends AbstractRoutingDataSource {
    // 当前使用的数据源标识
    public static ThreadLocal<String> name=new ThreadLocal<>();
    // 写库
    @Autowired
    DataSource dataSource1;
    // 读库
    @Autowired
    DataSource dataSource2;
    // 返回当前数据源标识
    @Override
    protected Object determineCurrentLookupKey() {
        return name.get();
    }
    @Override
    public void afterPropertiesSet() {
        // 为targetDataSources初始化所有数据源
        Map<Object, Object> targetDataSources=new HashMap<>();
        targetDataSources.put("W",dataSource1);
        targetDataSources.put("R",dataSource2);
        super.setTargetDataSources(targetDataSources);
        // 为defaultTargetDataSource 设置默认的数据源
        super.setDefaultTargetDataSource(dataSource1);
        super.afterPropertiesSet();
    }
}
```

将自己实现的DynamicDataSource注册成为默认的DataSource实例后，只需要在每次使用DataSource时，提前改变一下其中的name标识，就可以快速切换数据源。

```
@Component
@Aspect
public class DynamicDataSourceAspect implements Ordered {
    // 在每个访问数据库的方法执行前执行。
    @Before("within(com.tuling.dynamic.datasource.service.impl.*) &&
@annotation(wr)")
    public void before(JoinPoint point, WR wr){
        String name = wr.value();
        DynamicDataSource.name.set(name);
        System.out.println(name);
    }
```

```
    @Override
    public int getOrder() {
        return 0;
    }
}
```

> 完整示例参见01-dynamic_datasource模块。

2、使用MyBatis注册多个SqlSessionFactory

如果使用MyBatis框架，要注册多个数据源的话，就需要将MyBatis底层的DataSource、SqlSessionFactory、DataSourceTransactionManager这些核心对象一并进行手动注册。例如：

```
@Configuration
@MapperScan(basePackages = "com.tuling.datasource.dynamic.mybatis.mapper.r",
        sqlSessionFactoryRef="rSqlSessionFactory")
public class RMyBatisConfig {
    @Bean
    @ConfigurationProperties(prefix = "spring.datasource.datasource2")
    public DataSource dataSource2() {
        // 底层会自动拿到spring.datasource中的配置， 创建一个DruidDataSource
        return DruidDataSourceBuilder.create().build();
    }
    @Bean
    @Primary
    public SqlSessionFactory rSqlSessionFactory()
            throws Exception {
        final SqlSessionFactoryBean sessionFactory = new
SqlSessionFactoryBean();
        sessionFactory.setDataSource(dataSource2());
        // 指定主库对应的mapper.xml文件
        /*sessionFactory.setMapperLocations(new
PathMatchingResourcePatternResolver()
                .getResources("classpath:mapper/r/*.xml"));*/
        return sessionFactory.getObject();
    }
    @Bean
    public DataSourceTransactionManager rTransactionManager(){
        DataSourceTransactionManager dataSourceTransactionManager = new
DataSourceTransactionManager();
        dataSourceTransactionManager.setDataSource(dataSource2());
        return dataSourceTransactionManager;
    }

    @Bean
    public TransactionTemplate rTransactionTemplate(){
        return new TransactionTemplate(rTransactionManager());
    }
}
```

这样就完成了读库的注册。而读库与写库之间，就可以通过指定不同的Mapper和XML文件的地址来进行区分。

> 完整示例02-dynamic-mybatis模块。

3、使用dynamic-datasource框架

dynamic-datasource是MyBaits-plus作者设计的一个多数据源开源方案。使用这个框架需要引入对应的pom依赖

```
<dependency>
            <groupId>com.baomidou</groupId>
            <artifactId>dynamic-datasource-spring-boot-starter</artifactId>
            <version>3.5.0</version>
        </dependency>
```

这样就可以在SpringBoot的配置文件中直接配置多个数据源。

```
spring:
  datasource:
    dynamic:
      #设置默认的数据源或者数据源组,默认值即为master
      primary: master
      #严格匹配数据源,默认false. true未匹配到指定数据源时抛异常,false使用默认数据源
      strict: false
      datasource:
        master:
          url: jdbc:mysql://127.0.0.1:3306/datasource1?
serverTimezone=UTC&useUnicode=true&characterEncoding=UTF8&useSSL=false
          username: root
          password: 123456
          initial-size: 1
          min-idle: 1
          max-active: 20
          test-on-borrow: true
          driver-class-name: com.mysql.cj.jdbc.Driver
        slave_1:
          url: jdbc:mysql://127.0.0.1:3306/datasource2?
serverTimezone=UTC&useUnicode=true&characterEncoding=UTF8&useSSL=false
          username: root
          password: 123456
          initial-size: 1
          min-idle: 1
          max-active: 20
          test-on-borrow: true
          driver-class-name: com.mysql.cj.jdbc.Driver
```

这样就配置完成了master和slave_1两个数据库。

接下来在使用时，只要在对应的方法或者类上添加@DS注解即可。例如

