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：

```java
    private static List<Member> select(String sql) {
        List<Member> result = new ArrayList<>();
        Connection con = null;          //连接对象
        PreparedStatement pstm = null;  //语句集
        ResultSet rs = null;            //结果集
        try {
            //1、加载驱动类，千万不要忘记了
            Class.forName("com.mysql.jdbc.Driver");
            //2、建立连接
            con = DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/spring-db-de
            //3、创建语句集
            pstm = con.prepareStatement(sql);
            //4、执行语句集
            rs = pstm.executeQuery();
            while (rs.next()) {
                //纯粹的硬编码
                Member instance = new Member();
                instance.setId(rs.getLong("id"));
                instance.setName(rs.getString("name"));
                instance.setAge(rs.getInt("age"));
                instance.setAddr(rs.getString("addr"));
                result.add(instance);
            }
            //5、获取结果集
        } catch (Exception e) {
            e.printStackTrace();
        }
        //6、关闭结果集、关闭语句集、关闭连接
        finally {
            try {
                rs.close();
                pstm.close();
                con.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        return result;
    }
```

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：

```java
    private static List<Member> select(String sql) {
        List<Member> result = new ArrayList<>();
        Connection con = null;
        PreparedStatement pstm = null;
        ResultSet rs = null;
        try {
            //1、加载驱动类
            Class.forName("com.mysql.jdbc.Driver");
            //2、建立连接
            con = DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/spring-db-de
            //3、创建语句集
            pstm = con.prepareStatement(sql);
            //4、执行语句集
            rs = pstm.executeQuery();
            while (rs.next()) {
                Member instance = mapperRow(rs, rs.getRow());
                result.add(instance);
            }
            //5、获取结果集
        } catch (Exception e) {
            e.printStackTrace();
        }
        //6、关闭结果集、关闭语句集、关闭连接
        finally {
            try {
                rs.close();
                pstm.close();
                con.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        return result;
    }

    private static Member mapperRow(ResultSet rs, int i) throws Exception {
        Member instance = new Member();
        instance.setId(rs.getLong("id"));
        instance.setName(rs.getString("name"));
        instance.setAge(rs.getInt("age"));
        instance.setAddr(rs.getString("addr"));
        return instance;
    }
```

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：

```java
package cn.sitedev.orm.demo.entity;

import lombok.Data;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import java.io.Serializable;

@Entity
@Table(name="t_member")
@Data
public class Member implements Serializable {
    @Id private Long id;
    private String name;
    private String addr;
    private Integer age;

    @Override
    public String toString() {
        return "Member{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", addr='" + addr + '\'' +
                ", age=" + age +
                '}';
    }
}
```

对JDBC操作优化：

```java
    public static void main(String[] args) {
        //ORM，完成了一部分，只完成了从 数据表到对象的映射
        //对象到数据库表还没有
        //我传的条件是一条SQL语句，我还是在面向SQL编程
//        List<Member> result = select("select * from t_member");

        //这就是OO编程，ORM
        Member condition = new Member();
        condition.setName("TomCat");
        condition.setAge(2);

        //"select * from t_member where name = 'Tom' and age = 19"
        List<?> result = select(condition);
        System.out.println(Arrays.toString(result.toArray()));
    }

    public static List<?> select(Object condition) {
        List<Object> result = new ArrayList<>();

        Class<?> entityClass = condition.getClass();


        Connection con = null;          //连接对象
        PreparedStatement pstm = null;  //语句集
        ResultSet rs = null;            //结果集


        try {
            //1、加载驱动类，千万不要忘记了
            Class.forName("com.mysql.jdbc.Driver");
            //2、建立连接
            con = DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/spring-db-de

            Map<String, String> getFieldNameByColumn = new HashMap<String, String>();
            Map<String, String> getColumnByFieldName = new HashMap<String, String>();
            Field[] fields = entityClass.getDeclaredFields();
            for (Field field : fields) {
                field.setAccessible(true);
                String fieldName = field.getName();
                if (field.isAnnotationPresent(Column.class)) {
                    Column column = field.getAnnotation(Column.class);
                    String columnName = column.name();
                    getFieldNameByColumn.put(columnName, fieldName);
                    getColumnByFieldName.put(fieldName, columnName);
                } else {
                    //默认属性名就是列名
                    getFieldNameByColumn.put(fieldName, fieldName);
                    getColumnByFieldName.put(fieldName, fieldName);
                }
            }


            StringBuffer sql = new StringBuffer();
            //3、创建语句集
            Table table = entityClass.getAnnotation(Table.class);
            sql.append("select * from " + table.name() + " where 1=1 ");
            for (Field field : fields) {

                Object value = field.get(condition);
                if (null != value) {
                    if (String.class == field.getType()) {
                        sql.append(" and " + getColumnByFieldName.get(field.getName())
                    } else {
                        sql.append(" and " + getColumnByFieldName.get(field.getName())
                    }
                    //其他依次类推
                }
            }

            pstm = con.prepareStatement(sql.toString());

            //4、执行，获取结果集
            rs = pstm.executeQuery();

            int columnCounts = rs.getMetaData().getColumnCount();
            while (rs.next()) {
                Object instance = entityClass.newInstance();
                for (int i = 1; i <= columnCounts; i++) {
                    String columnName = rs.getMetaData().getColumnName(i);
                    Field field = entityClass.getDeclaredField(getFieldNameByColumn.get
                    field.setAccessible(true);
                    field.set(instance, rs.getObject(columnName));
                }
                result.add(instance);
            }


        } catch (Exception e) {
            e.printStackTrace();
        }
        //6、关闭结果集、关闭语句集、关闭连接
        finally {
            try {
                rs.close();
                pstm.close();
                con.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        return result;
    }
```

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：

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

```java

    /**
     * 获取分页结果
     * @param queryRule 查询条件
     * @param pageNo 页码
     * @param pageSize 每页条数
     * @return
     */
    Page<?> select(QueryRule queryRule,int pageNo,int pageSize) throws Exception;

    /**
     * 根据SQL获取列表
     * @param sql SQL语句
     * @param args 参数
     * @return
     */
    List<Map<String,Object>> selectBySql(String sql, Object... args) throws Exception;

    /**
     * 根据SQL获取分页
     * @param sql SQL语句
     * @param pageNo 页码
     * @param pageSize 每页条数
     * @return
     */
    Page<Map<String,Object>> selectBySqlToPage(String sql, Object [] param, int pageNo,
```

2、规定删除方法的接口模型为：

```java
    /**
     * 删除一条记录
     * @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
     * @return
     */
    boolean delete(T entity) throws Exception;

    /**
     * 批量删除
     * @param list
     * @return 返回受影响的行数
     * @throws Exception
     */
    int deleteAll(List<T> list) throws Exception;
```

3、规定插入方法的接口模型为：

```
    /**
     * 插入一条记录并返回插入后的ID
     * @param entity 只要entity不等于null，就执行插入
     * @return
     */
    PK insertAndReturnId(T entity) throws Exception;

    /**
     * 插入一条记录自增ID
     * @param entity
     * @return
     * @throws Exception
     */
    boolean insert(T entity) throws Exception;

    /**
     * 批量插入
     * @param list
     * @return 返回受影响的行数
     * @throws Exception
     */
    int insertAll(List<T> list) throws Exception;
```

4、规定修改方法的接口模型为：

```
    /**
     *  修改一条记录
     * @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
     * @return
     * @throws Exception
     */
    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

```java
package javax.core.common;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

/**
 * 分页对象. 包含当前页数据及分页信息如总记录数.
 * 能够支持JQuery EasyUI直接对接，能够支持和BootStrap Table直接对接
 */
public class Page<T> implements Serializable {

    private static final long serialVersionUID = 1L;

    private static final int DEFAULT_PAGE_SIZE = 20;

    private int pageSize = DEFAULT_PAGE_SIZE; // 每页的记录数

    private long start; // 当前页第一条数据在List中的位置,从0开始

    private List<T> rows; // 当前页中存放的记录,类型一般为List

    private long total; // 总记录数

    /**
     * 构造方法，只构造空页.
     */
    public Page() {
        this(0, 0, DEFAULT_PAGE_SIZE, new ArrayList<T>());
    }

    /**
     * 默认构造方法.
     *
     * @param start
     *            本页数据在数据库中的起始位置
     * @param totalSize
     *            数据库中总记录条数
     * @param pageSize
     *            本页容量
     * @param rows
     *            本页包含的数据
     */
    public Page(long start, long totalSize, int pageSize, List<T> rows) {
        this.pageSize = pageSize;
        this.start = start;
        this.total = totalSize;
        this.rows = rows;
    }

    /**
     * 取总记录数.
     */
    public long getTotal() {
        return this.total;
    }

    public void setTotal(long total) {
        this.total = total;
    }

    /**
     * 取总页数.
     */
    public long getTotalPageCount() {
        if (total % pageSize == 0){
            return total / pageSize;
        }else{
            return total / pageSize + 1;
        }
    }

    /**
     * 取每页数据容量.
     */
    public int getPageSize() {
        return pageSize;
    }

    /**
     * 取当前页中的记录.
     */
    public List<T> getRows() {
        return rows;
    }

    public void setRows(List<T> rows) {
        this.rows = rows;
    }

    /**
     * 取该页当前页码,页码从1开始.
     */
    public long getPageNo() {
        return start / pageSize + 1;
    }

    /**
     * 该页是否有下一页.
     */
    public boolean hasNextPage() {
        return this.getPageNo() < this.getTotalPageCount() - 1;
    }

    /**
     * 该页是否有上一页.
     */
    public boolean hasPreviousPage() {
        return this.getPageNo() > 1;
    }

    /**
     * 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.
     *
     * @see #getStartOfPage(int,int)
     */
    protected static int getStartOfPage(int pageNo) {
        return getStartOfPage(pageNo, DEFAULT_PAGE_SIZE);
    }

    /**
```

```java
     * 获取任一页第一条数据在数据集的位置.
     *
     * @param pageNo
     *            从1开始的页号
     * @param pageSize
     *            每页记录条数
     * @return 该页第一条数据
     */
    public static int getStartOfPage(int pageNo, int pageSize) {
        return (pageNo - 1) * pageSize;
    }

}
```

ResultMsg

```java
package javax.core.common;

import lombok.Data;

import java.io.Serializable;


//最底层设计
@Data
public class ResultMsg<T> implements Serializable {

    private static final long serialVersionUID = 2635002588308355785L;

    private int status; //状态码，系统的返回码
    private String msg;  //状态码的解释
    private T data;  //放任意结果

    public ResultMsg() {}

    public ResultMsg(int status) {
        this.status = status;
    }

    public ResultMsg(int status, String msg) {
        this.status = status;
        this.msg = msg;
    }

    public ResultMsg(int status, T data) {
        this.status = status;
        this.data = data;
    }

    public ResultMsg(int status, String msg, T data) {
        this.status = status;
        this.msg = msg;
        this.data = data;
    }
}
```

BaseDao

```java
package javax.core.common.jdbc;

import cn.sitedev.orm.framework.QueryRule;

import javax.core.common.Page;
import java.util.List;
import java.util.Map;

public interface BaseDao<T,PK> {
    /**
     * 获取列表
     * @param queryRule 查询条件
     * @return
     */
    List<T> select(QueryRule queryRule) throws Exception;

    /**
     * 获取分页结果
     * @param queryRule 查询条件
     * @param pageNo 页码
     * @param pageSize 每页条数
     * @return
     */
    Page<?> select(QueryRule queryRule,int pageNo,int pageSize) throws Exception;

    /**
     * 根据SQL获取列表
     * @param sql SQL语句
     * @param args 参数
     * @return
     */
    List<Map<String,Object>> selectBySql(String sql, Object... args) throws Exception;

    /**
     * 根据SQL获取分页
     * @param sql SQL语句
     * @param pageNo 页码
     * @param pageSize 每页条数
     * @return
     */
    Page<Map<String,Object>> selectBySqlToPage(String sql, Object [] param, int pageNo,






    /**
     * 删除一条记录
     * @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
     * @return
     */
    boolean delete(T entity) throws Exception;

    /**
     * 批量删除
     * @param list
     * @return 返回受影响的行数
     * @throws Exception
     */
    int deleteAll(List<T> list) throws Exception;

    /**
     * 插入一条记录并返回插入后的ID
     * @param entity 只要entity不等于null，就执行插入
     * @return
     */
    PK insertAndReturnId(T entity) throws Exception;

    /**
     * 插入一条记录自增ID
     * @param entity
     * @return
     * @throws Exception
     */
    boolean insert(T entity) throws Exception;

    /**
     * 批量插入
     * @param list
     * @return 返回受影响的行数
     * @throws Exception
     */
    int insertAll(List<T> list) throws Exception;

    /**
     *  修改一条记录
     * @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
     * @return
     * @throws Exception
     */
    boolean update(T entity) throws Exception;
}
```

QueryRule

```java
package cn.sitedev.orm.framework;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

