BeetlSQL 2.10中文文档

- 作者: 闲大赋,Gavin.King,Sue,Zhoupan,woate,Darren
- 社区 http://ibeetl.com
- qq群 219324263(满) 636321496

- 当前版本 2.10.45

1. BeetlSQL 特点

BeetSql是一个全功能DAO工具， 同时具有Hibernate 优点 & Mybatis优点功能，适用于承认以SQL为中心，同时又需求工具能自动能生成大量常用的SQL的应用。

- 开发效率
  - 无需注解，自动使用大量内置SQL，轻易完成增删改查功能，节省50%的开发工作量
  - 数据模型支持Pojo，也支持Map/List这种快速模型，也支持混合模型
  - SQL 模板基于Beetl实现，更容易写和调试，以及扩展
  - 可以针对单个表(或者视图）代码生成pojo类和sql模版，甚至是整个数据库。能减少代码编写工作量
- 维护性
  - SQL 以更简洁的方式，Markdown方式集中管理，同时方便程序开发和数据库SQL调试。
  - 可以自动将sql文件映射为dao接口类
  - 灵活直观的支持支持一对一，一对多，多对多关系映射而不引入复杂的OR Mapping概念和技术。
  - 具备Interceptor功能，可以调试，性能诊断SQL，以及扩展其他功能
- 其他
  - 内置支持主从数据库支持的开源工具
  - 性能数倍于JPA，MyBatis
  - 支持跨数据库平台，开发者所需工作减少到最小，目前跨数据库支持mysql,postgres,oracle,sqlserver,h2,sqllite,DB2.

2. 5分钟例子

2.1. 安装

maven 方式:

```
<dependency>
    <groupId>com.ibeetl</groupId>
    <artifactId>beetlsql</artifactId>
    <version>2.10.43</version>
</dependency>
<dependency>
  <groupId>com.ibeetl</groupId>
  <artifactId>beetl</artifactId>
  <version>${最新版本}</version>
</dependency>
```

或者依次下载beetlsql，beetl 最新版本 包放到classpath里

2.2. 准备工作

为了快速尝试BeetlSQL，需要准备一个Mysql数据库或者其他任何beetlsql支持的数据库，然后执行如下sql脚本

```
CREATE TABLE `user` (
      `id` int(11) NOT NULL AUTO_INCREMENT,
      `name` varchar(64) DEFAULT NULL,
      `age` int(4) DEFAULT NULL,
      `userName` varchar(64) DEFAULT NULL COMMENT '用户名称',
      `roleId` int(11) DEFAULT NULL COMMENT '用户角色',
      `create_date` datetime NULL DEFAULT NULL,
      PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

编写一个Pojo类，与数据库表对应（或者可以通过SQLManager的gen方法生成此类，参考一下节）

```java
import java.math.*;
import java.util.Date;

/*
*
* gen by beetlsql 2016-01-06
*/
public class User  {
    private Integer id ;
    private Integer age ;
    //用户角色
    private Integer roleId ;
    private String name ;
    //用户名称
    private String userName ;
    private Date createDate ;

}
```

> 主键需要通过注解来说明，如@AutoID，或者@AssignID等，但如果是自增主键，且属性是名字是id，则不需要注解，自动认为是自增主键

2.3. 代码例子

写一个java的Main方法，内容如下

```
ConnectionSource source = ConnectionSourceHelper.getSimple(driver, url,
userName, password);
DBStyle mysql = new MySqlStyle();
// sql语句放在classpagth的/sql 目录下
SQLLoader loader = new ClasspathLoader("/sql");
// 数据库命名跟java命名一样，所以采用DefaultNameConversion，还有一个是
UnderlinedNameConversion，下划线风格的，
UnderlinedNameConversion nc = new  UnderlinedNameConversion();
// 最后，创建一个SQLManager,DebugInterceptor 不是必须的，但可以通过它查看sql执行情
况
SQLManager sqlManager = new SQLManager(mysql,loader,source,nc,new
Interceptor[]{new DebugInterceptor()});


//使用内置的生成的sql 新增用户，如果需要获取主键，可以传入KeyHolder
User user = new User();
user.setAge(19);
user.setName("xiandafu");
sqlManager.insert(user);

//使用内置sql查询用户
int id = 1;
user = sqlManager.unique(User.class,id);

//模板更新,仅仅根据id更新值不为null的列
User newUser = new User();
newUser.setId(1);
newUser.setAge(20);
sqlManager.updateTemplateById(newUser);

//模板查询
User query = new User();
query.setName("xiandafu");
List<User> list = sqlManager.template(query);

//Query查询
Query userQuery = sqlManager.getQuery(User.class);
List<User> users =
userQuery.lambda().andEq(User::getName,"xiandafy").select();

//使用user.md 文件里的select语句，参考下一节。
User query2 = new User();
query.setName("xiandafu");
List<User> list2 = sqlManager.select("user.select",User.class,query2);

// 这一部分需要参考mapper一章
UserDao dao = sqlManager.getMapper(UserDao.class);
List<User> list3 = dao.select(query2);
```

> BeetlSql2.8.11 提供了 SQLManagerBuilder来链式创建SQLManager

2.4. SQL文件例子

通常一个项目还是有少量复杂sql，可能只有5，6行，也可能有上百行，放在单独的sql文件里更容易编写和维护，为了能执行上例的user.select,需要在classpath里建立一个sql目录（在src目录下建立一个sql目录，或者maven工程的resources目录。ClasspathLoader 配置成sql目录，参考上一节ClasspathLoader初始化的代码）以及下面的user.md 文件，内容如下

```
select
===
select * from user where 1=1
@if(!isEmpty(age)){
and age = #age#
@}
@if(!isEmpty(name)){
and name = #name#
@}
```

关于如何写sql模板，会稍后章节说明，如下是一些简单说明。

- 采用md格式，===上面是sql语句在本文件里的唯一标示，下面则是sql语句。
- @ 和回车符号是定界符号，可以在里面写beetl语句。
- "#" 是占位符号，生成sql语句得时候，将输出？，如果你想输出表达式值，需要用text函数，或者任何以db开头的函数，引擎则认为是直接输出文本。
- isEmpty是beetl的一个函数，用来判断变量是否为空或者是否不存在.
- 文件名约定为类名，首字母小写。

sql模板采用beetl原因是因为beetl 语法类似js，且对模板渲染做了特定优化，相比于mybatis，更加容易掌握和功能强大，可读性更好，也容易在java和数据库之间迁移sql语句

注意：sqlId 到sql文件的映射是通过类SQLIdNameConversion来完成的，默认提供了DefaultSQLIdNameConversion实现，即 以 "." 区分最后一部分是sql片段名字，前面转为为文件相对路径，如sqlId是user.select，则select是sql片段名字，user是文件名，beetlsql会在根目录下寻找/user.sql,/user.md ,也会找数据库方言目录下寻找，比如如果使用了mysql数据库，则优先寻找/mysql/user.md,/mysql/user.sql 然后在找/user.md,/user.sql.

如果sql是 test.user.select,则会在/test/user.md(sql) 或者 /mysql/test/user.md(sql) 下寻找“select”片段

2.5. 代码&sql生成

User类并非需要自己写，好的实践是可以在项目中专门写个类用来辅助生成pojo和sql片段，代码如下

```
public static void main(String[] args){
    SqlManager sqlManager  = ..... //同上面的例子
    sqlManager.genPojoCodeToConsole("user");
    sqlManager.genSQLTemplateToConsole("user");
}
```

> 注意:我经常在我的项目里写一个这样的辅助类，用来根据表或者视图生成各种代码和sql片段，以快速开发.

genPojoCodeToConsole 方法可以根据数据库表生成相应的Pojo代码，输出到控制台，开发者可以根据这些代码创建相应的类，如上例子，控制台将输出

```
package com.test;
import java.math.*;
import java.util.Date;
import java.sql.Timestamp;

/*
*
* gen by beetlsql 2016-01-06
*/
public class User  {
    private Integer id ;
    private Integer age ;
    //用户角色
    private Integer roleId ;
    private String name ;
    //用户名称
    private String userName ;
    private Date createDate ;

}
```

> 注意
>
> 生成属性的时候，id总是在前面，后面依次是类型为Integer的类型，最后面是日期类型，剩下的按照字母排序放到中间。

一旦有了User 类，如果你需要写sql语句，那么genSQLTemplateToConsole 将是个很好的辅助方法，可以输出一系列sql语句片段，你同样可以赋值粘贴到代码或者sql模板文件里（user.md),如上例所述，当调用genSQLTemplateToConsole的时候，生成如下

```
sample
===
* 注释

    select #use("cols")# from user where #use("condition")#

cols
===

    id,name,age,userName,roleId,create_date

updateSample
===


`id`=#id#,`name`=#name#,`age`=#age#,`userName`=#userName#,`roleId`=#roleId#
,`create_date`=#date#

condition
===

    1 = 1
    @if(!isEmpty(name)){
     and `name`=#name#
    @}
    @if(!isEmpty(age)){
     and `age`=#age#
    @}
```

beetlsql生成了用于查询，更新，条件的sql片段和一个简单例子。你可以按照你的需要copy到sql模板文件里.实际上，如果你熟悉gen方法，你可以直接gen代码和sql到你的工程里，甚至是整个数据库都可以调用genAll来一次生成

> 注意
>
> sql 片段的生成顺序按照数据库表定义的顺序显示

3. BeetlSQL 说明

3.1. 获得SQLManager

SQLManager 是系统的核心，他提供了所有的dao方法。获得SQLManager，可以直接构造SQLManager.并通过单例获取如：

```
ConnectionSource source = ConnectionSourceHelper.getSimple(driver, url, "",
userName, password);
DBStyle mysql = new MySqlStyle();
// sql语句放在classpagth的/sql 目录下
SQLLoader loader = new ClasspathLoader("/sql");
// 数据库命名跟java命名一样，所以采用DefaultNameConversion，还有一个是
UnderlinedNameConversion，下划线风格的
UnderlinedNameConversion nc = new  UnderlinedNameConversion();
// 最后，创建一个SQLManager,DebugInterceptor 不是必须的，但可以通过它查看sql执行情
况
SQLManager sqlManager = new SQLManager(mysql,loader,source,nc,new
Interceptor[]{new DebugInterceptor()});
```

更常见的是，已经有了DataSource，创建ConnectionSource 可以采用如下代码

```
ConnectionSource source = ConnectionSourceHelper.getSingle(datasource);
```

如果是主从Datasource

```
ConnectionSource source =
ConnectionSourceHelper.getMasterSlave(master,slaves)
```

关于使用Sharding-JDBC实现分库分表，参考主从一章

3.2. 查询API

3.2.1. 简单查询（自动生成sql）

- public T unique(Class clazz,Object pk) 根据主键查询，如果未找到，抛出异常.
- public T single(Class clazz,Object pk) 根据主键查询，如果未找到，返回null.

- public List all(Class clazz) 查询出所有结果集
- public List all(Class clazz, int start, int size) 翻页
- public int allCount(Class<?> clazz) 总数

3.2.2 （Query）单表查询

SQLManager提供Query类可以实现单表查询操作

```
SQLManager sql = ...
List<User> list =
sql.query(User.class).andEq("name","hi").orderBy("create_date").select();
```

sql.query(User.class) 返回了Query类用于单表查询

如果是Java8，则可以使用lambda表示列名

```
List<User> list1  = sql.lambdaQuery(User.class).andEq(User::getName,
"hi").orderBy(User::getCreateDate).select();
```

lamdba()方法返回了一个LamdbaQuery 类，列名支持采用lambda。

> 关于Query操作的具体用法，请参考25.1节
>
> Query对象通常适合在业务操作中使用，而不能代替通常的前端界面查询，前端界面查询推荐使用sqlId来查询

3.2.3 template查询

- public List template(T t) 根据模板查询，返回所有符合这个模板的数据库 同上，mapper可以提供额外的映射，如处理一对多，一对一
- public T templateOne(T t) 根据模板查询，返回一条结果，如果没有找到，返回null
- public List template(T t,int start,int size) 同上，可以翻页
- public long templateCount(T t) 获取符合条件的个数
- public List template(Class target,Object paras,long start, long size) 模板查询，参数是paras，可以是Map或者普通对象
- public long templateCount(Class target, Object paras) 获取符合条件个数

翻页的start，系统默认位从1开始，为了兼容各个数据库系统，会自动翻译成数据库习俗，比如start为1，会认为mysql，postgres从0开始（从start－1开始），oralce，sqlserver，db2从1开始（start－0）开始。

然而，如果你只用特定数据库，可以按照特定数据库习俗来，比如，你只用mysql，start为0代表起始纪录，需要配置

```
OFFSET_START_ZERO = true
```

这样，翻页参数start传入0即可。

注意:template查询方法根据模板查询并不包含时间字段，也不包含排序，然而，可以通过在pojo class上使用@TableTemplate() 或者日期字段的getter方法上使用@DateTemplate()来定制，如下:

```
@TableTemplate("order by id desc ")
public class User  {
    private Integer id ;
    private Integer age ;
    // ...
    @DateTemplate(accept="minDate,maxDate")
    public Date getDate() {
        return date;
    }
}
```

这样，模板查询将添加order by id desc ,以及date字段将按照日期范围来查询。 具体参考annotation一章

模板查询一般时间较为简单的查询，如用户登录验证

