通用 Mapper 笔记

1 引入

1.1 作用

替我们生成常用增删改查操作的 SQL 语句。

通用Mapper是Mybatis的一个插件，优化开发效率

1.2 代码官方发布地址

https://gitee.com/free

https://gitee.com/free/Mapper/wikis/1.1-java?parent=1.integration

1.3 前置知识

MyBatis

Spring

2 快速入门

2.1 创建测试数据

- SQL 语句

```
CREATE TABLE `tabple_emp` (
`emp_id`  int NOT NULL AUTO_INCREMENT ,
`emp_name`  varchar(500) NULL ,
`emp_salary`  double(15,5) NULL ,
`emp_age`  int NULL ,
PRIMARY KEY (`emp_id`)
);

INSERT INTO `tabple_emp` (`emp_name`, `emp_salary`, `emp_age`) VALUES ('tom', '1254.37', '27');
INSERT INTO `tabple_emp` (`emp_name`, `emp_salary`, `emp_age`) VALUES ('jerry', '6635.42', '38');
INSERT INTO `tabple_emp` (`emp_name`, `emp_salary`, `emp_age`) VALUES ('bob', '5560.11', '40');
INSERT INTO `tabple_emp` (`emp_name`, `emp_salary`, `emp_age`) VALUES ('kate', '2209.11', '22');
INSERT INTO `tabple_emp` (`emp_name`, `emp_salary`, `emp_age`) VALUES ('justin', '4203.15', '30');
```

- Java 实体类

考虑到基本数据类型在 Java 类中都有默认值，会导致 MyBatis 在执行相关操作时很难判断当前字段是否为 null，所以在 MyBatis 环境下使用 Java 实体类时尽量不要使用基本数据类型，都使用对应的包装类型。

```
public class Employee {

	private Integer empId;
	private String empName;
	private Double empSalary;
	private Integer empAge;

	public Employee() {

	}

	public Employee(Integer empId, String empName, Double empSalary, Integer empAge) {
		super();
		this.empId = empId;
		this.empName = empName;
		this.empSalary = empSalary;
		this.empAge = empAge;
	}

	@Override
	public String toString() {
		return "Employee [empId=" + empId + ", empName=" + empName + ", empSalary=" + empSalary + ", empAge=" + empAge
				+ "]";
	}

	public Integer getEmpId() {
		return empId;
	}

	public void setEmpId(Integer empId) {
		this.empId = empId;
	}

	public String getEmpName() {
		return empName;
	}

	public void setEmpName(String empName) {
		this.empName = empName;
	}

	public Double getEmpSalary() {
```

```
        return empSalary;
    }

    public void setEmpSalary(Double empSalary) {
        this.empSalary = empSalary;
    }

    public Integer getEmpAge() {
        return empAge;
    }

    public void setEmpAge(Integer empAge) {
        this.empAge = empAge;
    }

}
```

2.2 搭建 MyBatis+Spring 开发环境

2.3 集成 Mapper

- 加入 Maven 依赖信息

```
<dependency>
    <groupId>tk.mybatis</groupId>
    <artifactId>mapper</artifactId>
    <version>4.0.0-beta3</version>
</dependency>
```

- 修改 Spring 配置文件

替代原有的Mybatis类型

```
<!-- 整合通用 Mapper 所需要做的配置修改： -->
<!-- 原始全类名：org.mybatis.spring.mapper.MapperScannerConfigurer -->
<!-- 通用 Mapper 使用：tk.mybatis.spring.mapper.MapperScannerConfigurer -->
<bean class="tk.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="basePackage" value="com.atguigu.mapper.mappers"/>
</bean>
```

2.4 第一个操作

```
/**
 * 具体操作数据库的 Mapper 接口，需要继承通用 Mapper 提供的核心接口：
Mapper<Employee>
 * 泛型类型就是实体类的类型
 * @author Lenovo
```

```
 *
 */
public interface EmployeeMapper extends Mapper<Employee> {


}
```

3 常用注解

3.1@Table 注解

作用：建立实体类和数据库表之间的对应关系。

默认规则：实体类类名首字母小写作为表名。Employee 类→employee 表。

用法：在@Table 注解的 name 属性中指定目标数据库表的表名

```
@Table(name="tabple_emp")
public class Employee {

    private Integer empId;
    private String empName;
    private Double empSalary;
    private Integer empAge;
```