/**
 * 查询规则构造器，实现多条件复杂查询的条件拼接
 *  Role 角色，Rule 尺子，规则
 */
public final class QueryRule implements Serializable {
    private static final long serialVersionUID = 1L;
    public static final int ASC_ORDER = 101;
    public static final int DESC_ORDER = 102;
    public static final int LIKE = 1;
    public static final int IN = 2;
    public static final int NOTIN = 3;
    public static final int BETWEEN = 4;
    public static final int EQ = 5;
    public static final int NOTEQ = 6;
    public static final int GT = 7;
    public static final int GE = 8;
    public static final int LT = 9;
    public static final int LE = 10;
    public static final int ISNULL = 11;
    public static final int ISNOTNULL = 12;
    public static final int ISEMPTY = 13;
    public static final int ISNOTEMPTY = 14;
    public static final int AND = 201;
    public static final int OR = 202;
    private List<Rule> ruleList = new ArrayList<Rule>();
    private List<QueryRule> queryRuleList = new ArrayList<QueryRule>();
    private String propertyName;

    private QueryRule() {}

    private QueryRule(String propertyName) {
        this.propertyName = propertyName;
    }

    public static QueryRule getInstance() {
        return new QueryRule();
    }

    /**
     * 添加升序规则
     * @param propertyName
     * @return
     */
    public QueryRule addAscOrder(String propertyName) {
        this.ruleList.add(new Rule(ASC_ORDER, propertyName));
        return this;
    }

    /**
     * 添加降序规则
     * @param propertyName
     * @return
     */
    public QueryRule addDescOrder(String propertyName) {
        this.ruleList.add(new Rule(DESC_ORDER, propertyName));
        return this;
    }

    public QueryRule andIsNull(String propertyName) {
        this.ruleList.add(new Rule(ISNULL, propertyName).setAndOr(AND));
        return this;
    }

    public QueryRule andIsNotNull(String propertyName) {
        this.ruleList.add(new Rule(ISNOTNULL, propertyName).setAndOr(AND));
        return this;
    }

    public QueryRule andIsEmpty(String propertyName) {
        this.ruleList.add(new Rule(ISEMPTY, propertyName).setAndOr(AND));
        return this;
    }

    public QueryRule andIsNotEmpty(String propertyName) {
        this.ruleList.add(new Rule(ISNOTEMPTY, propertyName).setAndOr(AND));
        return this;
    }

    public QueryRule andLike(String propertyName, Object value) {
        this.ruleList.add(new Rule(LIKE, propertyName, new Object[] { value }).setAndOr
        return this;
    }

    public QueryRule andEqual(String propertyName, Object value) {
        this.ruleList.add(new Rule(EQ, propertyName, new Object[] { value }).setAndOr(
        return this;
    }

    public QueryRule andBetween(String propertyName, Object... values) {
        this.ruleList.add(new Rule(BETWEEN, propertyName, values).setAndOr(AND));
        return this;
    }

    public QueryRule andIn(String propertyName, List<Object> values) {
        this.ruleList.add(new Rule(IN, propertyName, new Object[] { values }).setAndOr(
        return this;
    }

    public QueryRule andIn(String propertyName, Object... values) {
        this.ruleList.add(new Rule(IN, propertyName, values).setAndOr(AND));
        return this;
    }

    public QueryRule andNotIn(String propertyName, List<Object> values) {
        this.ruleList.add(new Rule(NOTIN, propertyName, new Object[] { values }).setAnc
        return this;
    }

    public QueryRule orNotIn(String propertyName, Object... values) {
        this.ruleList.add(new Rule(NOTIN, propertyName, values).setAndOr(OR));
        return this;
    }


    public QueryRule andNotEqual(String propertyName, Object value) {
        this.ruleList.add(new Rule(NOTEQ, propertyName, new Object[] { value }).setAndC
        return this;
    }

    public QueryRule andGreaterThan(String propertyName, Object value) {
        this.ruleList.add(new Rule(GT, propertyName, new Object[] { value }).setAndOr(A
        return this;
    }

    public QueryRule andGreaterEqual(String propertyName, Object value) {
        this.ruleList.add(new Rule(GE, propertyName, new Object[] { value }).setAndOr(A
        return this;
    }

    public QueryRule andLessThan(String propertyName, Object value) {
        this.ruleList.add(new Rule(LT, propertyName, new Object[] { value }).setAndOr(A
        return this;
    }

    public QueryRule andLessEqual(String propertyName, Object value) {
        this.ruleList.add(new Rule(LE, propertyName, new Object[] { value }).setAndOr(A
        return this;
    }


    public QueryRule orIsNull(String propertyName) {
        this.ruleList.add(new Rule(ISNULL, propertyName).setAndOr(OR));
        return this;
    }

    public QueryRule orIsNotNull(String propertyName) {
        this.ruleList.add(new Rule(ISNOTNULL, propertyName).setAndOr(OR));
        return this;
    }

    public QueryRule orIsEmpty(String propertyName) {
        this.ruleList.add(new Rule(ISEMPTY, propertyName).setAndOr(OR));
        return this;
    }

    public QueryRule orIsNotEmpty(String propertyName) {
        this.ruleList.add(new Rule(ISNOTEMPTY, propertyName).setAndOr(OR));
        return this;
    }

    public QueryRule orLike(String propertyName, Object value) {
        this.ruleList.add(new Rule(LIKE, propertyName, new Object[] { value }).setAndOr
        return this;
    }

    public QueryRule orEqual(String propertyName, Object value) {
        this.ruleList.add(new Rule(EQ, propertyName, new Object[] { value }).setAndOr(
        return this;
    }

    public QueryRule orBetween(String propertyName, Object... values) {
        this.ruleList.add(new Rule(BETWEEN, propertyName, values).setAndOr(OR));
        return this;
    }

    public QueryRule orIn(String propertyName, List<Object> values) {
        this.ruleList.add(new Rule(IN, propertyName, new Object[] { values }).setAndOr(
        return this;
    }

    public QueryRule orIn(String propertyName, Object... values) {
        this.ruleList.add(new Rule(IN, propertyName, values).setAndOr(OR));
        return this;
    }

    public QueryRule orNotEqual(String propertyName, Object value) {
        this.ruleList.add(new Rule(NOTEQ, propertyName, new Object[] { value }).setAndO
        return this;
    }

    public QueryRule orGreaterThan(String propertyName, Object value) {
        this.ruleList.add(new Rule(GT, propertyName, new Object[] { value }).setAndOr(O
        return this;
    }

    public QueryRule orGreaterEqual(String propertyName, Object value) {
        this.ruleList.add(new Rule(GE, propertyName, new Object[] { value }).setAndOr(O
        return this;
    }

    public QueryRule orLessThan(String propertyName, Object value) {
        this.ruleList.add(new Rule(LT, propertyName, new Object[] { value }).setAndOr(O
        return this;
    }

    public QueryRule orLessEqual(String propertyName, Object value) {
        this.ruleList.add(new Rule(LE, propertyName, new Object[] { value }).setAndOr(O
        return this;
    }


    public List<Rule> getRuleList() {
        return this.ruleList;
    }

    public List<QueryRule> getQueryRuleList() {
        return this.queryRuleList;
    }

    public String getPropertyName() {
        return this.propertyName;
    }

    protected class Rule implements Serializable {
        private static final long serialVersionUID = 1L;
        private int type;    //规则的类型
        private String property_name;
        private Object[] values;
        private int andOr = AND;

        public Rule(int paramInt, String paramString) {
            this.property_name = paramString;
            this.type = paramInt;
        }

        public Rule(int paramInt, String paramString,
                    Object[] paramArrayOfObject) {
            this.property_name = paramString;
            this.values = paramArrayOfObject;
            this.type = paramInt;
        }

        public Rule setAndOr(int andOr){
            this.andOr = andOr;
            return this;
        }

        public int getAndOr(){
            return this.andOr;
        }

        public Object[] getValues() {
            return this.values;
        }

        public int getType() {
            return this.type;
        }

        public String getPropertyName() {
            return this.property_name;
        }
    }
}
```

Order

```java
package cn.sitedev.orm.framework;


/**
 * sql排序组件
 */
public class Order {
    private boolean ascending; //升序还是降序
    private String propertyName; //哪个字段升序，哪个字段降序

    public String toString() {
        return propertyName + ' ' + (ascending ? "asc" : "desc");
    }

    /**
     * Constructor for Order.
     */
    protected Order(String propertyName, boolean ascending) {
        this.propertyName = propertyName;
        this.ascending = ascending;
    }

    /**
     * Ascending order
     *
     * @param propertyName
     * @return Order
     */
    public static Order asc(String propertyName) {
        return new Order(propertyName, true);
    }

    /**
     * Descending order
     *
     * @param propertyName
     * @return Order
     */
    public static Order desc(String propertyName) {
        return new Order(propertyName, false);
    }

}
```

2.1. 基于Spring JDBC 实现关键功能

ClassMappings

```
package cn.sitedev.orm.framework;

import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import java.math.BigDecimal;
import java.sql.Date;
import java.sql.Timestamp;
import java.util.Arrays;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

public class ClassMappings {

    private ClassMappings(){}

     static final Set<Class<?>> SUPPORTED_SQL_OBJECTS = new HashSet<Class<?>>();

        static {
            //只要这里写了的，默认支持自动类型转换
            Class<?>[] classes = {
                    boolean.class, Boolean.class,
                    short.class, Short.class,
                    int.class, Integer.class,
                    long.class, Long.class,
                    float.class, Float.class,
                    double.class, Double.class,
                    String.class,
                    Date.class,
                    Timestamp.class,
                    BigDecimal.class
```

```java
            };
            SUPPORTED_SQL_OBJECTS.addAll(Arrays.asList(classes));
        }

        static boolean isSupportedSQLObject(Class<?> clazz) {
            return clazz.isEnum() || SUPPORTED_SQL_OBJECTS.contains(clazz);
        }

        public static Map<String, Method> findPublicGetters(Class<?> clazz) {
            Map<String, Method> map = new HashMap<String, Method>();
            Method[] methods = clazz.getMethods();
            for (Method method : methods) {
                if (Modifier.isStatic(method.getModifiers()))
                    continue;
                if (method.getParameterTypes().length != 0)
                    continue;
                if (method.getName().equals("getClass"))
                    continue;
                Class<?> returnType = method.getReturnType();
                if (void.class.equals(returnType))
                    continue;
                if(!isSupportedSQLObject(returnType)){
                    continue;
                }
                if ((returnType.equals(boolean.class)
                        || returnType.equals(Boolean.class))
                        && method.getName().startsWith("is")
                        && method.getName().length() > 2) {
                    map.put(getGetterName(method), method);
                    continue;
                }
                if ( ! method.getName().startsWith("get"))
                    continue;
                if (method.getName().length() < 4)
                    continue;
                map.put(getGetterName(method), method);
            }
            return map;
        }

        public static Field[] findFields(Class<?> clazz){
            return clazz.getDeclaredFields();
        }

        public static Map<String, Method> findPublicSetters(Class<?> clazz) {
            Map<String, Method> map = new HashMap<String, Method>();
            Method[] methods = clazz.getMethods();
            for (Method method : methods) {
                if (Modifier.isStatic(method.getModifiers()))
                    continue;
                if ( ! void.class.equals(method.getReturnType()))
                    continue;
                if (method.getParameterTypes().length != 1)
                    continue;
                if ( ! method.getName().startsWith("set"))
                    continue;
                if (method.getName().length() < 4)
                    continue;
                if(!isSupportedSQLObject(method.getParameterTypes()[0])){
                    continue;
                }
                map.put(getSetterName(method), method);
            }
            return map;
        }

        public static String getGetterName(Method getter) {
            String name = getter.getName();
            if (name.startsWith("is"))
                name = name.substring(2);
            else
                name = name.substring(3);
            return Character.toLowerCase(name.charAt(0)) + name.substring(1);
        }

        private static String getSetterName(Method setter) {
            String name = setter.getName().substring(3);
            return Character.toLowerCase(name.charAt(0)) + name.substring(1);
        }
}
```

EntityOperation

```java
package cn.sitedev.orm.framework;
```

```java

import org.apache.log4j.Logger;
import org.springframework.jdbc.core.RowMapper;

import javax.core.common.utils.StringUtils;
import javax.persistence.*;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.util.HashMap;
import java.util.Map;
import java.util.TreeMap;

/**
 * 实体对象的反射操作
 *
 * @param <T>
 */
public class EntityOperation<T> {
    private Logger log = Logger.getLogger(EntityOperation.class);
    public Class<T> entityClass = null; // 泛型实体Class对象
    public final Map<String, PropertyMapping> mappings;
    public final RowMapper<T> rowMapper;

    public final String tableName;
    public String allColumn = "*";
    public Field pkField;

    public EntityOperation(Class<T> clazz,String pk) throws Exception{
        if(!clazz.isAnnotationPresent(Entity.class)){
            throw new Exception("在" + clazz.getName() + "中没有找到Entity注解，不能做O
        }
        this.entityClass = clazz;
        Table table = entityClass.getAnnotation(Table.class);
        if (table != null) {
                this.tableName = table.name();
        } else {
                this.tableName =  entityClass.getSimpleName();
        }
        Map<String, Method> getters = ClassMappings.findPublicGetters(entityClass);
        Map<String, Method> setters = ClassMappings.findPublicSetters(entityClass);
        Field[] fields = ClassMappings.findFields(entityClass);
        fillPkFieldAndAllColumn(pk,fields);
        this.mappings = getPropertyMappings(getters, setters, fields);
        this.allColumn = this.mappings.keySet().toString().replace("[", "").replace("]"
        this.rowMapper = createRowMapper();
    }