```
User template = new User();
template.setName(...);
template.setPassword(...);
template.setStatus(1);
User user = sqlManager.templateOne(template);
```

3.2.4. 通过sqlid查询,sql语句在md文件里

- public List select(String sqlId, Class clazz, Map<String, Object> paras) 根据sqlid来查询，参数是个map
- public List select(String sqlId, Class clazz, Object paras) 根据sqlid来查询，参数是个pojo
- public List select(String sqlId, Class clazz) 根据sqlid来查询，无参数
- public T selectSingle(String id,Object paras, Class target) 根据sqlid查询，输入是Pojo，将对应的唯一值映射成指定的target对象，如果未找到，则返回空。需要注意的时候，有时候结果集本身是空，这时候建议使用unique
- public T selectSingle(String id,Map<String, Object> paras, Class target) 根据sqlid查询，输入是Map，将对应的唯一值映射成指定的target对象，如果未找到，则返回空。需要注意的时候，有时候结果集本身是空，这时候建议使用unique
- public T selectUnique(String id,Object paras, Class target) 根据sqlid查询，输入是Pojo或者Map，将对应的唯一值映射成指定的target对象,如果未找到，则抛出异常
- public T selectUnique(String id,Map<String, Object> paras, Class target) 根据sqlid查询，输入是Pojo或者Map，将对应的唯一值映射成指定的target对象,如果未找到，则抛出异常
- public Integer intValue(String id,Object paras) 查询结果映射成Integer，如果找不到，返回null，输入是object
- public Integer intValue(String id,Map paras) 查询结果映射成Integer，如果找不到，返回null，输入是map，其他还有 longValue，bigDecimalValue

> 注意，对于Map参数来说，有一个特殊的key叫着_root,它代表了查询根对象，sql语句中未能找到的变量都会在试图从_root 中查找，关于_root对象，可以参考第8章。 在Map中使用_root, 可以混合为sql提供参数

3.2.5 指定范围查询

- public List select(String sqlId, Class clazz, Map<String, Object> paras, int start, int size)， 查询指定范围
- public List select(String sqlId, Class clazz, Object paras, int start, int size) ，查询指定范围

> beetlsql 默认从1 开始，自动翻译为目标数据库的的起始行，如mysql的0，oracle的1
>
> 如果你想从0开始，参考11章，配置beetlsql

3.3 翻页查询API

```
public <T> void pageQuery(String sqlId,Class<T> clazz,PageQuery query)
```

BeetlSQL 提供一个PageQuery对象,用于web应用的翻页查询,BeetlSql假定有sqlId 和sqlId$count,俩个sqlId,并用这来个来翻页和查询结果总数.如:

```
queryNewUser
===
select * from user order by id desc ;

queryNewUser$count
===
select count(1) from user
```

对于俩个相似的sql语句,你可以使用use函数,把公共部分提炼出来.

大部分情况下,都不需要2个sql来完成,一个sql也可以,要求使用page函数或者pageTag标签,这样才能同时获得查询结果集总数和当前查询的结果

```
queryNewUser
===
select
@pageTag(){
a.*,b.name role_name
@}
from user a left join b ...
```

如上sql,会在pageQuery查询的时候转为俩条sql语句

```
select count(1) from user a left join b...
select a.*,b.name role_name  from user a left join b...
```

如果字段较多,为了输出方便,也可以使用pageTag,字段较少,用page函数也可以. ,具体参考pageTag和page函数说明.翻页代码如下

```
//从第一页开始查询,无参数
PageQuery query = new PageQuery();
sql.pageQuery("user.queryNewUser", User.class,query);
System.out.println(query.getTotalPage());
System.out.println(query.getTotalRow());
System.out.println(query.getPageNumber());
List<User> list = query.getList();
```

PageQuery 对象也提供了 orderBy属性，用于数据库排序，如 "id desc"

> ### 跨数据库支持
>
> 如果你打算使用PageQuery做翻页,且只想提供一个sql语句+page函数,那考虑到跨数据库,应该不要在这个sql语句里包含排序,因为大部分数据库都不支持. page函数生成的查询总数sql语句,因为包含了oder by,在大部分数据库都是会报错的的,比如:select count(1) form user order by name,在sqlserver,mysql,postgresql都会出错,oracle允许这种情况, 因此,如果你要使用一条sql

> 语句+page函数,建议排序用PageQuery对象里有排序属性oderBy,可用于排序,而不是放在sql语句里.
>
> 2.8版本以后也提供了标签函数 pageIgnoreTag，可以用在翻页查询里，当查询用作统计总数的时候，会忽略标签体内容，如
>
> ```
> select page("*") from xxx
> @pageIgnoreTag(){
>  order by id
> @}
> ```
>
> 如上语句，在求总数的时候，会翻译成 select count(1) from xxx
>
> 如果你不打算使用PageQuery+一条sql的方式,而是用两条sql来分别翻页查询和统计总数,那无所谓
>
> 或者你直接使用select 带有起始和读取总数的接口,也没有关系,可以在sql语句里包含排序

如果PageQuery对象的totalRow属性大于等于0，则表示已经知道总数，则不会在进行求总数查询

3.4. 更新API

添加，删除和更新均使用下面的API

3.4.1. 自动生成sql

- public void insert(Object paras) 插入paras到paras关联的表
- public void insert(Object paras,boolean autoAssignKey) 插入paras到paras对象关联的表,并且指定是否自动将数据库主键赋值到paras里,适用于对于自增或者序列类数据库产生的主健
- public void insertTemplate(Object paras) 插入paras到paras关联的表,忽略为null值或者为空值的属性
- public void insertTemplate(Object paras,boolean autoAssignKey) 插入paras到paras对象关联的表,并且指定是否自动将数据库主键赋值到paras里,忽略为null值或者为空值的属性，调用此方法，对应的数据库必须主键自增。
- public void insert(Class<?> clazz,Object paras) 插入paras到clazz关联的表
- public void insert(Class<?> clazz,Object paras,KeyHolder holder)，插入paras到clazz关联的表，如果需要主键，可以通过holder的getKey来获取，调用此方法，对应的数据库必须主键自增
- public int insert(Class clazz,Object paras,boolean autoAssignKey) 插入paras到clazz关联的表，并且指定是否自动将数据库主键赋值到paras里，调用此方法，对应的数据库必须主键自增。
- public int updateById(Object obj) 根据主键更新，所有值参与更新
- public int updateTemplateById(Object obj) 根据主键更新，属性为null的不会更新
- public int updateBatchTemplateById(Class clazz,List<?> list) 批量根据主键更新,属性为null的不会更新
- public int updateTemplateById(Class<?> clazz，Map paras) 根据主键更新，组件通过clazz的annotation表示，如果没有，则认为属性id是主键,属性为null的不会更新。
- public int[] updateByIdBatch(List<?> list) 批量更新
- public void insertBatch(Class clazz,List<?> list) 批量插入数据

- public int upsert(Object obj), 更新或者插入一条。先判断是否主键为空，如果为空，则插入，如果不为空，则从数据库 按照此主健取出一条，如果未取到，则插入一条，其他情况按照主键更新。插入后的自增或者序列主健
- int upsertByTemplate(Object obj) 同上，按照模板插入或者更新。

3.4.2. 通过sqlid更新（删除）

- public int insert(String sqlId,Object paras,KeyHolder holder) 根据sqlId 插入，并返回主键，主键id由paras对象所指定，调用此方法，对应的数据库表必须主键自增。
- public int insert(String sqlId,Object paras,KeyHolder holder,String keyName) 同上，主键由keyName指定
- public int insert(String sqlId,Map paras,KeyHolder holder,String keyName)，同上，参数通过map提供
- public int update(String sqlId, Object obj) 根据sqlid更新
- public int update(String sqlId, Map<String, Object> paras) 根据sqlid更新，输出参数是map
- public int[] updateBatch(String sqlId,List<?> list) 批量更新
- public int[] updateBatch(String sqlId,Map<String, Object>[] maps) 批量更新，参数是个数组，元素类型是map

3.5. 直接执行SQL模板

3.5.1. 直接执行sql模板语句

一下接口sql变量是sql模板

- public List execute(String sql,Class clazz, Object paras)
- public List execute(String sql,Class clazz, Map paras)
- public int executeUpdate(String sql,Object paras) 返回成功执行条数
- public int executeUpdate(String sql,Map paras) 返回成功执行条数

3.5.2. 直接执行JDBC sql语句

- 查询 public List execute(SQLReady p,Class clazz) SQLReady包含了需要执行的sql语句和参数，clazz是查询结果，如

```
List<User> list = sqlManager.execute(new SQLReady("select * from user where name=? and age = ?","xiandafu",18),User.class);)
```

- public PageQuery execute(SQLReady p, Class clazz, PageQuery pageQuery)

```
String  jdbcSql = "  select *from user order by id";
PageQuery query = new PageQuery(1,20);
query = sql.execute(new SQLReady(jdbcSql), User.class, query);
```

注意:sql参数通过SQLReady 传递，而不是PageQuery。

- 更新 public int executeUpdate(SQLReady p) SQLReady包含了需要执行的sql语句和参数，返回更新结果

- 直接使用Connection public T executeOnConnection(OnConnection call),使用者需要实现onConnection方法的call方法，如调用存储过程

```
List<User> users = sql.executeOnConnection(new OnConnection<List<User>>(){
    @Override
    public List<User> call(Connection conn) throws SQLException {
         CallableStatement cstmt = conn.prepareCall("{ ? = call md5( ? )
}");
         ResultSet rs = callableStatement.executeQuery();
         return
this.sqlManagaer.getDefaultBeanProcessors().toBeanList(rs,User.class);


    }
});
```

3.6. 其他

3.6.1. 强制使用主或者从

如果为SQLManager提供多个数据源，默认第一个为主库，其他为从库，更新语句将使用主库，查询语句使用从库库

可以强制SQLManager 使用主或者从

- public void useMaster(DBRunner f) DBRunner里的beetlsql调用将使用主数据库库
- public void useSlave(DBRunner f) DBRunner里的beetlsql调用将使用从数据库库

> 对于通常事务来说只读事务则从库，写操作事务则总是主库。关于主从支持，参考17章

3.6.2. 生成Pojo代码和SQ片段

用于开发阶段根据表名来生成pojo代码和相应的sql文件

- genPojoCodeToConsole(String table), 根据表名生成pojo类，输出到控制台.
- genSQLTemplateToConsole(String table),生成查询，条件，更新sql模板，输出到控制台。
- genPojoCode(String table,String pkg,String srcPath,GenConfig config) 根据表名，包名，生成路径，还有配置，生成pojo代码
- genPojoCode(String table,String pkg,GenConfig config) 同上，生成路径自动是项目src路径，或者src/main/java (如果是maven工程)
- genPojoCode(String table,String pkg),同上，采用默认的生成配置
- genSQLFile(String table), 同上，但输出到工程，成为一个sql模版,sql模版文件的位置在src目录下，或者src／main／resources（如果是maven）工程.
- genALL(String pkg,GenConfig config,GenFilter filter) 生成所有的pojo代码和sql模版，
- genBuiltInSqlToConsole(Class z) 根据类来生成内置的增删改查sql语句，并打印到控制台

```
sql.genAll("com.test", new GenConfig(), new GenFilter(){
    public boolean accept(String tableName){
        if(tableName.equalsIgnoreCase("user")){
            return true;
        }else{
            return false;
        }
        // return false
    }
});
```

第一个参数是pojo类包名，GenConfig是生成pojo的配置，GenFilter 是过滤，返回true的才会生成。如果GenFilter为null，则数据库所有表都要生成

> 警告
>
> 必须当心覆盖你掉你原来写好的类和方法，不要轻易使用genAll，如果你用了，最好立刻将其注释掉，或者在genFilter写一些逻辑保证不会生成所有的代码好sql模板文件

3.6.3. 悲观锁 lock

SQLManager 提供如下API实现悲观锁，clazz对应的数据库表，主键为pk的记录实现悲观锁

```
public <T> T lock(Class<T> clazz, Object pk)
```

相当于sql语句

```
select * from xxx where id = ? for update
```

lock 方法必须用在事务环境里才能生效。事务结束后，自动释放

> 乐观锁实现请参考@Version 注解

4. 命名转化，表和列名映射

Beetlsql 默认提供了三种列明和属性名的映射类，推荐使用UnderlinedNameConversion

- DefaultNameConversion 数据库名和java属性名保持一致，如数据库表User，对应Java类也是User，数据库列是sysytemId,则java属性也是systemId，反之亦然
- UnderlinedNameConversion 将数据库下划线去掉，首字母大写，如数据库是SYS_USER（oralce数据库的表和属性总是大写的), 则会改成SysUser
- JPA2NameConversion 支持JPA方式的映射，适合不能用确定的映射关系(2.7.4以前采用JPANameConversion过于简单，已经不用了)
- 自定义命名转化，如果以上3个都不合适,可以自己实现一个命名转化。实现DefaultNameConversion实现方式
- 因为数据库表和列都忽略大小写区别，所以，实现NameConversion也不需要考虑大小写

一般来讲，都建议数据库以下划线来区分单词，因此，使用UnderlinedNameConversion是个很好的选择

```
    public class DefaultNameConversion extends NameConversion {
    @Override
    public String getTableName(Class<?> c) {
        Table table = (Table)c.getAnnotation(Table.class);
        if(table!=null){
            return table.name();
        }
        return c.getSimpleName();
    }

