

课程目标

内容定位

1. 实现思路概述

1.1. 从ResultSet说起

1.2. 为什么需要ORM框架

2. 搭建基础架构

2.1. 基于Spring JDBC 实现关键功能

3. 动态数据源切换的底层原理

4. 运行效果演示

课程目标

- 1、了解Spring的JdbcTemplate的API设计思想。
- 2、基于Spring JdbcTemplate进行二次开发，实现ORM框架。

内容定位

彻底理解 JavaJDBC的操作原理，掌握ORM框架的实现逻辑。为学习MyBatis 框架打下基础。

1. 实现思路概述

1.1. 从ResultSet说起

说到ResultSet，对于有Java开发经验的小伙伴自然是熟悉不过了，不过我相信对于大多数人来说也算是最熟悉的陌生人。从ResultSet的取值操作大家都会，比如：

```
1     private static List<Member> select(String sql) {
2         List<Member> result = new ArrayList<>();
3         Connection con = null;           //连接对象
4         PreparedStatement pstmt = null;   //语句集
5         ResultSet rs = null;             //结果集
6         try {
7             //1、加载驱动类，千万不要忘记了
8             Class.forName("com.mysql.jdbc.Driver");
9             //2、建立连接
10            con = DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/spring-db-de
11            //3、创建语句集
12            pstmt = con.prepareStatement(sql);
```

```

13      //4、执行语句集
14      rs = pstmt.executeQuery();
15      while (rs.next()) {
16          //纯粹的硬编码
17          Member instance = new Member();
18          instance.setId(rs.getLong("id"));
19          instance.setName(rs.getString("name"));
20          instance.setAge(rs.getInt("age"));
21          instance.setAddr(rs.getString("addr"));
22          result.add(instance);
23      }
24      //5、获取结果集
25  } catch (Exception e) {
26      e.printStackTrace();
27  }
28  //6、关闭结果集、关闭语句集、关闭连接
29  finally {
30      try {
31          rs.close();
32          pstmt.close();
33          con.close();
34      } catch (Exception e) {
35          e.printStackTrace();
36      }
37  }
38  return result;
39  }

```

这是我们在没有使用框架以前的常规操作。随着业务和开发量的增加，我们发现这样在数据持久层这样的重复代码出现频次非常高。因此，我们首先就想到将非功能性代码和业务代码分离。首先我就会想到将ResultSet 封装数据的代码逻辑分离，增加一个mapperRow()方法，专门处理对结果的封装，代码如下：

```

1      private static List<Member> select(String sql) {
2          List<Member> result = new ArrayList<>();
3          Connection con = null;
4          PreparedStatement pstmt = null;
5          ResultSet rs = null;
6          try {
7              //1、加载驱动类
8              Class.forName("com.mysql.jdbc.Driver");

```

```

9      //2、建立连接
10     con = DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/spring-db-de
11     //3、创建语句集
12     pstmt = con.prepareStatement(sql);
13     //4、执行语句集
14     rs = pstmt.executeQuery();
15     while (rs.next()) {
16         Member instance = mapperRow(rs, rs.getRow());
17         result.add(instance);
18     }
19     //5、获取结果集
20 } catch (Exception e) {
21     e.printStackTrace();
22 }
23 //6、关闭结果集、关闭语句集、关闭连接
24 finally {
25     try {
26         rs.close();
27         pstmt.close();
28         con.close();
29     } catch (Exception e) {
30         e.printStackTrace();
31     }
32 }
33 return result;
34 }
35
36 private static Member mapperRow(ResultSet rs, int i) throws Exception {
37     Member instance = new Member();
38     instance.setId(rs.getLong("id"));
39     instance.setName(rs.getString("name"));
40     instance.setAge(rs.getInt("age"));
41     instance.setAddr(rs.getString("addr"));
42     return instance;
43 }

```

但在真实的业务场景中，这样的代码逻辑重复率实在太高，上面的改造只能应用Member这个类，换一个实体类又要重新封装，聪明的程序猿肯定不会通过纯体力劳动给每一个实体类写一个mapperRow()方法，一定会想到代码复用方案。我们不妨来做这样一个改造，代码如下：

先创建Member类：

```

1 package cn.sitedev.orm.demo.entity;
2
3 import lombok.Data;
4
5 import javax.persistence.Entity;
6 import javax.persistence.Id;
7 import javax.persistence.Table;
8 import java.io.Serializable;
9
10 @Entity
11 @Table(name="t_member")
12 @Data
13 public class Member implements Serializable {
14     @Id private Long id;
15     private String name;
16     private String addr;
17     private Integer age;
18
19     @Override
20     public String toString() {
21         return "Member{" +
22             "id=" + id +
23             ", name='" + name + '\'' +
24             ", addr='" + addr + '\'' +
25             ", age=" + age +
26             '}';
27     }
28 }

```

对JDBC操作优化:

```

1     public static void main(String[] args) {
2         //ORM, 完成了一部分, 只完成了从 数据表到对象的映射
3         //对象到数据库表还没有
4         //我传的条件是一条SQL语句, 我还是在面向SQL编程
5         //     List<Member> result = select("select * from t_member");
6
7         //这就是OO编程, ORM
8         Member condition = new Member();
9         condition.setName("TomCat");
10        condition.setAge(2);

```

```

11
12     //"select * from t_member where name = 'Tom' and age = 19"
13     List<?> result = select(condition);
14     System.out.println(Arrays.toString(result.toArray()));
15 }
16
17 public static List<?> select(Object condition) {
18     List<Object> result = new ArrayList<>();
19
20     Class<?> entityClass = condition.getClass();
21
22
23     Connection con = null;           //连接对象
24     PreparedStatement pstmt = null;  //语句集
25     ResultSet rs = null;             //结果集
26
27
28     try {
29         //1、加载驱动类，千万不要忘记了
30         Class.forName("com.mysql.jdbc.Driver");
31         //2、建立连接
32         con = DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/spring-db-de
33
34         Map<String, String> getFieldByNameByColumn = new HashMap<String, String>();
35         Map<String, String> getColumnByFieldName = new HashMap<String, String>();
36         Field[] fields = entityClass.getDeclaredFields();
37         for (Field field : fields) {
38             field.setAccessible(true);
39             String fieldName = field.getName();
40             if (field.isAnnotationPresent(Column.class)) {
41                 Column column = field.getAnnotation(Column.class);
42                 String columnName = column.name();
43                 getFieldByNameByColumn.put(columnName, fieldName);
44                 getColumnByFieldName.put(fieldName, columnName);
45             } else {
46                 //默认属性名就是列名
47                 getFieldByNameByColumn.put(fieldName, fieldName);
48                 getColumnByFieldName.put(fieldName, fieldName);
49             }
50         }
51
52
53         StringBuffer sql = new StringBuffer();

```

```

54 //3、创建语句集
55 Table table = entityClass.getAnnotation(Table.class);
56 sql.append("select * from " + table.name() + " where 1=1 ");
57 for (Field field : fields) {
58
59     Object value = field.get(condition);
60     if (null != value) {
61         if (String.class == field.getType()) {
62             sql.append(" and " + getColumnByFieldName.get(field.getName()))
63         } else {
64             sql.append(" and " + getColumnByFieldName.get(field.getName()))
65         }
66         //其他依次类推
67     }
68 }
69
70 pstmt = con.prepareStatement(sql.toString());
71
72 //4、执行，获取结果集
73 rs = pstmt.executeQuery();
74
75 int columnCounts = rs.getMetaData().getColumnCount();
76 while (rs.next()) {
77     Object instance = entityClass.newInstance();
78     for (int i = 1; i <= columnCounts; i++) {
79         String columnName = rs.getMetaData().getColumnName(i);
80         Field field = entityClass.getDeclaredField(getFieldNameByColumn.get(columnName));
81         field.setAccessible(true);
82         field.set(instance, rs.getObject(columnName));
83     }
84     result.add(instance);
85 }
86
87
88 } catch (Exception e) {
89     e.printStackTrace();
90 }
91 //6、关闭结果集、关闭语句集、关闭连接
92 finally {
93     try {
94         rs.close();
95         pstmt.close();
96         con.close();

```

```
97         } catch (Exception e) {
98             e.printStackTrace();
99         }
100     }
101     return result;
102 }
```

巧妙地利用反射机制，读取Class 信息和Annotation信息，将数据库表中的列和类中的字段进行关联映射并赋值，以减少重复代码。

1.2. 为什么需要ORM框架

通过上面的操作，其实我们已经了解ORM框架的基本实现原理。ORM是指对象关系映射（Object Relation Mapping），映射的不仅仅只是对象值，还有对象与对象之间的关系。例如一对多、多对多、一对一这样的表关系。现在市面上ORM框架也非常之多，有大家所熟知的Hibernate、Spring JDBC、MyBatis、JPA等。我在这里做一个简单的总结，如下表：

名称	特征	描述
Hibernate	全自动(档)	不需要写一句 SQL
MyBatis	半自动(档)	手自一体，支持简单的映射，复杂关系需要自己写 SQL
Spring JDBC	纯手动(档)	所有的 SQL 都要自己，它帮我们设计了一套标准流程

既然，市面上有这么多选择，我又为什么还要自己写ORM框架呢？

这得从我的一次空降担任架构师的经验说起。空降面临最大的难题就是如何取得团队小伙伴们的信任。

当时，团队总共就8人，每个人水平层次不齐，甚至有些还没接触过MySQL，诸如Redis 等缓存中间件就不需要谈。基本只会使用Hibernate的CRUD，而且已经影响到了系统性能。由于工期紧张，没有时间和精力给团队做系统培训，也为了兼顾可控性，于是就产生了自研ORM框架的想法。我做了这样的顶层设计，以降低团队小伙伴的存息成本，顶层接口统一参数、统一返回值，具体如下：

1、规定查询方法的接口模型为：

```
1 public interface BaseDao<T,PK> {
2     /**
3      * 获取列表
4      * @param queryRule 查询条件
5      * @return
6      */
7     List<T> select(QueryRule queryRule) throws Exception;
```

```

8
9  /**
10  * 获取分页结果
11  * @param queryRule 查询条件
12  * @param pageNo 页码
13  * @param pageSize 每页条数
14  * @return
15  */
16  Page<?> select(QueryRule queryRule,int pageNo,int pageSize) throws Exception;
17
18  /**
19  * 根据SQL获取列表
20  * @param sql SQL语句
21  * @param args 参数
22  * @return
23  */
24  List<Map<String,Object>> selectBySql(String sql, Object... args) throws Exception;
25
26  /**
27  * 根据SQL获取分页
28  * @param sql SQL语句
29  * @param pageNo 页码
30  * @param pageSize 每页条数
31  * @return
32  */
33  Page<Map<String,Object>> selectBySqlToPage(String sql, Object [] param, int pageNo,

```

2、规定删除方法的接口模型为：

```

1  /**
2  * 删除一条记录
3  * @param entity entity中的ID不能为空，如果ID为空，其他条件不能为空，都为空不予执行
4  * @return
5  */
6  boolean delete(T entity) throws Exception;
7
8  /**
9  * 批量删除
10  * @param list
11  * @return 返回受影响的行数
12  * @throws Exception

```



```
13      */
14      int deleteAll(List<T> list) throws Exception;
```

3、规定插入方法的接口模型为：

```
1      /**
2       * 插入一条记录并返回插入后的ID
3       * @param entity 只要entity不等于null，就执行插入
4       * @return
5       */
6      PK insertAndReturnId(T entity) throws Exception;
7
8      /**
9       * 插入一条记录自增ID
10     * @param entity
11     * @return
12     * @throws Exception
13     */
14     boolean insert(T entity) throws Exception;
15
16     /**
17     * 批量插入
18     * @param list
19     * @return 返回受影响的行数
20     * @throws Exception
21     */
22     int insertAll(List<T> list) throws Exception;
```

4、规定修改方法的接口模型为：

```
1      /**
2       * 修改一条记录
3       * @param entity entity中的ID不能为空，如果ID为空，其他条件不能为空，都为空不予执行
4       * @return
5       * @throws Exception
6       */
7      boolean update(T entity) throws Exception;
```

利用这一套基础的API，后面我又基于Redis、MongoDB、ElasticSearch、Hive、HBase各封装了一套，以此来讲降低团队学习成本。也大大提升了程序可控性，也更方便统一监控。

2. 搭建基础架构

Page

```
1 package javax.core.common;
2
3 import java.io.Serializable;
4 import java.util.ArrayList;
5 import java.util.List;
6
7 /**
8  * 分页对象。包含当前页数据及分页信息如总记录数。
9  * 能够支持JQuery EasyUI直接对接，能够支持和Bootstrap Table直接对接
10 */
11 public class Page<T> implements Serializable {
12
13     private static final long serialVersionUID = 1L;
14
15     private static final int DEFAULT_PAGE_SIZE = 20;
16
17     private int pageSize = DEFAULT_PAGE_SIZE; // 每页的记录数
18
19     private long start; // 当前页第一条数据在List中的位置,从0开始
20
21     private List<T> rows; // 当前页中存放的记录,类型一般为List
22
23     private long total; // 总记录数
24
25     /**
26      * 构造方法，只构造空页。
27      */
28     public Page() {
29         this(0, 0, DEFAULT_PAGE_SIZE, new ArrayList<T>());
30     }
31
32     /**
33      * 默认构造方法。
34      *
35      * @param start
```

```
36      *          本页数据在数据库中的起始位置
37      * @param totalSize
38      *          数据库中总记录条数
39      * @param pageSize
40      *          本页容量
41      * @param rows
42      *          本页包含的数据
43      */
44      public Page(long start, long totalSize, int pageSize, List<T> rows) {
45          this.pageSize = pageSize;
46          this.start = start;
47          this.total = totalSize;
48          this.rows = rows;
49      }
50
51      /**
52       * 取总记录数.
53       */
54      public long getTotal() {
55          return this.total;
56      }
57
58      public void setTotal(long total) {
59          this.total = total;
60      }
61
62      /**
63       * 取总页数.
64       */
65      public long getTotalPageCount() {
66          if (total % pageSize == 0){
67              return total / pageSize;
68          }else{
69              return total / pageSize + 1;
70          }
71      }
72
73      /**
74       * 取每页数据容量.
75       */
76      public int getPageSize() {
77          return pageSize;
78      }
```

```
79
80  /**
81   * 取当前页中的记录.
82   */
83  public List<T> getRows() {
84      return rows;
85  }
86
87  public void setRows(List<T> rows) {
88      this.rows = rows;
89  }
90
91  /**
92   * 取该页当前页码,页码从1开始.
93   */
94  public long getPageNo() {
95      return start / pageSize + 1;
96  }
97
98  /**
99   * 该页是否有下一页.
100  */
101  public boolean hasNextPage() {
102      return this.getPageNo() < this.getTotalPageCount() - 1;
103  }
104
105  /**
106   * 该页是否有上一页.
107   */
108  public boolean hasPreviousPage() {
109      return this.getPageNo() > 1;
110  }
111
112  /**
113   * 获取任一页第一条数据在数据集的位置, 每页条数使用默认值.
114   *
115   * @see #getStartOfPage(int,int)
116   */
117  protected static int getStartOfPage(int pageNo) {
118      return getStartOfPage(pageNo, DEFAULT_PAGE_SIZE);
119  }
120
121  /**
```

```

122     * 获取任一页第一条数据在数据集的位置.
123     *
124     * @param pageNo
125     *         从1开始的页号
126     * @param pageSize
127     *         每页记录条数
128     * @return 该页第一条数据
129     */
130     public static int getStartOfPage(int pageNo, int pageSize) {
131         return (pageNo - 1) * pageSize;
132     }
133
134 }

