MyBatis 通用 Mapper 实现原理及相关内容

本文会先介绍通用 Mapper 的简单原理，然后使用最简单的代码来实现这个过程，最后会介绍 MyBatis 最新版本中原生的通用方法用法。

一、基本原理

通用 Mapper 提供了一些通用的方法，这些通用方法是以接口的形式提供的，例如。

```
public interface SelectMapper<T> {
    /**
     * 根据实体中的属性值进行查询，查询条件使用等号
     */
    @SelectProvider(type = BaseSelectProvider.class, method =
"dynamicSQL")
    List<T> select(T record);
}
```


接口和方法都使用了泛型，使用该通用方法的接口需要指定泛型的类型。通过 Java 反射可以很容易得到接口泛型的类型信息，代码如下。

```
Type[] types = mapperClass.getGenericInterfaces();
Class<?> entityClass = null;
for (Type type : types) {
    if (type instanceof ParameterizedType) {
        ParameterizedType t = (ParameterizedType) type;
        //由于可能继承多个接口，需要判断父接口是否为 SelectMapper.class
```

择了这种方式。如果使用模板（如 FreeMarker，Velocity 和 beetl 等模板引擎）实现，自由度会更高，也能方便开发人员调整。

在 MyBatis 中，每一个方法（注解或 XML 方式）经过处理后，最终会构造成 `MappedStatement` 实例，这个对象包含了方法id（namespace+id）、结果映射、缓存配置、 `SqlSource` 等信息，和 SQL 关系最紧密的是其中的 `SqlSource` ，MyBatis 最终执行的 SQL 时就是通过这个接口的 `getBoundSql` 方法获取的。

在 MyBatis 中，使用 `@SelectProvider` 这种方式定义的方法，最终会构造成 `ProviderSqlSource` ，`ProviderSqlSource` 是一种处于中间的 `SqlSource` ，它本身不能作为最终执行时使用的 `SqlSource` ，但是他会根据指定方法返回的 SQL 去构造一个可用于最后执行的 `StaticSqlSource` ，`StaticSqlSource` 的特点就是静态 SQL，支持在 SQL 中使用 `#{param}` 方式的参数，但是不支持 `<if>` ，`<where>` 等标签。

为了能根据实体类动态生成支持动态 SQL 的方法，通用 Mapper 从这里入手，利用 `ProviderSqlSource` 可以生成正常的 `MappedStatement` ，并且可以直接利用 MyBatis 各种配置和命名空间的特点（这是通用 Mapper 选择这种方式的主要原因）。在生成 `MappedStatement` 后，“过河拆桥”般的利用完就把 `ProviderSqlSource` 替换掉了，正常情况下，`ProviderSqlSource` 根本就没有执行的机会。在通用 Mapper 定义的实现方法中，提供了 `MappedStatement` 作为参数，有了这个参数，我们就可以根据 ms 的 id（规范情况下是 `接口名.方法名` ）得到接口，通过接口的泛型可以获取实体类（`entityClass`），根据实体和表的关系我们可以拼出 XML 方式的动态 SQL，一个简单的方法如下。


```
/**
 * 查询全部结果
 *
 * @param ms
 * @return
 */
public String selectAll(MappedStatement ms) {
    final Class<?> entityClass = getEntityClass(ms);
    //修改返回值类型为实体类型
    setResultType(ms, entityClass);
```

```
/**
 * 重新设置SqlSource
 *
 * @param ms
 * @param sqlSource
 */
protected void setSqlSource(MappedStatement ms, SqlSource
sqlSource) {
    MetaObject msObject = SystemMetaObject.forObject(ms);
    msObject.setValue("sqlSource", sqlSource);
}
```

> `MetaObject` 是MyBatis 中很有用的工具类，MyBatis 的结果映射就是靠这种方式实现的。反射信息使用的 `DefaultReflectorFactory` ，这个类会缓存反射信息，因此 MyBatis 的结果映射的效率很高。

到这里核心的内容都已经说完了，虽然知道怎么去替换 `SqlSource` 了，但是！什么时候去替换呢？

这一直都是一个难题，如果不大量重写 MyBatis 的代码很难万无一失的完成这个任务。通用 Mapper 并没有去大量重写，主要是考虑到 MyBatis 以后的升级，也因此在某些特殊情况下，通用 Mapper 的方法会在没有被替换的情况下被调用，这个问题在将来的 MyBatis 3.5.x 版本中会以更友好的方式解决（目前的 `ProviderSqlSource` 已经比以前能实现更多的东西，后面会讲）。