     Map<String, PropertyMapping> getPropertyMappings(Map<String, Method> getters, Map<
        Map<String, PropertyMapping> mappings = new HashMap<String, PropertyMapping>();
        String name;
        for (Field field : fields) {
            if (field.isAnnotationPresent(Transient.class))
                continue;
            name = field.getName();
            if(name.startsWith("is")){
                name = name.substring(2);
            }
            name = Character.toLowerCase(name.charAt(0)) + name.substring(1);
            Method setter = setters.get(name);
            Method getter = getters.get(name);
            if (setter == null || getter == null){
                continue;
            }
            Column column = field.getAnnotation(Column.class);
            if (column == null) {
                mappings.put(field.getName(), new PropertyMapping(getter, setter, fielc
            } else {
                mappings.put(column.name(), new PropertyMapping(getter, setter, field))
            }
        }
        return mappings;
    }

    RowMapper<T> createRowMapper() {
            return new RowMapper<T>() {
                public T mapRow(ResultSet rs, int rowNum) throws SQLException {
                    try {
                        T t = entityClass.newInstance();
                        ResultSetMetaData meta = rs.getMetaData();
                        int columns = meta.getColumnCount();
                        String columnName;
                        for (int i = 1; i <= columns; i++) {
                            Object value = rs.getObject(i);
                            columnName = meta.getColumnName(i);
                            fillBeanFieldValue(t,columnName,value);
                        }
                        return t;
                    }catch (Exception e) {
                        throw new RuntimeException(e);
                    }
                }
            };
        }

    protected void fillBeanFieldValue(T t, String columnName, Object value) {
         if (value != null) {
             PropertyMapping pm = mappings.get(columnName);
             if (pm != null) {
                 try {
                    pm.set(t, value);
                } catch (Exception e) {
                    e.printStackTrace();
                }
             }
         }
    }

    private void fillPkFieldAndAllColumn(String pk, Field[] fields) {
        //设定主键
        try {
            if(!StringUtils.isEmpty(pk)){
                pkField = entityClass.getDeclaredField(pk);
                pkField.setAccessible(true);
            }
        } catch (Exception e) {
                log.debug("没找到主键列,主键列名必须与属性名相同");
        }
          for (int i = 0 ; i < fields.length ;i ++) {
              Field f = fields[i];
              if(StringUtils.isEmpty(pk)){
                  Id id = f.getAnnotation(Id.class);
                  if(id != null){
                      pkField = f;
                      break;
                  }
              }
            }
    }

    public T parse(ResultSet rs) {
        T t = null;
        if (null == rs) {
            return null;
        }
        Object value = null;
        try {
            t = (T) entityClass.newInstance();
            for (String columnName : mappings.keySet()) {
                try {
                    value = rs.getObject(columnName);
                } catch (Exception e) {
                    e.printStackTrace();
                }
                fillBeanFieldValue(t,columnName,value);
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        return t;
    }

    public Map<String, Object> parse(T t) {
        Map<String, Object> _map = new TreeMap<String, Object>();
        try {

            for (String columnName : mappings.keySet()) {
                Object value = mappings.get(columnName).getter.invoke(t);
                if (value == null)
                    continue;
                _map.put(columnName, value);

            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return _map;
    }

    public void println(T t) {
        try {
            for (String columnName : mappings.keySet()) {
                Object value = mappings.get(columnName).getter.invoke(t);
                if (value == null)
                    continue;
                System.out.println(columnName + " = " + value);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

class PropertyMapping {

    final boolean insertable;
    final boolean updatable;
    final String columnName;
    final boolean id;
    final Method getter;
    final Method setter;
    final Class enumClass;
    final String fieldName;

    public PropertyMapping(Method getter, Method setter, Field field) {
        this.getter = getter;
        this.setter = setter;
        this.enumClass = getter.getReturnType().isEnum() ? getter.getReturnType() : nul
        Column column = field.getAnnotation(Column.class);
        this.insertable = column == null || column.insertable();
        this.updatable = column == null || column.updatable();
        this.columnName = column == null ? ClassMappings.getGetterName(getter) : ("".ec
        this.id = field.isAnnotationPresent(Id.class);
        this.fieldName = field.getName();
    }

    @SuppressWarnings("unchecked")
    Object get(Object target) throws Exception {
        Object r = getter.invoke(target);
        return enumClass == null ? r : Enum.valueOf(enumClass, (String) r);
    }

    @SuppressWarnings("unchecked")
    void set(Object target, Object value) throws Exception {
        if (enumClass != null && value != null) {
            value = Enum.valueOf(enumClass, (String) value);
        }
        //BeanUtils.setProperty(target, fieldName, value);
        try {
             if(value != null){
                  setter.invoke(target, setter.getParameterTypes()[0].cast(value));
             }
        } catch (Exception e) {
            e.printStackTrace();
            /**
             * 出错原因如果是boolean字段 mysql字段类型 设置tinyint(1)
             */
            System.err.println(fieldName + "--" + value);
        }

    }
}
```

QueryRuleSqlBulider

```java
package cn.sitedev.orm.framework;

import cn.sitedev.orm.framework.QueryRule.Rule;
import org.apache.commons.lang.ArrayUtils;

import javax.core.common.utils.StringUtils;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.regex.Matcher;
import java.util.regex.Pattern;


/**
 * 根据QueryRule自动构建sql语句
 *
 */
public class QueryRuleSqlBuilder {
    private int CURR_INDEX = 0; //记录参数所在的位置
    private List<String> ;properties; //保存列名列表
    private List<Object> values; //保存参数值列表
    private List<Order> orders; //保存排序规则列表

    private String whereSql = "";
    private String orderSql = "";
    private Object [] valueArr = new Object[]{};
    private Map<Object,Object> valueMap = new HashMap<Object,Object>();

    /**
     * 或得查询条件
     * @return
     */
    public String getWhereSql(){
        return this.whereSql;
    }

    /**
     * 获得排序条件
     * @return
     */
    public String getOrderSql(){
        return this.orderSql;
    }

    /**
     * 获得参数值列表
     * @return
     */
    public Object [] getValues(){
        return this.valueArr;
    }

    /**
     * 获取参数列表
     * @return
     */
    public Map<Object,Object> getValueMap(){
        return this.valueMap;
    }

    /**
```

```java
     * 创建SQL构造器
     * @param queryRule
     */
    public QueryRuleSqlBuilder(QueryRule queryRule) {
        CURR_INDEX = 0;
        properties = new ArrayList<String>();
        values = new ArrayList<Object>();
        orders = new ArrayList<Order>();
        for (Rule rule : queryRule.getRuleList()) {
            switch (rule.getType()) {
            case QueryRule.BETWEEN:
                processBetween(rule);
                break;
            case QueryRule.EQ:
                processEqual(rule);
                break;
            case QueryRule.LIKE:
                processLike(rule);
                break;
            case QueryRule.NOTEQ:
                processNotEqual(rule);
                break;
            case QueryRule.GT:
                processGreaterThen(rule);
                break;
            case QueryRule.GE:
                processGreaterEqual(rule);
                break;
            case QueryRule.LT:
                processLessThen(rule);
                break;
            case QueryRule.LE:
                processLessEqual(rule);
                break;
            case QueryRule.IN:
                processIN(rule);
                break;
            case QueryRule.NOTIN:
                processNotIN(rule);
                break;
            case QueryRule.ISNULL:
                processIsNull(rule);
                break;
            case QueryRule.ISNOTNULL:
                processIsNotNull(rule);
                break;
            case QueryRule.ISEMPTY:
                processIsEmpty(rule);
                break;
            case QueryRule.ISNOTEMPTY:
                processIsNotEmpty(rule);
                break;
            case QueryRule.ASC_ORDER:
                processOrder(rule);
                break;
            case QueryRule.DESC_ORDER:
                processOrder(rule);
                break;
            default:
                throw new IllegalArgumentException("type " + rule.getType() + " not sup
            }
        }
        //拼装where语句
        appendWhereSql();
        //拼装排序语句
        appendOrderSql();
        //拼装参数值
        appendValues();
    }

    /**
     * 去掉order
     *
     * @param sql
     * @return
     */
    protected String removeOrders(String sql) {
        Pattern p = Pattern.compile("order\\\\s*by[\\w|\\W|\\s|\\S]*", Pattern.CASE_INS
        Matcher m = p.matcher(sql);
        StringBuffer sb = new StringBuffer();
        while (m.find()) {
            m.appendReplacement(sb, "");
        }
        m.appendTail(sb);
        return sb.toString();
    }
```

```java

    /**
     * 去掉select
     *
     * @param sql
     * @return
     */
    protected String removeSelect(String sql) {
        if(sql.toLowerCase().matches("from\\s+")){
            int beginPos = sql.toLowerCase().indexOf("from");
            return sql.substring(beginPos);
        }else{
            return sql;
        }
    }

    /**
     * 处理like
     * @param rule
     */
    private  void processLike(Rule rule) {
        if (ArrayUtils.isEmpty(rule.getValues())) {
            return;
        }
        Object obj = rule.getValues()[0];

        if (obj != null) {
            String value = obj.toString();
            if (!StringUtils.isEmpty(value)) {
                value = value.replace('*', '%');
                obj = value;
            }
        }
        add(rule.getAndOr(),rule.getPropertyName(),"like","%"+rule.getValues()[0]+"%");
    }

    /**
     * 处理between
     * @param rule
     */
    private  void processBetween(Rule rule) {
        if ((ArrayUtils.isEmpty(rule.getValues()))
                || (rule.getValues().length < 2)) {
```

```java
            return;
        }
        add(rule.getAndOr(),rule.getPropertyName(),"","between",rule.getValues()[0],"ar
        add(0,"","","",rule.getValues()[1],"");
    }

    /**
     * 处理 =
     * @param rule
     */
    private  void processEqual(Rule rule) {
        if (ArrayUtils.isEmpty(rule.getValues())) {
            return;
        }
        add(rule.getAndOr(),rule.getPropertyName(),"=",rule.getValues()[0]);
    }

    /**
     * 处理 <>
     * @param rule
     */
    private  void processNotEqual(Rule rule) {
        if (ArrayUtils.isEmpty(rule.getValues())) {
            return;
        }
        add(rule.getAndOr(),rule.getPropertyName(),"<>",rule.getValues()[0]);
    }

    /**
     * 处理 >
     * @param rule
     */
    private  void processGreaterThen(
            Rule rule) {
        if (ArrayUtils.isEmpty(rule.getValues())) {
            return;
        }
        add(rule.getAndOr(),rule.getPropertyName(),">",rule.getValues()[0]);
    }

    /**
     * 处理>=
     * @param rule
     */
    private  void processGreaterEqual(
            Rule rule) {
        if (ArrayUtils.isEmpty(rule.getValues())) {
            return;
        }
        add(rule.getAndOr(),rule.getPropertyName(),">=",rule.getValues()[0]);
    }

    /**
     * 处理<
     * @param rule
     */
    private  void processLessThen(Rule rule) {
        if (ArrayUtils.isEmpty(rule.getValues())) {
            return;
        }
        add(rule.getAndOr(),rule.getPropertyName(),"<",rule.getValues()[0]);
    }

    /**
     * 处理<=
     * @param rule
     */
    private  void processLessEqual(
            Rule rule) {
        if (ArrayUtils.isEmpty(rule.getValues())) {
            return;
        }
        add(rule.getAndOr(),rule.getPropertyName(),"<=",rule.getValues()[0]);
    }

    /**
     * 处理  is null
     * @param rule
     */
    private  void processIsNull(Rule rule) {
        add(rule.getAndOr(),rule.getPropertyName(),"is null",null);
    }

    /**
     * 处理 is not null
     * @param rule
     */
    private  void processIsNotNull(Rule rule) {
        add(rule.getAndOr(),rule.getPropertyName(),"is not null",null);
    }

    /**
     * 处理  <>''
     * @param rule
     */
    private  void processIsNotEmpty(Rule rule) {
        add(rule.getAndOr(),rule.getPropertyName(),"<>","''");
    }

    /**
     * 处理 =''
     * @param rule
     */
    private  void processIsEmpty(Rule rule) {
        add(rule.getAndOr(),rule.getPropertyName(),"=","''");
    }