    @Override
    public String getColName(Class<?> c, String attrName) {
        return attrName;
    }

    @Override
    public String getPropertyName(Class<?> c, String colName) {
        return colName;
    }

}
```

如果有特殊的表并不符合DefaultNameConversion实现方式，你可以重新实现上面的三个方法

在使用org.beetl.sql.core.JPA2NameConversion作为命令转化规则时，你可以使用以下JPA标签来帮助解析实体类到数据库的转换:

- javax.persistence.Table
- javax.persistence.Column
- javax.persistence.Transient

规则如下： 1 在类名前使用Table注解映射的表名，例如：@Table(name = "PF_TEST")，表示映射的表名是PF_TEST； 2 忽略静态变量以及被@Transient注解的属性； 3 默认属性名与库表的字段名保持一致，如果不一致时，可以使用@Column注解。

```
@Table(name = "PF_TEST")
public class TestEntity implements Serializable {
    public static String S="SSS";
    private String id;
    @Column(name = "login_name")
    private String loginName;
    private String password;
    private Integer age;
    private Long ttSize;
    private byte[] bigger;
    private String biggerClob;
    @Transient
    private String biggerStr;
    @AssignID
    public String getId() {
        return id;
    }
    getter setter...
}
```

5. 复合主键

beetlsql 支持复合主键，无需像其他dao工具那样创建一个特别的主键对象，主键对象就是实体对象本身

```
CREATE TABLE `party` (
  `id1` int(11) NOT NULL,
  `id2` int(11) NOT NULL,
  `name` varchar(45) DEFAULT NULL,
  PRIMARY KEY (`id1`,`id2`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

Party代码如下

```
public class Party  {
    @AssignID
    private Integer id1 ;
    @AssignID
    private Integer id2 ;
    private String name ;
    //忽略其他 getter setter方法
}
```

根据主键获取Party

```
Party key = new Party();
key.setId1(1);
key.setId2(2);
Party party = sql.unique(Party.class, key);
```

6. 使用Mapper

SQLManager 提供了所有需要知道的API，但通过sqlid来访问sql有时候还是很麻烦，因为需要手敲字符串，另外参数不是map就是para，对代码理解没有好处，BeetlSql支持Mapper，将sql文件映射到一个interface接口。接口的方法名与sql文件desqlId一一对应。

接口必须实现BaseMapper接口（后面可以自定义一个Base接口），它提供内置的CRUID方法，如insert,unique,template,templateOne ,updateById等

BaseMapper 具备数据库常见的操作，接口只需要定义额外的方法与sqlId同名即可。

```
public interface UserDao extends BaseMapper<User> {
    List<User> select(String name);
}
```

如上select将会对应如下md文件

```
select
===

    select * from user where name = #name#
```

如果你使用JDK8，不必为参数提供名称，自动对应。但必须保证java编译的时候开启-parameters选项。如果使用JDK8以下的版本，则可以使用@Param注解()

```
List<User> select(@Param("name") String name);
```

BeetlSql的mapper方法总会根据调用方法名字，返回值，以及参数映射到SQLManager相应的查询接口，比如返回类型是List，意味着发起SQLManager.select 查询，如果返回是一个Map或者Pojo，则发起一次selectSingle查询，如果返回定义为List，则表示查询实体，如果定义为List ，则对应的查询结果映射为Long

定义好接口后，可以通过SQLManager.getMapper 来获取一个Dao真正的实现

```
UserDao dao = sqlManager.getMapper(UserDao.class);
```

如果你使用Spring或者SpringBoot，可以参考Spring集成一章，了解如何自动注入Mapper

> Mapper 对应的sql文件默认根据实体来确定，如实体是UserInfo对象，则对应的sql文件是userInfo.md(sql),可以通过@SqlResource 注解来指定Mapper对应的sql文件。比如

```
@SqlResource("core.user")
public interface UserCoreDao extends BaseMapper<User> {
    List<User> select(String name);
}
@SqlResource("console.user")
public interface UserConsoleDao extends BaseMapper<User> {
    List<User> select(String name);
}
```

> 这样，这俩个mapper分别访问sql/core/user.md 和 sql/console/user.md

6.1. 内置CRUD

BaseMapper包含了内置的常用查询，如下

```
public interface BaseMapper<T> {

    /**
     * 通用插入，插入一个实体对象到数据库，所以字段将参与操作，除非你使用ColumnIgnore注解
     * @param entity
     */
    void insert(T entity);
    /**
     * （数据库表有自增主键调用此方法）如果实体对应的有自增主键，插入一个实体到数据库，设置assignKey为true的时候，将会获取此主键
     * @param entity
     * @param autDbAssignKey 是否获取自增主键
     */
    void insert(T entity,boolean autDbAssignKey);
    /**
     * 插入实体到数据库，对于null值不做处理
     * @param entity
     */
    void insertTemplate(T entity);
    /**
     * 如果实体对应的有自增主键，插入实体到数据库，对于null值不做处理,设置assignKey为true的时候，将会获取此主键
     * @param entity
     * @param autDbAssignKey
     */
    void insertTemplate(T entity,boolean autDbAssignKey);
    /**
     * 批量插入实体。此方法不会获取自增主键的值，如果需要，建议不适用批量插入，适用
     * <pre>
     * insert(T entity,true);
```

```
     * </pre>
     * @param list
     */
    void insertBatch(List<T> list);
    /**
     * （数据库表有自增主键调用此方法）如果实体对应的有自增主键，插入实体到数据库，自增主键值放到keyHolder里处理
     * @param entity
     * @return
     */
    KeyHolder insertReturnKey(T entity);

    /**
     * 根据主键更新对象，所以属性都参与更新。也可以使用主键ColumnIgnore来控制更新的时候忽略此字段
     * @param entity
     * @return
     */
    int updateById(T entity);
    /**
     * 根据主键更新对象，只有不为null的属性参与更新
     * @param entity
     * @return
     */
    int updateTemplateById(T entity);

    /**
     * 根据主键删除对象，如果对象是复合主键，传入对象本生即可
     * @param key
     * @return
     */
    int deleteById(Object key);

    /**
     * 根据主键获取对象，如果对象不存在，则会抛出一个Runtime异常
     * @param key
     * @return
     */
    T unique(Object key);
    /**
     * 根据主键获取对象，如果对象不存在，返回null
     * @param key
     * @return
     */
    T single(Object key);


    /**
```

```
     * 根据主键获取对象，如果在事物中执行会添加数据库行级锁(select * from table where id = ? for update)，如果对象不存在，返回null
     * @param key
     * @return
     */
    T lock(Object key);

    /**
     * 返回实体对应的所有数据库记录
     * @return
     */
    List<T> all();
    /**
     * 返回实体对应的一个范围的记录
     * @param start
     * @param size
     * @return
     */
    List<T> all(int start,int size);
    /**
     * 返回实体在数据库里的总数
     * @return
     */
    long allCount();

    /**
     * 模板查询，返回符合模板得所有结果。beetlsql将取出非null值（日期类型排除在外），从数据库找出完全匹配的结果集
     * @param entity
     * @return
     */
    List<T> template(T entity);


    /**
     * 模板查询，返回一条结果,如果没有，返回null
     * @param entity
     * @return
     */
    <T> T templateOne(T entity);

    List<T> template(T entity,int start,int size);
    /**
     * 符合模板得个数
     * @param entity
     * @return
     */
    long templateCount(T entity);
```

```
    /**
     * 单表的基于模板查询的翻页
     * @param query
     * @return
     */
    void templatePage(PageQuery<T> query);

    /**
     * 执行一个jdbc sql模板查询
     * @param sql
     * @param args
     * @return
     */
    List<T> execute(String sql,Object... args);
    /**
     * 执行一个更新的jdbc sql
     * @param sql
     * @param args
     * @return
     */
    int executeUpdate(String sql,Object... args );
    SQLManager getSQLManager();
    /**
     * 返回一个Query对象
     * @return
     */
    Query<T> createQuery();

     /**
     * 返回一个LambdaQuery对象
     * @return
     */
    LambdaQuery<T> createLambdaQuery();

}
```

> 内置BaseMapper 可以定制，参考文档最后一节25.6，设置自己的BaseMapper

6.2. sqlId查询

对于sqlId 是查询语句，返回值可以是任何类型，Mapper将视图将查询结果映射到定义的类型上，如下是一些常见例子

```
public interface UserDao extends BaseMapper<User> {
  // 使用"user.getCount"语句,无参数
  public int getCount();
  //使用"user.findById"语句，参数是id，返回User对象
  public User findById(@Param("id") Integer id);
  //使用"user.findById"语句，参数是id，返回User对象
  public List<User> findByName(@Param("name") String name);
  //使用user.findTop10OfIds语句, 查询结果映射为Long，比如“select id from user
limit 10
  public List<Long> findTop10OfIds();
  //返回一个Map，不建议这么做，最好返回一个实体，或者实体+Map的混合模型(参考BeetlSql模
型)
  public List<Map<String,Object> findUsers(@Param("name") String
name,@Param("departmentId") departmentId)

}
```

> 对于jdk8以上的，不必使用@Param注解。但必须保证java编译代码的时候开启-parameters选项

Mapper 查询有一个例外，如果第一个参数是一个JavaBean(即非java内置对象)，则默认为是_root对象，因此如下俩个接口定义是等价的

```
public List query(User template);
public List query2(@Param("_root") User template)
```

这俩个查询方法都将template视为一个_root对象，sql语句可以直接使用引用其属性。同样第一参数类型是Map的，BeetlSql也视为_root对象。

如果需要查询指定范围内的结果集，可以使用@RowStart,@RowSize , 将指示Mapper发起一次范围查询(参考3.2.3)

```
pulic List<User> selectRange(User data,Date maxTime,@RowStart int
start, @RowSize int size)
```

如上查询语句，类似这样调用了SQLManager

```
Map paras = new HashMap();
paras.put("_root",data);
paras.put("maxTime",maxTime);
List<User> list =
sqlManager.select("user.selectRanage",User.class,paras,start,size);
```

6.3. PageQuery 查询

PageQuery查询类似上一节的sqlId查询，不同的是，需要提供PageQuery参数以让Mapper理解为PageQuery查询，如下俩个是等价的

```
public void queryByCondtion(PageQuery query);
public PageQuery queryByCondtion(PageQuery query);
```

可以添加额外参数，但PageQuery 必须是第一个参数，如

```
public void queryByCondtion(PageQuery query,Date maxTime);
```

这类似如下SQLManager调用

```
query.setPara("maxTime",maxTime);
sqlManager.pageQuery("user.queryByCondtion",User.class,query)
```

也可以在方法中提供翻页参数来实现翻页查询，这时候返回值必须是PageQuery，如下

```
public PageQuery queryByCondtion(int pageNumber,int pageSize,String name);
```

这种情况下，前俩个参数必须是int或者long类型

6.4. 更新语句

更新语句返回的结果可以是void，或者int，如果是批量更新，则可以返回int[]

```
public int updaetUser(int id,String name);
public int updateUser(User user);
public int[] updateAll(List<User> list);
```

BeetlSql 通常根据返回值是int或者int[] 来判断是否是更新还是批量更新，如果没有返回值，会进一步判断第一个参数是否是集合或者数组，比如

```
public void updateAllByUser(List<User> list);
public void updateAllByIds(List<Integer> list);
```

Beetl会假设前者是批量更新，而后者只是普通更新。建议还是不要使用void，而使用int或者int[]来帮助区分

6.5. 插入语句

插入语句同更新语句，唯一不同的是插入语句有时候需要获取自增序列值，这时候使用KeyHolder作为返回参数

```
public KeyHolder insertSql(User user);
```

6.6. 使用JDBC SQL

可以通过@Sql注解直接在java中使用较为简单的sql语句，如下

```
@Sql(value=" update user set age = ? where id = ? ")
public void updateAge(int age,int id);
@Sql(value="select id from user where create_time<?")
public List<Long> selectIds(Date date)
```

此时方法参数与"?" 一一对应 也可以使用@Sql翻页，这要求方法参数前俩个必须是int或者long,返回结果使用PageQuery定义

```
@Sql(value="select * from user where create_time<?")
public PageQuery selectUser(int pageNumber,int pageSize,Date date)
```

6.7. Mapper中的注解

从上面我们已经了解了@Param注解，用于申明参数名字，如果使用jdk8，且打开了编译选项parameters，则可以去掉@Param注解 @RowStart和 @RowSize，用于查询中的范围查询。 @Sql注解则用于在java中构造一个简短的jdbc sql语句。

@SqlStatment 注解可以对接口参数进一步说明，他有如下属性

- type,用于说明sqlId是何种类型的语句，默认为auto，BeetlSql将会根据sqlId对应的Sql语句判断是否是查询，还是修改语句等，通常是根据sql语句的第一个词来判断，如果是select，表示查询，如果是insert，表示新增，如果update，drop，则是更新。如果Sql模板语句第一个词不包含这些，则需要用type做说明。如下是一个需要用到type的情况

```
selectUsers
===
use("otherSelect") and status=1;
```

因为beetlsql无法根据第一个单词确定操作类型，因此必须使用type=SqlStatementType.SELECT，来说明。

- params ,可以不用在接口参数上使用@Param，而直接使用params 属性，如下是等价的

```
@SqlStatement(params="name,age,_st,_sz")
public List<User> queryUser( String name, Integer age,int start, int size);

public List<User> queryUser( @Param(name) String name, @Param(age)
@RowStart Integer age,int start, @RowSize int size);
```