3.2@Column 注解

作用：建立实体类字段和数据库表字段之间的对应关系。

默认规则：

实体类字段：驼峰式命名

数据库表字段：使用“_”区分各个单词

用法：在@Column 注解的 name 属性中指定目标字段的字段名

```
@Column(name="emp_salary_apple")
private Double empSalary;//emp_salary_apple
```

3.3@Id 注解

通用 Mapper 在执行 xxxByPrimaryKey(key)方法时，有两种情况。

情况 1：没有使用@Id 注解明确指定主键字段

```
SELECT emp_id,emp_name,emp_salary_apple,emp_age FROM tabple_emp WHERE emp_id = ?
AND emp_name = ? AND emp_salary_apple = ? AND emp_age = ?
```

之所以会生成上面这样的 WHERE 子句是因为通用 Mapper 将实体类中的所有字段都拿来放在一起作为联合主键。

情况 2：使用@Id 主键明确标记和数据库表中主键字段对应的实体类字段。

必须使用@Id标注主键对应的实体类属性

```
@Id
private Integer empId;//emp_id
```

3.4 @GeneratedValue 注解

作用：让通用 Mapper 在执行 insert 操作之后将数据库自动生成的主键值回写到实体类对象中。

自增主键用法：

```
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
private Integer empId;//emp_id
```

序列主键用法：

```
@Id
@GeneratedValue(
  strategy = GenerationType.IDENTITY,
  generator = "select SEQ_ID.nextval from dual")
private Integer id;
```

应用场景：购物车结账

- 增加商品销量...
- 减少商品库存...
- 生成订单数据→封装到 Order 对象中→保存 Order 对象→数据库自动生成主键值→回写到实体类对象 Order 中
- 生成一系列订单详情数据→List<OrderItem>→在每一个 OrderItem 中设置 Order 对象的主键值作为外键→批量保存 List<OrderItem>

……

3.5 @Transient 主键

用于标记不与数据库表字段对应的实体类字段。

```
@Transient
private String otherThings; //非数据库表中字段
```

4 常用方法

4.1 selectOne 方法

- 通用 Mapper 替我们自动生成的 SQL 语句情况

```
==>  Preparing: SELECT emp_id,emp_name,emp_salary,emp_age FROM tabple_emp WHERE emp_name = ? AND emp_salary = ?
==> Parameters: bob(String), 5560.11(Double)
<==      Total: 1
```

- 实体类封装查询条件生成 WHERE 子句的规则
  - 使用非空的值生成 WHERE 子句

注意，delete方法如果传入的是null，则会将表中所有数据都删除

- 在条件表达式中使用“=”进行比较
- 要求必须返回一个实体类结果，如果有多个，则会抛出异常

```
Caused by: org.apache.ibatis.exceptions.TooManyResultsException: Expected one result (or null) to be returned by selectOne(), but found: 5
at org.apache.ibatis.session.defaults.DefaultSqlSession.selectOne(DefaultSqlSession.java:70)
at org.mybatis.spring.SqlSessionTemplate$SqlSessionInterceptor.invoke(SqlSessionTemplate.java:358)
```

-

4.2 xxxByPrimaryKey 方法

需要使用@Id 主键明确标记和数据库表主键字段对应的实体类字段，否则通用 Mapper 会将所有实体类字段作为联合主键。

4.3 xxxSelective 方法

非主键字段如果为 null 值，则不加入到 SQL 语句中。

5 QBC 查询

RowBounds分页是假的分页，其实还是会将所有的数据查询出来，然后在内存中分页，而不是在sql中加入limit子句进行分页；

5.1 概念

Query By Criteria

Criteria 是 Criterion 的复数形式。意思是：规则、标准、准则。在 SQL 语句中相当于查询条件。

QBC 查询是将查询条件通过 Java 对象进行模块化封装。

5.2 示例代码

```
//目标：WHERE (emp_salary>? AND emp_age<?) OR (emp_salary<? AND emp_age>?)
//1.创建 Example 对象
Example example = new Example(Employee.class);

//**********************
//i.设置排序信息
example.orderBy("empSalary").asc().orderBy("empAge").desc();

//ii.设置“去重”
example.setDistinct(true);

//iii.设置 select 字段
example.selectProperties("empName","empSalary");

//**********************

//2.通过 Example 对象创建 Criteria 对象
```