    /**
     * 处理in和not in
     * @param rule
     * @param name
     */
    private void inAndNotIn(Rule rule,String name){
        if (ArrayUtils.isEmpty(rule.getValues())) {
            return;
        }
        if ((rule.getValues().length == 1) && (rule.getValues()[0] != null)
                && (rule.getValues()[0] instanceof List)) {
            List<Object> list = (List) rule.getValues()[0];

            if ((list != null) && (list.size() > 0)){
                for (int i = 0; i < list.size(); i++) {
                    if(i == 0 && i == list.size() - 1){
                        add(rule.getAndOr(),rule.getPropertyName(),"",name + " (",list.
                    }else if(i == 0 && i < list.size() - 1){
                        add(rule.getAndOr(),rule.getPropertyName(),"",name + " (",list.
                    }
                    if(i > 0 && i < list.size() - 1){
                        add(0,"",",","",list.get(i),"");
                    }
                    if(i == list.size() - 1 && i != 0){
                        add(0,"",",","",list.get(i),")");
                    }
                }
            }
        } else {
            Object[] list =  rule.getValues();
            for (int i = 0; i < list.length; i++) {
                if(i == 0 && i == list.length - 1){
                    add(rule.getAndOr(),rule.getPropertyName(),"",name + " (",list[i],"
                }else if(i == 0 && i < list.length - 1){
                    add(rule.getAndOr(),rule.getPropertyName(),"",name + " (",list[i],"
                }
                if(i > 0 && i < list.length - 1){
                    add(0,"",",","",list[i],"");
                }
                if(i == list.length - 1 && i != 0){
                    add(0,"",",","",list[i],")");
                }
            }
        }
    }

    /**
     * 处理 not in
     * @param rule
     */
    private void processNotIN(Rule rule){
        inAndNotIn(rule,"not in");
    }

    /**
     * 处理 in
     * @param rule
     */
    private  void processIN(Rule rule) {
        inAndNotIn(rule,"in");
    }

    /**
     * 处理 order by
```

```java
     * @param rule 查询规则
     */
    private void processOrder(Rule rule) {
        switch (rule.getType()) {
        case QueryRule.ASC_ORDER:
            // propertyName非空
            if (!StringUtils.isEmpty(rule.getPropertyName())) {
                orders.add(Order.asc(rule.getPropertyName()));
            }
            break;
        case QueryRule.DESC_ORDER:
            // propertyName非空
            if (!StringUtils.isEmpty(rule.getPropertyName())) {
                orders.add(Order.desc(rule.getPropertyName()));
            }
            break;
        default:
            break;
        }
    }


    /**
     * 加入到sql查询规则队列
     * @param andOr and 或者 or
     * @param key 列名
     * @param split 列名与值之间的间隔
     * @param value 值
     */
    private  void add(int andOr,String key,String split ,Object value){
        add(andOr,key,split,"",value,"");
    }

    /**
     * 加入到sql查询规则队列
     * @param andOr and 或则 or
     * @param key 列名
     * @param split 列名与值之间的间隔
     * @param prefix 值前缀
     * @param value 值
     * @param suffix 值后缀
     */
    private  void add(int andOr,String key,String split ,String prefix,Object value,Str
        String andOrStr = (0 == andOr ? "" :(QueryRule.AND == andOr ? " and " : " or ")
        properties.add(CURR_INDEX, andOrStr + key + " " + split + prefix + (null != val
        if(null != value){
            values.add(CURR_INDEX,value);
            CURR_INDEX ++;
        }
    }


    /**
     * 拼装 where 语句
     */
    private void appendWhereSql(){
        StringBuffer whereSql = new StringBuffer();
        for (String p : properties) {
            whereSql.append(p);
        }
        this.whereSql = removeSelect(removeOrders(whereSql.toString()));
    }

    /**
     * 拼装排序语句
     */
    private void appendOrderSql(){
        StringBuffer orderSql = new StringBuffer();
        for (int i = 0 ; i < orders.size(); i ++) {
            if(i > 0 && i < orders.size()){
                orderSql.append(",");
            }
            orderSql.append(orders.get(i).toString());
        }
        this.orderSql = removeSelect(removeOrders(orderSql.toString()));
    }

    /**
     * 拼装参数值
     */
    private void appendValues(){
        Object [] val = new Object[values.size()];
        for (int i = 0; i < values.size(); i ++) {
            val[i] = values.get(i);
            valueMap.put(i, values.get(i));
        }
        this.valueArr = val;
    }

}
```

BaseDaoSupport

```java
package cn.sitedev.orm.framework;

import com.alibaba.fastjson.util.FieldInfo;
import com.alibaba.fastjson.util.TypeUtils;
import org.apache.log4j.Logger;
import org.springframework.dao.DataAccessException;
import org.springframework.dao.support.DataAccessUtils;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.PreparedStatementCreator;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.support.GeneratedKeyHolder;
import org.springframework.jdbc.support.KeyHolder;

import javax.core.common.Page;
import javax.core.common.jdbc.BaseDao;
import javax.core.common.utils.BeanUtils;
import javax.core.common.utils.DataUtils;
import javax.core.common.utils.GenericsUtils;
import javax.core.common.utils.StringUtils;
import javax.sql.DataSource;
import java.io.*;
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.sql.*;
import java.util.*;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/**
 * 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
 * 需要重写和实现以下三个方法
 *  //设定主键列
 *    private String getPKColumn() {return "id";}
 *    //重写对象反转为Map的方法
 *    protected Map<String, Object> parse(Object entity) {return utils.parse((Entity)e
 *    //重写结果反转为对象的方法
 *    protected Entity mapRow(ResultSet rs, int rowNum) throws SQLException {return ut
 *
 *
 */
public abstract class BaseDaoSupport<T extends Serializable, PK extends Serializable>
    private Logger log = Logger.getLogger(BaseDaoSupport.class);

    private String tableName = "";

    private JdbcTemplate jdbcTemplateWrite;
    private JdbcTemplate jdbcTemplateReadOnly;

    private DataSource dataSourceReadOnly;
    private DataSource dataSourceWrite;

    private EntityOperation<T> op;

    @SuppressWarnings("unchecked")
    protected BaseDaoSupport(){
        try{
    //        Class<T> entityClass = (Class<T>)((ParameterizedType) getClass().getGene
            Class<T> entityClass = GenericsUtils.getSuperClassGenricType(getClass(), 0
            op = new EntityOperation<T>(entityClass,this.getPKColumn());
            this.setTableName(op.tableName);
        }catch(Exception e){
            e.printStackTrace();
        }
    }

    protected String getTableName() {
        return tableName;
    }

    protected DataSource getDataSourceReadOnly() {
        return dataSourceReadOnly;
    }

    protected DataSource getDataSourceWrite() {
        return dataSourceWrite;
    }

```

```java
    /**
     * 动态切换表名
     */
    protected void setTableName(String tableName) {
        if(StringUtils.isEmpty(tableName)){
            this.tableName = op.tableName;
        }else{
            this.tableName = tableName;
        }
    }

    protected void setDataSourceWrite(DataSource dataSourceWrite) {
        this.dataSourceWrite = dataSourceWrite;
        jdbcTemplateWrite = new JdbcTemplate(dataSourceWrite);
    }

    protected void setDataSourceReadOnly(DataSource dataSourceReadOnly) {
        this.dataSourceReadOnly = dataSourceReadOnly;
        jdbcTemplateReadOnly = new JdbcTemplate(dataSourceReadOnly);
    }

    private JdbcTemplate jdbcTemplateReadOnly() {
        return this.jdbcTemplateReadOnly;
    }

    private JdbcTemplate jdbcTemplateWrite() {
        return this.jdbcTemplateWrite;
    }


    /**
     * 还原默认表名
     */
    protected void restoreTableName(){
        this.setTableName(op.tableName);
    }

    /**
     * 将对象解析为Map
     * @param entity
     * @return
     */
    protected Map<String,Object> parse(T entity){
        return op.parse(entity);
    }



    /**
     * 根据ID获取对象. 如果对象不存在，返回null.<br>
     */
    protected T get(PK id) throws Exception {
        return (T) this.doLoad(id, this.op.rowMapper);
    }

    /**
     * 获取全部对象. <br>
     *
     * @return 全部对象
     */
    protected List<T> getAll() throws Exception {
        String sql = "select " + op.allColumn + " from " + getTableName();
        return this.jdbcTemplateReadOnly().query(sql, this.op.rowMapper, new HashMap<S
    }

    /**
     * 插入并返回id
     * @param entity
     * @return
     */
    public PK insertAndReturnId(T entity) throws Exception{
        return (PK)this.doInsertRuturnKey(parse(entity));
    }

    /**
     * 插入一条记录
     * @param entity
     * @return
     */
    public boolean insert(T entity) throws Exception{
        return this.doInsert(parse(entity));
    }


    /**
     * 保存对象,如果对象存在则更新,否则插入.<br>
     * </code>
     * </pre>
     * @throws IllegalAccessException
     * @throws IllegalArgumentException
     */
    protected boolean save(T entity) throws Exception {
        PK pkValue = (PK)op.pkField.get(entity);
        if(this.exists(pkValue)){
            return this.doUpdate(pkValue, parse(entity)) > 0;
        }else{
            return this.doInsert(parse(entity));
        }
    }

    /**
     * 保存并返回新的id,如果对象存在则更新,否则插入
     * @param entity
     * @return
     * @throws IllegalAccessException
     * @throws IllegalArgumentException
     */
    protected PK saveAndReturnId(T entity) throws Exception{
        Object o = op.pkField.get(entity);
        if(null == o){
            return (PK)this.doInsertRuturnKey(parse(entity));
            //return (PK)id;
        }
        PK pkValue = (PK)o;
        if(this.exists(pkValue)){
            this.doUpdate(pkValue, parse(entity));
            return pkValue;
        }else{
            return (PK)this.doInsertRuturnKey(parse(entity));
        }
    }

    /**
     * 更新对象.<br>
     * 例如: 以下代码将对象更新到数据库
     * <pre>
     *         <code>
     * User entity = service.get(1);
     * entity.setName(&quot;zzz&quot;);
     * // 更新对象
     * service.update(entity);
     * </code>
     * </pre>
     *
     * @param entity 待更新对对象
     * @throws IllegalAccessException
     * @throws IllegalArgumentException
     */
    public boolean update(T entity) throws Exception {
        return this.doUpdate(op.pkField.get(entity), parse(entity)) > 0;
    }

    /**
     * 使用SQL语句更新对象.<br>
     * 例如：以下代码将更新id="0002"的name值更新为“张三”到数据库
     * <pre>
     *        <code>
     * String name = "张三";
     * String id = "0002";
     * String sql = "UPDATE SET name = ? WHERE id = ?";
     * // 更新对象
     * service.update(sql,name,id)
     * </code>
     * </pre>
     *
     * @param sql 更新sql语句
     * @param args 参数对象
     *
     * @return 更新记录数
     */
    protected int update(String sql,Object... args) throws Exception{
        return jdbcTemplateWrite().update(sql, args);
    }

    /**
     * 使用SQL语句更新对象.<br>
     * 例如：以下代码将更新id="0002"的name值更新为“张三”到数据库
     * <pre>
     *        <code>
     * Map<String,Object> map = new HashMap();
     * map.put("name","张三");
     * map.put("id","0002");
     * String sql = "UPDATE SET name = :name WHERE id = :id";
     * // 更新对象
     * service.update(sql,map)
     * </code>
     * </pre>
     *
     * @param sql 更新sql语句
     * @param paramMap 参数对象
     *
     * @return 更新记录数
     */
    protected int update(String sql,Map<String,?> paramMap) throws Exception{
        return jdbcTemplateWrite().update(sql, paramMap);
    }
    /**
     * 批量保存对象.<br>
     * 例如：以下代码将对象保存到数据库
     * <pre>
     *         <code>
     * List&lt;Role&gt; list = new ArrayList&lt;Role&gt;();
     * for (int i = 1; i &lt; 8; i++) {
     *     Role role = new Role();
     *     role.setId(i);
     *     role.setRolename(&quot;管理quot; + i);
     *     role.setPrivilegesFlag(&quot;1,2,3&quot;);
     *     list.add(role);
     * }
     * service.insertAll(list);
     * </code>
     * </pre>
     *
     * @param list 待保存的对象List
     * @throws InvocationTargetException
     * @throws IllegalArgumentException
     * @throws IllegalAccessException
     */
    public int insertAll(List<T> list) throws Exception {
        int count = 0 ,len = list.size(),step = 50000;
        Map<String, PropertyMapping> pm = op.mappings;
        int maxPage = (len % step == 0) ? (len / step) : (len / step + 1);
        for (int i = 1; i <= maxPage; i ++) {
            Page<T> page = pagination(list, i, step);
            String sql = "insert into " + getTableName() + "(" + op.allColumn + ") val
```