```
@Service
public class FrendImplService implements FrendService {

    @Autowired
    FrendMapper frendMapper;
    @Override
    @DS("slave")  // 从库，  如果按照下划线命名方式配置多个  ，  可以指定前缀即可（组名）
    public List<Frend> list() {
```

```
        return frendMapper.list();
    }
    @Override
    @DS("master")
    public void save(Frend frend) {
        frendMapper.save(frend);
    }
    @DS("master")
    @DSTransactional
    public void saveAll(){
        // 执行多数据源的操作
    }
}
```

> 完整示例查看03-dynamic_datasource_framework模块。

当前电商管理后台采用了第三种方式来进行多数据源的管理。

> oms订单数据除外。因为订单相关数据已经完成了分库分表，不能直接查。

三、自定义实现MyBatis-Plus逆向工程

多数据源的问题解决了，接下来开始进行实际开发时，你会发现，最麻烦的一件事情就是要创建与数据库表对应的POJO了。这些没什么难度，但是繁琐的内容会占据大量的开发时间。比如一个PmsProducr对象，有三四十个属性。这就需要开发一个庞大的POJO对象。相反，上层的CRUD操作则相当简单。只需要继承MyBatis-plus框架提供的BaseMapper接口即可。

```
@DS("goods")
public interface PmsProductMapper extends BaseMapper<PmsProduct> {
}
```

标准的CRUD操作完全都不需要进行声明，直接就可以拿来用。只需要补充一些复杂的SQL操作即可。接下来当然是希望能够用程序快速自动的生成这些POJO类了，这样可以节省大量的开发时间。

关于如何生成POJO类，你当然可以使用MyBatis的逆向工程或者MyBatis-plus的逆向工程，这些网上有大量的资料，我们这里就不多做介绍。但是，你会不会有一种感觉，这些通用的逆向工程虽然优秀，但是却都太过复杂。他们为了工具的通用性，做了很多对我们没有用的封装。你有没有想过自己做一个简单使用的逆向工程出来呢？做一些这样的思考会让你对枯燥的CRUD工作产生一些不一样的想法。

其实你可以思考一下，需要根据数据库的表创建出对应的POJO类，需要哪些信息？ 其实要的信息并不多。表名、列名、列类型、主键信息。有这些就差不多了。而这些信息，其实都可以从最简单的JDBC操作中获取到。

```
public static void main(String[] args) throws Exception {
        //mysql
        Class.forName("com.mysql.cj.jdbc.Driver");
        Properties props = new Properties();
        props.put("useInformationSchema", "true"); //mysql获取表注释需要加上这个属性
        props.put("user", "root");
        props.put("password", "root");
        Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/genserver?
serverTimezone=GMT%2B8&characterEncoding=utf-8&autoReconnect=true",props);
        System.out.println("========映射表信息==============");
```

```
        DatabaseMetaData meta = con.getMetaData();
        ResultSet tables = meta.getTables("genserver", "%", "black_info", new
String[] {"TABLE"});
        while(tables.next()) {
            ResultSetMetaData metaData = tables.getMetaData();
            System.out.println(metaData.getColumnCount());
            for(int i = 1 ; i <= metaData.getColumnCount(); i ++) {
                System.out.println(metaData.getColumnName(i)+" ==>
"+tables.getString(metaData.getColumnName(i)));
            }