_st,_sz 同@RowStart和@RowSize

6.8 使用接口默认方法

如果你使用JDK8，则可以在mapper中添加默认方法，有利于重用

```
public interface SysResourceDao extends BaseMapper<SysResource> {

    void page(PageQuery<SysResource> query);

    default List<SysResource> listChildren(Integer resourceId) {
        return createLambdaQuery()
                .andNotEq(SysResource::getStatus, 1)
                .andEq(SysResource::getPid, resourceId)
                .select();
    }
}
```

7. BeetlSQL Annotation

对于自动生成的sql，默认不需要任何annotaton，类名对应于表名（通过NameConversion类），getter方法的属性名对应于列明（也是通过NameConversion类），但有些情况还是需要annotation。

7.1. @AutoID 和 @AssignID ， @SeqID

- @AutoID,作用于属性字段或者getter方法，告诉beetlsql，这是自增主键,对应于数据自增长
- @AssignID，作用于属性字段或者getter方法，告诉beetlsql，这是程序设定

```
@AssignID()
public Long getId() {
    return id;
}
```

代码设定主键允许像@AssignID 传入id的生成策略以自动生成序列，beetl默认提供了一个snowflake算法，一个用于分布式环境的id生成器(https://github.com/twitter/snowflake)

```
@AssignID("simple")
public Long getId() {
    return id;
}
```

simple 是beetlsql提供的一个默认的snowflake实现，你可以通过sqlManager自己注册id生成器

```
sqlManager.addIdAutonGen("uuid2", new IDAutoGen(){
    @Override
    public Object nextID(String params) {
        return "hi"+new Random().nextInt(10000);
    }
});
```

```
@AssignID("uuid2")
public Long getId() {
    return id;
}
```

- @SeqID(name="xx_seq"，作用于getter方法，告诉beetlsql，这是序列主键。

对于属性名为id的自增主键，不需要加annotation，beetlsql默认就是@AutoID

> 备注
>
> - 对于支持多种数据库的，这些annotation可以叠加在一起

7.2. @Tail

@Tail作用于类上，表示该对象是混合模型，参考下一章混合模型,sql查询无法在pojo映射的列或者结果集将使用Tail指定的方法

7.3. 忽略属性

BeetlSql提供InsertIgnore,UpdateIgnore俩个注解,作用于属性字段或者getter方法，前者用于内置插入的时候忽略，后者用于内置更新的时候忽略。

```
@UpdateIgnore
public Date getBir(){
    return  bir;
}
```

在beetlsql较早版本提供了ColumnIgnore， 提供insert或者update属性用来忽略

```
@ColumnIgnore(insert=true,update=false)
public Date getBir(){
    return  bir;
}
```

如上例子，插入的时候忽略bir属性（往往是因为数据库指定了默认值为当前时间），更新的时候不能忽略 @ColumnIgnore的insert默认是true，update是false，因此也可以直接用 @ColumnIgnore()

7.4. @EnumMapping

对于Entity使用了枚举作为属性值，可以再枚举类上定义EnumMapping，指出如何将枚举与数据库值互相转化，有四种方法

- 如果没有使用@EnumMapping，则使用枚举的名称作为属性

- @EnumMapping(EnumMapping.EnumType.STRING) 同上，使用枚举名称作为属性，数据库对应列应该是字符列
- @EnumMapping(EnumMapping.EnumType.ORDINAL) 使用枚举的顺序作为属性，数据库对应列应该是int类型，用此作为映射需要防止重构枚举的时候导致数据库也重构，应该慎用
- @EnumMapping("xxx")，如果不是上面的的定义，则beetlsql会查找枚举类的xxx属性，用这个值作为属性，比如

```
@EnumMapping("value")
public enum Color {
    RED("RED",1),BLUE ("BLUE",2);
    private String name;
    private int value;
    private Color(String name, int value) {
        this.name = name;
        this.value = value;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getValue() {
        return value;
    }
    public void setValue(int value) {
        this.value = value;
    }

}
```

beetlsq 会获取枚举的value属性（调用getValue)来获取枚举属性值

7.5. @Table

标签 @Table(name="xxxx") 告诉beetlsql，此类对应xxxx表。比如数据库有User表，User类对应于User表，也可以创建一个UserQuery对象，也对应于User表

```
@Table(name="user")
public class QueryUser ..
```

注：可以为对象指定一个数据库shcema，如name="cms.user",此时将访问cms库（或者cms用户，对不同的数据库，称谓不一样）下的user数据表

> 考虑到跨数据库，最好采用大写方式，比如USER,CMS.USER，oracle 识别大写

7.6. @TableTemplate

- @TableTemplate() 用于模板查询，如果没有任何值，将按照主键降序排，也就是order by 主键名称 desc
- @DateTemplate()，作用于日期字段的属性字段或者getter方法，有俩个属性accept 和 compare 方法，分别表示 模板查询中，日期字段如果不为空，所在的日期范围，如

```
@DateTemplate(accept="minDate,maxDate",compare=">=,<")
    public Date getDate() {
}
```

在模板查询的时候，将会翻译成

```
@if(!isEmpty(minDate)){
 and date>=#minDate#
@}
@if(!isEmpty(maxDate)){
 and date<#maxDate#
@}
```

> 注意
>
> minDate,maxDate 是俩个额外的变量,需要定义到pojo类里，DateTemplate也可以有默认值，如果@DateTemplate()，相当于@DateTemplate(accept="min日期字段,max日期字段",compare=">=,<")

7.7. Mapper相关注解

Mapper 是将sql模板文件映射成一个具体的Dao方法类,这样方便代码开发和维护

Mapper中的注解，包括常用的 SqlStatement ， SqlStatementType ， Sql,Param 还有不常用的 RowSize ， RowStart，具体参考Mapper

7.8. ORMQuery

beetlsql 支持在实体类上增加ORMQuery注解,这样对于实体的查询,会触发懒加载,从而实现ORM 查询功能,具体参考ORM 查询一章

7.9. @Version

注解@Version作用在类型为int,long的属性或者getter方法上，用于乐观锁实现。

```
public class Credit   implements Serializable{
    private Integer id ;
    private Integer balance ;
    @Version
    private Integer version ;
```

当调用内置的updateById，或者updateTemlateById的时候，被@Version注解的字段将作为where条件的一部分

```
┌──────── Debug [credit._gen_updateTemplateById] ─────
├ SQL:    update `credit` set `balance`=?, `version`=`version`+1 where
`id` = ? and `version` = ?
├ 参数:     [15, 1, 5]
├ 位置:     org.beetl.sql.test.QuickTest.main(QuickTest.java:38)
├ 时间:     4ms
├ 更新:     [1]
└──────── Debug [credit._gen_updateTemplateById] ─────
```

> BeetlSQL 也支持悲观锁实现，即采用select for update 方式，只要调用SQLManager.lock(Class cls,Object key)就可以对cls对应的的表的主键为key的记录使用行锁。只有事务结束后，才释放此锁

7.10 @SqlResource

用在Mapper接口上，说明MD文件的位置，可以通过此注解指定一个在根目录下的某一个子目录位置。

```
@SqlResource("platform.sysDict")
public interface SysDictDao extends BaseMapper<SysDict> {
   public List<SysDict> findAllList(@Param(value = "type") String type);
}
```

如上findAllList方法对应的sql，将位于resources/sql/platform/sysDict.md(sql)里。

这通常用在系统数据库表较多或者有多个模块的时候。

8. BeetlSQL 数据模型

BeetlSQL是一个全功能DAO工具，支持的模型也很全面，包括

- Pojo, 也就是面向对象Java Object。Beetlsql操作将选取Pojo的属性和sql列的交集。额外属性和额外列将忽略.
- Map/List, 对于一些敏捷开发，可以直接使用Map/List 作为输入输出参数

```
List<Map<String,Object>> list =
sqlManager.select("user.find",Map.class,paras);
```

- 混合模型，推荐使用混合模型。兼具灵活性和更好的维护性。Pojo可以实现Tail（尾巴的意思），或者继承TailBean，这样查询出的ResultSet 除了按照pojo进行映射外，无法映射的值将按照列表/值保存。如下一个混合模型:

```
/*混合模型*/
public User extends TailBean{
    private int id ;
    private String name;
    private int roleId;
    /*以下是getter和setter 方法*/
}
```

对于sql语句:

```
selectUser
===
select u.*,r.name r_name from user u left join role r on u.roleId=r.id
.....
```

执行查询的时候

```
List<User> list = sqlManager.select("user.selectUser",User.class,paras);
for(User user:list){
    System.out.println(user.getId());
    System.out.println(user.get("rName"));
}
```

程序可以通过get方法获取到未被映射到pojo的值，也可以在模板里直接 ${user.rName} 显示（对于大多数模板引擎都支持）

另外一种更自由的实现混合模型的方法是在目标Pojo上采用注解@Tail，如果注解不带参数，则默认会调用set(String,Object) 方法来放置额外的查询属性，否则，依据注解的set参数来确定调用方法

```
@Tail(set="addValue")
public class User  {
    private Integer id ;
    private Integer age ;
    public User addValue(String str,Object ok){
        ext.put(str, ok);
        return this;
    }
```

不仅仅查询结果支持这种混合模型，查询条件也支持。BeetlSql在调用底层SQLScript的时候，传递的参数实际上是Map，包含了sql语句需要的键值对。有一个特殊的键是"_root", 如果Beetlsql未在Map中找到，则从"_root"中查找，"_root"可以是个Map，或者是个Pojo。从"_root" 继续查找

因为这个特性，BeetlSql的参数可以混合Map和Pojo。如下是一些常用例子

```
Map map = new HashMap();
User query = .....
map.put("_root",query);
map.put("maxAge",19);
map.put("minAge",15);
List<User> list = sqlManager.select("user.select",User.class,map);
```

sql语句可以使用user的所有属性，“user.select” 如下

```
select
===

select * from user where name = #name# and (age>=#minAge# and age<=maxAge)
```

翻页查询中的参数设置PageQuery.setParas 同样可以使用这种方式，或者直接通过PageQuery API完成

```
PageQuery query = .....
User user = ...
query.setParas(user);
query.setParas("maxAge",19);
query.setParas("minAge",15);
sqlManager.pageQuery("user.query",User.class,query)
```

实际上，在这种情况下，query对象最后构造的参数就是一个带有"_root"键值的Map，当然，setParas直接设置一个带“_root”的Map参数也可以。

> 注意，不要使用TailBean的方法的set 来设置额外参数，尽管可以，但未来考虑不支持。因为TailBean还是放置查询结果。

9. Markdown方式管理

BeetlSQL集中管理SQL语句，SQL 可以按照业务逻辑放到一个文件里，文件名的扩展名是md或者sql。如User对象放到user.md 或者 user.sql里，文件可以按照模块逻辑放到一个目录下。文件格式抛弃了XML格式，采用了Markdown，原因是

- XML格式过于复杂，书写不方便
- XML 格式有保留符号，写SQL的时候也不方便，如常用的< 符号 必须转义
- MD 格式本身就是一个文档格式，也容易通过浏览器阅读和维护

目前SQL文件格式非常简单，仅仅是sqlId 和sql语句本身，如下

```
文件一些说明，放在头部可有可无，如果有说明，可以是任意文字
SQL标示
===
以*开头的注释
SQL语句

SQL标示2
===
SQL语句 2
```

注意:SQL语句从2.7.10版本后 sql部分也可以包含在markdow 的``` 格式

所有SQL文件建议放到一个sql目录，sql目录有多个子目录，表示数据库类型，这是公共SQL语句放到sql目录下，特定数据库的sql语句放到各自自目录下 当程序获取SQL语句得时候，先会根据数据库找特定数据库下的sql语句，如果未找到，会寻找sql下的。如下代码

```
List<User> list = sqlManager.select("user.select",User.class);
```

SqlManager 会根据当前使用的数据库，先找sql/mysql/user.md 文件，确认是否有select语句，如果没有，则会寻找sql/user.md

> 注
>
> - 注释是以* 开头，注释语句不作为sql语句
> - 默认的ClasspathLoader采用了这种方法，你可以实现SQLLoader来实现自己的格式和sql存储方式，如数据库存储

10. SQL 注释

对于采用Markdown方式，可以采用多种方式对sql注释。

- 采用sql 自己的注释符号，"-- " ,优点是适合java和数据库sql之间互相迁移，如

```
select * from user where
--  status 代表状态
statu = 1
```

- 采用beetl注释

```
select * from user where
@ /* 这些sql语句被注释掉
statu = 1
@ */
```

- 在sqlId 的=== 紧挨着的下一行 后面连续使用“*”作为sql整个语句注释

```
selectByUser
===
* 这个sql语句用来查询用户的
* status =1 表示查找有效用户

select * from user where status = 1
```

11. (重要) 配置beetlsql

beetlsql 配置文件是 btsql-ext.properties，位于classpath 根目录下，如果没有此文件，beetlsql将使用系统默认配置，如 * 是开发模式，beetlsql每次运行sql都会检测sql文件是否变化，并重新加载 * 字符集，是系统默认的字符集 * 翻页默认总是从1开始，对于oralce数据库来说，翻页起始参数正合适。对于mysql其他数据库来说，beetlsql，翻页参数变成n-1.一般你不需要关心