```java
            StringBuffer valstr = new StringBuffer();
            Object[] values = new Object[pm.size() * page.getRows().size()];
            for (int j = 0; j < page.getRows().size(); j ++) {
                if(j > 0 && j < page.getRows().size()){ valstr.append(","); }
                valstr.append("(");
                int k = 0;
                for (PropertyMapping p : pm.values()) {
                    values[(j * pm.size()) + k] = p.getter.invoke(page.getRows().get(j
                    if(k > 0 && k < pm.size()){ valstr.append(","); }
                    valstr.append("?");
                    k ++;
                }
                valstr.append(")");
            }
            int result = jdbcTemplateWrite().update(sql + valstr.toString(), values);
            count += result;
        }

        return count;
    }


    protected boolean replaceOne(T entity) throws Exception{
        return this.doReplace(parse(entity));
    }


    protected int replaceAll(List<T> list) throws Exception {
        int count = 0 ,len = list.size(),step = 50000;
        Map<String, PropertyMapping> pm = op.mappings;
        int maxPage = (len % step == 0) ? (len / step) : (len / step + 1);
        for (int i = 1; i <= maxPage; i ++) {
            Page<T> page = pagination(list, i, step);
            String sql = "replace into " + getTableName() + "(" + op.allColumn + ") va
            StringBuffer valstr = new StringBuffer();
            Object[] values = new Object[pm.size() * page.getRows().size()];
            for (int j = 0; j < page.getRows().size(); j ++) {
                if(j > 0 && j < page.getRows().size()){ valstr.append(","); }
                valstr.append("(");
                int k = 0;
                for (PropertyMapping p : pm.values()) {
                    values[(j * pm.size()) + k] = p.getter.invoke(page.getRows().get(j
                    if(k > 0 && k < pm.size()){ valstr.append(","); }
                    valstr.append("?");
                    k ++;
                }
                valstr.append(")");
            }
            int result = jdbcTemplateWrite().update(sql + valstr.toString(), values);
            count += result;
        }
        return count;
    }


    /**
     * 删除对象.<br>
     * 例如：以下删除entity对应的记录
     * <pre>
     *          <code>
     * service.delete(entity);
     * </code>
     * </pre>
     *
     * @param entity 待删除的实体对象
     */
    public boolean delete(T entity) throws Exception {
        return this.doDelete(op.pkField.get(entity)) > 0;
    }

    /**
     * 删除对象.<br>
     * 例如：以下删除entity对应的记录
     * <pre>
     *          <code>
     * service.deleteAll(entityList);
     * </code>
     * </pre>
     *
     * @param list 待删除的实体对象列表
     * @throws InvocationTargetException
     * @throws IllegalArgumentException
     * @throws IllegalAccessException
     */
    public int deleteAll(List<T> list) throws Exception {
        String pkName = op.pkField.getName();
        int count = 0 ,len = list.size(),step = 1000;
        Map<String, PropertyMapping> pm = op.mappings;
        int maxPage = (len % step == 0) ? (len / step) : (len / step + 1);
        for (int i = 1; i <= maxPage; i ++) {
            StringBuffer valstr = new StringBuffer();
            Page<T> page = pagination(list, i, step);
            Object[] values = new Object[page.getRows().size()];

            for (int j = 0; j < page.getRows().size(); j ++) {
                if(j > 0 && j < page.getRows().size()){ valstr.append(","); }
                values[j] = pm.get(pkName).getter.invoke(page.getRows().get(j));
                valstr.append("?");
            }

            String sql = "delete from " + getTableName() + " where " + pkName + " in (
            int result = jdbcTemplateWrite().update(sql, values);
            count += result;
        }
        return count;
    }

    /**
     * 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>
     * 例如：以下删除主键唯一的记录
     * <pre>
     *         <code>
     * service.deleteByPK(1);
     * </code>
     * </pre>
     *
     * @param id 序列化对id
     */
    protected void deleteByPK(PK id)  throws Exception {
        this.doDelete(id);
    }

    /**
     * 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>
     * 例如：以下删除主键唯一的记录
     * <pre>
     *         <code>
     * service.delete(1);
     * </code>
     * </pre>
     *
     * @param id 序列化对id
     *
     * @return 删除是否成功
     */
//    protected boolean delete(PK id)  throws Exception {
//        return this.doDelete(id) > 0;
//    }

    /**
     * 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>
     * 例如，如下语句查找id=5的唯一记录：
     *
     * <pre>
     *     <code>
     * User user = service.selectUnique(User.class, &quot;id&quot;, 5);
     * </code>
     * </pre>
     *
     * @param propertyName 属性名
     * @param value 属性值
     * @return 符合条件的唯一对象 or null if not found.
     */
    protected T selectUnique(String propertyName,Object value) throws Exception {
        QueryRule queryRule = QueryRule.getInstance();
        queryRule.andEqual(propertyName, value);
        return this.selectUnique(queryRule);
    }

    /**
     * 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
     *
     * <pre>
     *         <code>
     * boolean user2Exist = service.exists(User.class, 2);
     * </code>
     * </pre>
     * @param id 序列化对象id
     * @return 存在返回true，否则返回false
     */
    protected boolean exists(PK id)  throws Exception {
        return null != this.doLoad(id, this.op.rowMapper);
    }

    /**
     * 查询满足条件的记录数，使用hql.<br>
     * 例如：查询User里满足条件?name like "%ca%" 的记录数
     *
     * <pre>
     *         <code>
     * long count = service.getCount(&quot;from User where name like ?&quot;, &quot;%c
     * </code>
     * </pre>
     *
     * @param queryRule
     * @return 满足条件的记录数
     */
    protected long getCount(QueryRule queryRule) throws Exception {
        QueryRuleSqlBuilder bulider = new QueryRuleSqlBuilder(queryRule);
        Object [] values = bulider.getValues();
        String ws = removeFirstAnd(bulider.getWhereSql());
        String whereSql = ("".equals(ws) ? ws : (" where " + ws));
        String countSql = "select count(1) from " + getTableName() + whereSql;
        return (Long) this.jdbcTemplateReadOnly().queryForMap(countSql, values).get("c
    }

    /**
     * 根据某个属性值倒序获得第一个最大值
     * @param propertyName
     * @return
     */
    protected T getMax(String propertyName) throws Exception{
        QueryRule queryRule = QueryRule.getInstance();
        queryRule.addDescOrder(propertyName);
        Page<T> result = this.select(queryRule,1,1);
        if(null == result.getRows() || 0 == result.getRows().size()){
            return null;
        }else{
            return result.getRows().get(0);
        }
    }

    /**
     * 查询函数，使用查询规
     * 例如以下代码查询条件为匹配的数据
     *
     * <pre>
     *        <code>
     * QueryRule queryRule = QueryRule.getInstance();
     * queryRule.addLike(&quot;username&quot;, user.getUsername());
     * queryRule.addLike(&quot;monicker&quot;, user.getMonicker());
     * queryRule.addBetween(&quot;id&quot;, lowerId, upperId);
     * queryRule.addDescOrder(&quot;id&quot;);
     * queryRule.addAscOrder(&quot;username&quot;);
     * list = userService.select(User.class, queryRule);
     * </code>
     * </pre>
     *
     * @param queryRule 查询规则
     * @return 查询出的结果List
     */
    public List<T> select(QueryRule queryRule) throws Exception{
        QueryRuleSqlBuilder bulider = new QueryRuleSqlBuilder(queryRule);
        String ws = removeFirstAnd(bulider.getWhereSql());
        String whereSql = ("".equals(ws) ? ws : (" where " + ws));
        String sql = "select " + op.allColumn + " from " + getTableName() + whereSql;
        Object [] values = bulider.getValues();
        String orderSql = bulider.getOrderSql();
        orderSql = (StringUtils.isEmpty(orderSql) ? " " : (" order by " + orderSql));
        sql += orderSql;
        log.debug(sql);
        return (List<T>) this.jdbcTemplateReadOnly().query(sql, this.op.rowMapper, val
    }

    /**
     * 根据SQL语句执行查询，参数为Map
     * @param sql 语句
     * @param pamam 为Map，key为属性名，value为属性值
     * @return 符合条件的所有对象
     */
    protected List<Map<String,Object>> selectBySql(String sql,Map<String,?> pamam) thr
        return this.jdbcTemplateReadOnly().queryForList(sql,pamam);
    }

    /**
     * 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>
     * @param sql 语句
     * @param pamam 为Map，key为属性名，value为属性值
     * @return 符合条件的唯一对象，没符合条件的记录返回null.
     */
    protected Map<String,Object> selectUniqueBySql(String sql,Map<String,?> pamam) thr
        List<Map<String,Object>> list = selectBySql(sql,pamam);
        if (list.size() == 0) {
            return null;
        } else if (list.size() == 1) {
            return list.get(0);
        } else {
            throw new IllegalStateException("findUnique return " + list.size() + " rec
        }
    }

    /**
     * 根据SQL语句执行查询，参数为Object数组对象
     * @param sql 查询语句
     * @param args 为Object数组
     * @return 符合条件的所有对象
     */
    public List<Map<String,Object>> selectBySql(String sql,Object... args) throws Exce
        return this.jdbcTemplateReadOnly().queryForList(sql,args);
    }

    /**
     * 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>
     * @param sql 查询语句
     * @param args 为Object数组
     * @return 符合条件的唯一对象，没符合条件的记录返回null.
     */
    protected Map<String,Object> selectUniqueBySql(String sql,Object... args) throws E
        List<Map<String,Object>> list = selectBySql(sql, args);
        if (list.size() == 0) {
            return null;
        } else if (list.size() == 1) {
            return list.get(0);
        } else {
            throw new IllegalStateException("findUnique return " + list.size() + " rec
        }
    }

    /**
     * 根据SQL语句执行查询，参数为List对象
     * @param sql 查询语句
     * @param list<Object>对象
     * @return 符合条件的所有对象
     */
    protected List<Map<String,Object>> selectBySql(String sql,List<Object> list) throw
        return this.jdbcTemplateReadOnly().queryForList(sql,list.toArray());
    }

    /**
     * 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>
     * @param sql 查询语句
     * @param listParam 属性值List
     * @return 符合条件的唯一对象，没符合条件的记录返回null.
     */
    protected Map<String,Object> selectUniqueBySql(String sql,List<Object> listParam)
        List<Map<String,Object>> listMap = selectBySql(sql, listParam);
        if (listMap.size() == 0) {
            return null;
        } else if (listMap.size() == 1) {
            return listMap.get(0);
        } else {
            throw new IllegalStateException("findUnique return " + listMap.size() + "
        }
    }

    /**
     * 分页查询函数，使用查询规则<br>
     * 例如以下代码查询条件为匹配的数据
     *
     * <pre>
     *        <code>
     * QueryRule queryRule = QueryRule.getInstance();
     * queryRule.addLike(&quot;username&quot;, user.getUsername());
     * queryRule.addLike(&quot;monicker&quot;, user.getMonicker());
     * queryRule.addBetween(&quot;id&quot;, lowerId, upperId);
     * queryRule.addDescOrder(&quot;id&quot;);
     * queryRule.addAscOrder(&quot;username&quot;);
     * page = userService.select(queryRule, pageNo, pageSize);
     * </code>
     * </pre>
     *
     * @param queryRule 查询规则
     * @param pageNo 页号,从1开始
     * @param pageSize  每页的记录条数
     * @return 查询出的结果Page
     */
    public Page<T> select(QueryRule queryRule,final int pageNo, final int pageSize) th
        QueryRuleSqlBuilder bulider = new QueryRuleSqlBuilder(queryRule);
        Object [] values = bulider.getValues();
        String ws = removeFirstAnd(bulider.getWhereSql());
        String whereSql = ("".equals(ws) ? ws : (" where " + ws));
        String countSql = "select count(1) from " + getTableName() + whereSql;
        long count = (Long) this.jdbcTemplateReadOnly().queryForMap(countSql, values).
        if (count == 0) {
            return new Page<T>();
        }
        long start = (pageNo - 1) * pageSize;
        // 有数据的情况下，继续查询
        String orderSql = bulider.getOrderSql();
        orderSql = (StringUtils.isEmpty(orderSql) ? " " : (" order by " + orderSql));
        String sql = "select " + op.allColumn +" from " + getTableName() + whereSql +
        List<T> list = (List<T>) this.jdbcTemplateReadOnly().query(sql, this.op.rowMap
        log.debug(sql);
        return new Page<T>(start, count, pageSize, list);
    }


    /**
     * 分页查询特殊SQL语句
     * @param sql 语句
     * @param param  查询条件
     * @param pageNo    页码
     * @param pageSize    每页内容
     * @return
     */
    protected Page<Map<String,Object>> selectBySqlToPage(String sql, Map<String,?> par
        String countSql = "select count(1) from (" + sql + ") a";
        long count = (Long) this.jdbcTemplateReadOnly().queryForMap(countSql,param).ge

//        long count = this.jdbcTemplateReadOnly().queryForMap(countSql, param);
        if (count == 0) {
            return new Page<Map<String,Object>>();
        }
        long start = (pageNo - 1) * pageSize;
        // 有数据的情况下，继续查询
        sql = sql + " limit " + start + "," + pageSize;
        List<Map<String,Object>> list = (List<Map<String,Object>>) this.jdbcTemplateRe
        log.debug(sql);
        return new Page<Map<String,Object>>(start, count, pageSize, list);
    }