```

ResultMsg

```

1  package javax.core.common;
2
3  import lombok.Data;
4
5  import java.io.Serializable;
6
7
8  //最底层设计
9  @Data
10 public class ResultMsg<T> implements Serializable {
11
12     private static final long serialVersionUID = 2635002588308355785L;
13
14     private int status; //状态码，系统的返回码
15     private String msg; //状态码的解释
16     private T data; //放任意结果
17
18     public ResultMsg() {}
19
20     public ResultMsg(int status) {
21         this.status = status;
22     }
23
24     public ResultMsg(int status, String msg) {
25         this.status = status;

```

```

26         this.msg = msg;
27     }
28
29     public ResultMsg(int status, T data) {
30         this.status = status;
31         this.data = data;
32     }
33
34     public ResultMsg(int status, String msg, T data) {
35         this.status = status;
36         this.msg = msg;
37         this.data = data;
38     }
39 }

```

BaseDao

```

1 package javax.core.common.jdbc;
2
3 import cn.sitedev.orm.framework.QueryRule;
4
5 import javax.core.common.Page;
6 import java.util.List;
7 import java.util.Map;
8
9 public interface BaseDao<T,PK> {
10     /**
11      * 获取列表
12      * @param queryRule 查询条件
13      * @return
14      */
15     List<T> select(QueryRule queryRule) throws Exception;
16
17     /**
18      * 获取分页结果
19      * @param queryRule 查询条件
20      * @param pageNo 页码
21      * @param pageSize 每页条数
22      * @return
23      */
24     Page<?> select(QueryRule queryRule,int pageNo,int pageSize) throws Exception;

```

```
25
26  /**
27   * 根据SQL获取列表
28   * @param sql SQL语句
29   * @param args 参数
30   * @return
31   */
32  List<Map<String,Object>> selectBySql(String sql, Object... args) throws Exception;
33
34  /**
35   * 根据SQL获取分页
36   * @param sql SQL语句
37   * @param pageNo 页码
38   * @param pageSize 每页条数
39   * @return
40   */
41  Page<Map<String,Object>> selectBySqlToPage(String sql, Object [] param, int pageNo,
42
43
44
45
46
47  /**
48   * 删除一条记录
49   * @param entity entity中的ID不能为空，如果ID为空，其他条件不能为空，都为空不予执行
50   * @return
51   */
52  boolean delete(T entity) throws Exception;
53
54  /**
55   * 批量删除
56   * @param list
57   * @return 返回受影响的行数
58   * @throws Exception
59   */
60  int deleteAll(List<T> list) throws Exception;
61
62  /**
63   * 插入一条记录并返回插入后的ID
64   * @param entity 只要entity不等于null，就执行插入
65   * @return
66   */
67  PK insertAndReturnId(T entity) throws Exception;
```

```

68
69     /**
70      * 插入一条记录自增ID
71      * @param entity
72      * @return
73      * @throws Exception
74      */
75     boolean insert(T entity) throws Exception;
76
77     /**
78      * 批量插入
79      * @param list
80      * @return 返回受影响的行数
81      * @throws Exception
82      */
83     int insertAll(List<T> list) throws Exception;
84
85     /**
86      * 修改一条记录
87      * @param entity entity中的ID不能为空，如果ID为空，其他条件不能为空，都为空不予执行
88      * @return
89      * @throws Exception
90      */
91     boolean update(T entity) throws Exception;
92 }

```

QueryRule

```

1 package cn.sitedev.orm.framework;
2
3 import java.io.Serializable;
4 import java.util.ArrayList;
5 import java.util.List;
6
7 /**
8  * 查询规则构造器，实现多条件复杂查询的条件拼接
9  * Role 角色，Rule 尺子，规则
10 */
11 public final class QueryRule implements Serializable {
12     private static final long serialVersionUID = 1L;
13     public static final int ASC_ORDER = 101;

```



```
14     public static final int DESC_ORDER = 102;
15     public static final int LIKE = 1;
16     public static final int IN = 2;
17     public static final int NOTIN = 3;
18     public static final int BETWEEN = 4;
19     public static final int EQ = 5;
20     public static final int NOTEQ = 6;
21     public static final int GT = 7;
22     public static final int GE = 8;
23     public static final int LT = 9;
24     public static final int LE = 10;
25     public static final int ISNULL = 11;
26     public static final int ISNOTNULL = 12;
27     public static final int ISEMPY = 13;
28     public static final int ISNOTEMPTY = 14;
29     public static final int AND = 201;
30     public static final int OR = 202;
31     private List<Rule> ruleList = new ArrayList<Rule>();
32     private List<QueryRule> queryRuleList = new ArrayList<QueryRule>();
33     private String propertyName;
34
35     private QueryRule() {}
36
37     private QueryRule(String propertyName) {
38         this.propertyName = propertyName;
39     }
40
41     public static QueryRule getInstance() {
42         return new QueryRule();
43     }
44
45     /**
46      * 添加升序规则
47      * @param propertyName
48      * @return
49      */
50     public QueryRule addAscOrder(String propertyName) {
51         this.ruleList.add(new Rule(ASC_ORDER, propertyName));
52         return this;
53     }
54
55     /**
56      * 添加降序规则
```

```
57     * @param propertyName
58     * @return
59     */
60     public QueryRule addDescOrder(String propertyName) {
61         this.ruleList.add(new Rule(DESC_ORDER, propertyName));
62         return this;
63     }
64
65     public QueryRule andIsNull(String propertyName) {
66         this.ruleList.add(new Rule(ISNULL, propertyName).setAndOr(AND));
67         return this;
68     }
69
70     public QueryRule andIsNotNull(String propertyName) {
71         this.ruleList.add(new Rule(ISNOTNULL, propertyName).setAndOr(AND));
72         return this;
73     }
74
75     public QueryRule andIsEmpty(String propertyName) {
76         this.ruleList.add(new Rule(ISEMPY, propertyName).setAndOr(AND));
77         return this;
78     }
79
80     public QueryRule andIsNotEmpty(String propertyName) {
81         this.ruleList.add(new Rule(ISNOTEMPTY, propertyName).setAndOr(AND));
82         return this;
83     }
84
85     public QueryRule andLike(String propertyName, Object value) {
86         this.ruleList.add(new Rule(LIKE, propertyName, new Object[] { value }).setAndOr(AND));
87         return this;
88     }
89
90     public QueryRule andEqual(String propertyName, Object value) {
91         this.ruleList.add(new Rule(EQ, propertyName, new Object[] { value }).setAndOr(AND));
92         return this;
93     }
94
95     public QueryRule andBetween(String propertyName, Object... values) {
96         this.ruleList.add(new Rule(BETWEEN, propertyName, values).setAndOr(AND));
97         return this;
98     }
99
```

```
100 public QueryRule andIn(String propertyName, List<Object> values) {
101     this.ruleList.add(new Rule(IN, propertyName, new Object[] { values })).setAndOr(
102     return this;
103 }
104
105 public QueryRule andIn(String propertyName, Object... values) {
106     this.ruleList.add(new Rule(IN, propertyName, values).setAndOr(AND));
107     return this;
108 }
109
110 public QueryRule andNotIn(String propertyName, List<Object> values) {
111     this.ruleList.add(new Rule(NOTIN, propertyName, new Object[] { values })).setAndOr(AND);
112     return this;
113 }
114
115 public QueryRule orNotIn(String propertyName, Object... values) {
116     this.ruleList.add(new Rule(NOTIN, propertyName, values).setAndOr(OR));
117     return this;
118 }
119
120
121 public QueryRule andNotEqual(String propertyName, Object value) {
122     this.ruleList.add(new Rule(NOTEQ, propertyName, new Object[] { value })).setAndOr(AND);
123     return this;
124 }
125
126 public QueryRule andGreaterThan(String propertyName, Object value) {
127     this.ruleList.add(new Rule(GT, propertyName, new Object[] { value })).setAndOr(AND);
128     return this;
129 }
130
131 public QueryRule andGreaterEqual(String propertyName, Object value) {
132     this.ruleList.add(new Rule(GE, propertyName, new Object[] { value })).setAndOr(AND);
133     return this;
134 }
135
136 public QueryRule andLessThan(String propertyName, Object value) {
137     this.ruleList.add(new Rule(LT, propertyName, new Object[] { value })).setAndOr(AND);
138     return this;
139 }
140
141 public QueryRule andLessEqual(String propertyName, Object value) {
142     this.ruleList.add(new Rule(LE, propertyName, new Object[] { value })).setAndOr(AND);
```

```
143         return this;
144     }
145
146
147     public QueryRule orIsNull(String propertyName) {
148         this.ruleList.add(new Rule(ISNULL, propertyName).setAndOr(OR));
149         return this;
150     }
151
152     public QueryRule orIsNotNull(String propertyName) {
153         this.ruleList.add(new Rule(ISNOTNULL, propertyName).setAndOr(OR));
154         return this;
155     }
156
157     public QueryRule orIsEmpty(String propertyName) {
158         this.ruleList.add(new Rule(ISEMPY, propertyName).setAndOr(OR));
159         return this;
160     }
161
162     public QueryRule orIsNotEmpty(String propertyName) {
163         this.ruleList.add(new Rule(ISNOTEMPTY, propertyName).setAndOr(OR));
164         return this;
165     }
166
167     public QueryRule orLike(String propertyName, Object value) {
168         this.ruleList.add(new Rule(LIKE, propertyName, new Object[] { value }).setAndOr(OR));
169         return this;
170     }
171
172     public QueryRule orEqual(String propertyName, Object value) {
173         this.ruleList.add(new Rule(EQ, propertyName, new Object[] { value }).setAndOr(OR));
174         return this;
175     }
176
177     public QueryRule orBetween(String propertyName, Object... values) {
178         this.ruleList.add(new Rule(BETWEEN, propertyName, values).setAndOr(OR));
179         return this;
180     }
181
182     public QueryRule orIn(String propertyName, List<Object> values) {
183         this.ruleList.add(new Rule(IN, propertyName, new Object[] { values }).setAndOr(OR));
184         return this;
185     }
```

```
186
187     public QueryRule orIn(String propertyName, Object... values) {
188         this.ruleList.add(new Rule(IN, propertyName, values).setAndOr(OR));
189         return this;
190     }
191
192     public QueryRule orNotEqual(String propertyName, Object value) {
193         this.ruleList.add(new Rule(NOTEQ, propertyName, new Object[] { value }).setAndOr(C));
194         return this;
195     }
196
197     public QueryRule orGreaterThan(String propertyName, Object value) {
198         this.ruleList.add(new Rule(GT, propertyName, new Object[] { value }).setAndOr(C));
199         return this;
200     }
201
202     public QueryRule orGreaterEqual(String propertyName, Object value) {
203         this.ruleList.add(new Rule(GE, propertyName, new Object[] { value }).setAndOr(C));
204         return this;
205     }
206
207     public QueryRule orLessThan(String propertyName, Object value) {
208         this.ruleList.add(new Rule(LT, propertyName, new Object[] { value }).setAndOr(C));
209         return this;
210     }
211
212     public QueryRule orLessEqual(String propertyName, Object value) {
213         this.ruleList.add(new Rule(LE, propertyName, new Object[] { value }).setAndOr(C));
214         return this;
215     }
216
217
218     public List<Rule> getRuleList() {
219         return this.ruleList;
220     }
221
222     public List<QueryRule> getQueryRuleList() {
223         return this.queryRuleList;
224     }
225
226     public String getPropertyName() {
227         return this.propertyName;
228     }
```

```
229
230     protected class Rule implements Serializable {
231         private static final long serialVersionUID = 1L;
232         private int type;    //规则的类型
233         private String property_name;
234         private Object[] values;
235         private int andOr = AND;
236
237         public Rule(int paramInt, String paramString) {
238             this.property_name = paramString;
239             this.type = paramInt;
240         }
241
242         public Rule(int paramInt, String paramString,
243             Object[] paramArrayOfObject) {
244             this.property_name = paramString;
245             this.values = paramArrayOfObject;
246             this.type = paramInt;
247         }
248
249         public Rule setAndOr(int andOr){
250             this.andOr = andOr;
251             return this;
252         }
253
254         public int getAndOr(){
255             return this.andOr;
256         }
257
258         public Object[] getValues() {
259             return this.values;
260         }
261
262         public int getType() {
263             return this.type;
264         }
265
266         public String getPropertyName() {
267             return this.property_name;
268         }
269     }
270 }
```

Order

```
1 package cn.sitedev.orm.framework;
2
3
4 /**
5  * sql排序组件
6  */
7 public class Order {
8     private boolean ascending; //升序还是降序
9     private String propertyName; //哪个字段升序，哪个字段降序
10
11     public String toString() {
12         return propertyName + ' ' + (ascending ? "asc" : "desc");
13     }
14
15     /**
16      * Constructor for Order.
17      */
18     protected Order(String propertyName, boolean ascending) {
19         this.propertyName = propertyName;
20         this.ascending = ascending;
21     }
22
23     /**
24      * Ascending order
25      *
26      * @param propertyName
27      * @return Order
28      */
29     public static Order asc(String propertyName) {
30         return new Order(propertyName, true);
31     }
32
33     /**
34      * Descending order
35      *
36      * @param propertyName
37      * @return Order
38      */
39     public static Order desc(String propertyName) {
40         return new Order(propertyName, false);
41     }
42 }
```

```
41     }
42
43 }
```

2.1. 基于Spring JDBC 实现关键功能

ClassMappings

```
1 package cn.sitedev.orm.framework;
2
3 import java.lang.reflect.Field;
4 import java.lang.reflect.Method;
5 import java.lang.reflect.Modifier;
6 import java.math.BigDecimal;
7 import java.sql.Date;
8 import java.sql.Timestamp;
9 import java.util.Arrays;
10 import java.util.HashMap;
11 import java.util.HashSet;
12 import java.util.Map;
13 import java.util.Set;
14
15 public class ClassMappings {
16
17     private ClassMappings(){}
18
19     static final Set<Class<?>> SUPPORTED_SQL_OBJECTS = new HashSet<Class<?>>();
20
21     static {
22         //只要这里写了的，默认支持自动类型转换
23         Class<?>[] classes = {
24             boolean.class, Boolean.class,
25             short.class, Short.class,
26             int.class, Integer.class,
27             long.class, Long.class,
28             float.class, Float.class,
29             double.class, Double.class,
30             String.class,
31             Date.class,
32             Timestamp.class,
33             BigDecimal.class
```



```

34         };
35         SUPPORTED_SQL_OBJECTS.addAll(Arrays.asList(classes));
36     }
37
38     static boolean isSupportedSQLObject(Class<?> clazz) {
39         return clazz.isEnum() || SUPPORTED_SQL_OBJECTS.contains(clazz);
40     }
41
42     public static Map<String, Method> findPublicGetters(Class<?> clazz) {
43         Map<String, Method> map = new HashMap<String, Method>();
44         Method[] methods = clazz.getMethods();
45         for (Method method : methods) {
46             if (Modifier.isStatic(method.getModifiers()))
47                 continue;
48             if (method.getParameterTypes().length != 0)
49                 continue;
50             if (method.getName().equals("getClass"))
51                 continue;
52             Class<?> returnType = method.getReturnType();
53             if (void.class.equals(returnType))
54                 continue;
55             if (!isSupportedSQLObject(returnType)){
56                 continue;
57             }
58             if ((returnType.equals(boolean.class)
59                 || returnType.equals(Boolean.class))
60                 && method.getName().startsWith("is")
61                 && method.getName().length() > 2) {
62                 map.put(getGetterName(method), method);
63                 continue;
64             }
65             if ( ! method.getName().startsWith("get"))
66                 continue;
67             if (method.getName().length() < 4)
68                 continue;
69             map.put(getGetterName(method), method);
70         }
71         return map;
72     }
73
74     public static Field[] findFields(Class<?> clazz){
75         return clazz.getDeclaredFields();
76     }