11.1. 开发模式和产品模式

beetlsql默认是开发模式，因此修改md的sql文件，不需要重启。但建议线上不要使用开发模式，因为此模式会每次sql调用都会检测md文件是否变化。可以通过修改/btsql-ext.properties ,修改如下属性改为产品模式

```
PRODUCT_MODE = true
```

11.2. NameConversion

数据库字段名与java属性名的映射关系必须配置正确，否则会导致各种问题，如下是一些建议

字段名字是user_id, java属性名是userId, 则使用UnderlinedNameConversion 字段名是userId, java属性名是userId，则使用DefaultNameConversion

如果是其他映射关系，可以考虑自己实现NameConversion接口

11.3. 模板字符集

默认sql模板文件采用的是系统默认字符集，可以更改配置采用指定的字符集

```
CHARSET = UTF-8
```

11.4. 翻页起始参数是0还是1

默认认为1对应于翻页的第一条记录，如果你习惯mysql 那种0对应于第一条记录，则需要配置OFFSET_START_ZERO，设置为true

```
OFFSET_START_ZERO =  true
```

无论是从0开始还是从1开始，都不影响beetlsql根据特定数据库翻译成目标数据库的sql语句，这只是一个约定好的习惯，beetlsql会处理跨数据库翻页的

11.5. 自定义方法和标签函数

可以在sql模板中使用自定义方法和标签函数，具体请参考beetl使用说明，如下是默认配置

```
FN.use = org.beetl.sql.core.engine.UseFunction
FN.globalUse = org.beetl.sql.core.engine.GlobalUseFunction
FN.text = org.beetl.sql.core.engine.TextFunction
FN.join = org.beetl.sql.ext.JoinFunction
FN.isEmpty=org.beetl.sql.ext.EmptyExpressionFunction
FN.page=org.beetl.sql.core.engine.PageQueryFuntion
TAG.trim= org.beetl.sql.core.engine.TrimTag
TAG.pageTag= org.beetl.sql.core.engine.PageQueryTag
```

EmptyExpressionFunction 用在很多地方,如template 类操作,where语句里的条件判断,它 沿用了beetl习惯,对于不存在的变量,或者为null的变量,都返回true,同时如果是字符串,为空字符串也返回true,数组,集合也是这样,有些项目,认为空字符串应该算有值而不应该返回true,你可以参考EmptyExpressionFunction的实现,按照项目语义来定义isEmpty

11.6. isEmpty 和 isNotEmpty

模板类查询和模板更新，以及Sql语句里的判断都依赖于isEmpty函数判断变量是否存在以及是否为null，2.8.4以前版本对空字符串也认为是空，2.8.4之后版本则仅仅判断对象是否存在以及是否为null

```
where 1=1
@if(!isEmpty(content)){
  and   content = #content#
@}
```

如上代码，如果content存在，且不为null，则进入if代码块

如果想兼容以前的判断的方式，即认为空字符串也是空（不推荐这么用了），则需要在btsql-ext.properties，再次使以前的实现方式

，

```
FN.isEmpty = org.beetl.ext.fn.EmptyExpressionFunction
FN.isNotEmpty = org.beetl.ext.fn.IsNotEmptyExpressionFunction
```

你也可以使用beetl的安全输出来表示，比如上面的sql代码，可以用安全输出

```
where 1=1
@if(null!=content!){
  and content = #content
@}
```

如上代码，content！是安全表达式，如果不存在或者为null，则为null，然后通过与null比较。

12. SQL 模板基于Beetl实现

SQL语句可以动态生成，基于Beetl语言，这是因为

- beetl执行效率高效 ，因此对于基于模板的动态sql语句，采用beetl非常合适
- beetl 语法简单易用，可以通过半猜半试的方式实现，杜绝myBatis这样难懂难记得语法。BeetlSql学习曲线几乎没有
- 利用beetl可以定制定界符号，完全可以将sql模板定界符好定义为数据库sql注释符号，这样容易在数据库中测试，如下也是sql模板（定义定界符为"--:" 和 null,null是回车意思);

```
selectByCond
===
select * from user where 1=1
--:if(age!=null)
    age=#age#
--:}
```

- beetl 错误提示非常友好，减少写SQL脚本编写维护时间
- beetl 能容易与本地类交互（直接访问Java类），能执行一些具体的业务逻辑 ，也可以直接在sql模板中写入模型常量，即使sql重构，也会提前解析报错
- beetl语句易于扩展，提供各种函数，比如分表逻辑函数，跨数据库的公共函数等

如果不了解beetl，可先自己尝试按照js语法来写sql模板，如果还有疑问，可以查阅官网 http://ibeetl.com

13. Beetl 入门

Beetl 语法类似js，java，如下做简要说明，使用可以参考 http://ibeetl.com , 或者在线体验 http://ibeetl.com/beetlonline/

13.1. 定界符号

默认的定界符号是@ 和 回车。 里面可以放控制语句，表达式等语，，占位符号是##,占位符号默认是输出？，并在执行sql的传入对应的值。如果想在占位符号输出变量值，则需要使用text函数

```
@if(!isEmpty(name)){
    and name = #name#
}
```

如果想修改定界符，可以增加一个/btsql-ext.properties. 设置如下属性

```
DELIMITER_PLACEHOLDER_START=#
DELIMITER_PLACEHOLDER_END=#
DELIMITER_STATEMENT_START=@
DELIMITER_STATEMENT_END=
```

beetlsql 的其他属性也可以在此文件里设置

13.2. 变量

通过程序传入的变量叫全局变量，可以在sql模板里使用，也可以定义变量，如

```
@var count = 3;
@var status = {"a":1}  //json变量
```

13.3. 算数表达式

同js，如a+1-b%30, i++ 等

```
select * from user where name like #'%'+name+'%'#
```

13.4. 逻辑表达式

有“&&”“||”，还有“！”，分别表示与，或，非， beetl也支持三元表达式

```
#user.gender==1?'女':'男'#
```

13.5. 控制语句

- if else 这个同java，c，js。
- for,循环语句，如for(id:ids){}

```
select * from user where status in (
@for(id in ids){
    #id#  #text(idLP.last?"":"," )#
@}
```

> 注意
>
> - 变量名+LP 是一个内置变量，包含了循环状态，具体请参考beetl文档，text方法表示直接输出文本而不是符号“？”
> - 关于 sql中的in，可以使用内置的join方法更加方便

- while 循环语句，如`while(i<count))`

13.6. 访问变量属性

- 如果是对象，直接访问属性名，user.name
- 如果是Map，用key访问 map["key"];
- 如果是数组或者list，用索引访问，如list[1],list[i];
- 可以直采用java方式访问变量的方法和属性，如静态类Constatns

```
public class Constatns{
    public static int   RUNNING = 0;
    public static User getUser(){}
}
```

直接以java方式访问，需要再变量符号前加上@，可以在模板里访问

```
select * from user where status = #@Constatns.RUNNING# and id =
#@Constatns.getUser().getId()#
```

注意，如果Constants 类 没有导入进beetl，则需要带包名，导入beetl方法是配置 IMPORT_PACKAGE=包名.;包名.

13.7. 判断对象非空

可以采用isEmpty判断变量表达式是否为空(为null)，是否存在，如果是字符串，是否是空字符串，如

```
if(isEmpty(user)||isEmpty(role.name))
```

也可以用传统方法判断，如

```
if(user==null) or if(role.name!=null))
```

变量有可能不存在，可用hasH函数或者需要使用安全输出符号，如

```
if(null==user.name!))
//or
if(has(user))
```

变量表达式后面跟上"!" 表示如果变量不存在，则为！后面的值，如果！后面没有值，则为null

13.8. 调用方法

同js，唯一值得注意的是，在占位符里调用text方法，会直接输出变量而不是“? ”，其他以db开头的方式也是这样。架构师可以设置SQLPlaceholderST.textFunList.add(xxxx) 来决定那些方法在占位符号里可以直接输出文本而不是符号"?"

beetl提供了很多内置方法，如print，debug,isEmpty,date等，具体请参考文档

13.9. 自定义方法

通过配置btsql-ext.properties, 可以注册自己定义的方法在beetlsql里使用，如注册一个返回当前年份的函数，可以在btsql-ext.properties加如下代码

```
FN.db.year= com.xxx.YearFunction
```

这样在模板里,可以调用db.year() 获得当前年份。YearFunction 需要实现Function的 call方法，如下是个简单代码

```
public class YearFunction implements Function{
    public String call(Object[] paras, Context ctx){
        return "2015";
    }
}
```

关于如何完成自定义方法，请参考 ibeetl 官方文档

13.10. 内置方法

- print println 输出，同js，如print("table1");
- has， 判断是否有此全局变量;
- isEmpty 判断表达式是否为空，不存在，空字符串，空集合都返回true;
- debug 将变量输出到控制台，如 debug(user);
- text 输出，但可用于占位符号里
- page 函数，用于在PageQuery翻页里，根据上下问决定输出count(1) 或者count(*),如果有参数，则按照参数输出
- join, 用逗号连接集合或者数组，并输出? ，用于in，如

```
select * from user where status in ( #join(ids)#)
            -- 输出成  select * from user where status in (?,?,?)
```

- use 参数是同一个md文件的sqlid，类似mybatis的 sql功能，如

```
condtion
===
where 1=1 and name = #name#

selectUser
===
select * from user #use("condition")#
```

globalUse 参数是其他文件的globalUse，如globalUse("share.accessControl"),将访问share.md(sql)文件的accessControl片段

- db.dynamicSql类似use功能,但第一个参数是sql片段，而不是sqlId

```
queryUsers
===
@ var sql = "id=#xxx#";
select #page("*")# from user where 1=1 and #db.dynamicSql(sql,{xxx:1})#
```

- page 用于pagequery,但beetlsql 使用pagequery查询,会将sql模板翻译成带有count(1),和列表名的俩个sql语句,因此必须使用page函数或者pageTag标签

```
queryNewUser
===
select #page()# from user
```

如果无参数,则在查询的时候解释成 *,如果有参数,则解释成列名,如 page("a.name,a.id,b.name role_name") ,如果列名较多,可以使用pageTag

13.11. 标签功能

- beetlsql 提供了trim标签函数，用于删除标签体最后一个逗号，这可以帮助拼接条件sql，如

```
updateStatus
===

update user set
@trim(){
    @if(!isEmpty(age){
        age = #age# ,
    @} if(!isEmpty(status){
        status = #status#,
    @}
@}
where id = #id#
```

trim 标签可以删除 标签体里的最后一个逗号.trim 也可以实现类似mybatis的功能，通过传入trim参数prefix，prefixOverrides来完成。具体参考标签api 文档

- pageTag,同page函数,用于pageQuery,如

```
queryNewUser
===
select
@pageTag(){
    id,name,status
@}
from user
```

注:可以参考beetl官网 了解如何开发自定义标签以及注册标签函数

- pageIgnoreTag，该标签的作用是在生成分页查询的count语句时，忽略sql语句里的某些内容，如：order by 。pageIgnoreTag与pageTag标签组合使用，组合如下

```
queryNewUser
===
select
@pageTag(){
    id,name,status
@}
from user
@pageIgnoreTag(){
    order by a.createTime
@}
```

因为count语句，无需要排序语句部分，而且，有些数据库，如SQLServer并不支持count语句被排序，因此可以使用pageIgnoreTag来解决夸数据库问题

- where

该标签复用TrimTag，其工作过程是：判断where 里的sql内容是否为空，如果为空就不输出空字符串，如果不为空则判断sql是否以AND或OR开头，如果是，则去掉。例如模板内容如下：

```
queryNewUser
===
select a.* from user a
@where(){
@if(!isEmpty(age){
    and a.age=#age#
@}
@if(!isEmpty(status){
    and a.status=#status#
@}
@}
```

将生成

```
select a.* from user a
where
a.age=?
and a.status=?
```

当然，如果你不用where，也可用where 1=1 来解决，我看并没有多大差别。

14. Debug功能

Debug 期望能在控制台或者日志系统输出执行的sql语句，参数，执行结果以及执行时间，可以采用系统内置的DebugInterceptor 来完成，在构造SQLManager的时候，传入即可

```
SqlManager sqlManager = new SqlManager(source,mysql,loader,nc ,new
Interceptor[]{new DebugInterceptor() });
```

或者通过spring，jfianl这样框架配置完成。使用后，执行beetlsql，会有类似输出

```
┌──────── Debug [user.selectUserAndDepartment] ─────
├ SQL:    select * from user where 1 = 1
├ 参数:     []
├ 位置:     org.beetl.sql.test.QuickTest.main(QuickTest.java:47)
├ 时间:     23ms
├ 结果:     [3]
└──────── Debug [user.selectUserAndDepartment] ─────
```

beetlsql会分别输出 执行前的sql和参数，以及执行后的结果和耗费的时间。你可以参考DebugInterceptor 实现自己的调试输出

DebugInterceptor 还允许设置“位置”的实现，这是因为很多业务系统封装了Beetlsql，更希望"位置"打印的是业务系统，而非封装类的位置，具体可参考DebugInterceptor 源码

> 默认情况下，Spring 集成会自动集成DebugInterceptor，如果想取消，需要自己修改集成代码，不像SQLManager 传入DebugInterceptor