    /**
     * 分页查询特殊SQL语句
     * @param sql 语句
     * @param param  查询条件
     * @param pageNo    页码
     * @param pageSize    每页内容
     * @return
     */
    public Page<Map<String,Object>> selectBySqlToPage(String sql, Object [] param, fir
        String countSql = "select count(1) from (" + sql + ") a";

        long count = (Long) this.jdbcTemplateReadOnly().queryForMap(countSql,param).ge
//        long count = this.jdbcTemplateReadOnly().queryForLong(countSql, param);
        if (count == 0) {
            return new Page<Map<String,Object>>();
        }
        long start = (pageNo - 1) * pageSize;
        sql = sql + " limit " + start + "," + pageSize;
        List<Map<String,Object>> list = (List<Map<String,Object>>) this.jdbcTemplateRe
        log.debug(sql);
        return new Page<Map<String,Object>>(start, count, pageSize, list);
    }

    /**
     * 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>
     * 例如，如下语句查找sex=1,age=18的所有记录：
     *
     * <pre>
     *     <code>
     * Map properties = new HashMap();
     * properties.put(&quot;sex&quot;, &quot;1&quot;);
     * properties.put(&quot;age&quot;, 18);
     * User user = service.selectUnique(properties);
     * </code>
     * </pre>
     *
     * @param properties 属性值Map，key为属性名，value为属性值
     * @return 符合条件的唯一对象，没符合条件的记录返回null.
     */
    protected T selectUnique(Map<String, Object> properties) throws Exception {
        QueryRule queryRule = QueryRule.getInstance();
        for (String key : properties.keySet()) {
            queryRule.andEqual(key, properties.get(key));
        }
        return selectUnique(queryRule);
    }

    /**
     * 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>
     * <pre>
     *     <code>
     * QueryRule queryRule = QueryRule.getInstance();
     * queryRule.addLike(&quot;username&quot;, user.getUsername());
     * queryRule.addLike(&quot;monicker&quot;, user.getMonicker());
     * queryRule.addBetween(&quot;id&quot;, lowerId, upperId);
     * User user = service.selectUnique(queryRule);
     * </code>
     * </pre>
     *
     * @param queryRule  查询规则
     * @return 符合条件的唯一对象，没符合条件的记录返回null.
     */
    protected T selectUnique(QueryRule queryRule) throws Exception {
        List<T> list = select(queryRule);
        if (list.size() == 0) {
            return null;
        } else if (list.size() == 1) {
            return list.get(0);
        } else {
            throw new IllegalStateException("findUnique return " + list.size() + " rec
        }
    }


    /**
     * 根据当前list进行相应的分页返回
     * @param objList
     * @param pageNo
     * @param pageSize
     * @return Page
     */
    protected Page<T> pagination(List<T> objList, int pageNo, int pageSize) throws Exc
        List<T> objectArray = new ArrayList<T>(0);
        int startIndex = (pageNo - 1) * pageSize;
        int endIndex = pageNo * pageSize;
        if(endIndex >= objList.size()){
            endIndex = objList.size();
        }
        for (int i = startIndex; i < endIndex; i++) {
            objectArray.add(objList.get(i));
        }
        return new Page<T>(startIndex, objList.size(), pageSize, objectArray);
    }

    /**
     * 合并PO List对象.(如果POJO中的值为null,则继续使用PO中的值）
     *
     * @param pojoList  传入的POJO的List
     * @param poList 传入的PO的List
     * @param idName ID字段名称
     */
    protected void mergeList(List<T> pojoList, List<T> poList, String idName) throws E
        mergeList(pojoList, poList, idName, false);
    }

    /**
     * 合并PO List对象.
     *
     * @param pojoList 传入的POJO的List
     * @param poList 传入的PO的List
     * @param idName  ID字段名称
     * @param isCopyNull 是否拷贝null(当POJO中的值为null时， 如果isCopyNull=ture,则用nu
     */
    protected void mergeList(List<T> pojoList, List<T> poList, String idName,boolean i
        Map<Object, Object> map = new HashMap<Object, Object>();
        Map<String, PropertyMapping> pm = op.mappings;
        for (Object element : pojoList) {
            Object key;
            try {
                key = pm.get(idName).getter.invoke(element);
                map.put(key, element);
            } catch (Exception e) {
                throw new IllegalArgumentException(e);
            }
        }
        for (Iterator<T> it = poList.iterator(); it.hasNext();) {
            T element = it.next();
            try {
                Object key = pm.get(idName).getter.invoke(element);
                if (!map.containsKey(key)) {
                    delete(element);
                    it.remove();
                } else {
                    DataUtils.copySimpleObject(map.get(key), element, isCopyNull);
                }
            } catch (Exception e) {
                throw new IllegalArgumentException(e);
            }
        }
        T[] pojoArray = (T[])pojoList.toArray();
        for (int i = 0; i < pojoArray.length; i++) {
            T element = pojoArray[i];
            try {
                Object key = pm.get(idName).getter.invoke(element);
                if (key == null) {
                    poList.add(element);
                }
            } catch (Exception e) {
                throw new IllegalArgumentException(e);
            }
        }
    }

    private String removeFirstAnd(String sql){
        if(StringUtils.isEmpty(sql)){return sql;}
        return sql.trim().toLowerCase().replaceAll("^\\s*and", "") + " ";
    }

    private EntityOperation<T> getOp(){
        return this.op;
    }



    /**
     * ResultSet -> Object
     *
     * @param <T>
     *
     * @param rs
     * @param obj
     */
    private <T> T populate(ResultSet rs, T obj) {
        try {
            ResultSetMetaData metaData = rs.getMetaData(); // 取得结果集的元元素
            int colCount = metaData.getColumnCount(); // 取得所有列的个数
            Field[] fields = obj.getClass().getDeclaredFields();
            for (int i = 0; i < fields.length; i++) {
                Field f = fields[i];
                // rs的游标从1开始，需要注意
                for (int j = 1; j <= colCount; j++) {
                    Object value = rs.getObject(j);
                    String colName = metaData.getColumnName(j);
                    if (!f.getName().equalsIgnoreCase(colName)) {
                        continue;
                    }

                    // 如果列名中有和字段名一样的，则设置值
                    try {
                        BeanUtils.copyProperty(obj, f.getName(), value);
                    } catch (Exception e) {
                        log.warn("BeanUtils.copyProperty error, field name: "
                                + f.getName() + ", error: " + e);
                    }

                }
            }
        } catch (Exception e) {
            log.warn("populate error...." + e);
        }
        return obj;
    }

    /**
     * 封装一下JdbcTemplate的queryForObject（默认查不到会抛异常）方法，
     *
     * @param sql
     * @param mapper
     * @param args
     * @return 如查询不到，返回null，不抛异常；查询到多个，也抛出异常
     */
    private <T> T selectForObject(String sql, RowMapper<T> mapper,
            Object... args) {
        List<T> results = this.jdbcTemplateReadOnly().query(sql, mapper, args);
        return DataAccessUtils.singleResult(results);
    }

    protected byte[] getBlobColumn(ResultSet rs, int columnIndex)
            throws SQLException {
        try {
            Blob blob = rs.getBlob(columnIndex);
            if (blob == null) {
                return null;
            }

            InputStream is = blob.getBinaryStream();
            ByteArrayOutputStream bos = new ByteArrayOutputStream();

            if (is == null) {
                return null;
            } else {
                byte buffer[] = new byte[64];
                int c = is.read(buffer);
                while (c > 0) {
                    bos.write(buffer, 0, c);
                    c = is.read(buffer);
                }
                return bos.toByteArray();
            }
        } catch (IOException e) {
            throw new SQLException(
                    "Failed to read BLOB column due to IOException: "
                            + e.getMessage());
        }
    }

    protected void setBlobColumn(PreparedStatement stmt, int parameterIndex,
            byte[] value) throws SQLException {
        if (value == null) {
            stmt.setNull(parameterIndex, Types.BLOB);
        } else {
            stmt.setBinaryStream(parameterIndex,
                    new ByteArrayInputStream(value), value.length);
        }
    }

    protected String getClobColumn(ResultSet rs, int columnIndex)
            throws SQLException {
        try {
            Clob clob = rs.getClob(columnIndex);
            if (clob == null) {
                return null;
            }

            StringBuffer ret = new StringBuffer();
            InputStream is = clob.getAsciiStream();

            if (is == null) {
                return null;
            } else {
                byte buffer[] = new byte[64];
                int c = is.read(buffer);
                while (c > 0) {
                    ret.append(new String(buffer, 0, c));
                    c = is.read(buffer);
                }
                return ret.toString();
            }
        } catch (IOException e) {
            throw new SQLException(
                    "Failed to read CLOB column due to IOException: "
                            + e.getMessage());
        }
    }

    protected void setClobColumn(PreparedStatement stmt, int parameterIndex,
            String value) throws SQLException {
        if (value == null) {
            stmt.setNull(parameterIndex, Types.CLOB);
        } else {
            stmt.setAsciiStream(parameterIndex,
                    new ByteArrayInputStream(value.getBytes()), value.length());
        }
    }

    /**
```

```java
     * 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）
     * @param <T>
     *
     * @param sql
     * @param rowMapper
     * @param args
     * @param pageNo
     * @param pageSize
     * @return
     */
    private <T> Page simplePageQuery(String sql, RowMapper<T> rowMapper, Map<String, ?
        long start = (pageNo - 1) * pageSize;
        return simplePageQueryByStart(sql,rowMapper,args,start,pageSize);
    }

    /**
     *
     * @param sql
     * @param rowMapper
     * @param args
     * @param start
     * @param pageSize
     * @return
     */
    private <T> Page simplePageQueryByStart(String sql, RowMapper<T> rowMapper, Map<St
        // 首先查询总数
        String countSql = "select count(*) " + removeSelect(removeOrders(sql));

        long count = (Long) this.jdbcTemplateReadOnly().queryForMap(countSql,args).get
//        long count = this.jdbcTemplateReadOnly().queryForLong(countSql, args);
        if (count == 0) {
            log.debug("no result..");
            return new Page();
        }
        // 有数据的情况下，继续查询
        sql = sql + " limit " + start + "," + pageSize;
        log.debug(StringUtils.format("[Execute SQL]sql:{0},params:{1}", sql, args));
        List<T> list = this.jdbcTemplateReadOnly().query(sql, rowMapper, args);
        return new Page(start, count, (int)pageSize, list);
    }

    protected long queryCount(String sql,Map<String, ?> args){
        String countSql = "select count(1) " + removeSelect(removeOrders(sql));

        return (Long)this.jdbcTemplateReadOnly().queryForMap(countSql, args).get("cour
    }

    protected <T> List<T> simpleListQueryByStart(String sql, RowMapper<T> rowMapper,
            Map<String, ?> args, long start, long pageSize) {

        sql = sql + " limit " + start + "," + pageSize;
        log.debug(StringUtils.format("[Execute SQL]sql:{0},params:{1}", sql, args));
        List<T> list = this.jdbcTemplateReadOnly().query(sql, rowMapper, args);
        if(list == null){
            return new ArrayList<T>();
        }
        return list;
    }

    /**
     * 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）
     *
     * @param sql
     * @param rm
     * @param args
     * @param pageNo
     * @param pageSize
     * @return
     */
    private Page simplePageQueryNotT(String sql, RowMapper rm, Map<String, ?> args, lo
        // 首先查询总数
        String countSql = "select count(*) " + removeSelect(removeOrders(sql));
        long count = (Long)this.jdbcTemplateReadOnly().queryForMap(countSql, args).get
        if (count == 0) {
            log.debug("no result..");
            return new Page();
        }
        // 有数据的情况下，继续查询
        long start = (pageNo - 1) * pageSize;
        sql = sql + " limit " + start + "," + pageSize;
        log.debug(StringUtils.format("[Execute SQL]sql:{0},params:{1}", sql, args));
        List list = this.jdbcTemplateReadOnly().query(sql, rm, args);
        return new Page(start, count, (int)pageSize, list);
    }

    /**
```

```
     * 去掉order
     *
     * @param sql
     * @return
     */
    private String removeOrders(String sql) {
        Pattern p = Pattern.compile("order\\s*by[\\w|\\W|\\s|\\S]*", Pattern.CASE_INSE
        Matcher m = p.matcher(sql);
        StringBuffer sb = new StringBuffer();
        while (m.find()) {
            m.appendReplacement(sb, "");
        }
        m.appendTail(sb);
        return sb.toString();
    }

    /**
     * 去掉select
     *
     * @param sql
     * @return
     */
    private String removeSelect(String sql) {
        int beginPos = sql.toLowerCase().indexOf("from");
        return sql.substring(beginPos);
    }


    private long getMaxId(String table, String column) {
        String sql = "SELECT max(" + column + ") FROM " + table + " ";
        long maxId = (Long)this.jdbcTemplateReadOnly().queryForMap(sql).get("max(" + c
        return maxId;
    }