```

```

77
78     public static Map<String, Method> findPublicSetters(Class<?> clazz) {
79         Map<String, Method> map = new HashMap<String, Method>();
80         Method[] methods = clazz.getMethods();
81         for (Method method : methods) {
82             if (Modifier.isStatic(method.getModifiers()))
83                 continue;
84             if ( ! void.class.equals(method.getReturnType()))
85                 continue;
86             if (method.getParameterTypes().length != 1)
87                 continue;
88             if ( ! method.getName().startsWith("set"))
89                 continue;
90             if (method.getName().length() < 4)
91                 continue;
92             if(!isSupportedSQLObject(method.getParameterTypes()[0])){
93                 continue;
94             }
95             map.put(getSetterName(method), method);
96         }
97         return map;
98     }
99
100     public static String getGetterName(Method getter) {
101         String name = getter.getName();
102         if (name.startsWith("is"))
103             name = name.substring(2);
104         else
105             name = name.substring(3);
106         return Character.toLowerCase(name.charAt(0)) + name.substring(1);
107     }
108
109     private static String getSetterName(Method setter) {
110         String name = setter.getName().substring(3);
111         return Character.toLowerCase(name.charAt(0)) + name.substring(1);
112     }
113 }

```

EntityOperation

```

1 package cn.sitedev.orm.framework;

```

```
2
3 import org.apache.log4j.Logger;
4 import org.springframework.jdbc.core.RowMapper;
5
6 import javax.core.common.utils.StringUtils;
7 import javax.persistence.*;
8 import java.lang.reflect.Field;
9 import java.lang.reflect.Method;
10 import java.sql.ResultSet;
11 import java.sql.ResultSetMetaData;
12 import java.sql.SQLException;
13 import java.util.HashMap;
14 import java.util.Map;
15 import java.util.TreeMap;
16
17 /**
18  * 实体对象的反射操作
19  *
20  * @param <T>
21  */
22 public class EntityOperation<T> {
23     private Logger log = Logger.getLogger(EntityOperation.class);
24     public Class<T> entityClass = null; // 泛型实体Class对象
25     public final Map<String, PropertyMapping> mappings;
26     public final RowMapper<T> rowMapper;
27
28     public final String tableName;
29     public String allColumn = "*";
30     public Field pkField;
31
32     public EntityOperation(Class<T> clazz,String pk) throws Exception{
33         if(!clazz.isAnnotationPresent(Entity.class)){
34             throw new Exception("在" + clazz.getName() + "中没有找到Entity注解，不能做O
35         }
36         this.entityClass = clazz;
37         Table table = entityClass.getAnnotation(Table.class);
38         if (table != null) {
39             this.tableName = table.name();
40         } else {
41             this.tableName = entityClass.getSimpleName();
42         }
43         Map<String, Method> getters = ClassMappings.findPublicGetters(entityClass);
44         Map<String, Method> setters = ClassMappings.findPublicSetters(entityClass);
```

```

45     Field[] fields = ClassMappings.findFields(entityClass);
46     fillPkFieldAndAllColumn(pk, fields);
47     this.mappings = getPropertyMappings(getters, setters, fields);
48     this.allColumn = this.mappings.keySet().toString().replace("[", "").replace("]"
49     this.rowMapper = createRowMapper();
50 }
51
52 Map<String, PropertyMapping> getPropertyMappings(Map<String, Method> getters, Map<
53     Map<String, PropertyMapping> mappings = new HashMap<String, PropertyMapping>();
54     String name;
55     for (Field field : fields) {
56         if (field.isAnnotationPresent(Transient.class))
57             continue;
58         name = field.getName();
59         if(name.startsWith("is")){
60             name = name.substring(2);
61         }
62         name = Character.toLowerCase(name.charAt(0)) + name.substring(1);
63         Method setter = setters.get(name);
64         Method getter = getters.get(name);
65         if (setter == null || getter == null){
66             continue;
67         }
68         Column column = field.getAnnotation(Column.class);
69         if (column == null) {
70             mappings.put(field.getName(), new PropertyMapping(getter, setter, field
71         } else {
72             mappings.put(column.name(), new PropertyMapping(getter, setter, field))
73         }
74     }
75     return mappings;
76 }
77
78 RowMapper<T> createRowMapper() {
79     return new RowMapper<T>() {
80         public T mapRow(ResultSet rs, int rowNum) throws SQLException {
81             try {
82                 T t = entityClass.newInstance();
83                 ResultSetMetaData meta = rs.getMetaData();
84                 int columns = meta.getColumnCount();
85                 String columnName;
86                 for (int i = 1; i <= columns; i++) {
87                     Object value = rs.getObject(i);

```

```

88         columnName = meta.getColumnName(i);
89         fillBeanFieldValue(t,columnName,value);
90     }
91     return t;
92 }catch (Exception e) {
93     throw new RuntimeException(e);
94 }
95 }
96 };
97 }
98
99 protected void fillBeanFieldValue(T t, String columnName, Object value) {
100     if (value != null) {
101         PropertyMapping pm = mappings.get(columnName);
102         if (pm != null) {
103             try {
104                 pm.set(t, value);
105             } catch (Exception e) {
106                 e.printStackTrace();
107             }
108         }
109     }
110 }
111
112 private void fillPkFieldAndAllColumn(String pk, Field[] fields) {
113     //设定主键
114     try {
115         if(!StringUtils.isEmpty(pk)){
116             pkField = entityClass.getDeclaredField(pk);
117             pkField.setAccessible(true);
118         }
119     } catch (Exception e) {
120         log.debug("没找到主键列,主键列名必须与属性名相同");
121     }
122     for (int i = 0 ; i < fields.length ;i ++ ) {
123         Field f = fields[i];
124         if(StringUtils.isEmpty(pk)){
125             Id id = f.getAnnotation(Id.class);
126             if(id != null){
127                 pkField = f;
128                 break;
129             }
130         }

```

```

131     }
132 }
133
134 public T parse(ResultSet rs) {
135     T t = null;
136     if (null == rs) {
137         return null;
138     }
139     Object value = null;
140     try {
141         t = (T) entityClass.newInstance();
142         for (String columnName : mappings.keySet()) {
143             try {
144                 value = rs.getObject(columnName);
145             } catch (Exception e) {
146                 e.printStackTrace();
147             }
148             fillBeanFieldValue(t, columnName, value);
149         }
150     } catch (Exception ex) {
151         ex.printStackTrace();
152     }
153     return t;
154 }
155
156 public Map<String, Object> parse(T t) {
157     Map<String, Object> _map = new TreeMap<String, Object>();
158     try {
159
160         for (String columnName : mappings.keySet()) {
161             Object value = mappings.get(columnName).getter.invoke(t);
162             if (value == null)
163                 continue;
164             _map.put(columnName, value);
165
166         }
167     } catch (Exception e) {
168         e.printStackTrace();
169     }
170     return _map;
171 }
172
173 public void println(T t) {

```

```

174     try {
175         for (String columnName : mappings.keySet()) {
176             Object value = mappings.get(columnName).getter.invoke(t);
177             if (value == null)
178                 continue;
179             System.out.println(columnName + " = " + value);
180         }
181     } catch (Exception e) {
182         e.printStackTrace();
183     }
184 }
185 }
186
187 class PropertyMapping {
188
189     final boolean insertable;
190     final boolean updatable;
191     final String columnName;
192     final boolean id;
193     final Method getter;
194     final Method setter;
195     final Class enumClass;
196     final String fieldName;
197
198     public PropertyMapping(Method getter, Method setter, Field field) {
199         this.getter = getter;
200         this.setter = setter;
201         this.enumClass = getter.getReturnType().isEnum() ? getter.getReturnType() : null;
202         Column column = field.getAnnotation(Column.class);
203         this.insertable = column == null || column.insertable();
204         this.updatable = column == null || column.updatable();
205         this.columnName = column == null ? ClassMappings.getGetterName(getter) : ("".ec
206         this.id = field.isAnnotationPresent(Id.class);
207         this.fieldName = field.getName();
208     }
209
210     @SuppressWarnings("unchecked")
211     Object get(Object target) throws Exception {
212         Object r = getter.invoke(target);
213         return enumClass == null ? r : Enum.valueOf(enumClass, (String) r);
214     }
215
216     @SuppressWarnings("unchecked")

```

```

217 void set(Object target, Object value) throws Exception {
218     if (enumClass != null && value != null) {
219         value = Enum.valueOf(enumClass, (String) value);
220     }
221     //BeanUtils.setProperty(target, fieldName, value);
222     try {
223         if(value != null){
224             setter.invoke(target, setter.getParameterTypes()[0].cast(value));
225         }
226     } catch (Exception e) {
227         e.printStackTrace();
228         /**
229          * 出错原因如果是boolean字段 mysql字段类型 设置tinyint(1)
230          */
231         System.err.println(fieldName + "--" + value);
232     }
233
234 }
235 }

```

QueryRuleSqlBulider

```

1 package cn.sitedev.orm.framework;
2
3 import cn.sitedev.orm.framework.QueryRule.Rule;
4 import org.apache.commons.lang.ArrayUtils;
5
6 import javax.core.common.utils.StringUtils;
7 import java.util.ArrayList;
8 import java.util.HashMap;
9 import java.util.List;
10 import java.util.Map;
11 import java.util.regex.Matcher;
12 import java.util.regex.Pattern;
13
14
15 /**
16  * 根据QueryRule自动构建sql语句
17  *
18  */
19 public class QueryRuleSqlBuilder {

```



```
20     private int CURR_INDEX = 0; //记录参数所在的位置
21     private List<String> ;properties; //保存列名列表
22     private List<Object> values; //保存参数值列表
23     private List<Order> orders; //保存排序规则列表
24
25     private String whereSql = "";
26     private String orderSql = "";
27     private Object [] valueArr = new Object[]{};
28     private Map<Object,Object> valueMap = new HashMap<Object,Object>();
29
30     /**
31      * 或得查询条件
32      * @return
33      */
34     public String getWhereSql(){
35         return this.whereSql;
36     }
37
38     /**
39      * 获得排序条件
40      * @return
41      */
42     public String getOrderSql(){
43         return this.orderSql;
44     }
45
46     /**
47      * 获得参数值列表
48      * @return
49      */
50     public Object [] getValues(){
51         return this.valueArr;
52     }
53
54     /**
55      * 获取参数列表
56      * @return
57      */
58     public Map<Object,Object> getValueMap(){
59         return this.valueMap;
60     }
61
62     /**
```

```
63      * 创建SQL构造器
64      * @param queryRule
65      */
66      public QueryRuleSqlBuilder(QueryRule queryRule) {
67          CURR_INDEX = 0;
68          properties = new ArrayList<String>();
69          values = new ArrayList<Object>();
70          orders = new ArrayList<Order>();
71          for (Rule rule : queryRule.getRuleList()) {
72              switch (rule.getType()) {
73                  case QueryRule.BETWEEN:
74                      processBetween(rule);
75                      break;
76                  case QueryRule.EQ:
77                      processEqual(rule);
78                      break;
79                  case QueryRule.LIKE:
80                      processLike(rule);
81                      break;
82                  case QueryRule.NOTEQ:
83                      processNotEqual(rule);
84                      break;
85                  case QueryRule.GT:
86                      processGreaterThan(rule);
87                      break;
88                  case QueryRule.GE:
89                      processGreaterEqual(rule);
90                      break;
91                  case QueryRule.LT:
92                      processLessThen(rule);
93                      break;
94                  case QueryRule.LE:
95                      processLessEqual(rule);
96                      break;
97                  case QueryRule.IN:
98                      processIN(rule);
99                      break;
100                 case QueryRule.NOTIN:
101                     processNotIN(rule);
102                     break;
103                 case QueryRule.ISNULL:
104                     processIsNull(rule);
105                     break;
```

```

106         case QueryRule.ISNOTNULL:
107             processIsNotNull(rule);
108             break;
109         case QueryRule.ISEMPY:
110             processIsEmpty(rule);
111             break;
112         case QueryRule.ISNOTEMPTY:
113             processIsNotEmpty(rule);
114             break;
115         case QueryRule.ASC_ORDER:
116             processOrder(rule);
117             break;
118         case QueryRule.DESC_ORDER:
119             processOrder(rule);
120             break;
121         default:
122             throw new IllegalArgumentException("type " + rule.getType() + " not sup
123     }
124 }
125 //拼装where语句
126 appendWhereSql();
127 //拼装排序语句
128 appendOrderSql();
129 //拼装参数值
130 appendValues();
131 }
132
133 /**
134  * 去掉order
135  *
136  * @param sql
137  * @return
138  */
139 protected String removeOrders(String sql) {
140     Pattern p = Pattern.compile("order\\\\s*by[\\\\w|\\\\W|\\\\s|\\\\S]*", Pattern.CASE_INSENSITIVE);
141     Matcher m = p.matcher(sql);
142     StringBuffer sb = new StringBuffer();
143     while (m.find()) {
144         m.appendReplacement(sb, "");
145     }
146     m.appendTail(sb);
147     return sb.toString();
148 }