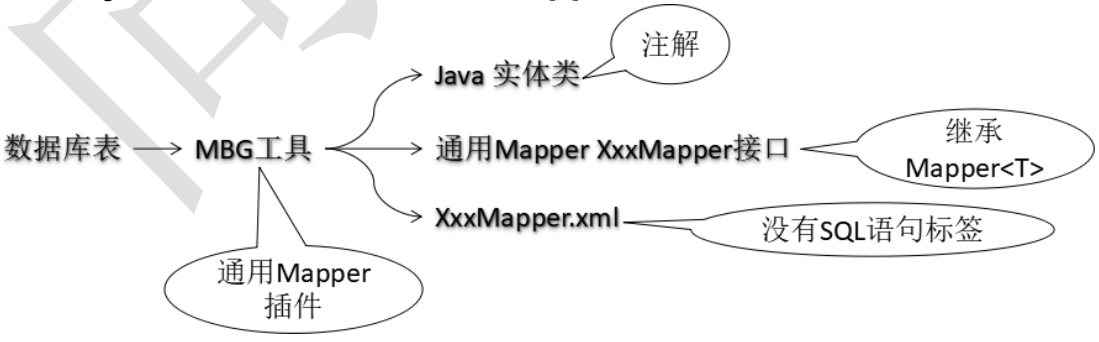
步骤：
1. 创建Example对象；
2. 通过example创建Criteria对象；
3. 每一个criteria对象代表sql语句中的一个括号括起来的条件；
4. 如果要连接多个criteria对象，则使用example对象连接；
5. 最后将example对象传入xxxByExample()方法中即可；

排序，去重，select的字段都是在example中设置的

```
Criteria criteria01 = example.createCriteria();
Criteria criteria02 = example.createCriteria();

//3.在两个 Criteria 对象中分别设置查询条件
//property 参数：实体类的属性名
//value 参数：实体类的属性值
criteria01.andGreaterThan("empSalary", 3000)
          .andLessThan("empAge", 25);

criteria02.andLessThan("empSalary", 5000)
          .andGreaterThan("empAge", 30);

//4.使用 OR 关键词组装两个 Criteria 对象
example.or(criteria02);

//5.执行查询
List<Employee> empList = employeeService.getEmpListByExample(example);

for (Employee employee : empList) {
    System.out.println(employee);
}
```