    /**
     * 生成简单对象UPDATE语句，简化sql拼接
     * @param tableName
     * @param pkName
     * @param pkValue
     * @param params
     * @return
     */
    private String makeSimpleUpdateSql(String tableName, String pkName, Object pkValue
```

```java
        if(StringUtils.isEmpty(tableName) || params == null || params.isEmpty()){
            return "";
        }

        StringBuffer sb = new StringBuffer();
        sb.append("update ").append(tableName).append(" set ");
        //添加参数
        Set<String> set = params.keySet();
        int index = 0;
        for (String key : set) {
//             sb.append(key).append(" = :").append(key);
             sb.append(key).append(" = ?");
             if(index != set.size() - 1){
                 sb.append(",");
             }
             index++;
        }
//        sb.append(" where ").append(pkName).append(" = :").append(pkName) ;
        sb.append(" where ").append(pkName).append(" = ?");
        params.put("where_" + pkName,params.get(pkName));

        return sb.toString();
    }


    /**
     * 生成简单对象UPDATE语句，简化sql拼接
     * @param pkName
     * @param pkValue
     * @param params
     * @return
     */
    private String makeSimpleUpdateSql(String pkName, Object pkValue, Map<String, Obje
        if(StringUtils.isEmpty(getTableName()) || params == null || params.isEmpty()){
            return "";
        }

        StringBuffer sb = new StringBuffer();
        sb.append("update ").append(getTableName()).append(" set ");
        //添加参数
        Set<String> set = params.keySet();
        int index = 0;
        for (String key : set) {
            sb.append(key).append(" = :").append(key);
            if(index != set.size() - 1){
                sb.append(",");
            }
            index++;
        }
        sb.append(" where ").append(pkName).append(" = :").append(pkName) ;

        return sb.toString();
    }



    /**
     * 生成对象INSERT语句, 简化sql拼接
     * @param tableName
     * @param params
     * @return
     */
    private String makeSimpleReplaceSql(String tableName, Map<String, Object> params){
        if(StringUtils.isEmpty(tableName) || params == null || params.isEmpty()){
            return "";
        }
        StringBuffer sb = new StringBuffer();
        sb.append("replace into ").append(tableName);

        StringBuffer sbKey = new StringBuffer();
        StringBuffer sbValue = new StringBuffer();

        sbKey.append("(");
        sbValue.append("(");
        //添加参数
        Set<String> set = params.keySet();
        int index = 0;
        for (String key : set) {
            sbKey.append(key);
            sbValue.append(" :").append(key);
            if(index != set.size() - 1){
                sbKey.append(",");
                sbValue.append(",");
            }
            index++;
        }
        sbKey.append(")");
        sbValue.append(")");

        sb.append(sbKey).append("VALUES").append(sbValue);

        return sb.toString();
    }

    /**
     * 生成对象INSERT语句，简化sql拼接
     * @param tableName
     * @param params
     * @return
     */
    private String makeSimpleReplaceSql(String tableName, Map<String, Object> params,L
        if(StringUtils.isEmpty(tableName) || params == null || params.isEmpty()){
            return "";
        }
        StringBuffer sb = new StringBuffer();
        sb.append("replace into ").append(tableName);

        StringBuffer sbKey = new StringBuffer();
        StringBuffer sbValue = new StringBuffer();

        sbKey.append("(");
        sbValue.append("(");
        //添加参数
        Set<String> set = params.keySet();
        int index = 0;
        for (String key : set) {
            sbKey.append(key);
            sbValue.append(" ?");
            if(index != set.size() - 1){
                sbKey.append(",");
                sbValue.append(",");
            }
            index++;
            values.add(params.get(key));
        }
        sbKey.append(")");
        sbValue.append(")");

        sb.append(sbKey).append("VALUES").append(sbValue);
```

```java

        return sb.toString();
    }



    /**
     * 生成对象INSERT语句，简化sql拼接
     * @param tableName
     * @param params
     * @return
     */
    private String makeSimpleInsertSql(String tableName, Map<String, Object> params){
        if(StringUtils.isEmpty(tableName) || params == null || params.isEmpty()){
            return "";
        }
        StringBuffer sb = new StringBuffer();
        sb.append("insert into ").append(tableName);

        StringBuffer sbKey = new StringBuffer();
        StringBuffer sbValue = new StringBuffer();

        sbKey.append("(");
        sbValue.append("(");
        //添加参数
        Set<String> set = params.keySet();
        int index = 0;
        for (String key : set) {
            sbKey.append(key);
//            sbValue.append(" :").append(key);
            sbValue.append(" ?");
            if(index != set.size() - 1){
                sbKey.append(",");
                sbValue.append(",");
            }
            index++;
        }
        sbKey.append(")");
        sbValue.append(")");

        sb.append(sbKey).append("VALUES").append(sbValue);

        return sb.toString();
    }

    /**
     * 生成对象INSERT语句，简化sql拼接
     * @param tableName
     * @param params
     * @return
     */
    private String makeSimpleInsertSql(String tableName, Map<String, Object> params,Li
        if(StringUtils.isEmpty(tableName) || params == null || params.isEmpty()){
            return "";
        }
        StringBuffer sb = new StringBuffer();
        sb.append("insert into ").append(tableName);

        StringBuffer sbKey = new StringBuffer();
        StringBuffer sbValue = new StringBuffer();

        sbKey.append("(");
        sbValue.append("(");
        //添加参数
        Set<String> set = params.keySet();
        int index = 0;
        for (String key : set) {
            sbKey.append(key);
            sbValue.append(" ?");
            if(index != set.size() - 1){
                sbKey.append(",");
                sbValue.append(",");
            }
            index++;
            values.add(params.get(key));
        }
        sbKey.append(")");
        sbValue.append(")");

        sb.append(sbKey).append("VALUES").append(sbValue);

        return sb.toString();
    }


    private Serializable doInsertRuturnKey(Map<String,Object> params){
        final List<Object> values = new ArrayList<Object>();
        final String sql = makeSimpleInsertSql(getTableName(),params,values);
        KeyHolder keyHolder = new GeneratedKeyHolder();
        final JdbcTemplate jdbcTemplate = new JdbcTemplate(getDataSourceWrite());
          try {

                jdbcTemplate.update(new PreparedStatementCreator() {
                public PreparedStatement createPreparedStatement(

                        Connection con) throws SQLException {
                    PreparedStatement ps = con.prepareStatement(sql,Statement.RETURN_G

                    for (int i = 0; i < values.size(); i++) {
                        ps.setObject(i+1, values.get(i)==null?null:values.get(i));

                    }
                    return ps;
                }

           }, keyHolder);
          } catch (DataAccessException e) {
              log.error("error",e);
          }



        if (keyHolder == null) { return ""; }


        Map<String, Object> keys = keyHolder.getKeys();
        if (keys == null || keys.size() == 0 || keys.values().size() == 0) {
            return "";
        }
        Object key = keys.values().toArray()[0];
        if (key == null || !(key instanceof Serializable)) {
            return "";
        }
        if (key instanceof Number) {
            //Long k = (Long) key;
            Class clazz = key.getClass();
//              return clazz.cast(key);
            return (clazz == int.class || clazz == Integer.class) ? ((Number) key).int


        } else if (key instanceof String) {
            return (String) key;
        } else {
            return (Serializable) key;
        }


    }


    /**
     * 生成默认的对象UPDATE语句，简化sql拼接
     * @param pkValue
     * @param params
     * @return
     */
    private String makeDefaultSimpleUpdateSql(Object pkValue, Map<String, Object> para
        return this.makeSimpleUpdateSql(getTableName(), getPKColumn(), pkValue, params
    }

    /**
     * 生成默认的对象INSERT语句，简化sql拼接
     * @param params
     * @return
     */
    private String makeDefaultSimpleInsertSql(Map<String, Object> params){
        return this.makeSimpleInsertSql(this.getTableName(), params);
    }

    /**
     * 获取一个实例对象
     * @param tableName
     * @param pkName
     * @param pkValue
     * @param rm
     * @return
     */
    private Object doLoad(String tableName, String pkName, Object pkValue, RowMapper r
        StringBuffer sb = new StringBuffer();
        sb.append("select * from ").append(tableName).append(" where ").append(pkName)
        List<Object> list = this.jdbcTemplateReadOnly().query(sb.toString(), rm, pkVal
        if(list == null || list.isEmpty()){
```

```java
            return null;
        }
        return list.get(0);
    }

    /**
     * 获取默认的实例对象
     * @param <T>
     * @param pkValue
     * @param rowMapper
     * @return
     */
    private <T> T doLoad(Object pkValue, RowMapper<T> rowMapper){
        Object obj = this.doLoad(getTableName(), getPKColumn(), pkValue, rowMapper);
        if(obj != null){
            return (T)obj;
        }
        return null;
    }


    /**
     * 删除实例对象，返回删除记录数
     * @param tableName
     * @param pkName
     * @param pkValue
     * @return
     */
    private int doDelete(String tableName, String pkName, Object pkValue) {
        StringBuffer sb = new StringBuffer();
        sb.append("delete from ").append(tableName).append(" where ").append(pkName).a
        int ret = this.jdbcTemplateWrite().update(sb.toString(), pkValue);
        return ret;
    }

    /**
     * 删除默认实例对象，返回删除记录数
     * @param pkValue
     * @return
     */
    private int doDelete(Object pkValue){
        return this.doDelete(getTableName(), getPKColumn(), pkValue);
    }

    /**
     * 更新实例对象，返回删除记录数
     * @param tableName
     * @param pkName
     * @param pkValue
     * @param params
     * @return
     */
    private int doUpdate(String tableName, String pkName, Object pkValue, Map<String,
        params.put(pkName, pkValue);
        String sql = this.makeSimpleUpdateSql(tableName, pkName, pkValue, params);
        int ret = this.jdbcTemplateWrite().update(sql, params.values().toArray());
        return ret;
    }

    /**
     * 更新实例对象，返回删除记录数
     * @param pkName
     * @param pkValue
     * @param params
     * @return
     */
    private int doUpdate( String pkName, Object pkValue, Map<String, Object> params){
        params.put(pkName, pkValue);
        String sql = this.makeSimpleUpdateSql( pkName, pkValue, params);
        int ret = this.jdbcTemplateWrite().update(sql, params.values().toArray());
        return ret;
    }

    /**
     * 更新实例对象，返回删除记录数
     * @param pkValue
     * @param params
     * @return
     */
    private int doUpdate(Object pkValue, Map<String, Object> params){
        //
        String sql = this.makeDefaultSimpleUpdateSql(pkValue, params);
        params.put(this.getPKColumn(), pkValue);
        int ret = this.jdbcTemplateWrite().update(sql, params.values().toArray());
        return ret;
    }


    private boolean doReplace(Map<String, Object> params) {
        String sql = this.makeSimpleReplaceSql(this.getTableName(), params);
        int ret = this.jdbcTemplateWrite().update(sql, params.values().toArray());
        return ret > 0;
    }

    private boolean doReplace(String tableName, Map<String, Object> params){
        String sql = this.makeSimpleReplaceSql(tableName, params);
        int ret = this.jdbcTemplateWrite().update(sql, params.values().toArray());
        return ret > 0;
    }


    /**
     * 插入
     * @param tableName
     * @param params
     * @return
     */
    private boolean doInsert(String tableName, Map<String, Object> params){
        String sql = this.makeSimpleInsertSql(tableName, params);
        int ret = this.jdbcTemplateWrite().update(sql, params.values().toArray());
        return ret > 0;
    }

    /**
     * 插入
     * @param params
     * @return
     */
    private boolean doInsert(Map<String, Object> params) {
        String sql = this.makeSimpleInsertSql(this.getTableName(), params);
        int ret = this.jdbcTemplateWrite().update(sql, params.values().toArray());
        return ret > 0;
    }

    /**
     * 获取主键列名称 建议子类重写
     * @return
     */
    protected abstract String getPKColumn();

    protected abstract void setDataSource(DataSource dataSource);

    private Map<String,Object> convertMap(Object obj){
        Map<String,Object> map = new HashMap<String,Object>();

         List<FieldInfo> getters = TypeUtils.computeGetters(obj.getClass(), null);
         for(int i=0,len=getters.size();i<len;i++){
             FieldInfo fieldInfo = getters.get(i);
             String name = fieldInfo.getName();
            try {
                Object value = fieldInfo.get(obj);
                map.put(name,value);
            } catch (Exception e) {
                log.error(String.format("convertMap error object:%s  field: %s",obj.tc
            }
         }

        return map;
    }

}
```

3. 动态数据源切换的底层原理

DynamicDataSourceEntry

```
package javax.core.common.jdbc.datasource;


import org.aspectj.lang.JoinPoint;

/**
 * 动态切换数据源
 *
 */
public class DynamicDataSourceEntry {

    // 默认数据源
    public final static String DEFAULT_SOURCE = null;

```

```java
    private final static ThreadLocal<String> local = new ThreadLocal<String>();

    /**
     * 清空数据源
     */
    public void clear() {
        local.remove();
    }

    /**
     * 获取当前正在使用的数据源名字
     *
     * @return String
     */
    public String get() {
         return local.get();
    }