针对不同的运行环境，需要用不同的方式去替换。当使用纯 MyBatis（没有Spring）方式运行时，替换很简单，因为会在系统中初始化 `SqlSessionFactory` ，可以初始化的时候进行替换，这个时候也不会出现前面提到的问题。替换的方式也很简单，通过 `SqlSessionFactory` 可以得到 `SqlSession` ，然后就能得到 `Configuration` ，通过 `configuration.getMappedStatements()` 就能得到所有的 `MappedStatement` ，循环判断其中的方法是否为通用接口提供的方法，如果是就按照前面的方式替换就可以了。

在使用 Spring 的情况下，以继承的方式重写了 `MapperScannerConfigurer` 和

```
public interface BaseMapper<T> {
    @SelectProvider(type = SelectMethodProvider.class, method =
"select")
    List<T> select(T entity);
}
```

这里定义了一个简单的 `select` 方法，这个方法判断参数中的属性是否为空，不为空的字段会作为查询条件进行查询，下面是对应的 Provider。

```
public class SelectMethodProvider {
    public String select(Object params) {
        return "什么都不是!";
    }
}
```

这里的 Provider 方法不会最终执行，只是为了在初始化时可以生成对应的 `MappedStatement`。

2. 替换 SqlSource

下面代码为了简单，都指定的 `BaseMapper` 接口，并且没有特别的校验。

```
public class SimpleMapperHelper {
    public static final XMLLanguageDriver XML_LANGUAGE_DRIVER
            = new XMLLanguageDriver();
    /**
     * 获取泛型类型
     */
    public static Class getEntityClass(Class<?> mapperClass){
        Type[] types = mapperClass.getGenericInterfaces();
        Class<?> entityClass = null;
        for (Type type : types) {
```

```
    /**
     * 替换 SqlSource
     */
    public static void changeMs(MappedStatement ms) throws
Exception {
        String msId = ms.getId();
        //标准msId为 包名.接口名.方法名
        int lastIndex = msId.lastIndexOf(".");
        String methodName = msId.substring(lastIndex + 1);
        String interfaceName = msId.substring(0, lastIndex);
        Class<?> mapperClass = Class.forName(interfaceName);
        //判断是否继承了通用接口
        if(BaseMapper.class.isAssignableFrom(mapperClass)){
            //判断当前方法是否为通用 select 方法
            if (methodName.equals("select")) {
                Class entityClass = getEntityClass(mapperClass);
                //必须使用<script>标签包裹代码
                StringBuffer sqlBuilder = new StringBuffer("
<script>");
                //简单使用类名作为包名
                sqlBuilder.append("select * from ")
                        .append(entityClass.getSimpleName());
                Field[] fields = entityClass.getDeclaredFields();
                sqlBuilder.append(" <where> ");
                for (Field field : fields) {
                    //使用 if 标签进行动态判断
                    sqlBuilder.append("<if test=\"")
.append(field.getName()).append("!=null\">");
                    //字段名直接作为列名
                    sqlBuilder.append(" and
").append(field.getName())
                            .append(" = #{")
                            .append(field.getName())
                            .append("}");
                    sqlBuilder.append("</if>");
                }
```

```
}
```

changeMs 方法简单的从 msId 开始，获取接口和实体信息，通过反射回去字段信息，使用 <if> 标签动态判断属性值，这里的写法和 XML 中一样，使用 XMLLanguageDriver 处理时需要在外面包上 <script> 标签。生成 SqlSource 后，通过反射替换了原值。

3. 测试

针对上面代码，提供一个 country 表和对应的各种类。

实体类。

```
public class Country {
  private Long   id;
  private String countryname;
  private String countrycode;
  //省略 getter,setter
}
```

Mapper 接口。


```
public interface CountryMapper extends BaseMapper<Country> {

}
```

启动 MyBatis 的公共类。

```
public class SqlSessionHelper {
    private static SqlSessionFactory sqlSessionFactory;
```

```
                runner.setLogWriter(null);
                runner.runScript(reader);
                reader.close();
            } finally {
                if (session != null) {
                    session.close();
                }
            }
        } catch (IOException ignore) {
            ignore.printStackTrace();
        }
    }

    public static SqlSession getSqlSession() {
        return sqlSessionFactory.openSession();
    }

}
```