```

```
149
150  /**
151   * 去掉select
152   *
153   * @param sql
154   * @return
155   */
156  protected String removeSelect(String sql) {
157      if(sql.toLowerCase().matches("from\\s+")){
158          int beginPos = sql.toLowerCase().indexOf("from");
159          return sql.substring(beginPos);
160      }else{
161          return sql;
162      }
163  }
164
165  /**
166   * 处理like
167   * @param rule
168   */
169  private void processLike(Rule rule) {
170      if (ArrayUtils.isEmpty(rule.getValues())) {
171          return;
172      }
173      Object obj = rule.getValues()[0];
174
175      if (obj != null) {
176          String value = obj.toString();
177          if (!StringUtils.isEmpty(value)) {
178              value = value.replace('*', '%');
179              obj = value;
180          }
181      }
182      add(rule.getAndOr(), rule.getPropertyName(), "like", "%" + rule.getValues()[0] + "%");
183  }
184
185  /**
186   * 处理between
187   * @param rule
188   */
189  private void processBetween(Rule rule) {
190      if ((ArrayUtils.isEmpty(rule.getValues()))
191          || (rule.getValues().length < 2)) {
```

```
192         return;
193     }
194     add(rule.getAndOr(),rule.getPropertyName(),"","between",rule.getValues()[0],"and",rule.getValues()[1]);
195     add(0,"",""," ",rule.getValues()[1],"");
196 }
197
198 /**
199  * 处理 =
200  * @param rule
201  */
202 private void processEqual(Rule rule) {
203     if (ArrayUtils.isEmpty(rule.getValues())) {
204         return;
205     }
206     add(rule.getAndOr(),rule.getPropertyName(),"=",rule.getValues()[0]);
207 }
208
209 /**
210  * 处理 <>
211  * @param rule
212  */
213 private void processNotEqual(Rule rule) {
214     if (ArrayUtils.isEmpty(rule.getValues())) {
215         return;
216     }
217     add(rule.getAndOr(),rule.getPropertyName(),"<>",rule.getValues()[0]);
218 }
219
220 /**
221  * 处理 >
222  * @param rule
223  */
224 private void processGreaterThen(
225     Rule rule) {
226     if (ArrayUtils.isEmpty(rule.getValues())) {
227         return;
228     }
229     add(rule.getAndOr(),rule.getPropertyName(),">",rule.getValues()[0]);
230 }
231
232 /**
233  * 处理 >=
234  * @param rule
```

```
235     */
236     private void processGreaterEqual(
237         Rule rule) {
238         if (ArrayUtils.isEmpty(rule.getValues())) {
239             return;
240         }
241         add(rule.getAndOr(), rule.getPropertyName(), ">=", rule.getValues()[0]);
242     }
243
244     /**
245     * 处理<
246     * @param rule
247     */
248     private void processLessThen(Rule rule) {
249         if (ArrayUtils.isEmpty(rule.getValues())) {
250             return;
251         }
252         add(rule.getAndOr(), rule.getPropertyName(), "<", rule.getValues()[0]);
253     }
254
255     /**
256     * 处理<=
257     * @param rule
258     */
259     private void processLessEqual(
260         Rule rule) {
261         if (ArrayUtils.isEmpty(rule.getValues())) {
262             return;
263         }
264         add(rule.getAndOr(), rule.getPropertyName(), "<=", rule.getValues()[0]);
265     }
266
267     /**
268     * 处理 is null
269     * @param rule
270     */
271     private void processIsNull(Rule rule) {
272         add(rule.getAndOr(), rule.getPropertyName(), "is null", null);
273     }
274
275     /**
276     * 处理 is not null
277     * @param rule
```

```

278     */
279     private void processIsNotNull(Rule rule) {
280         add(rule.getAndOr(),rule.getPropertyName(),"is not null",null);
281     }
282
283     /**
284      * 处理 <>'
285      * @param rule
286      */
287     private void processIsNotEmpty(Rule rule) {
288         add(rule.getAndOr(),rule.getPropertyName(),"<>", "'");
289     }
290
291     /**
292      * 处理 ='
293      * @param rule
294      */
295     private void processIsEmpty(Rule rule) {
296         add(rule.getAndOr(),rule.getPropertyName(),"=", "'");
297     }
298
299
300     /**
301      * 处理in和not in
302      * @param rule
303      * @param name
304      */
305     private void inAndNotIn(Rule rule,String name){
306         if (ArrayUtils.isEmpty(rule.getValues())) {
307             return;
308         }
309         if ((rule.getValues().length == 1) && (rule.getValues()[0] != null)
310             && (rule.getValues()[0] instanceof List)) {
311             List<Object> list = (List) rule.getValues()[0];
312
313             if ((list != null) && (list.size() > 0)){
314                 for (int i = 0; i < list.size(); i++) {
315                     if(i == 0 && i == list.size() - 1){
316                         add(rule.getAndOr(),rule.getPropertyName(),"",name + " (" ,list.
317                     }else if(i == 0 && i < list.size() - 1){
318                         add(rule.getAndOr(),rule.getPropertyName(),"",name + " (" ,list.
319                     }
320                     if(i > 0 && i < list.size() - 1){

```

```

321         add(0, "", "", "", list.get(i), "");
322     }
323     if(i == list.size() - 1 && i != 0){
324         add(0, "", "", "", list.get(i), "");
325     }
326 }
327 }
328 } else {
329     Object[] list = rule.getValues();
330     for (int i = 0; i < list.length; i++) {
331         if(i == 0 && i == list.length - 1){
332             add(rule.getAndOr(), rule.getPropertyName(), "", name + " (" ,list[i], "
333         }else if(i == 0 && i < list.length - 1){
334             add(rule.getAndOr(), rule.getPropertyName(), "", name + " (" ,list[i], "
335         }
336         if(i > 0 && i < list.length - 1){
337             add(0, "", "", "", list[i], "");
338         }
339         if(i == list.length - 1 && i != 0){
340             add(0, "", "", "", list[i], "");
341         }
342     }
343 }
344 }
345
346 /**
347  * 处理 not in
348  * @param rule
349  */
350 private void processNotIN(Rule rule){
351     inAndNotIn(rule, "not in");
352 }
353
354 /**
355  * 处理 in
356  * @param rule
357  */
358 private void processIN(Rule rule) {
359     inAndNotIn(rule, "in");
360 }
361
362 /**
363  * 处理 order by

```



```

364     * @param rule 查询规则
365     */
366     private void processOrder(Rule rule) {
367         switch (rule.getType()) {
368             case QueryRule.ASC_ORDER:
369                 // propertyName非空
370                 if (!StringUtils.isEmpty(rule.getPropertyName())) {
371                     orders.add(Order.asc(rule.getPropertyName()));
372                 }
373                 break;
374             case QueryRule.DESC_ORDER:
375                 // propertyName非空
376                 if (!StringUtils.isEmpty(rule.getPropertyName())) {
377                     orders.add(Order.desc(rule.getPropertyName()));
378                 }
379                 break;
380             default:
381                 break;
382         }
383     }
384
385
386     /**
387     * 加入到sql查询规则队列
388     * @param andOr and 或者 or
389     * @param key 列名
390     * @param split 列名与值之间的间隔
391     * @param value 值
392     */
393     private void add(int andOr,String key,String split ,Object value){
394         add(andOr,key,split,"",value,"");
395     }
396
397     /**
398     * 加入到sql查询规则队列
399     * @param andOr and 或则 or
400     * @param key 列名
401     * @param split 列名与值之间的间隔
402     * @param prefix 值前缀
403     * @param value 值
404     * @param suffix 值后缀
405     */
406     private void add(int andOr,String key,String split ,String prefix,Object value,String

```

```

407     String andOrStr = (0 == andOr ? "" : (QueryRule.AND == andOr ? " and " : " or "))
408     properties.add(CURR_INDEX, andOrStr + key + " " + split + prefix + (null != val
409     if(null != value){
410         values.add(CURR_INDEX,value);
411         CURR_INDEX ++;
412     }
413 }
414
415
416 /**
417  * 拼装 where 语句
418  */
419 private void appendWhereSql(){
420     StringBuffer whereSql = new StringBuffer();
421     for (String p : properties) {
422         whereSql.append(p);
423     }
424     this.whereSql = removeSelect(removeOrders(whereSql.toString()));
425 }
426
427 /**
428  * 拼装排序语句
429  */
430 private void appendOrderSql(){
431     StringBuffer orderSql = new StringBuffer();
432     for (int i = 0 ; i < orders.size(); i ++) {
433         if(i > 0 && i < orders.size()){
434             orderSql.append(",");
435         }
436         orderSql.append(orders.get(i).toString());
437     }
438     this.orderSql = removeSelect(removeOrders(orderSql.toString()));
439 }
440
441 /**
442  * 拼装参数值
443  */
444 private void appendValues(){
445     Object [] val = new Object[values.size()];
446     for (int i = 0; i < values.size(); i ++) {
447         val[i] = values.get(i);
448         valueMap.put(i, values.get(i));
449     }

```

```
450         this.valueArr = val;
451     }
452
453 }
```

BaseDaoSupport

```
1 package cn.sitedev.orm.framework;
2
3 import com.alibaba.fastjson.util.FieldInfo;
4 import com.alibaba.fastjson.util.TypeUtils;
5 import org.apache.log4j.Logger;
6 import org.springframework.dao.DataAccessException;
7 import org.springframework.dao.support.DataAccessUtils;
8 import org.springframework.jdbc.core.JdbcTemplate;
9 import org.springframework.jdbc.core.PreparedStatementCreator;
10 import org.springframework.jdbc.core.RowMapper;
11 import org.springframework.jdbc.support.GeneratedKeyHolder;
12 import org.springframework.jdbc.support.KeyHolder;
13
14 import javax.core.common.Page;
15 import javax.core.common.jdbc.BaseDao;
16 import javax.core.common.utils.BeanUtils;
17 import javax.core.common.utils.DataUtils;
18 import javax.core.common.utils.GenericsUtils;
19 import javax.core.common.utils.StringUtils;
20 import javax.sql.DataSource;
21 import java.io.*;
22 import java.lang.reflect.Field;
23 import java.lang.reflect.InvocationTargetException;
24 import java.sql.*;
25 import java.util.*;
26 import java.util.regex.Matcher;
27 import java.util.regex.Pattern;
28
29 /**
30  * BaseDao 扩展类,主要功能是支持自动拼装sql语句, 必须继承方可使用
31  * 需要重写和实现以下三个方法
32  * //设定主键列
33  * private String getPKColumn() {return "id";}
34  * //重写对象反转为Map的方法
```

```

35 *     protected Map<String, Object> parse(Object entity) {return utils.parse((Entity)e
36 *     //重写结果反转为对象的方法
37 *     protected Entity mapRow(ResultSet rs, int rowNum) throws SQLException {return ut
38 *
39 *
40 */
41 public abstract class BaseDaoSupport<T extends Serializable, PK extends Serializable>
42     private Logger log = Logger.getLogger(BaseDaoSupport.class);
43
44     private String tableName = "";
45
46     private JdbcTemplate jdbcTemplateWrite;
47     private JdbcTemplate jdbcTemplateReadOnly;
48
49     private DataSource dataSourceReadOnly;
50     private DataSource dataSourceWrite;
51
52     private EntityOperation<T> op;
53
54     @SuppressWarnings("unchecked")
55     protected BaseDaoSupport(){
56         try{
57             //      Class<T> entityClass = (Class<T>)((ParameterizedType) getClass().getGene
58             Class<T> entityClass = GenericsUtils.getSuperClassGenricType(getClass(), 0
59             op = new EntityOperation<T>(entityClass, this.getPKColumn());
60             this.setTableName(op.tableName);
61         }catch(Exception e){
62             e.printStackTrace();
63         }
64     }
65
66     protected String getTableName() {
67         return tableName;
68     }
69
70     protected DataSource getDataSourceReadOnly() {
71         return dataSourceReadOnly;
72     }
73
74     protected DataSource getDataSourceWrite() {
75         return dataSourceWrite;
76     }
77

```

```
78  /**
79   * 动态切换表名
80   */
81  protected void setTableName(String tableName) {
82      if(StringUtils.isEmpty(tableName)){
83          this.tableName = op.tableName;
84      }else{
85          this.tableName = tableName;
86      }
87  }
88
89  protected void setDataSourceWrite(DataSource dataSourceWrite) {
90      this.dataSourceWrite = dataSourceWrite;
91      jdbcTemplateWrite = new JdbcTemplate(dataSourceWrite);
92  }
93
94  protected void setDataSourceReadOnly(DataSource dataSourceReadOnly) {
95      this.dataSourceReadOnly = dataSourceReadOnly;
96      jdbcTemplateReadOnly = new JdbcTemplate(dataSourceReadOnly);
97  }
98
99  private JdbcTemplate jdbcTemplateReadOnly() {
100      return this.jdbcTemplateReadOnly;
101  }
102
103  private JdbcTemplate jdbcTemplateWrite() {
104      return this.jdbcTemplateWrite;
105  }
106
107
108  /**
109   * 还原默认表名
110   */
111  protected void restoreTableName(){
112      this.setTableName(op.tableName);
113  }
114
115  /**
116   * 将对象解析为Map
117   * @param entity
118   * @return
119   */
120  protected Map<String,Object> parse(T entity){
```

```
121         return op.parse(entity);
122     }
123
124
125
126     /**
127      * 根据ID获取对象. 如果对象不存在, 返回null.<br>
128      */
129     protected T get(PK id) throws Exception {
130         return (T) this.doLoad(id, this.op.rowMapper);
131     }
132
133     /**
134      * 获取全部对象. <br>
135      *
136      * @return 全部对象
137      */
138     protected List<T> getAll() throws Exception {
139         String sql = "select " + op.allColumn + " from " + getTableName();
140         return this.jdbcTemplateReadOnly().query(sql, this.op.rowMapper, new HashMap<S
141     }
142
143     /**
144      * 插入并返回id
145      * @param entity
146      * @return
147      */
148     public PK insertAndReturnId(T entity) throws Exception{
149         return (PK)this.doInsertRuturnKey(parse(entity));
150     }
151
152     /**
153      * 插入一条记录
154      * @param entity
155      * @return
156      */
157     public boolean insert(T entity) throws Exception{
158         return this.doInsert(parse(entity));
159     }
160
161
162     /**
163      * 保存对象,如果对象存在则更新,否则插入.<br>
```

```

164     * </code>
165     * </pre>
166     * @throws IllegalAccessException
167     * @throws IllegalArgumentException
168     */
169     protected boolean save(T entity) throws Exception {
170         PK pkValue = (PK)op.pkField.get(entity);
171         if(this.exists(pkValue)){
172             return this.doUpdate(pkValue, parse(entity)) > 0;
173         }else{
174             return this.doInsert(parse(entity));
175         }
176     }
177
178     /**
179     * 保存并返回新的id,如果对象存在则更新,否则插入
180     * @param entity
181     * @return
182     * @throws IllegalAccessException
183     * @throws IllegalArgumentException
184     */
185     protected PK saveAndReturnId(T entity) throws Exception{
186         Object o = op.pkField.get(entity);
187         if(null == o){
188             return (PK)this.doInsertReturnKey(parse(entity));
189             //return (PK)id;
190         }
191         PK pkValue = (PK)o;
192         if(this.exists(pkValue)){
193             this.doUpdate(pkValue, parse(entity));
194             return pkValue;
195         }else{
196             return (PK)this.doInsertReturnKey(parse(entity));
197         }
198     }
199
200     /**
201     * 更新对象.<br>
202     * 例如： 以下代码将对象更新到数据库
203     * <pre>
204     *         <code>
205     *         User entity = service.get(1);
206     *         entity.setName("&quot;zzz&quot;");

```

```

207     * // 更新对象
208     * service.update(entity);
209     * </code>
210     * </pre>
211     *
212     * @param entity 待更新对象
213     * @throws IllegalAccessException
214     * @throws IllegalArgumentException
215     */
216     public boolean update(T entity) throws Exception {
217         return this.doUpdate(op.pkField.get(entity), parse(entity)) > 0;
218     }
219
220     /**
221     * 使用SQL语句更新对象.<br>
222     * 例如：以下代码将更新id="0002"的name值更新为“张三”到数据库
223     * <pre>
224     *         <code>
225     * String name = "张三";
226     * String id = "0002";
227     * String sql = "UPDATE SET name = ? WHERE id = ?";
228     * // 更新对象
229     * service.update(sql,name,id)
230     * </code>
231     * </pre>
232     *
233     * @param sql 更新sql语句
234     * @param args 参数对象
235     *
236     * @return 更新记录数
237     */
238     protected int update(String sql,Object... args) throws Exception{
239         return jdbcTemplateWrite().update(sql, args);
240     }
241
242     /**
243     * 使用SQL语句更新对象.<br>
244     * 例如：以下代码将更新id="0002"的name值更新为“张三”到数据库
245     * <pre>
246     *         <code>
247     * Map<String,Object> map = new HashMap();
248     * map.put("name","张三");
249     * map.put("id","0002");