15. 缓存功能

同DebugInterceptor构造方式一样， SimpleCacheInterceptor能缓存指定的sql查询结果

```
List<String> lcs = new ArrayList<String>();
lcs.add("user");
SimpleCacheInterceptor cache =new SimpleCacheInterceptor(lcs);
Interceptor[] inters = new Interceptor[]{ new DebugInterceptor(),cache};
SQLManager  sql = new SQLManager(style,loader,cs,new
UnderlinedNameConversion(), inters);
for(int i=0;i<2;i++){
    sql.select("user.queryUser", User.class, null);
}
```

如上例子，指定所有namespace为user查询都将被缓存，如果此namespace有更新操作，则缓存清除，输出如下

```
┌──────── Debug [user.queryUser] ─────
├ SQL:    select * from User where 1 =1
├ 参数:     []
├ 位置:     org.beetl.sql.test.QuickTest.main(QuickTest.java:54)
├ 时间:     52ms
├ 结果:     [9]
└──────── Debug [user.queryUser] ─────

┌──────── Debug [user.queryUser] ─────
├ SQL:    select * from User where 1 =1
├ 参数:     []
├ 位置:     org.beetl.sql.test.QuickTest.main(QuickTest.java:54)
├ 时间:     0ms
├ 结果:     [9]
└──────── Debug [user.queryUser] ─────
```

第二条查询的时间为0，这是因为直接使用了缓存缘故。

SimpleCacheInterceptor 构造的时候接受一个类列表，所有sqlid的namespace，比如“user.queryUser” 的namespace是“user”，如果beetlsql的查询sqlid此列表里，将参与缓存处理，否则，不参与缓存处理

默认的缓存实现是使用内存Map，也可以使用其他实现方式，比如redies，只需要实现如下接口

```
public static interface CacheManager{
    public void initCache(String ns);
    public void putCache(String ns,Object key,Object value);
    public Object getCache(String ns,Object key);
    public void clearCache(String ns);
    public boolean containCache(String ns,Object key);
}
```

16. Interceptor功能

BeetlSql可以在执行sql前后执行一系列的Intercetor，从而有机会执行各种扩展和监控，这比已知的通过数据库连接池做Interceptor更加容易。如下Interceptor都是有可能的

- 监控sql执行较长时间语句，打印并收集。TimeStatInterceptor 类完成
- 对每一条sql语句执行后输出其sql和参数，也可以根据条件只输出特定sql集合的sql。便于用户调试。DebugInterceptor完成
- 对sql预计解析，汇总sql执行情况（未完成，需要集成第三方sql分析工具）

你也可以自行扩展Interceptor类，来完成特定需求。 如下，在执行数据库操作前会执行before，通过ctx可以获取执行的上下文参数，数据库成功执行后，会执行after方法

```
public interface Interceptor {
    public void before(InterceptorContext ctx);
    public void after(InterceptorContext ctx);
}
```

InterceptorContext 如下，包含了sqlId，实际得sql，和实际得参数, 也包括执行结果result。对于查询，执行结果是查询返回的结果集条数，对于更新，返回的是成功条数，如果是批量更新，则是一个数组。可以参考源码DebugInterceptor

```
public class InterceptorContext {
    private String sqlId;
    private String sql;
    private  List<SQLParameter> paras;
    private boolean isUpdate = false ;
    private Object result ;
    private Map<String,Object> env  = null;
}
```

17. 内置支持主从数据库

BeetlSql管理数据源，如果只提供一个数据源，则认为读写均操作此数据源，如果提供多个，则默认第一个为写库，其他为读库。用户在开发代码的时候，无需关心操作的是哪个数据库，因为调用sqlScript 的 select相关api的时候，总是去读取从库，add/update/delete 的时候，总是读取主库（如下是主从实现原理，大部分情况下无需关心如何实现)

```
sqlManager.insert(User.class,user) // 操作主库，如果只配置了一个数据源，则无所谓主从
sqlManager.unique(id,User.class) //读取从库
```

主从库的逻辑是由ConnectionSource来决定的，如下DefaultConnectionSource 的逻辑

```
@Override
public Connection getConn(String sqlId,boolean isUpdate,String sql,List<?>
paras){
    if(this.slaves==null||this.slaves.length==0) return
this.getWriteConn(sqlId,sql,paras);
    if(isUpdate) return this.getWriteConn(sqlId,sql,paras);
    int status  = forceStatus.get();
    if(status ==0||status==1){
        return this.getReadConn(sqlId, sql, paras);
    }else{
        return this.getWriteConn(sqlId,sql,paras);
    }
}
```

- forceStatus 可以强制SQLManager 使用主或者从数据库。参考api SQLManager.useMaster(DBRunner f) ， SQLManager.useSlave(DBRunner f)

对于不同的ConnectionSource 完成逻辑不一样，对于spring，jfinal这样的框架，如果sqlManager在事务环境里，总是操作主数据库，如果是只读事务环境 则操作从数据库。如果没有事务环境，则根据sql是查询还是更新来决定。

如下是SpringConnectionSource 提供的主从逻辑

```
public Connection getConn(String sqlId,boolean isUpdate,String sql,List
paras){
    //只有一个数据源
    if(this.slaves==null||this.slaves.length==0) return
this.getWriteConn(sqlId,sql,paras);
    //如果是更新语句，也得走master
    if(isUpdate) return this.getWriteConn(sqlId,sql,paras);
    //如果api强制使用
    int status  = forceStatus.get();
    if(status==1){
        return this.getReadConn(sqlId, sql, paras);
    }else if(status ==2){
        return this.getWriteConn(sqlId,sql,paras);
    }
    //在事物里都用master，除了readonly事物
    boolean inTrans =
TransactionSynchronizationManager.isActualTransactionActive();
    if(inTrans){
        boolean  isReadOnly =
TransactionSynchronizationManager.isCurrentTransactionReadOnly();
        if(!isReadOnly){
            return this.getWriteConn(sqlId,sql,paras);
        }
    }
     return this.getReadConn(sqlId, sql, paras);
}
```

注意，对于使用者来说，无需关心本节说的内容，仅仅供要定制主从逻辑的架构师。

18. Sharding-JDBC支持

开发者可以使用Sharding-JDBC来支持数据库分库分表，Sharding-JDBC 对外提供了一个封装好的数据源。只要将此DataSource当成普通数据源配置给BeetlSQL即可，唯一的问题是因为 Sharding-JDBC的数据源提供的MetaData功能较弱，因此构造BeetlSQL的ConnectionSource的时候还需要额外指定一个真正的数据源

```
final DataSource one =  ....
finbal DataSource two = ....
final Map<String, DataSource> result = new HashMap<>(3, 1);
result.put("one",one);
result.put("two",two);

DataSource shardingDataSource =
MasterSlaveDataSourceFactory.createDataSource(result,....)

BeetlSqlDataSource connectionSource = new BeetlSqlDataSource(){
    @Override
    public Connection getMetaData() {
        return one.getConnection();
    }
}
connectionSource.setMaster(shardingDataSource);
1628
```

connectionSource 唯一不同地方是重载了getMetaData，从而从一个真实的DataSource上获取数据库元信息

> 对于分表应用，比如创建了user001,user002.. User对象必须对应user表，如果未创建user表，这样会报错，可以通过SQLManager.addVirtualTable("user001","user"),告诉BeetlSQL，通过user001来获得“user”的metadata信息

19. 跨数据库平台

如前所述，BeetlSql 可以通过sql文件的管理和搜索来支持跨数据库开发，如前所述，先搜索特定数据库，然后再查找common。另外BeetlSql也提供了一些夸数据库解决方案

- DbStyle 描述了数据库特性，注入insert语句，翻页语句都通过其子类完成，用户无需操心
- 提供一些默认的函数扩展，代替各个数据库的函数，如时间和时间操作函数date等
- MySqlStyle mysql 数据库支持
- OracleStyle oralce支持
- PostgresStyle postgresql数据库支持
- 其他还有SQLServer,H2,SQLLite ， DB2数据库支持

20. 代码生成

20.1. 生成pojo 和 md文件

beetsql支持调用SQLManager.gen... 方法生成表对应的pojo类，如：

```
SQLManager sqlManager = new SQLManager(style,loader,cs,new
DefaultNameConversion(), new Interceptor[]{new DebugInterceptor()});
//sql.genPojoCodeToConsole("userRole"); 快速生成，显示到控制台
// 或者直接生成java文件
GenConfig config = new GenConfig();
config.preferBigDecimal(true);
config.setBaseClass("com.test.User");
sqlManager.genPojoCode("UserRole","com.test",config);
```

生成的路径位于工程的scr目录下，beetlsql自动判断是传统java项目还是maven项目，以使得生成的代码和sql放到正确的位置上。你也可以通过调用GenKit.setSrcPathRelativeToSrc 来设置代码生成路径，调用setResourcePathRelativeToSrc来设置生成的sql文件路径

config 类用来配置生成喜好,目前支持生成pojo是否继承某个基类, 是否用BigDecimal代替Double,是否采用Date而不是Timestamp来表示日期，是否是直接输出到控制台而不是文件等 生成的代码如下：

```
package com.test;
import java.math.*;
import java.sql.*;
public class UserRole extends com.test.User{
    private Integer id;

    /* 数据库注释 */
    private String userName;
}
```

也可以自己设定输出模版，通过GenConfig.initTemplate(String classPath),指定模版文件在classpath 的路径，或者直接设置一个字符串模版 GenConfig.initStringTemplate. 系统默认的模版如下：

```
package ${package};
${imports}
/*
* ${comment}
* gen by beetsql ${date(),"yyyy-MM-dd"}
*/
public class ${className} ${!isEmpty(ext)?"extends "+ext} {
    @for(attr in attrs){
        @if(!isEmpty(attr.comment)){
        //${attr.comment}
        @}
        private ${attr.type} ${attr.name} ;
    @}
}
```

如果你需要给beetl模板注册函数等扩展，可以在生成sqlManager.genPojoCode之前，先注册扩展函数

```
SourceGen.gt.registerFunction("xxx",yourFunction)
```

这样就可以再代码模板里使用扩展函数了

> 注意，GenConfig构造函数期望的是一个classpath路径，并非文件路径

20.2. 生成更多的代码

可以实现MapperCodeGen的genCode接口，然后添加到 GenConfig里，这样再生成代码后，也会调用自定义的MapperCodeGen来生成更多代码。如系统内置的生成Mapper的代码

```
MapperCodeGen mapper = new MapperCodeGen("com.dao");
config.codeGens.add(mapper);
sql.genPojoCodeToConsole("user", config);
```

这样，除了生成pojo代码外，还生成mapper代码，内置的mapper代码实现如下，供参考

```
public class MapperCodeGen implements CodeGen {
    String pkg = null;
    public MapperCodeGen(){

    }
    public MapperCodeGen(String pkg){
        this.pkg = pkg;
    }
    public static String mapperTemplate="";
    static {
        mapperTemplate =
GenConfig.getTemplate("/org/beetl/sql/ext/gen/mapper.btl");
    }

    @Override
    public void genCode(String entityPkg, String entityClass, TableDesc
tableDesc,GenConfig config,boolean isDisplay) {
        if(pkg==null){
            pkg = entityPkg;
        }
        Template template = SourceGen.gt.getTemplate(mapperTemplate);
        String mapperClass = entityClass+"Dao";
        template.binding("className", mapperClass);
        template.binding("package",pkg);
        template.binding("entityClass", entityClass);

        String mapperHead = "import "+entityPkg+".*;"+SourceGen.CR;
        template.binding("imports", mapperHead);
        String mapperCode = template.render();
        if(isDisplay){
            System.out.println();
            System.out.println(mapperCode);
        }else{
            try {
                SourceGen.saveSourceFile(GenKit.getJavaSRCPath(), pkg,
mapperClass, mapperCode);
            } catch (IOException e) {
                throw new RuntimeException("mapper代码生成失败",e);
            }
        }


    }

}
```

21. 直接使用SQLResult

有时候，也许你只需要SQL及其参数列表，然后传给你自己的dao工具类，这时候你需要SQLResult，它包含了你需要的sql，和sql参数。 SQLManager 有如下方法，你需要传入sqlid，和参数即可

```
public SQLResult getSQLResult(String id, Map<String, Object> paras)
```

paras 是一个map，如果你只有一个pojo作为参数，你可以使用*“root” 作为key，这样sql模版找不到名称对应的属性值的时候，会寻找*root 对象，如果存在，则取其同名属性。

SQLResult 如下：

```
public class SQLResult {
    public String jdbcSql;
    public List<SQLParameter> jdbcPara;
    public Object[] toObjectArray(){}
}
```

jdbcSql是渲染过后的sql，jdbcPara 是对应的参数描述，toObjectArray 是sql对应的参数值。

SQLParameter 用来描述参数，主要包含了

- value: 参数值
- expression ， 参数对应的表达式,如下sql

```
select * from user where id = #id#
```