6 逆向工程

6.1 原生 MyBatis 逆向工程和通用 Mapper 逆向工程对比

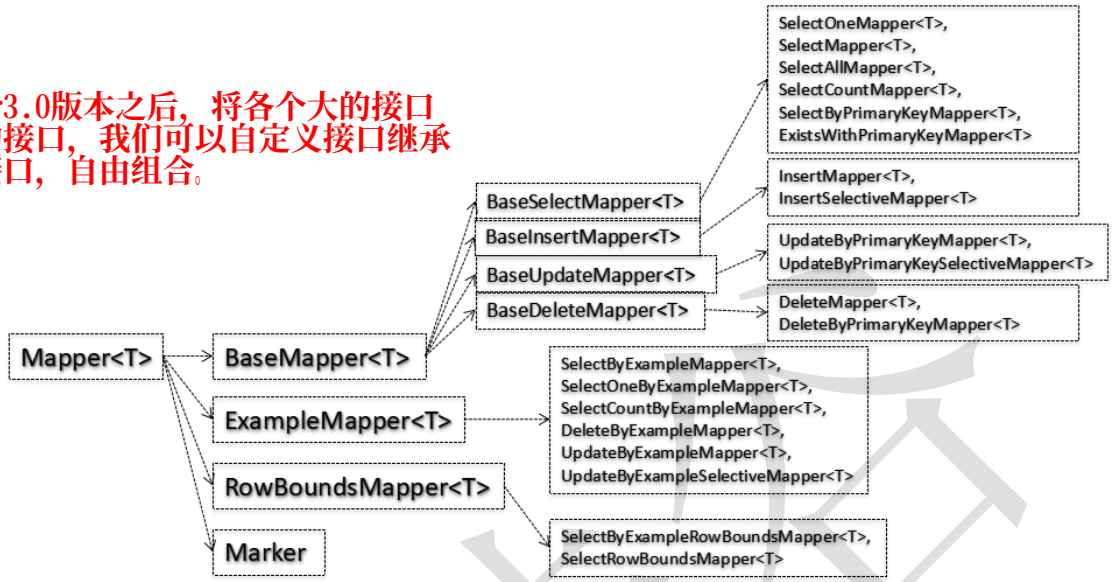


6.2 参考文档地址

https://github.com/abel533/Mapper/wiki/4.1.mappergenerator

7 自定义 Mapper<T>接口

通用Mapper3.0版本之后，将各个大的接口拆分为小的接口，我们可以自定义接口继承需要的小接口，自由组合。



7.1 用途

让我们可以根据开发的实际需要对 Mapper<T>接口进行定制。

7.2 创建自定义 Mapper<T>接口

```
public interface MyMapper<T>
        extends SelectAllMapper<T>,SelectByExampleMapper<T> {

}
```

自由选择

```
public interface EmployeeMapper extends MyMapper<Employee> {

}
```

7.3 配置 MapperScannerConfigurer 注册 MyMapper<T>

```
<bean class="tk.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="basePackage" value="com.atguigu.mapper.mappers"/>
    <property name="properties">
        <value>
            mappers=com.atguigu.mapper.mine_mappers.MyMapper
        </value>
    </property>
```

```
</bean>
```

8 通用 Mapper 接口扩展

8.1 说明

这里的扩展是指增加通用 Mapper 没有提供的功能。

8.2 举例

通用 Mapper 官方文档中使用一个批量 insert 作为扩展功能的例子：

tk.mybatis.mapper.additional.insert.InsertListMapper<T>

tk.mybatis.mapper.additional.insert.InsertListProvider

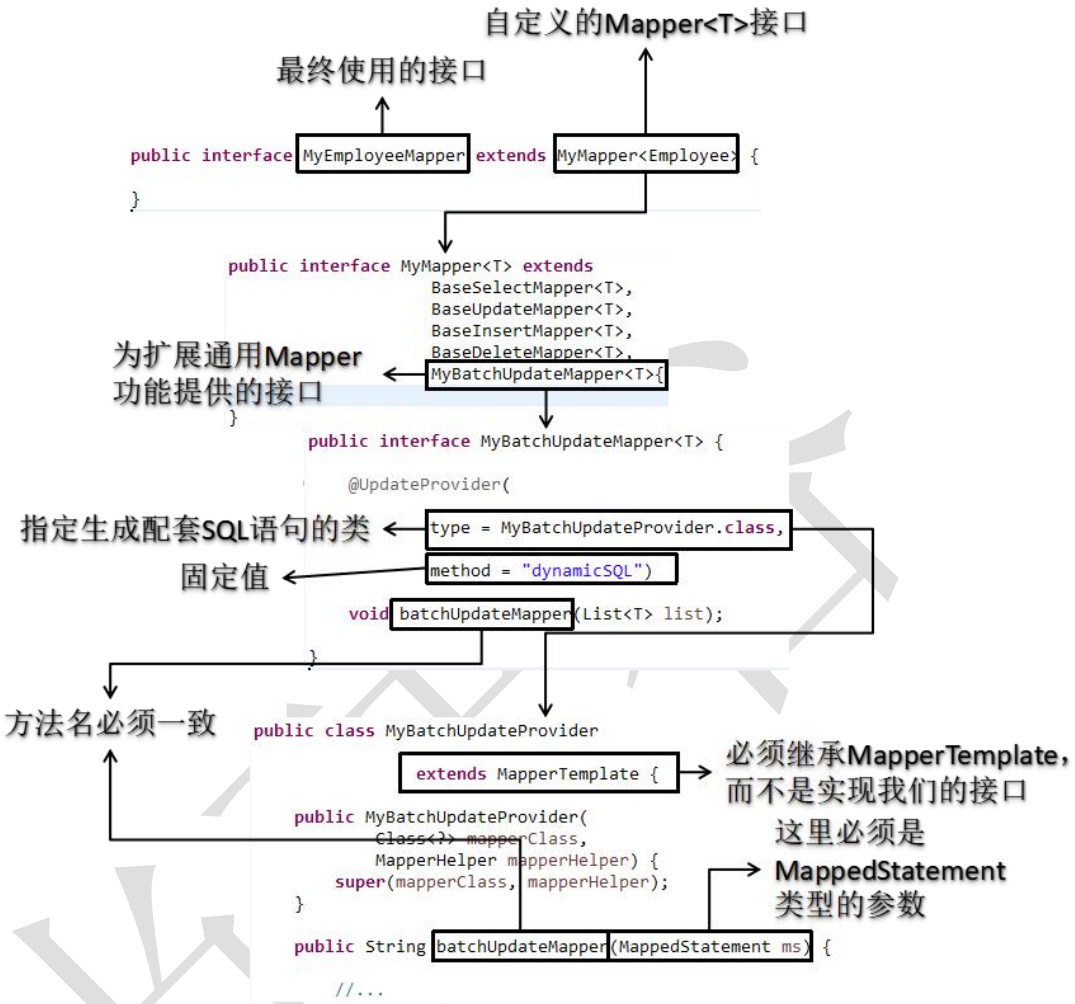
我们来仿照写一个批量 update。假设我们想生成下面这样的 SQL 语句：

```
UPDATE tabple_emp SET emp_name=?,emp_age=?,emp_salary=? where emp_id=? ;
UPDATE tabple_emp SET emp_name=?,emp_age=?,emp_salary=? where emp_id=? ;
UPDATE tabple_emp SET emp_name=?,emp_age=?,emp_salary=? where emp_id=? ;
……
```