```



```

250     * String sql = "UPDATE SET name = :name WHERE id = :id";
251     * // 更新对象
252     * service.update(sql,map)
253     * </code>
254     * </pre>
255     *
256     * @param sql 更新sql语句
257     * @param paramMap 参数对象
258     *
259     * @return 更新记录数
260     */
261     protected int update(String sql,Map<String,?> paramMap) throws Exception{
262         return jdbcTemplateWrite().update(sql, paramMap);
263     }
264     /**
265     * 批量保存对象.<br>
266     * 例如： 以下代码将对象保存到数据库
267     * <pre>
268     *         <code>
269     * List<Role> list = new ArrayList<Role>();
270     * for (int i = 1; i <= 8; i++) {
271     *     Role role = new Role();
272     *     role.setId(i);
273     *     role.setRolename("管理" + i);
274     *     role.setPrivilegesFlag("1,2,3");
275     *     list.add(role);
276     * }
277     * service.insertAll(list);
278     * </code>
279     * </pre>
280     *
281     * @param list 待保存的对象List
282     * @throws InvocationTargetException
283     * @throws IllegalArgumentException
284     * @throws IllegalAccessException
285     */
286     public int insertAll(List<T> list) throws Exception {
287         int count = 0 ,len = list.size(),step = 50000;
288         Map<String, PropertyMapping> pm = op.mappings;
289         int maxPage = (len % step == 0) ? (len / step) : (len / step + 1);
290         for (int i = 1; i <= maxPage; i++) {
291             Page<T> page = pagination(list, i, step);
292             String sql = "insert into " + getTableName() + "(" + op.allColumn + ") val

```

```

293         StringBuffer valstr = new StringBuffer();
294         Object[] values = new Object[pm.size() * page.getRows().size()];
295         for (int j = 0; j < page.getRows().size(); j++) {
296             if(j > 0 && j < page.getRows().size()){ valstr.append(","); }
297             valstr.append("(");
298             int k = 0;
299             for (PropertyMapping p : pm.values()) {
300                 values[(j * pm.size()) + k] = p.getter.invoke(page.getRows().get(j));
301                 if(k > 0 && k < pm.size()){ valstr.append(","); }
302                 valstr.append("?");
303                 k++;
304             }
305             valstr.append(")");
306         }
307         int result = jdbcTemplateWrite().update(sql + valstr.toString(), values);
308         count += result;
309     }
310
311     return count;
312 }
313
314
315 protected boolean replaceOne(T entity) throws Exception{
316     return this.doReplace(parse(entity));
317 }
318
319
320 protected int replaceAll(List<T> list) throws Exception {
321     int count = 0 ,len = list.size(),step = 50000;
322     Map<String, PropertyMapping> pm = op.mappings;
323     int maxPage = (len % step == 0) ? (len / step) : (len / step + 1);
324     for (int i = 1; i <= maxPage; i++) {
325         Page<T> page = pagination(list, i, step);
326         String sql = "replace into " + getTableName() + "(" + op.allColumn + ") va
327         StringBuffer valstr = new StringBuffer();
328         Object[] values = new Object[pm.size() * page.getRows().size()];
329         for (int j = 0; j < page.getRows().size(); j++) {
330             if(j > 0 && j < page.getRows().size()){ valstr.append(","); }
331             valstr.append("(");
332             int k = 0;
333             for (PropertyMapping p : pm.values()) {
334                 values[(j * pm.size()) + k] = p.getter.invoke(page.getRows().get(j));
335                 if(k > 0 && k < pm.size()){ valstr.append(","); }

```

```

336         valstr.append("?");
337         k ++;
338     }
339     valstr.append(")");
340 }
341 int result = jdbcTemplateWrite().update(sql + valstr.toString(), values);
342 count += result;
343 }
344 return count;
345 }
346
347
348 /**
349  * 删除对象.<br>
350  * 例如：以下删除entity对应的记录
351  * <pre>
352  *         <code>
353  * service.delete(entity);
354  * </code>
355  * </pre>
356  *
357  * @param entity 待删除的实体对象
358  */
359 public boolean delete(T entity) throws Exception {
360     return this.doDelete(op.pkField.get(entity)) > 0;
361 }
362
363 /**
364  * 删除对象.<br>
365  * 例如：以下删除entity对应的记录
366  * <pre>
367  *         <code>
368  * service.deleteAll(entityList);
369  * </code>
370  * </pre>
371  *
372  * @param list 待删除的实体对象列表
373  * @throws InvocationTargetException
374  * @throws IllegalArgumentException
375  * @throws IllegalAccessException
376  */
377 public int deleteAll(List<T> list) throws Exception {
378     String pkName = op.pkField.getName();

```

```

379     int count = 0 ,len = list.size(),step = 1000;
380     Map<String, PropertyMapping> pm = op.mappings;
381     int maxPage = (len % step == 0) ? (len / step) : (len / step + 1);
382     for (int i = 1; i <= maxPage; i++) {
383         StringBuffer valstr = new StringBuffer();
384         Page<T> page = pagination(list, i, step);
385         Object[] values = new Object[page.getRows().size()];
386
387         for (int j = 0; j < page.getRows().size(); j++) {
388             if(j > 0 && j < page.getRows().size()){ valstr.append(","); }
389             values[j] = pm.get(pkName).getter.invoke(page.getRows().get(j));
390             valstr.append("?");
391         }
392
393         String sql = "delete from " + getTableName() + " where " + pkName + " in (
394             int result = jdbcTemplateWrite().update(sql, values);
395             count += result;
396         }
397         return count;
398     }
399
400     /**
401     * 根据ID删除对象.如果有记录则删之，没有记录也不报异常<br>
402     * 例如：以下删除主键唯一的记录
403     * <pre>
404     *         <code>
405     * service.deleteByPK(1);
406     * </code>
407     * </pre>
408     *
409     * @param id 序列化对id
410     */
411     protected void deleteByPK(PK id) throws Exception {
412         this.doDelete(id);
413     }
414
415     /**
416     * 根据ID删除对象.如果有记录则删之，没有记录也不报异常<br>
417     * 例如：以下删除主键唯一的记录
418     * <pre>
419     *         <code>
420     * service.delete(1);
421     * </code>

```

```

422     * </pre>
423     *
424     * @param id 序列化对id
425     *
426     * @return 删除是否成功
427     */
428 //     protected boolean delete(PK id) throws Exception {
429 //         return this.doDelete(id) > 0;
430 //     }
431
432 /**
433     * 根据属性名查询出内容等于属性值的唯一对象，没符合条件的记录返回null.<br>
434     * 例如，如下语句查找id=5的唯一记录：
435     *
436     * <pre>
437     *     <code>
438     * User user = service.selectUnique(User.class, &quot;id&quot;, 5);
439     * </code>
440     * </pre>
441     *
442     * @param propertyName 属性名
443     * @param value 属性值
444     * @return 符合条件的唯一对象 or null if not found.
445     */
446 protected T selectUnique(String propertyName, Object value) throws Exception {
447     QueryRule queryRule = QueryRule.getInstance();
448     queryRule.andEqual(propertyName, value);
449     return this.selectUnique(queryRule);
450 }
451
452 /**
453     * 根据主键判断对象是否存在。例如：以下代码判断id=2的User记录是否存在
454     *
455     * <pre>
456     *     <code>
457     * boolean user2Exist = service.exists(User.class, 2);
458     * </code>
459     * </pre>
460     * @param id 序列化对象id
461     * @return 存在返回true，否则返回false
462     */
463 protected boolean exists(PK id) throws Exception {
464     return null != this.doLoad(id, this.op.rowMapper);

```

```

465     }
466
467     /**
468      * 查询满足条件的记录数，使用hql.<br>
469      * 例如：查询User里满足条件?name like "%ca%" 的记录数
470      *
471      * <pre>
472      *         <code>
473      * long count = service.getCount("&quot;from User where name like ?&quot;;, &quot;%c
474      * </code>
475      * </pre>
476      *
477      * @param queryRule
478      * @return 满足条件的记录数
479      */
480     protected long getCount(QueryRule queryRule) throws Exception {
481         QueryRuleSqlBuilder bulider = new QueryRuleSqlBuilder(queryRule);
482         Object [] values = bulider.getValues();
483         String ws = removeFirstAnd(bulider.getWhereSql());
484         String whereSql = ("".equals(ws) ? ws : (" where " + ws));
485         String countSql = "select count(1) from " + getTableName() + whereSql;
486         return (Long) this.jdbcTemplateReadOnly().queryForMap(countSql, values).get("c
487     }
488
489     /**
490      * 根据某个属性值倒序获得第一个最大值
491      * @param propertyName
492      * @return
493      */
494     protected T getMax(String propertyName) throws Exception{
495         QueryRule queryRule = QueryRule.getInstance();
496         queryRule.addDescOrder(propertyName);
497         Page<T> result = this.select(queryRule,1,1);
498         if(null == result.getRows() || 0 == result.getRows().size()){
499             return null;
500         }else{
501             return result.getRows().get(0);
502         }
503     }
504
505     /**
506      * 查询函数，使用查询规
507      * 例如以下代码查询条件为匹配的数据

```

```

508 *
509 * <pre>
510 *     <code>
511 * QueryRule queryRule = QueryRule.getInstance();
512 * queryRule.addLike(&quot;username&quot;; user.getUsername());
513 * queryRule.addLike(&quot;monicker&quot;; user.getMonicker());
514 * queryRule.addBetween(&quot;id&quot;; lowerId, upperId);
515 * queryRule.addDescOrder(&quot;id&quot;);
516 * queryRule.addAscOrder(&quot;username&quot;);
517 * list = userService.select(User.class, queryRule);
518 * </code>
519 * </pre>
520 *
521 * @param queryRule 查询规则
522 * @return 查询出的结果List
523 */
524 public List<T> select(QueryRule queryRule) throws Exception{
525     QueryRuleSqlBuilder bulider = new QueryRuleSqlBuilder(queryRule);
526     String ws = removeFirstAnd(bulider.getWhereSql());
527     String whereSql = ("".equals(ws) ? ws : (" where " + ws));
528     String sql = "select " + op.allColumn + " from " + getTableName() + whereSql;
529     Object [] values = bulider.getValues();
530     String orderSql = bulider.getOrderSql();
531     orderSql = (StringUtils.isEmpty(orderSql) ? " " : (" order by " + orderSql));
532     sql += orderSql;
533     log.debug(sql);
534     return (List<T>) this.jdbcTemplateReadOnly().query(sql, this.op.rowMapper, val
535 }
536
537 /**
538 * 根据SQL语句执行查询，参数为Map
539 * @param sql 语句
540 * @param pamam 为Map，key为属性名，value为属性值
541 * @return 符合条件的所有对象
542 */
543 protected List<Map<String,Object>> selectBySql(String sql,Map<String,?> pamam) thr
544     return this.jdbcTemplateReadOnly().queryForList(sql,pamam);
545 }
546
547 /**
548 * 根据SQL语句查询符合条件的唯一对象，没符合条件的记录返回null.<br>
549 * @param sql 语句
550 * @param pamam 为Map，key为属性名，value为属性值

```

```

551     * @return 符合条件的唯一对象，没符合条件的记录返回null.
552     */
553     protected Map<String,Object> selectUniqueBySql(String sql,Map<String,?> pamam) throws Exception {
554         List<Map<String,Object>> list = selectBySql(sql,pamam);
555         if (list.size() == 0) {
556             return null;
557         } else if (list.size() == 1) {
558             return list.get(0);
559         } else {
560             throw new IllegalStateException("findUnique return " + list.size() + " records");
561         }
562     }
563
564     /**
565     * 根据SQL语句执行查询，参数为Object数组对象
566     * @param sql 查询语句
567     * @param args 为Object数组
568     * @return 符合条件的所有对象
569     */
570     public List<Map<String,Object>> selectBySql(String sql,Object... args) throws Exception {
571         return this.jdbcTemplateReadOnly().queryForList(sql,args);
572     }
573
574     /**
575     * 根据SQL语句查询符合条件的唯一对象，没符合条件的记录返回null.<br>
576     * @param sql 查询语句
577     * @param args 为Object数组
578     * @return 符合条件的唯一对象，没符合条件的记录返回null.
579     */
580     protected Map<String,Object> selectUniqueBySql(String sql,Object... args) throws Exception {
581         List<Map<String,Object>> list = selectBySql(sql, args);
582         if (list.size() == 0) {
583             return null;
584         } else if (list.size() == 1) {
585             return list.get(0);
586         } else {
587             throw new IllegalStateException("findUnique return " + list.size() + " records");
588         }
589     }
590
591     /**
592     * 根据SQL语句执行查询，参数为List对象
593     * @param sql 查询语句

```



```

594     * @param list<Object>对象
595     * @return 符合条件的所有对象
596     */
597     protected List<Map<String,Object>> selectBySql(String sql,List<Object> list) throw
598         return this.jdbcTemplateReadOnly().queryForList(sql,list.toArray());
599 }
600
601 /**
602     * 根据SQL语句查询符合条件的唯一对象，没符合条件的记录返回null.<br>
603     * @param sql 查询语句
604     * @param listParam 属性值List
605     * @return 符合条件的唯一对象，没符合条件的记录返回null.
606     */
607     protected Map<String,Object> selectUniqueBySql(String sql,List<Object> listParam)
608         List<Map<String,Object>> listMap = selectBySql(sql, listParam);
609         if (listMap.size() == 0) {
610             return null;
611         } else if (listMap.size() == 1) {
612             return listMap.get(0);
613         } else {
614             throw new IllegalStateException("findUnique return " + listMap.size() + "
615         }
616     }
617
618 /**
619     * 分页查询函数，使用查询规则<br>
620     * 例如以下代码查询条件为匹配的数据
621     *
622     * <pre>
623     *         <code>
624     * QueryRule queryRule = QueryRule.getInstance();
625     * queryRule.addLike(&quot;username&quot;;, user.getUsername());
626     * queryRule.addLike(&quot;monicker&quot;;, user.getMonicker());
627     * queryRule.addBetween(&quot;id&quot;;, lowerId, upperId);
628     * queryRule.addDescOrder(&quot;id&quot;;);
629     * queryRule.addAscOrder(&quot;username&quot;;);
630     * page = userService.select(queryRule, pageNo, pageSize);
631     * </code>
632     * </pre>
633     *
634     * @param queryRule 查询规则
635     * @param pageNo 页号,从1开始
636     * @param pageSize 每页的记录条数

```

```

637     * @return 查询出的结果Page
638     */
639     public Page<T> select(QueryRule queryRule,final int pageNo, final int pageSize) th
640         QueryRuleSqlBuilder bulider = new QueryRuleSqlBuilder(queryRule);
641         Object [] values = bulider.getValues();
642         String ws = removeFirstAnd(bulider.getWhereSql());
643         String whereSql = ("".equals(ws) ? ws : (" where " + ws));
644         String countSql = "select count(1) from " + getTableName() + whereSql;
645         long count = (Long) this.jdbcTemplateReadOnly().queryForMap(countSql, values).
646         if (count == 0) {
647             return new Page<T>();
648         }
649         long start = (pageNo - 1) * pageSize;
650         // 有数据的情况下，继续查询
651         String orderSql = bulider.getOrderSql();
652         orderSql = (StringUtil.isEmpty(orderSql) ? " " : (" order by " + orderSql));
653         String sql = "select " + op.allColumn + " from " + getTableName() + whereSql +
654         List<T> list = (List<T>) this.jdbcTemplateReadOnly().query(sql, this.op.rowMap
655         log.debug(sql);
656         return new Page<T>(start, count, pageSize, list);
657     }
658
659
660     /**
661     * 分页查询特殊SQL语句
662     * @param sql 语句
663     * @param param 查询条件
664     * @param pageNo 页码
665     * @param pageSize 每页内容
666     * @return
667     */
668     protected Page<Map<String,Object>> selectBySqlToPage(String sql, Map<String,?> par
669         String countSql = "select count(1) from (" + sql + ") a";
670         long count = (Long) this.jdbcTemplateReadOnly().queryForMap(countSql,param).ge
671
672         //         long count = this.jdbcTemplateReadOnly().queryForMap(countSql, param);
673         if (count == 0) {
674             return new Page<Map<String,Object>>();
675         }
676         long start = (pageNo - 1) * pageSize;
677         // 有数据的情况下，继续查询
678         sql = sql + " limit " + start + "," + pageSize;
679         List<Map<String,Object>> list = (List<Map<String,Object>>) this.jdbcTemplateRe

```

```

680         log.debug(sql);
681         return new Page<Map<String,Object>>(start, count, pageSize, list);
682     }
683
684
685     /**
686      * 分页查询特殊SQL语句
687      * @param sql 语句
688      * @param param 查询条件
689      * @param pageNo 页码
690      * @param pageSize 每页内容
691      * @return
692      */
693     public Page<Map<String,Object>> selectBySqlToPage(String sql, Object [] param, fir
694         String countSql = "select count(1) from (" + sql + ") a";
695
696         long count = (Long) this.jdbcTemplateReadOnly().queryForMap(countSql,param).get(0,Long.class);
697         // long count = this.jdbcTemplateReadOnly().queryForLong(countSql, param);
698         if (count == 0) {
699             return new Page<Map<String,Object>>();
700         }
701         long start = (pageNo - 1) * pageSize;
702         sql = sql + " limit " + start + "," + pageSize;
703         List<Map<String,Object>> list = (List<Map<String,Object>>) this.jdbcTemplateRe
704         log.debug(sql);
705         return new Page<Map<String,Object>>(start, count, pageSize, list);
706     }
707
708     /**
709      * 根据<属性名和属属性值Map查询符合条件的唯一对象，没符合条件的记录返回null.<br>
710      * 例如，如下语句查找sex=1,age=18的所有记录：
711      *
712      * <pre>
713      *     <code>
714      *     Map properties = new HashMap();
715      *     properties.put("sex", "1");
716      *     properties.put("age", 18);
717      *     User user = service.selectUnique(properties);
718      *     </code>
719      * </pre>
720      *
721      * @param properties 属性值Map，key为属性名，value为属性值
722      * @return 符合条件的唯一对象，没符合条件的记录返回null.