则expression 就是字符串id

- type，expression 类型，因为sql里有可能是一个复杂的表达式，因此type有如下值

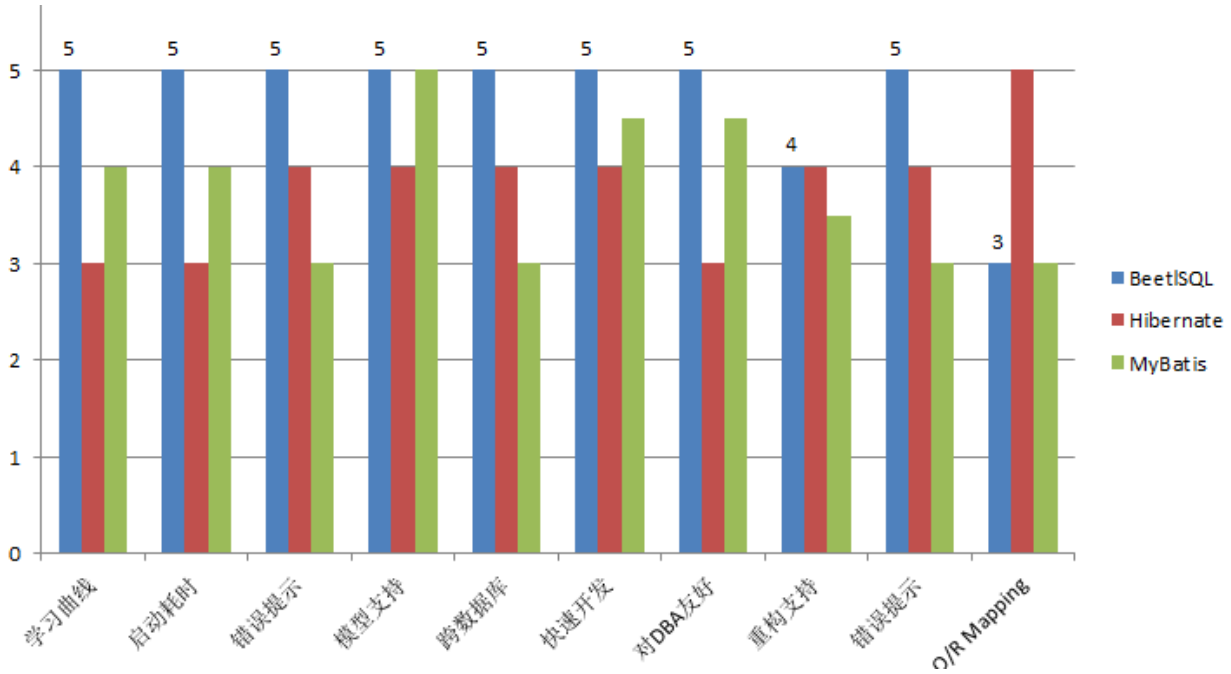
  NAME_GENEARL:简单的表达式，如id

  NAME_EXPRESSION：复杂表达式，比如函数调用，逻辑运算表达式

对于开发者来说，只需要关心sql对应的参数值即可，因此可以调用toObjectArray得到。

22. Hibernate,MyBatis,BeetlSQL 对比

https://my.oschina.net/xiandafu/blog/617542 提供了12项对比并给与评分。在犹豫使用BeetlSQL，可以参考这个全面的对比文章



23. ORM

> 注意
>
> - BeetlSql的Pojo类与数据库表对应，如果Pojo有对应的属性，可以直接映射到属性上，这个同其他ORM工具一样，如果没有关系映射相关特性，实现@Tail 或者继承TailBean（或者实现Tail接口），额外的关系映射才会放到tail属性里供查询的时候调用。
> - 要注意的是，beetlsql的orm 仅仅限于查询结果集，而不包括新增，更新，删除。这些需要调用sqlManager的api直接操作就行了而不像JPA那样还需要成为容器管理对象才能更新
> - 无论是sql语句里配置orm查询,还是通过注解来配置orm查询,都不强求数据库一定有此映射关系,只要当运行调用的时候才会触发orm查询.对于eager查询,当调用beetlsql的时候,会触发ORM查询,对于lazy查询,并不会触发,只会在获取关系属性的时候的,再触发.

23.1. sql语句里的ORM查询

beetlsql 关系映射是在sql语句里通过orm.single和 orm.many,orm.lazySingle,orm.lazyMany 函数进行申明。beetlsql会根据这些申明来完成关系映射

orm.single,orm.many ,orm.lazySingle,orm.lazyMany函数名字本身说明了是一对一，还是一对多或者多对多。以及是直接加载还是懒加载。函数可以放在sql语句任何地方，建议放到头部或者尾部，参数格式有俩种形式，

- 使用模板方式查询关系对象，orm.single({"departmentId","id"},"Department") 第一个参数申明了关系映射，即sql查询结果里属性（非字段名），对应到关系表的查询属性， 如User对象里，departmentId应到Department对象的id，beetlsql会根据此关系发起一次template查询。映射的结果集放在第二个参数Department类里，如果Department与User类在同一个包下，可以省略包名，否则需要加上类包名
- 使用sqlId来查询关系对象，
  orm.single({"departmentId","id"},"user.selectDepatment","Department") 第一个参数还是映射关系，第二个参数是一sql查询id，beetlsql将查询此sql语句，将结果集放到第三个参数Deparmtent类里

- lazy 意味着当调用的时候再加载。如果在事务外调用，并不会像hibernate，JPA那样报错，beetlsql会再用一个数据库连接去查询。一般来讲，如果业务代码确定要用，建议不用lazy方式。因为lazy不会有查询优化，性能可能慢一些.需要注意的是，Beetlsql并非通过proxy技术来实现lazy加载，因此对于lazy加载，你需要继承TailBean
- 映射关系可以用别名，如User对象有myDepartment属性，则映射可以写成 orm.single({"departmentId","id"},"Department",{"alias":"myDepartment"})

如上查询关系对象，结果放到对应的属性上（lazy加载不能放到属性上），或者放到tail属性里，名称就是类名小写开头，如

```
User user =
sqlManager.select("user.selectUserAndDepartment",User.class,paras);
Department
//dept = user.getDepartment();
dept = user.get("department");
```

如下是个例子，假设user表与department表示一对一关系，user.departmentId对应于deparment.id,因此关系映射是{"departmentId":"id"} user与 role表示多对多关系，通过user_role做关联

```
selectUserAndDepartment
===
    select * from user where user_id=#userId#
    @orm.single({"departmentId":"id"},"Department");
    @orm.many({"id":"userId"},"user.selectRole","Role");

selectRole
===

    select r.* from user_role ur left join role r on ur.role_id=r.id

    where ur.user_id=#userId#
```

java代码的样子:

```
User user =
sqlManager.select("user.selectUserAndDepartment",User.class,paras);
Department dept = user.get("department");
List<Role> roles = user.get("role");
```

完整的orm查询例子可以参
考 https://code.csdn.net/xiandafu/beetlsql_orm_sample/tree/master 还有文档

http://my.oschina.net/xiandafu/blog/735809

> 注意

> lazy方式只能继承TailBean才行，如果实现Tail接口，或者@Tail，你取出来的对象并不是你期望的Pojo，而是LazyEntity，你还需要调用get()方法获取到期数据库查询结果 ，可以参看TailBean代码来实现你的Lazy支持

```
public class TailBean implements Tail {
    protected Map<String,Object> extMap = new HashMap<String,Object>();
    boolean hasLazy = false ;
    public Object get(String key){
        if(hasLazy){
            Object o = extMap.get(key);
            if(o instanceof LazyEntity ){
                LazyEntity lazyEntity = (LazyEntity)o;
                try{
                    Object real = lazyEntity.get();
                    extMap.put(key, real);
                    return real;
                }catch(RuntimeException ex){
                    throw new
BeetlSQLException(BeetlSQLException.ORM_LAZY_ERROR,"Lazy Load
Error:"+key+","+ex.getMessage(),ex);
                }
            }else{
                return o;
            }
        }else{
            return extMap.get(key);
        }
    }

    public void set(String key,Object value){
        if(value instanceof LazyEntity ){
            hasLazy = true;
        }
        this.extMap.put(key, value);

    }
}
```

23.2. ORM 注解

不必要为每个sql语句写映射关系或者是使用一个公共的映射关系,可以在映射的实体类上注解映射关系从而实现懒加载机制,如

```
@OrmQuery(
    value={

@OrmCondition(target=Department.class,attr="departmentId",targetAttr="id",t
ype=OrmQuery.Type.ONE,lazy=false),

@OrmCondition(target=ProductOrder.class,attr="id",targetAttr="userId"
,type=OrmQuery.Type.MANY),
        @OrmCondition(target=Role.class,attr="id",targetAttr="userId"
,sqlId="user.selectRole",type=OrmQuery.Type.MANY)
    }
)
public class User   extends TailBean {

    private Integer id ;
    private String name ;
    private Integer departmentId;

    //忽略getter setter
}
```

OrmQuery 标注在类上,OrmCondition 声明了一个懒加载关系.因此,在以后beetlsql的关于此实体的查询,都可以进一步使用懒加载查询.

如果sql语句里也申明了ORM查询,则会和注解的里ORM注解做合并,遇到映射同一个实体,则以sql语句里的映射配置为准

24. 集成和Demo

24.1. Spring集成和Demo

集成提供了Mapper类的自动注入以及SQLManager的自动注入,以及与spring事务集成

```
<!-- DAO接口所在包名，Spring会自动查找其下的类 -->
<bean name="beetlSqlScannerConfigurer"
class="org.beetl.sql.ext.spring4.BeetlSqlScannerConfigurer">
    <!-- 哪些类可以自动注入 -->
    <property name="basePackage" value="org.beetl.sql.ext.spring4"/>
    <!-- 通过类后缀 来自动注入Dao -->
    <property name="daoSuffix" value="Dao"/>
    <property name="sqlManagerFactoryBeanName"
value="sqlManagerFactoryBean"/>
</bean>
<bean id="sqlManagerFactoryBean"
class="org.beetl.sql.ext.spring4.SqlManagerFactoryBean">
   <property name="cs" >
        <bean  class="org.beetl.sql.ext.spring4.BeetlSqlDataSource">
            <property name="masterSource" ref="dataSource"></property>
        </bean>
    </property>
    <property name="dbStyle">
        <bean class="org.beetl.sql.core.db.H2Style"/>
    </property>
    <property name="sqlLoader">
        <bean class="org.beetl.sql.core.ClasspathLoader">
            <property name="sqlRoot" value="/sql"></property>
        </bean>
    </property>
     <property name="nc">
        <bean class="org.beetl.sql.core.UnderlinedNameConversion"/>
    </property>
    <property name="interceptors">
        <list>
            <bean class="org.beetl.sql.ext.DebugInterceptor"></bean>
        </list>
    </property>
</bean>
```

- BeetlSqlScannerConfigurer 根据包名和类后缀来自动注入Dao类,如果没有Dao,可以不配置此项
- cs: 指定ConnectionSource，可以用系统提供的DefaultConnectionSource，支持按照CRUD决定主从。例子里只有一个master库
- dbStyle: 数据库类型，目前只支持org.beetl.sql.core.db.MySqlStyle，以及OralceSytle，PostgresStyle，SQLiteStyle，SqlServerStyle，H2Style
- sqlLoader: sql语句加载来源
- nc: 命名转化，有默认的DefaultNameConversion，数据库跟类名一致，还有有数据库下划线的UnderlinedNameConversion,JPANameConversion,
- interceptors:DebugInterceptor 用来打印sql语句，参数和执行时间

注意： 任何使用了Transactional 注解的，将统一使用Master数据源，例外的是@Transactional(readOnly=true),这将让Beetsql选择从数据库。

```
@Service
public class MyServiceImpl implements MyService {

    @Autowired
    UserDao dao; // UserDao extends BaseMapper<User>

    @Autowired
    SQLManager sql;

    @Override
    @Transactional()
    public int total(User user) {
        int total = list .size();
        dao.deleteById(3);
        User u =new User();
        u.id = 3;
        u.name="hello";
        u.age = 12;
        dao.insert(u);
        return total;

    }

}
```

其他集成配置还包括:

- functions 配置扩展函数
- tagFactorys 配置扩展标签
- configFileResource 扩展配置文件位置，beetlsql将读取此配置文件覆盖beetlsql默认选项
- defaultSchema 数据库访问schema

> 参考
>
> 可以参考demo https://git.oschina.net/xiandafu/springbeetlsql

24.2. SpringBoot集成

```
<dependency>
    <groupId>com.ibeetl</groupId>
    <artifactId>beetl-framework-starter</artifactId>
    <version>1.1.73.RELEASE</version>
</dependency>
```

```
@Configuration
public class DataSourceConfig {

 @Bean(name="datasource")
  public DataSource datasource(Environment env) {
    HikariDataSource ds = new HikariDataSource();
    ds.setJdbcUrl(env.getProperty("spring.datasource.url"));
    ds.setUsername(env.getProperty("spring.datasource.username"));
    ds.setPassword(env.getProperty("spring.datasource.password"));
    ds.setDriverClassName(env.getProperty("spring.datasource.driver-class-
name"));
    return ds;
  }
}
```

提供如下配置

beetl-framework-starter 会读取application.properites如下配置

- beetlsql.sqlPath，默认为/sql, 作为存放sql文件的根目录，位于/resources/sql目录下
- beetlsql.nameConversion: 默认是org.beetl.sql.core.UnderlinedNameConversion,能将下划线分割的数据库命名风格转化为java驼峰命名风格，还有常用的DefaultNameConversion，数据库命名完全和Java命名一直，以及JPA2NameConversion，兼容JPA命名
- beetl-beetlsql.dev：默认是true，即向控制台输出执行时候的sql，参数，执行时间，以及执行的位置，每次修改sql文件的时候，自动检测sql文件修改.
- beetlsql.daoSuffix：默认为Dao。
- beetlsql.basePackage：默认为com，此选项配置beetlsql.daoSuffix来自动扫描com包极其子包下的所有以Dao结尾的Mapper类。以本章例子而言，你可以配置“com.bee.sample.ch5.dao”
- beetlsql.dbStyle ：数据库风格，默认是org.beetl.sql.core.db.MySqlStyle.对应不同的数据库，其他还有OracleStyle，PostgresStyle,SqlServerStyle,DB2SqlStyle,SQLiteStyle,H2Style