为了生成上面那样的 SQL 语句，我们需要使用到 MyBatis 的 foreach 标签。

```
<foreach collection="list" item="record" separator=";" >
    UPDATE tabple_emp
    SET emp_name=#{record.empName},
    emp_age=#{record.empAge},
    emp_salary=#{record.empSalary}
    where emp_id=#{record.empId}
</foreach>
```

8.3 我们需要提供的接口和实现类



8.4参考代码

```
public String batchUpdateMapper(MappedStatement ms) {

    //1.实体类类型对象
    Class<?> entityClass = super.getEntityClass(ms);

    //2.表名对象
    String tableName = super.tableName(entityClass);

    StringBuilder sqlBuilder = new StringBuilder();

    //3.开始拼SQL
    sqlBuilder.append("<foreach collection=\"list\" item=\"record\" separator=\";\" >");

    //4.拼update子句
    String updateClause = SqlHelper.updateTable(entityClass, tableName);

    sqlBuilder.append(updateClause);

    //5.拼set子句
    //i.获取所有列信息
    Set<EntityColumn> columns = EntityHelper.getColumns(entityClass);

    sqlBuilder.append("<set>");

    String idColumn = null;
    String idValue = null;

    //ii.遍历所有列
    for (EntityColumn entityColumn : columns) {
        //iii.判断当前列是否为主键列
        if(entityColumn.isId()) {

            //iv.缓存主键列的列名和值
            idColumn = entityColumn.getColumn();
            idValue = entityColumn.getColumnHolder("record");

        }else{

            //v.获取非主键列的列名
            String columnName = entityColumn.getColumn();

            //vi.返回格式如:#{entityName.age,jdbcType=NUMERIC,typeHandler=MyTypeHandler}
            String columnHolder = entityColumn.getColumnHolder("record");

            sqlBuilder.append(columnName).append("=").append(columnHolder).append(",");

        }

    }

    sqlBuilder.append("</set>");

    //vii.拼where子句
    sqlBuilder.append("where ").append(idColumn).append("=").append(idValue);

    sqlBuilder.append("</foreach>");

    return sqlBuilder.toString();
}
```

9 二级缓存

9.1 MyBatis 配置文件开启二级缓存功能

```
<settings>
    <setting name="cacheEnabled" value="true"/>
</settings>
```

9.2 在 XxxMapper 接口上使用@CacheNamespace 注解

```
@CacheNamespace
public interface EmployeeMapper extends MyMapper<Employee> {


}
```