            System.out.println(tables.getString("TABLE_NAME")+" --->>>
"+tables.getString("REMARKS"));
        }
        System.out.println("========映射列信息==============");
        ResultSet columns = meta.getColumns("genserver", "%", "black_info",
"%");
        while(columns.next()) {
            String columnName = columns.getString("COLUMN_NAME");
            String columnType = columns.getString("TYPE_NAME");
            int datasize = columns.getInt("COLUMN_SIZE");
            int digits = columns.getInt("DECIMAL_DIGITS");
            int nullable = columns.getInt("NULLABLE");
            String remarks = columns.getString("REMARKS");
            System.out.println(columnName+" "+columnType+" "+datasize+"
"+digits+" "+ nullable+" "+remarks);
        }
        System.out.println("========映射主键信息==============");
        ResultSet primaryKeys = meta.getPrimaryKeys("genserver", "%",
"black_info");
        while(primaryKeys.next()) {
            ResultSetMetaData metaData = primaryKeys.getMetaData();
            System.out.println(metaData.getColumnCount());
            for(int i = 1 ; i <= metaData.getColumnCount(); i ++) {
                System.out.println(metaData.getColumnName(i)+" ==>
"+primaryKeys.getString(metaData.getColumnName(i)));
            }
        }
    }
```

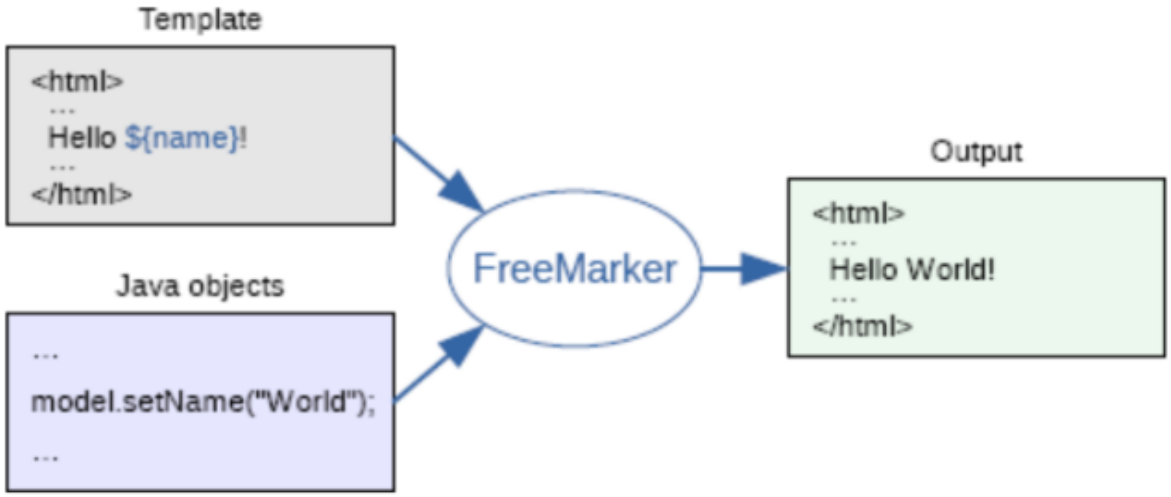
接下来如何将这些信息拼凑成一个POJO呢？你可以使用一个StringBuffer，一点点拼接出POJO的完整代码，再一次输出到文件当中，这没有问题。但是这样显然会比较麻烦，而且容易出错。

> MyBatis的逆向工程使用的就是这种方式。

对于这种问题，其实可以用模版引擎来做。将代码中静态的部分写到模版当中，然后将动态部分交由模版生成。最为常用的模版引擎就是freemarker了。大部分场景下，freemarker通常是用来生成静态HTML页面的。比如在我们的电商场景中，就实现了对产品单品页的静态化功能。

> 使用静态化功能，需要你创建 %{user_home}\template\ftl\目录下放置report.ftl模版文件，同时需要提前创建 %{user_home}\template\report目录

freemarker是一个基于模版和数据输出文本的通用工具。只需要准备好动态的业务数据，以及基于FTL语言编写的模版文件，就可以快生成静态的文本。



如果你对freemarker不是很了解，可以从这个示例中快速理解freemarker模版引擎。这个引擎上手非常简单，对于有开发经验的你，肯定没什么问题。按照以下几个步骤就可以快速上手freemarker了。

1、引入maven依赖

```
<dependency>
    <groupId>org.freemarker</groupId>
    <artifactId>freemarker</artifactId>
    <version>2.3.23</version>
</dependency>
```

2、构建后台数据

```
public class FreemarkerTest {
    public static void main(String[] args) throws Exception {
        // 第一步：创建一个Configuration对象，直接new一个对象。构造方法的参数就是
freemarker对于的版本号。
        Configuration configuration = new
Configuration(Configuration.getVersion());
        // 第二步：设置模板文件所在的路径。
        configuration.setDirectoryForTemplateLoading(new File("D:\\ftl"));
        // 第三步：设置模板文件使用的字符集。一般就是utf-8. 注意版本。新版本不需要
//          configuration.setDefaultEncoding("UTF-8");
        // 第四步：加载一个模板，创建一个模板对象。
        Template template = configuration.getTemplate("test.ftl");
        // 第五步：创建一个模板使用的数据集，可以是pojo也可以是map。一般是Map。
        Map dataModel = new HashMap<>();
```