```

```

723     */
724     protected T selectUnique(Map<String, Object> properties) throws Exception {
725         QueryRule queryRule = QueryRule.getInstance();
726         for (String key : properties.keySet()) {
727             queryRule.andEqual(key, properties.get(key));
728         }
729         return selectUnique(queryRule);
730     }
731
732     /**
733      * 根据查询规则查询符合条件的唯一象，没符合条件的记录返回null.<br>
734      * <pre>
735      *     <code>
736      *     QueryRule queryRule = QueryRule.getInstance();
737      *     queryRule.addLike(&quot;username&quot;, user.getUsername());
738      *     queryRule.addLike(&quot;monicker&quot;, user.getMonicker());
739      *     queryRule.addBetween(&quot;id&quot;, lowerId, upperId);
740      *     User user = service.selectUnique(queryRule);
741      *     </code>
742      * </pre>
743      *
744      * @param queryRule 查询规则
745      * @return 符合条件的唯一对象，没符合条件的记录返回null.
746      */
747     protected T selectUnique(QueryRule queryRule) throws Exception {
748         List<T> list = select(queryRule);
749         if (list.size() == 0) {
750             return null;
751         } else if (list.size() == 1) {
752             return list.get(0);
753         } else {
754             throw new IllegalStateException("findUnique return " + list.size() + " records");
755         }
756     }
757
758
759     /**
760      * 根据当前list进行相应的分页返回
761      * @param objList
762      * @param pageNo
763      * @param pageSize
764      * @return Page
765      */

```

```

766     protected Page<T> pagination(List<T> objList, int pageNo, int pageSize) throws Exc
767         List<T> objectArray = new ArrayList<T>(0);
768         int startIndex = (pageNo - 1) * pageSize;
769         int endIndex = pageNo * pageSize;
770         if(endIndex >= objList.size()){
771             endIndex = objList.size();
772         }
773         for (int i = startIndex; i < endIndex; i++) {
774             objectArray.add(objList.get(i));
775         }
776         return new Page<T>(startIndex, objList.size(), pageSize, objectArray);
777     }
778
779     /**
780      * 合并PO List对象.(如果POJO中的值为null,则继续使用PO中的值)
781      *
782      * @param pojoList 传入的POJO的List
783      * @param poList 传入的PO的List
784      * @param idName ID字段名称
785      */
786     protected void mergeList(List<T> pojoList, List<T> poList, String idName) throws E
787         mergeList(pojoList, poList, idName, false);
788     }
789
790     /**
791      * 合并PO List对象.
792      *
793      * @param pojoList 传入的POJO的List
794      * @param poList 传入的PO的List
795      * @param idName ID字段名称
796      * @param isCopyNull 是否拷贝null(当POJO中的值为null时, 如果isCopyNull=true,则用nu
797      */
798     protected void mergeList(List<T> pojoList, List<T> poList, String idName,boolean i
799         Map<Object, Object> map = new HashMap<Object, Object>();
800         Map<String, PropertyMapping> pm = op.mappings;
801         for (Object element : pojoList) {
802             Object key;
803             try {
804                 key = pm.get(idName).getter.invoke(element);
805                 map.put(key, element);
806             } catch (Exception e) {
807                 throw new IllegalArgumentException(e);
808             }

```

```

809     }
810     for (Iterator<T> it = poList.iterator(); it.hasNext();) {
811         T element = it.next();
812         try {
813             Object key = pm.get(idName).getter.invoke(element);
814             if (!map.containsKey(key)) {
815                 delete(element);
816                 it.remove();
817             } else {
818                 DataUtils.copySimpleObject(map.get(key), element, isCopyNull);
819             }
820         } catch (Exception e) {
821             throw new IllegalArgumentException(e);
822         }
823     }
824     T[] pojoArray = (T[])pojoList.toArray();
825     for (int i = 0; i < pojoArray.length; i++) {
826         T element = pojoArray[i];
827         try {
828             Object key = pm.get(idName).getter.invoke(element);
829             if (key == null) {
830                 poList.add(element);
831             }
832         } catch (Exception e) {
833             throw new IllegalArgumentException(e);
834         }
835     }
836 }
837
838 private String removeFirstAnd(String sql){
839     if(StringUtils.isEmpty(sql)){return sql;}
840     return sql.trim().toLowerCase().replaceAll("^\\s*and", "") + " ";
841 }
842
843 private EntityOperation<T> getOp(){
844     return this.op;
845 }
846
847
848
849 /**
850  * ResultSet -> Object
851  *

```

```

852     * @param <T>
853     *
854     * @param rs
855     * @param obj
856     */
857     private <T> T populate(ResultSet rs, T obj) {
858         try {
859             ResultSetMetaData metaData = rs.getMetaData(); // 取得结果集的元元素
860             int colCount = metaData.getColumnCount(); // 取得所有列的个数
861             Field[] fields = obj.getClass().getDeclaredFields();
862             for (int i = 0; i < fields.length; i++) {
863                 Field f = fields[i];
864                 // rs的游标从1开始，需要注意
865                 for (int j = 1; j <= colCount; j++) {
866                     Object value = rs.getObject(j);
867                     String colName = metaData.getColumnName(j);
868                     if (!f.getName().equalsIgnoreCase(colName)) {
869                         continue;
870                     }
871
872                     // 如果列名中有和字段名一样的，则设置值
873                     try {
874                         BeanUtils.copyProperty(obj, f.getName(), value);
875                     } catch (Exception e) {
876                         log.warn("BeanUtils.copyProperty error, field name: "
877                             + f.getName() + ", error: " + e);
878                     }
879
880                 }
881             }
882         } catch (Exception e) {
883             log.warn("populate error...." + e);
884         }
885         return obj;
886     }
887
888     /**
889     * 封装一下JdbcTemplate的queryForObject（默认查不到会抛异常）方法，
890     *
891     * @param sql
892     * @param mapper
893     * @param args
894     * @return 如查询不到，返回null，不抛异常；查询到多个，也抛出异常

```

```

895     */
896     private <T> T selectForObject(String sql, RowMapper<T> mapper,
897         Object... args) {
898         List<T> results = this.jdbcTemplateReadOnly().query(sql, mapper, args);
899         return DataAccessUtils.singleResult(results);
900     }
901
902     protected byte[] getBlobColumn(ResultSet rs, int columnIndex)
903         throws SQLException {
904         try {
905             Blob blob = rs.getBlob(columnIndex);
906             if (blob == null) {
907                 return null;
908             }
909
910             InputStream is = blob.getBinaryStream();
911             ByteArrayOutputStream bos = new ByteArrayOutputStream();
912
913             if (is == null) {
914                 return null;
915             } else {
916                 byte buffer[] = new byte[64];
917                 int c = is.read(buffer);
918                 while (c > 0) {
919                     bos.write(buffer, 0, c);
920                     c = is.read(buffer);
921                 }
922                 return bos.toByteArray();
923             }
924         } catch (IOException e) {
925             throw new SQLException(
926                 "Failed to read BLOB column due to IOException: "
927                 + e.getMessage());
928         }
929     }
930
931     protected void setBlobColumn(PreparedStatement stmt, int parameterIndex,
932         byte[] value) throws SQLException {
933         if (value == null) {
934             stmt.setNull(parameterIndex, Types.BLOB);
935         } else {
936             stmt.setBinaryStream(parameterIndex,
937                 new ByteArrayInputStream(value), value.length);

```



```

938     }
939 }
940
941 protected String getClobColumn(ResultSet rs, int columnIndex)
942     throws SQLException {
943     try {
944         Clob clob = rs.getClob(columnIndex);
945         if (clob == null) {
946             return null;
947         }
948
949         StringBuffer ret = new StringBuffer();
950         InputStream is = clob.getAsciiStream();
951
952         if (is == null) {
953             return null;
954         } else {
955             byte buffer[] = new byte[64];
956             int c = is.read(buffer);
957             while (c > 0) {
958                 ret.append(new String(buffer, 0, c));
959                 c = is.read(buffer);
960             }
961             return ret.toString();
962         }
963     } catch (IOException e) {
964         throw new SQLException(
965             "Failed to read CLOB column due to IOException: "
966             + e.getMessage());
967     }
968 }
969
970 protected void setClobColumn(PreparedStatement stmt, int parameterIndex,
971     String value) throws SQLException {
972     if (value == null) {
973         stmt.setNull(parameterIndex, Types.CLOB);
974     } else {
975         stmt.setAsciiStream(parameterIndex,
976             new ByteArrayInputStream(value.getBytes()), value.length());
977     }
978 }
979
980 /**

```

```

981      * 分页查询支持，支持简单的sql查询分页（复杂的查询，请自行编写对应的方法）
982      * @param <T>
983      *
984      * @param sql
985      * @param rowMapper
986      * @param args
987      * @param pageNo
988      * @param pageSize
989      * @return
990      */
991     private <T> Page simplePageQuery(String sql, RowMapper<T> rowMapper, Map<String, ?> args,
992         long start = (pageNo - 1) * pageSize;
993         return simplePageQueryByStart(sql, rowMapper, args, start, pageSize);
994     }
995
996     /**
997      *
998      * @param sql
999      * @param rowMapper
1000     * @param args
1001     * @param start
1002     * @param pageSize
1003     * @return
1004     */
1005     private <T> Page simplePageQueryByStart(String sql, RowMapper<T> rowMapper, Map<String, ?> args,
1006         // 首先查询总数
1007         String countSql = "select count(*) " + removeSelect(removeOrders(sql));
1008
1009         long count = (Long) this.jdbcTemplateReadOnly().queryForMap(countSql, args).get("count");
1010     //     long count = this.jdbcTemplateReadOnly().queryForLong(countSql, args);
1011         if (count == 0) {
1012             log.debug("no result..");
1013             return new Page();
1014         }
1015         // 有数据的情况下，继续查询
1016         sql = sql + " limit " + start + "," + pageSize;
1017         log.debug(StringUtils.format("[Execute SQL]sql:{0},params:{1}", sql, args));
1018         List<T> list = this.jdbcTemplateReadOnly().query(sql, rowMapper, args);
1019         return new Page(start, count, (int)pageSize, list);
1020     }
1021
1022     protected long queryCount(String sql, Map<String, ?> args){
1023         String countSql = "select count(1) " + removeSelect(removeOrders(sql));

```

```

1024
1025         return (Long)this.jdbcTemplateReadOnly().queryForMap(countSql, args).get("count");
1026     }
1027
1028     protected <T> List<T> simpleListQueryByStart(String sql, RowMapper<T> rowMapper,
1029         Map<String, ?> args, long start, long pageSize) {
1030
1031         sql = sql + " limit " + start + "," + pageSize;
1032         log.debug(StringUtils.format("[Execute SQL]sql:{0},params:{1}", sql, args));
1033         List<T> list = this.jdbcTemplateReadOnly().query(sql, rowMapper, args);
1034         if(list == null){
1035             return new ArrayList<T>();
1036         }
1037         return list;
1038     }
1039
1040     /**
1041      * 分页查询支持，支持简单的sql查询分页（复杂的查询，请自行编写对应的方法）
1042      *
1043      * @param sql
1044      * @param rm
1045      * @param args
1046      * @param pageNo
1047      * @param pageSize
1048      * @return
1049      */
1050     private Page simplePageQueryNotT(String sql, RowMapper rm, Map<String, ?> args, long count) {
1051         // 首先查询总数
1052         String countSql = "select count(*) " + removeSelect(removeOrders(sql));
1053         long count = (Long)this.jdbcTemplateReadOnly().queryForMap(countSql, args).get("count");
1054         if (count == 0) {
1055             log.debug("no result..");
1056             return new Page();
1057         }
1058         // 有数据的情况下，继续查询
1059         long start = (pageNo - 1) * pageSize;
1060         sql = sql + " limit " + start + "," + pageSize;
1061         log.debug(StringUtils.format("[Execute SQL]sql:{0},params:{1}", sql, args));
1062         List list = this.jdbcTemplateReadOnly().query(sql, rm, args);
1063         return new Page(start, count, (int)pageSize, list);
1064     }
1065
1066     /**

```

```

1067     * 去掉order
1068     *
1069     * @param sql
1070     * @return
1071     */
1072     private String removeOrders(String sql) {
1073         Pattern p = Pattern.compile("order\\s*by[\\w|\\W|\\s|\\S]*", Pattern.CASE_INSE
1074         Matcher m = p.matcher(sql);
1075         StringBuffer sb = new StringBuffer();
1076         while (m.find()) {
1077             m.appendReplacement(sb, "");
1078         }
1079         m.appendTail(sb);
1080         return sb.toString();
1081     }
1082
1083     /**
1084     * 去掉select
1085     *
1086     * @param sql
1087     * @return
1088     */
1089     private String removeSelect(String sql) {
1090         int beginPos = sql.toLowerCase().indexOf("from");
1091         return sql.substring(beginPos);
1092     }
1093
1094
1095     private long getMaxId(String table, String column) {
1096         String sql = "SELECT max(" + column + ") FROM " + table + " ";
1097         long maxId = (Long)this.jdbcTemplateReadOnly().queryForMap(sql).get("max(" + c
1098         return maxId;
1099     }
1100
1101     /**
1102     * 生成简单对象UPDATE语句，简化sql拼接
1103     * @param tableName
1104     * @param pkName
1105     * @param pkValue
1106     * @param params
1107     * @return
1108     */
1109     private String makeSimpleUpdateSql(String tableName, String pkName, Object pkValue