10 类型处理器：TypeHandler

10.1 简单类型和复杂类型

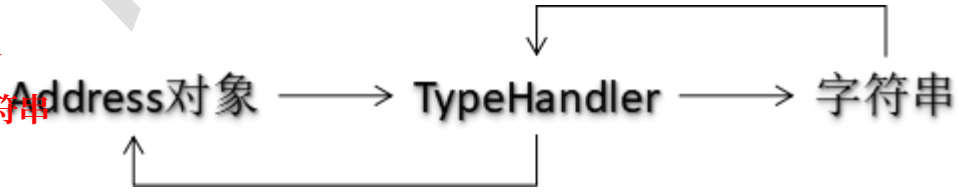
基本数据类型	byte short int long double float char boolean
引用数据类型	类、接口、数组、枚举……
简单类型	只有一个值的类型
复杂类型	有多个值的类型

※通用 Mapper 默认情况下会忽略复杂类型，对复杂类型不进行“从类到表”的映射。

10.2 Address 处理

10.2.1 自定义类型转换器

当我们需要将一个复杂对象存入到表中时，需要使用TypeHandler进行转换为字符串存入到表中



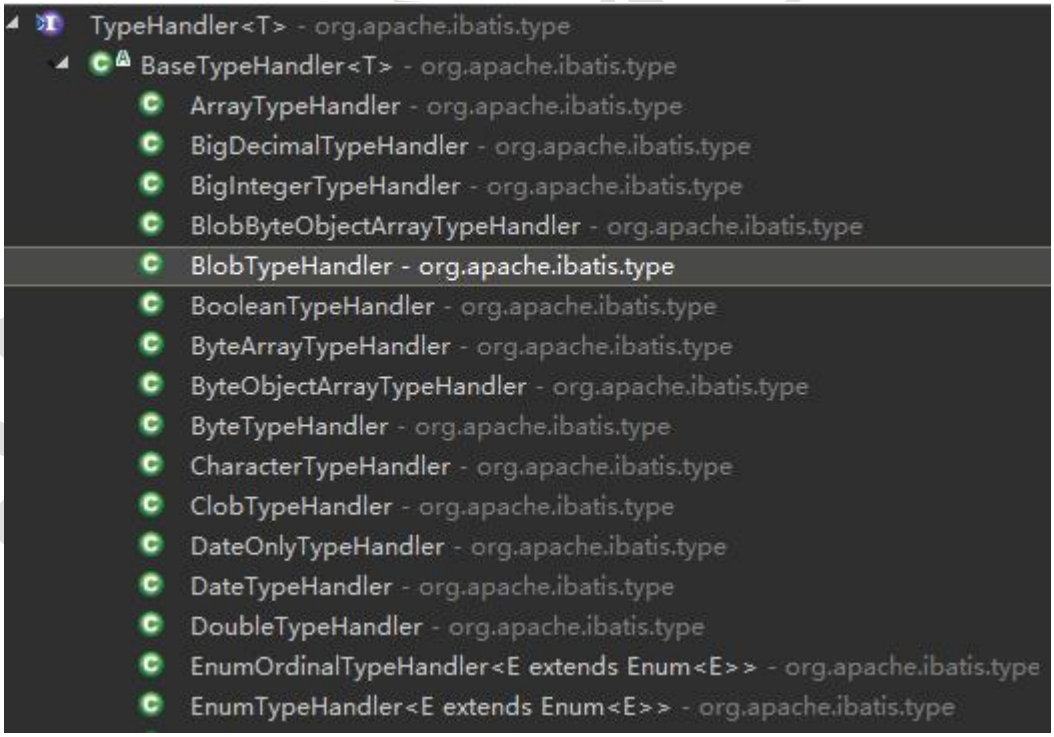
10.2.2 TypeHandler 接口

```
public interface TypeHandler<T> {


    //将 parameter 设置到 ps 对象中，位置是 i
```

```
    //在这个方法中将 parameter 转换为字符串
    void setParameter(PreparedStatement ps, int i, T parameter, JdbcType jdbcType) throws SQLException;

    //根据列名从 ResultSet 中获取数据，通常是字符串形式
    //将字符串还原为 Java 对象，以 T 类型返回
    T getResult(ResultSet rs, String columnName) throws SQLException;

    T getResult(ResultSet rs, int columnIndex) throws SQLException;

    T getResult(CallableStatement cs, int columnIndex) throws SQLException;

}
```