    /**
     * 还原指定切面的数据源
     *
     * @param join
     */
    public void restore(JoinPoint join) {
        local.set(DEFAULT_SOURCE);
    }

    /**
     * 还原当前切面的数据源
     */
    public void restore() {
        local.set(DEFAULT_SOURCE);
    }

    /**
     * 设置已知名字的数据源
     *
     * @param source
     */
    public void set(String source) {
        local.set(source);
    }

    /**
     * 根据年份动态设置数据源
     * @param year
     */
    public void set(int year) {
        local.set("DB_" + year);
    }
}
```

DynamicDataSource

```java
package javax.core.common.jdbc.datasource;

import org.springframework.jdbc.datasource.lookup.AbstractRoutingDataSource;

/**
 * 动态数据源
 *
 *
 */
public class DynamicDataSource extends AbstractRoutingDataSource {


    //entry的目的，主要是用来给每个数据源打个标记
    private DynamicDataSourceEntry dataSourceEntry;

    @Override
    protected Object determineCurrentLookupKey() {
        return this.dataSourceEntry.get();
    }

    public void setDataSourceEntry(DynamicDataSourceEntry dataSourceEntry) {
        this.dataSourceEntry = dataSourceEntry;
    }

    public DynamicDataSourceEntry getDataSourceEntry(){
            return this.dataSourceEntry;
    }

}
```

4. 运行效果演示

创建 Member实体类

```java
package cn.sitedev.orm.demo.entity;

import lombok.Data;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import java.io.Serializable;

@Entity
@Table(name="t_member")
@Data
public class Member implements Serializable {
    @Id private Long id;
    private String name;
    private String addr;
    private Integer age;

    @Override
    public String toString() {
        return "Member{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", addr='" + addr + '\'' +
                ", age=" + age +
                '}';
    }
}
```

创建 Order实体类

```java
package cn.sitedev.orm.demo.entity;

import lombok.Data;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Table;
import java.io.Serializable;

@Entity
@Table(name="t_order")
@Data
public class Order implements Serializable {
    private Long id;
    @Column(name="mid")
    private Long memberId;
    private String detail;
    private Long createTime;
    private String createTimeFmt;

    @Override
    public String toString() {
        return "Order{" +
                "id=" + id +
                ", memberId=" + memberId +
                ", detail='" + detail + '\'' +
                ", createTime=" + createTime +
                ", createTimeFmt='" + createTimeFmt + '\'' +
                '}';
    }
}
```

创建 MemberDao

```java
package cn.sitedev.orm.demo.dao;

import cn.sitedev.orm.demo.entity.Member;
import cn.sitedev.orm.framework.BaseDaoSupport;
import cn.sitedev.orm.framework.QueryRule;
import org.springframework.stereotype.Repository;

import javax.annotation.Resource;
import javax.core.common.Page;
import javax.sql.DataSource;
import java.util.List;
import java.util.Map;
```

```java

@Repository
public class MemberDao extends BaseDaoSupport<Member,Long> {

    @Override
    protected String getPKColumn() {
        return "id";
    }

    @Resource(name="dataSource")
    public void setDataSource(DataSource dataSource){
        super.setDataSourceReadOnly(dataSource);
        super.setDataSourceWrite(dataSource);
    }


    public List<Member> selectAll() throws  Exception{
        QueryRule queryRule = QueryRule.getInstance();
        queryRule.andLike("name","Mic%");
        return super.select(queryRule);
    }


    public Page<Member> selectForPage(int pageNo,int pageSize) throws Exception{
        QueryRule queryRule = QueryRule.getInstance();
        queryRule.andLike("name","Tom%");
        Page<Member> page = super.select(queryRule,pageNo,pageSize);
        return page;
    }

    public void select() throws Exception{
        String sql = "";
        List<Map<String,Object>> result = super.selectBySql(sql);
//        System.out.println(JSON.parseObject(JSON.toJSONString(result)),Member.class);
    }

    public boolean insert(Member entity) throws Exception{
        super.setTableName("t_mmmmm");
        return super.insert(entity);
    }
}
```

创建 OrderDao

```java
package cn.sitedev.orm.demo.dao;

import cn.sitedev.orm.demo.entity.Order;
import cn.sitedev.orm.framework.BaseDaoSupport;
import org.springframework.stereotype.Repository;

import javax.annotation.Resource;
import javax.core.common.jdbc.datasource.DynamicDataSource;
import javax.sql.DataSource;
import java.text.SimpleDateFormat;
import java.util.Date;


@Repository
public class OrderDao extends BaseDaoSupport<Order, Long> {

    private SimpleDateFormat yearFormat = new SimpleDateFormat("yyyy");
    private SimpleDateFormat fullDataFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss
    private DynamicDataSource dataSource;
    @Override
    protected String getPKColumn() {return "id";}

    @Resource(name="dynamicDataSource")
    public void setDataSource(DataSource dataSource) {
        this.dataSource = (DynamicDataSource)dataSource;
        this.setDataSourceReadOnly(dataSource);
        this.setDataSourceWrite(dataSource);
    }

    /**
     * @throws Exception
     *
     */
    public boolean insertOne(Order order) throws Exception{
        //约定优于配置
        Date date = null;
        if(order.getCreateTime() == null){
            date = new Date();
            order.setCreateTime(date.getTime());
        }else {
            date = new Date(order.getCreateTime());
        }
        Integer dbRouter = Integer.valueOf(yearFormat.format(date));
        System.out.println("自动分配到【DB_" + dbRouter + "】数据源");
        this.dataSource.getDataSourceEntry().set(dbRouter);

        order.setCreateTimeFmt(fullDataFormat.format(date));

        Long orderId = super.insertAndReturnId(order);
        order.setId(orderId);
        return orderId > 0;
    }


}
```

修改db.properties文件

```properties
#sysbase database mysql config

#mysql.jdbc.driverClassName=com.mysql.jdbc.Driver
#mysql.jdbc.url=jdbc:mysql://127.0.0.1:3306/gp-vip-spring-db-demo?characterEncoding=UTF
#mysql.jdbc.username=root
#mysql.jdbc.password=123456

db2019.mysql.jdbc.driverClassName=com.mysql.jdbc.Driver
db2019.mysql.jdbc.url=jdbc:mysql://127.0.0.1:3306/spring-db-2019?characterEncoding=UTF-
db2019.mysql.jdbc.username=root
db2019.mysql.jdbc.password=root

db2020.mysql.jdbc.driverClassName=com.mysql.jdbc.Driver
db2020.mysql.jdbc.url=jdbc:mysql://127.0.0.1:3306/spring-db-2020?characterEncoding=UTF-
db2020.mysql.jdbc.username=root
db2020.mysql.jdbc.password=root

#alibaba druid config

dbPool.initialSize=1
dbPool.minIdle=1
dbPool.maxActive=200
dbPool.maxWait=60000
dbPool.timeBetweenEvictionRunsMillis=60000
dbPool.minEvictableIdleTimeMillis=300000
dbPool.validationQuery=SELECT 'x'
dbPool.testWhileIdle=true
dbPool.testOnBorrow=false
dbPool.testOnReturn=false
dbPool.poolPreparedStatements=false
dbPool.maxPoolPreparedStatementPerConnectionSize=20
dbPool.filters=stat,log4j,wall
```

修改application-db.xml文件

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.sprin
        http://www.springframework.org/schema/tx http://www.springframework.org/schema/
        http://www.springframework.org/schema/context http://www.springframework.org/sc
           http://www.springframework.org/schema/aop http://www.springframework.org/sch

    <bean id="datasourcePool" abstract="true" class="com.alibaba.druid.pool.DruidDataSo
        <property name="initialSize" value="${dbPool.initialSize}" />
        <property name="minIdle" value="${dbPool.minIdle}" />
        <property name="maxActive" value="${dbPool.maxActive}" />
        <property name="maxWait" value="${dbPool.maxWait}" />
        <property name="timeBetweenEvictionRunsMillis" value="${dbPool.timeBetweenEvict
        <property name="minEvictableIdleTimeMillis" value="${dbPool.minEvictableIdleTim
        <property name="validationQuery" value="${dbPool.validationQuery}" />
        <property name="testWhileIdle" value="${dbPool.testWhileIdle}" />
        <property name="testOnBorrow" value="${dbPool.testOnBorrow}" />
        <property name="testOnReturn" value="${dbPool.testOnReturn}" />
        <property name="poolPreparedStatements" value="${dbPool.poolPreparedStatements}
        <property name="maxPoolPreparedStatementPerConnectionSize" value="${dbPool.maxP
        <property name="filters" value="${dbPool.filters}" />
    </bean>

    <bean id="dataSource2019" parent="datasourcePool">
        <property name="driverClassName" value="${db2019.mysql.jdbc.driverClassName}" /
```

```xml
        <property name="url" value="${db2019.mysql.jdbc.url}" />
        <property name="username" value="${db2019.mysql.jdbc.username}" />
        <property name="password" value="${db2019.mysql.jdbc.password}" />
    </bean>

    <bean id="dataSource" parent="datasourcePool">
        <property name="driverClassName" value="${db2020.mysql.jdbc.driverClassName}" /
        <property name="url" value="${db2020.mysql.jdbc.url}" />
        <property name="username" value="${db2020.mysql.jdbc.username}" />
        <property name="password" value="${db2020.mysql.jdbc.password}" />
    </bean>


    <bean id="dynamicDataSourceEntry"  class="javax.core.common.jdbc.datasource.Dynamic

    <bean id="dynamicDataSource" class="javax.core.common.jdbc.datasource.DynamicDataSo
        <property name="dataSourceEntry" ref="dynamicDataSourceEntry"></property>
        <property name="targetDataSources">
            <map>
                <entry key="DB_2020" value-ref="dataSource"></entry>
                <entry key="DB_2019" value-ref="dataSource2019"></entry>
            </map>
        </property>
        <property name="defaultTargetDataSource" ref="dataSource" />
    </bean>

</beans>
```

编写测试用例

```java
package cn.sitedev.orm.test;

import cn.sitedev.orm.demo.dao.MemberDao;
import cn.sitedev.orm.demo.dao.OrderDao;
import cn.sitedev.orm.demo.entity.Member;
import cn.sitedev.orm.demo.entity.Order;
import com.alibaba.fastjson.JSON;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import javax.core.common.Page;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.List;

@ContextConfiguration(locations = {"classpath:application-context.xml"})
@RunWith(SpringJUnit4ClassRunner.class)
public class OrmTest {

    private SimpleDateFormat sdf = new SimpleDateFormat("yyyyMMddHHmmdd");

    @Autowired private MemberDao memberDao;

    @Autowired private OrderDao orderDao;

    //ORM（对象关系映射 Object Relation Mapping）
    //Hibernate/Spring JDBC/MyBatis/JPA 一对多、多对多、一对一

    //Hibernate 全自动档  不需要写一句SQL语句
    //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
    //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

    //为什么有了MyBatis我还要自己的手写ORM框架呢？
    //1、用MyBatis，我可控性无法保证
    //2、我有不敢用Hibernate，高级玩家玩的，
    //3、没有时间自己从0到1写一个ORM框架
    //4、站在巨人的肩膀上再升级，做二次开发

    //约定优于配置
    //1、先制定顶层接口,参数返回值全部统一
    // List<?> Page<?>  select(QueryRule queryRule)
    // 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
    // ReturnId  insert(T entity) 只要entity不等于null
    // 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

    //基于JDBC封装了一套
    //基于Redis封装了一套
    //基于MongoDB
    //基于ElasticSearch
    //基于Hive
    //基于HBase

    //QueryRule

    @Test
    public void testSelectForPage(){
        try {
           Page page = memberDao.selectForPage(2, 3);
            System.out.println("总条数：  " + page.getTotal());
            System.out.println("当前第几页：" + page.getPageNo());
            System.out.println("每页多少条：" + page.getPageSize());
            System.out.println("本页的数据：" + JSON.toJSONString(page.getRows(),true))
        }catch (Exception e){
            e.printStackTrace();
        }
    }



    @Test
    public void testSelectAllForMember(){
        try {
            List<Member> result = memberDao.selectAll();
            System.out.println(JSON.toJSONString(result,true));
//              System.out.println(Arrays.toString(result.toArray()));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    @Test
//    @Ignore
    public void testInsertMember(){
        try {
            for (int age = 25; age < 35; age++) {
                Member member = new Member();
                member.setAge(age);
                member.setName("Tom");
                member.setAddr("Hunan Changsha");
                memberDao.insert(member);
            }
        }catch (Exception e){
            e.printStackTrace();
        }

    }


    @Test
//    @Ignore
    public void testInsertOrder(){
        try {
            Order order = new Order();
            order.setMemberId(1L);
            order.setDetail("历史订单");
            Date date = sdf.parse("20190426123456");
            order.setCreateTime(date.getTime());
            orderDao.insertOne(order);
        }catch (Exception e){
            e.printStackTrace();
        }
    }

}
```