```

```

1110         if(StringUtils.isEmpty(tableName) || params == null || params.isEmpty()){
1111             return "";
1112         }
1113
1114         StringBuffer sb = new StringBuffer();
1115         sb.append("update ").append(tableName).append(" set ");
1116         //添加参数
1117         Set<String> set = params.keySet();
1118         int index = 0;
1119         for (String key : set) {
1120             // sb.append(key).append(" = :").append(key);
1121             sb.append(key).append(" = ?");
1122             if(index != set.size() - 1){
1123                 sb.append(",");
1124             }
1125             index++;
1126         }
1127         // sb.append(" where ").append(pkName).append(" = :").append(pkName) ;
1128         sb.append(" where ").append(pkName).append(" = ?");
1129         params.put("where_" + pkName,params.get(pkName));
1130
1131         return sb.toString();
1132     }
1133
1134
1135     /**
1136      * 生成简单对象UPDATE语句，简化sql拼接
1137      * @param pkName
1138      * @param pkValue
1139      * @param params
1140      * @return
1141      */
1142     private String makeSimpleUpdateSql(String pkName, Object pkValue, Map<String, Object> params) {
1143         if(StringUtils.isEmpty(getTableName()) || params == null || params.isEmpty()){
1144             return "";
1145         }
1146
1147         StringBuffer sb = new StringBuffer();
1148         sb.append("update ").append(getTableName()).append(" set ");
1149         //添加参数
1150         Set<String> set = params.keySet();
1151         int index = 0;
1152         for (String key : set) {

```

```

1153         sb.append(key).append(" = :").append(key);
1154         if(index != set.size() - 1){
1155             sb.append(",");
1156         }
1157         index++;
1158     }
1159     sb.append(" where ").append(pkName).append(" = :").append(pkName) ;
1160
1161     return sb.toString();
1162 }
1163
1164
1165
1166 /**
1167  * 生成对象INSERT语句，简化sql拼接
1168  * @param tableName
1169  * @param params
1170  * @return
1171  */
1172 private String makeSimpleReplaceSql(String tableName, Map<String, Object> params){
1173     if(StringUtils.isEmpty(tableName) || params == null || params.isEmpty()){
1174         return "";
1175     }
1176     StringBuffer sb = new StringBuffer();
1177     sb.append("replace into ").append(tableName);
1178
1179     StringBuffer sbKey = new StringBuffer();
1180     StringBuffer sbValue = new StringBuffer();
1181
1182     sbKey.append("(");
1183     sbValue.append("(");
1184     //添加参数
1185     Set<String> set = params.keySet();
1186     int index = 0;
1187     for (String key : set) {
1188         sbKey.append(key);
1189         sbValue.append(" :").append(key);
1190         if(index != set.size() - 1){
1191             sbKey.append(",");
1192             sbValue.append(",");
1193         }
1194         index++;
1195     }

```

```

1196         sbKey.append(")");
1197         sbValue.append(")");
1198
1199         sb.append(sbKey).append("VALUES").append(sbValue);
1200
1201         return sb.toString();
1202     }
1203
1204     /**
1205      * 生成对象INSERT语句，简化sql拼接
1206      * @param tableName
1207      * @param params
1208      * @return
1209      */
1210     private String makeSimpleReplaceSql(String tableName, Map<String, Object> params, List<Object> values) {
1211         if(StringUtils.isEmpty(tableName) || params == null || params.isEmpty()){
1212             return "";
1213         }
1214         StringBuffer sb = new StringBuffer();
1215         sb.append("replace into ").append(tableName);
1216
1217         StringBuffer sbKey = new StringBuffer();
1218         StringBuffer sbValue = new StringBuffer();
1219
1220         sbKey.append("(");
1221         sbValue.append("(");
1222         //添加参数
1223         Set<String> set = params.keySet();
1224         int index = 0;
1225         for (String key : set) {
1226             sbKey.append(key);
1227             sbValue.append(" ?");
1228             if(index != set.size() - 1){
1229                 sbKey.append(",");
1230                 sbValue.append(",");
1231             }
1232             index++;
1233             values.add(params.get(key));
1234         }
1235         sbKey.append(")");
1236         sbValue.append(")");
1237
1238         sb.append(sbKey).append("VALUES").append(sbValue);

```

```
1239
1240     return sb.toString();
1241 }
1242
1243
1244
1245 /**
1246  * 生成对象INSERT语句，简化sql拼接
1247  * @param tableName
1248  * @param params
1249  * @return
1250  */
1251 private String makeSimpleInsertSql(String tableName, Map<String, Object> params){
1252     if(StringUtils.isEmpty(tableName) || params == null || params.isEmpty()){
1253         return "";
1254     }
1255     StringBuffer sb = new StringBuffer();
1256     sb.append("insert into ").append(tableName);
1257
1258     StringBuffer sbKey = new StringBuffer();
1259     StringBuffer sbValue = new StringBuffer();
1260
1261     sbKey.append("(");
1262     sbValue.append("(");
1263     //添加参数
1264     Set<String> set = params.keySet();
1265     int index = 0;
1266     for (String key : set) {
1267         sbKey.append(key);
1268         // sbValue.append(" :").append(key);
1269         sbValue.append(" ?");
1270         if(index != set.size() - 1){
1271             sbKey.append(",");
1272             sbValue.append(",");
1273         }
1274         index++;
1275     }
1276     sbKey.append(")");
1277     sbValue.append(")");
1278
1279     sb.append(sbKey).append("VALUES").append(sbValue);
1280
1281     return sb.toString();
```



```

1282     }
1283
1284     /**
1285      * 生成对象INSERT语句，简化sql拼接
1286      * @param tableName
1287      * @param params
1288      * @return
1289      */
1290     private String makeSimpleInsertSql(String tableName, Map<String, Object> params, Li
1291         if(StringUtils.isEmpty(tableName) || params == null || params.isEmpty()){
1292             return "";
1293         }
1294         StringBuffer sb = new StringBuffer();
1295         sb.append("insert into ").append(tableName);
1296
1297         StringBuffer sbKey = new StringBuffer();
1298         StringBuffer sbValue = new StringBuffer();
1299
1300         sbKey.append("(");
1301         sbValue.append("(");
1302         //添加参数
1303         Set<String> set = params.keySet();
1304         int index = 0;
1305         for (String key : set) {
1306             sbKey.append(key);
1307             sbValue.append(" ?");
1308             if(index != set.size() - 1){
1309                 sbKey.append(",");
1310                 sbValue.append(",");
1311             }
1312             index++;
1313             values.add(params.get(key));
1314         }
1315         sbKey.append(")");
1316         sbValue.append(")");
1317
1318         sb.append(sbKey).append("VALUES").append(sbValue);
1319
1320         return sb.toString();
1321     }
1322
1323
1324     private Serializable doInsertReturnKey(Map<String, Object> params){

```

```

1325     final List<Object> values = new ArrayList<Object>();
1326     final String sql = makeSimpleInsertSql(getTableName(),params,values);
1327     KeyHolder keyHolder = new GeneratedKeyHolder();
1328     final JdbcTemplate jdbcTemplate = new JdbcTemplate(getDataSourceWrite());
1329     try {
1330
1331         jdbcTemplate.update(new PreparedStatementCreator() {
1332             public PreparedStatement createPreparedStatement(
1333
1334                 Connection con) throws SQLException {
1335                 PreparedStatement ps = con.prepareStatement(sql,Statement.RETURN_G
1336
1337                 for (int i = 0; i < values.size(); i++) {
1338                     ps.setObject(i+1, values.get(i)==null?null:values.get(i));
1339
1340                 }
1341                 return ps;
1342             }
1343
1344             }, keyHolder);
1345     } catch (DataAccessException e) {
1346         log.error("error",e);
1347     }
1348
1349
1350
1351     if (keyHolder == null) { return ""; }
1352
1353
1354     Map<String, Object> keys = keyHolder.getKeys();
1355     if (keys == null || keys.size() == 0 || keys.values().size() == 0) {
1356         return "";
1357     }
1358     Object key = keys.values().toArray()[0];
1359     if (key == null || !(key instanceof Serializable)) {
1360         return "";
1361     }
1362     if (key instanceof Number) {
1363         //Long k = (Long) key;
1364         Class clazz = key.getClass();
1365         // return clazz.cast(key);
1366         return (clazz == int.class || clazz == Integer.class) ? ((Number) key).int
1367

```

```

1368
1369     } else if (key instanceof String) {
1370         return (String) key;
1371     } else {
1372         return (Serializable) key;
1373     }
1374
1375
1376 }
1377
1378
1379 /**
1380  * 生成默认的对象UPDATE语句，简化sql拼接
1381  * @param pkValue
1382  * @param params
1383  * @return
1384  */
1385 private String makeDefaultSimpleUpdateSql(Object pkValue, Map<String, Object> para
1386     return this.makeSimpleUpdateSql(getTableName(), getPKColumn(), pkValue, params
1387 }
1388
1389 /**
1390  * 生成默认的对象INSERT语句，简化sql拼接
1391  * @param params
1392  * @return
1393  */
1394 private String makeDefaultSimpleInsertSql(Map<String, Object> params){
1395     return this.makeSimpleInsertSql(this.getTableName(), params);
1396 }
1397
1398 /**
1399  * 获取一个实例对象
1400  * @param tableName
1401  * @param pkName
1402  * @param pkValue
1403  * @param rm
1404  * @return
1405  */
1406 private Object doLoad(String tableName, String pkName, Object pkValue, RowMapper r
1407     StringBuffer sb = new StringBuffer();
1408     sb.append("select * from ").append(tableName).append(" where ").append(pkName)
1409     List<Object> list = this.jdbcTemplateReadOnly().query(sb.toString(), rm, pkVal
1410     if(list == null || list.isEmpty()){

```

```

1411         return null;
1412     }
1413     return list.get(0);
1414 }
1415
1416 /**
1417  * 获取默认的实例对象
1418  * @param <T>
1419  * @param pkValue
1420  * @param rowMapper
1421  * @return
1422  */
1423 private <T> T doLoad(Object pkValue, RowMapper<T> rowMapper){
1424     Object obj = this.doLoad(getTableName(), getPKColumn(), pkValue, rowMapper);
1425     if(obj != null){
1426         return (T)obj;
1427     }
1428     return null;
1429 }
1430
1431
1432 /**
1433  * 删除实例对象，返回删除记录数
1434  * @param tableName
1435  * @param pkName
1436  * @param pkValue
1437  * @return
1438  */
1439 private int doDelete(String tableName, String pkName, Object pkValue) {
1440     StringBuffer sb = new StringBuffer();
1441     sb.append("delete from ").append(tableName).append(" where ").append(pkName).append(" = ");
1442     int ret = this.jdbcTemplateWrite().update(sb.toString(), pkValue);
1443     return ret;
1444 }
1445
1446 /**
1447  * 删除默认实例对象，返回删除记录数
1448  * @param pkValue
1449  * @return
1450  */
1451 private int doDelete(Object pkValue){
1452     return this.doDelete(getTableName(), getPKColumn(), pkValue);
1453 }

```

```
1454
1455  /**
1456   * 更新实例对象，返回删除记录数
1457   * @param tableName
1458   * @param pkName
1459   * @param pkValue
1460   * @param params
1461   * @return
1462   */
1463 private int doUpdate(String tableName, String pkName, Object pkValue, Map<String,
1464     params.put(pkName, pkValue);
1465     String sql = this.makeSimpleUpdateSql(tableName, pkName, pkValue, params);
1466     int ret = this.jdbcTemplateWrite().update(sql, params.values().toArray());
1467     return ret;
1468 }
1469
1470 /**
1471   * 更新实例对象，返回删除记录数
1472   * @param pkName
1473   * @param pkValue
1474   * @param params
1475   * @return
1476   */
1477 private int doUpdate( String pkName, Object pkValue, Map<String, Object> params){
1478     params.put(pkName, pkValue);
1479     String sql = this.makeSimpleUpdateSql( pkName, pkValue, params);
1480     int ret = this.jdbcTemplateWrite().update(sql, params.values().toArray());
1481     return ret;
1482 }
1483
1484 /**
1485   * 更新实例对象，返回删除记录数
1486   * @param pkValue
1487   * @param params
1488   * @return
1489   */
1490 private int doUpdate(Object pkValue, Map<String, Object> params){
1491     //
1492     String sql = this.makeDefaultSimpleUpdateSql(pkValue, params);
1493     params.put(this.getPKColumn(), pkValue);
1494     int ret = this.jdbcTemplateWrite().update(sql, params.values().toArray());
1495     return ret;
1496 }
```

```
1497
1498
1499     private boolean doReplace(Map<String, Object> params) {
1500         String sql = this.makeSimpleReplaceSql(this.getTableName(), params);
1501         int ret = this.jdbcTemplateWrite().update(sql, params.values().toArray());
1502         return ret > 0;
1503     }
1504
1505     private boolean doReplace(String tableName, Map<String, Object> params){
1506         String sql = this.makeSimpleReplaceSql(tableName, params);
1507         int ret = this.jdbcTemplateWrite().update(sql, params.values().toArray());
1508         return ret > 0;
1509     }
1510
1511
1512     /**
1513      * 插入
1514      * @param tableName
1515      * @param params
1516      * @return
1517      */
1518     private boolean doInsert(String tableName, Map<String, Object> params){
1519         String sql = this.makeSimpleInsertSql(tableName, params);
1520         int ret = this.jdbcTemplateWrite().update(sql, params.values().toArray());
1521         return ret > 0;
1522     }
1523
1524     /**
1525      * 插入
1526      * @param params
1527      * @return
1528      */
1529     private boolean doInsert(Map<String, Object> params) {
1530         String sql = this.makeSimpleInsertSql(this.getTableName(), params);
1531         int ret = this.jdbcTemplateWrite().update(sql, params.values().toArray());
1532         return ret > 0;
1533     }
1534
1535     /**
1536      * 获取主键列名称 建议子类重写
1537      * @return
1538      */
1539     protected abstract String getPKColumn();
```

```

1540
1541     protected abstract void setDataSource(DataSource dataSource);
1542
1543     private Map<String,Object> convertMap(Object obj){
1544         Map<String,Object> map = new HashMap<String,Object>();
1545
1546         List<FieldInfo> getters = TypeUtils.computeGetters(obj.getClass(), null);
1547         for(int i=0,len=getters.size();i<len;i++){
1548             FieldInfo fieldInfo = getters.get(i);
1549             String name = fieldInfo.getName();
1550             try {
1551                 Object value = fieldInfo.get(obj);
1552                 map.put(name,value);
1553             } catch (Exception e) {
1554                 log.error(String.format("convertMap error object:%s field: %s",obj.tc
1555             }
1556         }
1557
1558         return map;
1559     }
1560
1561 }

```

3. 动态数据源切换的底层原理

DynamicDataSourceEntry

```

1 package javax.core.common.jdbc.datasource;
2
3
4 import org.aspectj.lang.JoinPoint;
5
6 /**
7  * 动态切换数据源
8  *
9  */
10 public class DynamicDataSourceEntry {
11
12     // 默认数据源
13     public final static String DEFAULT_SOURCE = null;
14