配置文件。

```
<?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>
  <environments default="development">
    <environment id="development">
      <transactionManager type="JDBC">
        <property name="" value=""/>
      </transactionManager>
      <dataSource type="UNPOOLED">
        <property name="driver" value="org.hsqldb.jdbcDriver"/>
        <property name="url" value="jdbc:hsqldb:mem:basetest"/>
        <property name="username" value="sa"/>
      </dataSource>
```

```
create table country (
  id integer,
  countryname varchar(32),
  countrycode varchar(2)
);

insert into country (id, countryname, countrycode)
values(1,'Angola','AO');
insert into country (id, countryname, countrycode)
values(23,'Botswana','BW');
-- 省略部分
insert into country (id, countryname, countrycode)
values(34,'Chile','CL');
insert into country (id, countryname, countrycode)
values(35,'China','CN');
insert into country (id, countryname, countrycode)
values(36,'Colombia','CO');
```

测试代码。

```
public class SimpleTest {

    public static void main(String[] args) throws Exception {
        SqlSession sqlSession = SqlSessionHelper.getSqlSession();
        Configuration configuration =
sqlSession.getConfiguration();
        HashSet<MappedStatement> mappedStatements
                = new HashSet<MappedStatement>
(configuration.getMappedStatements());
        //如果注释下面替换步骤就会出错
        for (MappedStatement ms : mappedStatements) {
            SimpleMapperHelper.changeMs(ms);
        }
        //替换后执行该方法
        CountryMapper mapper =
sqlSession.getMapper(CountryMapper.class);
```

通过简化版的处理过程应该可以和前面的内容联系起来，从而理解通用 Mapper 的简单处理过程。

> 完整代码下载：
>
> 链接：http://pan.baidu.com/s/1slNXfHF 密码：273u

三、最新的 ProviderSqlSource

早期的 `ProviderSqlSource` 有个缺点就是定义的方法要么没有参数，要么只能是 `Object parameterObject` 参数，这个参数最终的形式在开发时也不容易一次写对，因为不同形式的接口的参数会被 MyBatis 处理成不同的形式，可以参考 深入了解MyBatis参数。由于没有提供接口和类型相关的参数，因此无法根据类型实现通用的方法。

在最新的 3.4.5 版本中，`ProviderSqlSource` 增加了一个额外可选的 `ProviderContext` 参数，这个类如下。


```
/**
 * The context object for sql provider method.
 *
 * @author Kazuki Shimizu
 * @since 3.4.5
 */
public final class ProviderContext {

  private final Class<?> mapperType;
  private final Method mapperMethod;

  /**
   * Constructor.
   *
   * @param mapperType A mapper interface type that specified
```

```
    return mapperType;
  }

  /**
   * Get a mapper method that specified provider.
   *
   * @return A mapper method that specified provider
   */
  public Method getMapperMethod() {
    return mapperMethod;
  }

}
```

有了这个参数后，就能获取到接口和当前执行的方法信息，因此我们已经可以实现通用方法了。

下面是一个官方测试中的简单例子，定义的通用接口如下。

```
public interface BaseMapper<T> {

  @SelectProvider(type= OurSqlBuilder.class,
                  method= "buildSelectByIdProviderContextOnly")
  @ContainsLogicalDelete
  T selectById(Integer id);

  @Retention(RetentionPolicy.RUNTIME)
  @Target(ElementType.METHOD)
  @interface ContainsLogicalDelete {
    boolean value() default false;
  }

  @Retention(RetentionPolicy.RUNTIME)
  @Target(ElementType.TYPE)
  @interface Meta {
    String tableName();
```

```
    final boolean containsLogicalDelete =
context.getMapperMethod().
            getAnnotation(BaseMapper.ContainsLogicalDelete.class)
!= null;
    //获取接口上的元注解（不是实体）
    final String tableName = context.getMapperType().
            getAnnotation(BaseMapper.Meta.class).tableName();
    return new SQL(){{
      SELECT("*");
      FROM(tableName);
      WHERE("id = #{id}");
      if (!containsLogicalDelete){
        WHERE("logical_delete = ${Constants.LOGICAL_DELETE_OFF}");
      }
    }}.toString();
  }
```

这里相比之前，可以获取到更多的信息，SQL 也不只是固定表的查询，可以根据 `@Meta` 注解制定方法查询的表名，和原来一样的是，最终还是返回一个简单的 SQL 字符串，仍然不支持动态 SQL 的标签。

下面是实现的接口。


```
@BaseMapper.Meta(tableName = "users")
public interface Mapper extends BaseMapper<User> {

}
```

上面实现的方法中，注解从接口获取的，因此这里也是在 `Mapper` 上配置的 `Meta` 注解。

按照前面通用 Mapper 中的介绍，在实现方法中是可以获取接口泛型类型 `User` 的，因此如果把注解定义在类上也是可行的。

现在看起来已经很不错了，但是还不支持动态 SQL，还不能缓存根据 SQL 生成的