```
        //向数据集中添加数据
        dataModel.put("hello", "图灵学院电商VIP");
        // 第六步：创建一个Writer对象，一般创建一FileWriter对象，指定生成的文件名。
        Writer out = new FileWriter(new File("D:\\ftl\\out\\test.html"));
        // 第七步：调用模板对象的process方法输出文件。
        template.process(dataModel, out);
        // 第八步：关闭流。
        out.close();
    }
}
```

3、编写ftl模版文件

最简单模版文件就长这样

```
<h1>
${hello}
</h1>
```

执行完成后，就会将模版中的${hello}部分替换成 图灵学院电商VIP

一个ftl模版文件，是由少数几个动态标签加上其他静态的内容组成。动态标签包含以下几种：

- 普通参数

  例如${hello}

- list标签

```
<#list studentList as student>
    ${student.id}/${studnet.name}
</#list>
```

- if条件标签

```
<#if student_index % 2 == 0>
<#else>
</#if>
```

在if标签中，还可以进行简单的null值判断

```
<#if a??>
a不为空时。。
<#else>
a为空时###
</#if>
```

- 日期标签

```
当前日期: ${date?date}
当前时间:${date?time}
当前日期和时间:${date?datetime}
自定义日期格式:${date?string("yyyyMM/dd HH:mm: ss")}
```

- 包含标签

```
<#include "hello.ftl"/>
```

接下来如果你发挥一些想象，freemarker既然可以生成html文件，那是不是可以用来生成java源文件呢？显然是可以的。

> 示例参见EntityGeneratorTest.java

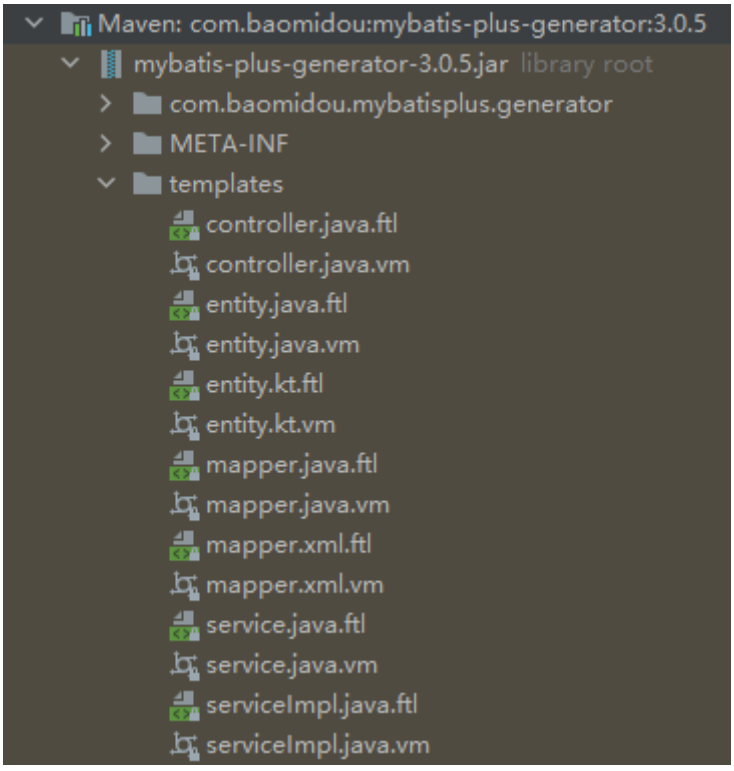
能够自己生成POJO了，那是不是可以把Service、Mapper、Controller等等这些重复性的代码一起生成呢？实际上，如果你有这种规范化的思想，你甚至可以将前台页面都一并生成了。减少大部分的复制粘贴的重复工作。

> 示例查看GenUI

最后，有了这个示例后，再来理解MyBatis-plus的逆向工程就非常容易了。 引入对应的依赖

```
        <dependency>
            <groupId>com.baomidou</groupId>
            <artifactId>mybatis-plus-generator</artifactId>
            <version>3.0.5</version>
        </dependency>
```

之后进入引入的jar包中，就能看到，MyBatis-plus的逆向工程也是使用freemarker和velocity提供的模版完成的逆向工程。



> vm是velocity框架的模版文件。velocity是和freemarker功能类似的一个模版引擎。

> 后续在设计秒杀场景时，也会使用freemarker自动生成前端商品单品页，实现动态页面静态化。

四、思维拓展

最后，发挥一下你自己的想象力，你还可以给这样简单的CRUD项目还能添加哪些与众不同的，实用的设计？