```

```
15     private final static ThreadLocal<String> local = new ThreadLocal<String>();
16
17     /**
18      * 清空数据源
19      */
20     public void clear() {
21         local.remove();
22     }
23
24     /**
25      * 获取当前正在使用的数据源名字
26      *
27      * @return String
28      */
29     public String get() {
30         return local.get();
31     }
32
33     /**
34      * 还原指定切面的数据源
35      *
36      * @param join
37      */
38     public void restore(JoinPoint join) {
39         local.set(DEFAULT_SOURCE);
40     }
41
42     /**
43      * 还原当前切面的数据源
44      */
45     public void restore() {
46         local.set(DEFAULT_SOURCE);
47     }
48
49     /**
50      * 设置已知名字的数据源
51      *
52      * @param source
53      */
54     public void set(String source) {
55         local.set(source);
56     }
57
```



```

58     /**
59      * 根据年份动态设置数据源
60      * @param year
61      */
62     public void set(int year) {
63         local.set("DB_" + year);
64     }
65 }

```

DynamicDataSource

```

1 package javax.core.common.jdbc.datasource;
2
3 import org.springframework.jdbc.datasource.lookup.AbstractRoutingDataSource;
4
5 /**
6  * 动态数据源
7  *
8  *
9  */
10 public class DynamicDataSource extends AbstractRoutingDataSource {
11
12
13     //entry的目的，主要是用来给每个数据源打个标记
14     private DynamicDataSourceEntry dataSourceEntry;
15
16     @Override
17     protected Object determineCurrentLookupKey() {
18         return this.dataSourceEntry.get();
19     }
20
21     public void setDataSourceEntry(DynamicDataSourceEntry dataSourceEntry) {
22         this.dataSourceEntry = dataSourceEntry;
23     }
24
25     public DynamicDataSourceEntry getDataSourceEntry(){
26         return this.dataSourceEntry;
27     }
28
29 }

```

4. 运行效果演示

创建 Member实体类

```
1 package cn.sitedev.orm.demo.entity;
2
3 import lombok.Data;
4
5 import javax.persistence.Entity;
6 import javax.persistence.Id;
7 import javax.persistence.Table;
8 import java.io.Serializable;
9
10 @Entity
11 @Table(name="t_member")
12 @Data
13 public class Member implements Serializable {
14     @Id private Long id;
15     private String name;
16     private String addr;
17     private Integer age;
18
19     @Override
20     public String toString() {
21         return "Member{" +
22             "id=" + id +
23             ", name='" + name + '\'' +
24             ", addr='" + addr + '\'' +
25             ", age=" + age +
26             '}';
27     }
28 }
```

创建 Order实体类

```
1 package cn.sitedev.orm.demo.entity;
2
3 import lombok.Data;
4
5 import javax.persistence.Column;
```

```

6  import javax.persistence.Entity;
7  import javax.persistence.Table;
8  import java.io.Serializable;
9
10 @Entity
11 @Table(name="t_order")
12 @Data
13 public class Order implements Serializable {
14     private Long id;
15     @Column(name="mid")
16     private Long memberId;
17     private String detail;
18     private Long createTime;
19     private String createTimeFmt;
20
21     @Override
22     public String toString() {
23         return "Order{" +
24             "id=" + id +
25             ", memberId=" + memberId +
26             ", detail='" + detail + '\'' +
27             ", createTime=" + createTime +
28             ", createTimeFmt='" + createTimeFmt + '\'' +
29             '}';
30     }
31 }

```

创建 MemberDao

```

1  package cn.sitedev.orm.demo.dao;
2
3  import cn.sitedev.orm.demo.entity.Member;
4  import cn.sitedev.orm.framework.BaseDaoSupport;
5  import cn.sitedev.orm.framework.QueryRule;
6  import org.springframework.stereotype.Repository;
7
8  import javax.annotation.Resource;
9  import javax.core.common.Page;
10 import javax.sql.DataSource;
11 import java.util.List;
12 import java.util.Map;

```

```
13
14 @Repository
15 public class MemberDao extends BaseDaoSupport<Member,Long> {
16
17     @Override
18     protected String getPKColumn() {
19         return "id";
20     }
21
22     @Resource(name="dataSource")
23     public void setDataSource(DataSource dataSource){
24         super.setDataSourceReadOnly(dataSource);
25         super.setDataSourceWrite(dataSource);
26     }
27
28
29     public List<Member> selectAll() throws Exception{
30         QueryRule queryRule = QueryRule.getInstance();
31         queryRule.andLike("name", "Mic%");
32         return super.select(queryRule);
33     }
34
35
36     public Page<Member> selectForPage(int pageNo,int pageSize) throws Exception{
37         QueryRule queryRule = QueryRule.getInstance();
38         queryRule.andLike("name", "Tom%");
39         Page<Member> page = super.select(queryRule,pageNo,pageSize);
40         return page;
41     }
42
43     public void select() throws Exception{
44         String sql = "";
45         List<Map<String,Object>> result = super.selectBySql(sql);
46         //      System.out.println(JSON.parseObject(JSON.toJSONString(result)),Member.class);
47     }
48
49     public boolean insert(Member entity) throws Exception{
50         super.setTableName("t_mmmmm");
51         return super.insert(entity);
52     }
53 }
```

创建 OrderDao

```
1 package cn.sitedev.orm.demo.dao;
2
3 import cn.sitedev.orm.demo.entity.Order;
4 import cn.sitedev.orm.framework.BaseDaoSupport;
5 import org.springframework.stereotype.Repository;
6
7 import javax.annotation.Resource;
8 import javax.core.common.jdbc.datasource.DynamicDataSource;
9 import javax.sql.DataSource;
10 import java.text.SimpleDateFormat;
11 import java.util.Date;
12
13
14 @Repository
15 public class OrderDao extends BaseDaoSupport<Order, Long> {
16
17     private SimpleDateFormat yearFormat = new SimpleDateFormat("yyyy");
18     private SimpleDateFormat fullDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
19     private DynamicDataSource dataSource;
20
21     @Override
22     protected String getPKColumn() {return "id";}
23
24     @Resource(name="dynamicDataSource")
25     public void setDataSource(DataSource dataSource) {
26         this.dataSource = (DynamicDataSource)dataSource;
27         this.setDataSourceReadOnly(dataSource);
28         this.setDataSourceWrite(dataSource);
29     }
30
31     /**
32      * @throws Exception
33      */
34     public boolean insertOne(Order order) throws Exception{
35         //约定优于配置
36         Date date = null;
37         if(order.getCreateTime() == null){
38             date = new Date();
39             order.setCreateTime(date.getTime());
40         }else {
```

```

41         date = new Date(order.getCreateTime());
42     }
43     Integer dbRouter = Integer.valueOf(yearFormat.format(date));
44     System.out.println("自动分配到【DB_】 + dbRouter + "】数据源");
45     this.dataSource.getDataSourceEntry().set(dbRouter);
46
47     order.setCreateTimeFmt(fullDateFormat.format(date));
48
49     Long orderId = super.insertAndReturnId(order);
50     order.setId(orderId);
51     return orderId > 0;
52 }
53
54
55 }

```

修改db.properties文件

```

1  #sysbase database mysql config
2
3  #mysql.jdbc.driverClassName=com.mysql.jdbc.Driver
4  #mysql.jdbc.url=jdbc:mysql://127.0.0.1:3306/gp-vip-spring-db-demo?characterEncoding=UTF-
5  #mysql.jdbc.username=root
6  #mysql.jdbc.password=123456
7
8  db2019.mysql.jdbc.driverClassName=com.mysql.jdbc.Driver
9  db2019.mysql.jdbc.url=jdbc:mysql://127.0.0.1:3306/spring-db-2019?characterEncoding=UTF-
10 db2019.mysql.jdbc.username=root
11 db2019.mysql.jdbc.password=root
12
13 db2020.mysql.jdbc.driverClassName=com.mysql.jdbc.Driver
14 db2020.mysql.jdbc.url=jdbc:mysql://127.0.0.1:3306/spring-db-2020?characterEncoding=UTF-
15 db2020.mysql.jdbc.username=root
16 db2020.mysql.jdbc.password=root
17
18 #alibaba druid config
19
20 dbPool.initialSize=1
21 dbPool.minIdle=1
22 dbPool.maxActive=200
23 dbPool.maxWait=60000

```

```

24 dbPool.timeBetweenEvictionRunsMillis=60000
25 dbPool.minEvictableIdleTimeMillis=300000
26 dbPool.validationQuery=SELECT 'x'
27 dbPool.testWhileIdle=true
28 dbPool.testOnBorrow=false
29 dbPool.testOnReturn=false
30 dbPool.poolPreparedStatements=false
31 dbPool.maxPoolPreparedStatementPerConnectionSize=20
32 dbPool.filters=stat,log4j,wall

```

修改application-db.xml文件

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xmlns:tx="http://www.springframework.org/schema/tx"
5     xmlns:aop="http://www.springframework.org/schema/aop"
6     xmlns:context="http://www.springframework.org/schema/context"
7     xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.sprin
8     http://www.springframework.org/schema/tx http://www.springframework.org/schema/
9     http://www.springframework.org/schema/context http://www.springframework.org/sc
10     http://www.springframework.org/schema/aop http://www.springframework.org/sch
11
12     <bean id="datasourcePool" abstract="true" class="com.alibaba.druid.pool.DruidDataSo
13         <property name="initialSize" value="${dbPool.initialSize}" />
14         <property name="minIdle" value="${dbPool.minIdle}" />
15         <property name="maxActive" value="${dbPool.maxActive}" />
16         <property name="maxWait" value="${dbPool.maxWait}" />
17         <property name="timeBetweenEvictionRunsMillis" value="${dbPool.timeBetweenEvict
18         <property name="minEvictableIdleTimeMillis" value="${dbPool.minEvictableIdleTim
19         <property name="validationQuery" value="${dbPool.validationQuery}" />
20         <property name="testWhileIdle" value="${dbPool.testWhileIdle}" />
21         <property name="testOnBorrow" value="${dbPool.testOnBorrow}" />
22         <property name="testOnReturn" value="${dbPool.testOnReturn}" />
23         <property name="poolPreparedStatements" value="${dbPool.poolPreparedStatements}
24         <property name="maxPoolPreparedStatementPerConnectionSize" value="${dbPool.maxP
25         <property name="filters" value="${dbPool.filters}" />
26     </bean>
27
28     <bean id="dataSource2019" parent="datasourcePool">
29         <property name="driverClassName" value="${db2019.mysql.jdbc.driverClassName}" /

```

```

30     <property name="url" value="${db2019.mysql.jdbc.url}" />
31     <property name="username" value="${db2019.mysql.jdbc.username}" />
32     <property name="password" value="${db2019.mysql.jdbc.password}" />
33 </bean>
34
35 <bean id="dataSource" parent="datasourcePool">
36     <property name="driverClassName" value="${db2020.mysql.jdbc.driverClassName}" /
37     <property name="url" value="${db2020.mysql.jdbc.url}" />
38     <property name="username" value="${db2020.mysql.jdbc.username}" />
39     <property name="password" value="${db2020.mysql.jdbc.password}" />
40 </bean>
41
42
43 <bean id="dynamicDataSourceEntry" class="javax.core.common.jdbc.datasource.Dynamic
44
45 <bean id="dynamicDataSource" class="javax.core.common.jdbc.datasource.DynamicDataSo
46     <property name="dataSourceEntry" ref="dynamicDataSourceEntry"></property>
47     <property name="targetDataSources">
48         <map>
49             <entry key="DB_2020" value-ref="dataSource"></entry>
50             <entry key="DB_2019" value-ref="dataSource2019"></entry>
51         </map>
52     </property>
53     <property name="defaultTargetDataSource" ref="dataSource" />
54 </bean>
55
56 </beans>

```

编写测试用例

...

```

1 package cn.sitedev.orm.test;
2
3 import cn.sitedev.orm.demo.dao.MemberDao;
4 import cn.sitedev.orm.demo.dao.OrderDao;
5 import cn.sitedev.orm.demo.entity.Member;
6 import cn.sitedev.orm.demo.entity.Order;
7 import com.alibaba.fastjson.JSON;
8 import org.junit.Test;
9 import org.junit.runner.RunWith;
10 import org.springframework.beans.factory.annotation.Autowired;
11 import org.springframework.test.context.ContextConfiguration;

```



```

12 import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
13
14 import javax.core.common.Page;
15 import java.text.SimpleDateFormat;
16 import java.util.Date;
17 import java.util.List;
18
19 @ContextConfiguration(locations = {"classpath:application-context.xml"})
20 @RunWith(SpringJUnit4ClassRunner.class)
21 public class OrmTest {
22
23     private SimpleDateFormat sdf = new SimpleDateFormat("yyyyMMddHHmmdd");
24
25     @Autowired private MemberDao memberDao;
26
27     @Autowired private OrderDao orderDao;
28
29     //ORM（对象关系映射 Object Relation Mapping）
30     //Hibernate/Spring JDBC/MyBatis/JPA 一对多、多对多、一对一
31
32     //Hibernate 全自动档 不需要写一句SQL语句
33     //MyBatis 半自动（手自一体） 支持简单的映射，复杂关系，需要自己写SQL
34     //Spring JDBC 全手动挡，所有的SQL都要自己写，它帮我们设计了一套标准 模板模式
35
36     //为什么有了MyBatis我还要自己的手写ORM框架呢？
37     //1、用MyBatis，我可控性无法保证
38     //2、我不敢用Hibernate，高级玩家玩的，
39     //3、没有时间自己从0到1写一个ORM框架
40     //4、站在巨人的肩膀上再升级，做二次开发
41
42     //约定优于配置
43     //1、先制定顶层接口,参数返回值全部统一
44     // List<?> Page<?> select(QueryRule queryRule)
45     // Int delete(T entity) entity中的ID不能为空，如果ID为空，其他条件不能为空，都为空
46     // ReturnId insert(T entity) 只要entity不等于null
47     // Int update(T entity) entity中的ID不能为空，如果ID为空，其他条件不能为空，都为空
48
49     //基于JDBC封装了一套
50     //基于Redis封装了一套
51     //基于MongoDB
52     //基于ElasticSearch
53     //基于Hive
54     //基于HBase

```

```
55
56 //QueryRule
57
58 @Test
59 public void testSelectForPage(){
60     try {
61         Page page = memberDao.selectForPage(2, 3);
62         System.out.println("总条数: " + page.getTotal());
63         System.out.println("当前第几页: " + page.getPageNo());
64         System.out.println("每页多少条: " + page.getPageSize());
65         System.out.println("本页的数据: " + JSON.toJSONString(page.getRows(),true))
66     }catch (Exception e){
67         e.printStackTrace();
68     }
69 }
70
71
72
73 @Test
74 public void testSelectAllForMember(){
75     try {
76         List<Member> result = memberDao.selectAll();
77         System.out.println(JSON.toJSONString(result,true));
78 //         System.out.println(Arrays.toString(result.toArray()));
79     } catch (Exception e) {
80         e.printStackTrace();
81     }
82 }
83
84 @Test
85 // @Ignore
86 public void testInsertMember(){
87     try {
88         for (int age = 25; age < 35; age++) {
89             Member member = new Member();
90             member.setAge(age);
91             member.setName("Tom");
92             member.setAddr("Hunan Changsha");
93             memberDao.insert(member);
94         }
95     }catch (Exception e){
96         e.printStackTrace();
97     }
98 }
```

```
98
99     }
100
101
102     @Test
103     //    @Ignore
104     public void testInsertOrder(){
105         try {
106             Order order = new Order();
107             order.setMemberId(1L);
108             order.setDetail("历史订单");
109             Date date = sdf.parse("20190426123456");
110             order.setCreateTime(date.getTime());
111             orderDao.insertOne(order);
112         }catch (Exception e){
113             e.printStackTrace();
114         }
115     }
116
117 }
```