获取字符串后根据之前存入时自定义的规则将字符串还原为Java实体类对象

10.2.3 继承树



10.2.4 BaseTypeHandler 类中的抽象方法说明

```
//将 parameter 对象转换为字符串存入到 ps 对象的 i 位置
public abstract void setNonNullParameter(
    PreparedStatement ps,
    int i,
    T parameter,
    JdbcType jdbcType) throws SQLException;
```

```
//从结果集中获取数据库对应查询结果
//将字符串还原为原始的 T 类型对象
public abstract T getNullableResult(
        ResultSet rs,
        String columnName) throws SQLException;

public abstract T getNullableResult(
        ResultSet rs,
        int columnIndex) throws SQLException;

public abstract T getNullableResult(
        CallableStatement cs,
        int columnIndex) throws SQLException;
```

10.2.5 自定义类型转换器类

```
public class AddressTypeHandler extends BaseTypeHandler<Address> {
    ……
```

10.2.6 注册自定义类型转换器

- 方法一 字段级别：@ColumnType 注解

定义完类型转换器后，我们需要注册该转换器，一种是直接在实体类属性上注册，另一种是在配置文件中配置

```
@Table(name="table_user")
public class User {

    @Id
    private Integer userId;

    private String userName;

    @ColumnType(typeHandler=AddressTypeHandler.class)
    private Address address;

    private SeasonEnum season;
```

- 方法二 全局级别：在 MyBatis 配置文件中配置 typeHandlers

```
@Column
private Address address;
```

```
<typeHandlers>
    <!-- handler属性：指定自定义类型转换器全类名 -->
    <!-- javaType属性：指定需要使用“自定义类型转换器”进行类型处理的实体类型 -->
    <typeHandler
        handler="com.atguigu.mapper.handlers.AddressTypeHandler"
        javaType="com.atguigu.mapper.entities.Address"/>
</typeHandlers>
```

10.3 枚举类型

10.3.1 办法一：让通用 Mapper 把枚举类型作为简单类型处理

- 增加一个通用 Mapper 的配置项

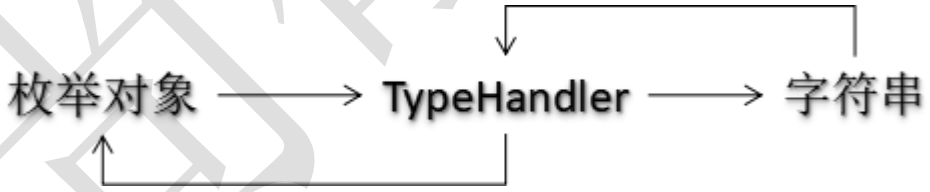
  在 Spring 配置文件中找到 MapperScannerConfigurer

```
<bean class="tk.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="basePackage" value="com.atguigu.mapper.mappers"/>
    <property name="properties">
        <value>
            enumAsSimpleType=true
        </value>
    </property>
</bean>
```

- 本质

  使用了 org.apache.ibatis.type.EnumTypeHandler<E>

10.3.2 办法二：为枚举类型配置对应的类型处理器



- 类型处理器
  - 内置
    - org.apache.ibatis.type.EnumTypeHandler<E>
      - 在数据库中存储枚举值本身
    - org.apache.ibatis.type.EnumOrdinalTypeHandler<E>
      - 在数据库中仅仅存储枚举值的索引
  - 自定义
- 内置枚举类型处理器注册
  - 不能使用@ColumnType 注解

```
34    @ColumnType(typeHandler=EnumTypeHandler.class)
35    private SeasonEnum seas
36
```

Type mismatch: cannot convert from Class<EnumTypeHandler> to Class<? extends TypeHandler<?>>

  - 需要在 MyBatis 配置文件中配置专门的类型处理器并在字段上使用

@Column 注解

```
<typeHandlers>
    <!-- handler属性：指定自定义类型转换器全类名 -->
    <!-- javaType属性：指定需要使用“自定义类型转换器”进行类型处理的实体类型 -->
    <typeHandler
        handler="com.atguigu.mapper.handlers.AddressTypeHandler"
        javaType="com.atguigu.mapper.entities.Address"/>

    <typeHandler
        handler="org.apache.ibatis.type.EnumTypeHandler"
        javaType="com.atguigu.mapper.entities.SeasonEnum"/>

</typeHandlers>
```

※注意：加@Column 注解的作用是让通用 Mapper 不忽略枚举类型。