如果你想配置主从或者指定一个已经配置好的数据源，可以自己创建一个 BeetlSqlDataSource的Bean，比如，在你的配置代码里

```
@Bean
public BeetlSqlDataSource beetlSqlDataSource(@Qualifier("master")
 DataSource dataSource,@Qualifier("slave")  DataSource slave){
    BeetlSqlDataSource source = new BeetlSqlDataSource();
    source.setMasterSource(dataSource);
    source.setSlaves(new DataSource[]{slave});
    return source;
}
```

注意，可以通过Application.properties 配置如下属性禁用BeetlSQL或者禁用Beetl

```
beetlsql.enabled=false
beetl.enabled=false
```

> 如果不满足你要求，你也可以采用java config方式自己配置，或者参考beetl-framework-starter源码，参考 demo ，http://git.oschina.net/xiandafu/springboot_beetl_beetlsql，自己完成

可以实现BeetlSqlCustomize接口来定制BeetlSQL，比如

```
@Configuration
public MyConfig{
  @Bean
  public BeetlSqlCustomize beetlSqlCustomize(){
    return  new BeetlSqlCustomize(){
      public void customize(SqlManagerFactoryBean sqlManagerFactoryBean){
        //....
      }
    };
  }
}
```

可以掉用SqlManagerFactoryBean来配置，或者获得SQLManager 进一步配置

24.3. SpringBoot集成多数据源

单数据源情况，BeetlSQL配置方式如上一节所示，BeetlSQL会自动根据单数据源配置好BeetlSQL，多数据源情况下，需要配置指定的多数据源，如下俩个数据源

```
@Configuration
public class DataSourceConfig {

    @Bean(name = "a")
    public DataSource datasource(Environment env) {
        HikariDataSource ds = new HikariDataSource();
        ds.setJdbcUrl(env.getProperty("spring.datasource.a.url"));
        ds.setUsername(env.getProperty("spring.datasource.a.username"));
        ds.setPassword(env.getProperty("spring.datasource.a.password"));
        ds.setDriverClassName(env.getProperty("spring.datasource.a.driver-
class-name"));
        return ds;
    }

    @Bean(name = "b")
    public DataSource datasourceOther(Environment env) {
        HikariDataSource ds = new HikariDataSource();
        ds.setJdbcUrl(env.getProperty("spring.datasource.b.url"));
        ds.setUsername(env.getProperty("spring.datasource.b.username"));
        ds.setPassword(env.getProperty("spring.datasource.b.password"));
        ds.setDriverClassName(env.getProperty("spring.datasource.b.driver-
class-name"));
        return ds;
    }
```

对于数据源a，b，需要配置beetlsql如下配置

```
beetlsql.ds.a.basePackage=com.bee.sample.ch5.xxxdao.
beetlsql.ds.b.basePackage=com.bee.sample.ch5.yyyydao
beetlsql.mutiple.datasource=a,b
```

以beetlsql.ds 为前缀，需要分别配置每个数据源的basePackage,nameConversion等配置，如果没有，则使用默认配置 beetlsql.mutiple.datasource 则配置了多个数据源列表。

如果需要定制每一个SQLManager，需要提供BeetlSqlMutipleSourceCustomize

```
@Bean
    public BeetlSqlMutipleSourceCustomize beetlSqlCustomize() {
        return new BeetlSqlMutipleSourceCustomize() {
            @Override
            public void customize(String dataSource,SQLManager sqlManager)
{

            }

        };
    }
```

24.4. JFinal集成和Demo

在configPlugin 里配置BeetlSql

```
JFinalBeetlSql.init();
```

默认会采用c3p0 作为数据源，其配置来源于jfinal 配置，如果你自己提供数据源或者主从，可以如下

```
JFinalBeetlSql.init(master,slaves);
```

由于使用了Beetlsql，因此你无需再配置 **数据库连接池插件**，和**ActiveRecordPlugin**,可以删除相关配置。

在controller里，可以通过JFinalBeetlSql.dao 方法获取到SQLManager

```
SQLManager dao = JFinalBeetlSql.dao();
BigBlog blog = getModel(BigBlog.class);
dao.insert(BigBlog.class, blog);
```

如果想控制事物，还需要注册Trans

```
public void configInterceptor(Interceptors me) {
    me.addGlobalActionInterceptor(new Trans());
}
```

然后业务方法使用

```
@Before(Trans.class)
public void doXXX(){....}
```

这样，方法执行完毕才会提交事物，任何RuntimeException将回滚，如果想手工控制回滚.也可以通过

```
Trans.commit()
Trans.rollback()
```

如果习惯了JFinal Record模式，建议用户创建一个BaseBean，封装SQLManager CRUD 方法即可。然后其他模型继承此BaseBean

> #### 注意
>
> 可以通过jfinal属性文件来配置sqlManager，比如 PropKit.use("config.txt", "UTF-8"),然后可以配置 sql.nc,sql.root，sql.interceptor，sql.dbStyle，具体参考源代码
>
> JFinalBeetlSql.initProp

> 参考
>
> 可以参考demo https://git.oschina.net/xiandafu/jfinal_beet_beetsql_btjson
>
> demo https://code.csdn.net/xiandafu/beetlsql_orm_sample/tree/master

25. 高级部分

25.1 Query对象

在实际应用场景中大部分时候是在针对单表进行操作，单独的写一条单表操作的SQL较为繁琐，为了能进行高效、快捷、优雅的进行单表操作，Query查询器诞生了。

Query使用方式和风格介绍

我们以一个 User表为例，查询模糊查询用户名包含 "t" ，并且delete_time 不为空的数据库，按照id倒序。

```
Query<User> query = sqlManager.query(User.class);
List<User> list = query.andLike("name", "%t%")
    .andIsNotNull("delete_time")
    .orderBy("id desc").select();
```

从上面的例子可以看出，Query是使用链式调用，看起来就像一个完整的sql一般，使用方式遵从用户平时SQL编写习惯，所以用户在使用过程中需遵循SQL格式。 所有的条件列完之后，再调用select（要执行的方法：select，insert，update，count 等等）；

这里有的同学可以看出来，直接使用数据库字段，这样不妥啊！要是重构怎么办。虽然大部分时候建立的数据库字段不会重命名，BeetlSql 还是支持列名重构，代码如下：

```
List<User> list1  = sql.sql.lambdaQuery(User.class)
    .andEq(User::getName, "hi")
    .orderBy(User::getCreateDate)
    .select();
```

> 这种方式必须在JDK8以上

为了方便，下面的例子都采用数据库字段的形式进行，示例数据库为MySql；

Query主要操作简介

Query接口分为俩类：

一部分是触发查询和更新操作，api分别是

- select 触发查询，返回指定的对象列表
- single 触发查询，返回一个对象，如果没有，返回null
- unique 触发查询，返回一个对象，如果没有，或者有多个，抛出异常
- count 对查询结果集求总数
- delete 删除符合条件的结果集

- update 全部字段更新，包括更新null值
- updateSelective 更新选中的结果集（null不更新）
- insert 全部字段插入，包括插入null值
- insertSelective 有选择的插入，null不插入

另外一部分是各种条件：

标准sql操作符	and操作	or操作
==,!=	andEq,andNotEq	orEq,orNotEq
>,>=	andGreat,andGreatEq	orGreat,orGreatEq
<,<=	andLess,andLessEq	orLess,orLessEq
LIKE,NOT LIKE	andLike,andNotLike	orLike,orNotLike
IS NULL,IS NOT NULL	andIsNull,andIsNotNull	orIsNull,orIsNotNull
	andIn ,andNotIn	orIn ,orNotIn
BETWEEN ,NOT BETWEEN	andBetween,andNotBetween	orBetween,orNotBetween
and (.....)	and	or

标准sql	Query方法
限制结果结范围，依赖于不同数据库翻页	limit
ORDER BY	orderBY
GROUP BY	groupBy
HAVING	having

查询器获取

查询器直接通过 sqlManager 获取，多个sqlManager 可以获取各自 的Query。 获取查询器时，我们泛型一下我们是针对哪个对象（对应的哪张表）进行的操作。

```
Query<User> query = sqlManager.query(User.class);
```

Mapper接口也提供了获取Query的方法，比如

```
UserDao dao = sqlManager.getMapper(UserDao.class);
Query<User> query = dao.createQuery();
```

SELECT简单的条件查询

我们还是以User为例，我们需要查询这条SQL

```
SELECT * FROM `user` WHERE `id` BETWEEN 1 AND 1640 AND `name` LIKE '%t%'
AND `create_time` IS NOT NULL ORDER BY id desc
```

直接上代码：

```
Query<User> query = sqlManager.query(User.class);
List<User> list = query.andBetween("id", 1, 1640)
    .andLike("name", "%t%")
    .andIsNotNull("create_time")
    .orderBy("id desc").select();
```

是不是感觉和写SQL一样爽。

如果我们只要查询其中的几个字段怎么办？比如我只要name和id字段，SQL如下：

```
SELECT name,id FROM `user`
```

Query也提供了定制字段的方法，只要传入你需要的字段名即可：

```
Query<User> query = sqlManager.query(User.class);
List<User> list = query.select("name", "id");
```

比如时间比较大小：

```
SELECT name,id FROM `user` WHERE `id` = 1637 AND `create_time` < now() AND
`name` = 'test'
```

```
Query<User> query = sqlManager.query(User.class);
List<User> list = query.andEq("id", 1637)
    .andLess("create_time", new Date())
    .andEq("name", "test")
    .select("name", "id");
```

有的同学会说，OR子句怎么用，和AND一样简单：

```
SELECT * FROM `user` WHERE `name` = 'new name' OR `id` = 1637 limit 0 , 10
```

```
query.andEq("name", "new name")
    .orEq("id", 1637)
    .limit(1, 10)
    .select();
```

为了兼容其他数据库，这里limit都是统一从1开始哦，后面也会提到。

复杂的条件查询

下面就开始进阶了，要进行一条复杂的条件查询SQL，就要用到 query.condition() 方法，产生一个新的条件，比如我们要查询下面这条SQL

```
-- SQL:
SELECT * FROM `user` WHERE ( `id` IN( ? , ? , ? ) AND `name` LIKE ? )OR (
`id` = ? )
-- 参数: [1637, 1639, 1640, %t%, 1640]
```

```
Query<User> query = sqlManager.query(User.class);
List<User> list = query
    .or(query.condition()
        .andIn("id", Arrays.asList(1637, 1639, 1640))
        .andLike("name", "%t%"))
    .or(query.condition().andEq("id", 1640))
    .select();
```

复杂的条件查询，只需要调用 or() 方法 和 and()方法 ，然后使用 query.condition()生成一个新的条件传入就行； 比如下面这条SQL

```
-- SQL:
SELECT * FROM `user` WHERE ( `id` IN( ? , ? , ? ) AND `name` LIKE ? )AND
`id` = ? OR ( `name` = ? )
-- 参数: [1637, 1639, 1640, %t%, 1640, new name2]
```

```
Query<User> query = sqlManager.query(User.class);
List<User> list = query
    .and(query.condition()
    .andIn("id", Arrays.asList(1637, 1639, 1640))
    .andLike("name", "%t%"))
    .andEq("id", 1640)
    .or(query.condition().andEq("name","new name2"))
    .select();
```

INSERT操作

学会条件查询之后，其他操作就简单了，我们看下insert。

全量插入**insert** 方法

```
-- SQL:
insert into `user` (`name`,`department_id`,`create_time`) VALUES (?,?,?)
-- 参数: [new name, null, null]
```

```
    User record = new User();
    record.setName("new name");
    Query<User> query = sqlManager.query(User.class);
    int count = query.insert(record);
```

全量插入，会对所有的值进行插入，即使这个值是NULL；返回影响的行数；

选择插入**insertSelective**方法

```
-- SQL:
insert into `user` ( `name`,`create_time` ) VALUES ( ?,? )
-- 参数: [new name2, now()]
```

```
User record = new User();
record.setName("new name2");
record.setCreateTime(new Date());
Query<User> query = sqlManager.query(User.class);
int count = query.insertSelective(record);
```

insertSelective方法，对user进行了一次有选择性的插入。NULL值的字段不插入；返回影响的行数；

UPDATE操作

update和insert类似,有全量更新和选择更新的方法；

全量更新 **update** 方法

```
-- SQL:
update `user` set `name`=?,`department_id`=?,`create_time`=? WHERE `id` = ?
AND `create_time` < ? AND `name` = ?
-- 参数: [new name, null, null, 1637, now(), test]
```

```
User record = new User();
record.setName("new name");
Query<User> query = sqlManager.query(User.class);
int count = query.andEq("id", 1637)
    .andLess("create_time", new Date())
    .andEq("name", "test")
    .update(record);
```

全量更新，会对所有的值进行更新，即使这个值是NULL；返回影响的行数；

选择更新 **updateSelective** 方法

```
-- SQL:
update `user` set `name`=? WHERE `id` = ? AND `create_time` < ? AND `name`
= ?
-- 参数: [new name, 1637, now(), test]
```

```
User record = new User();
record.setName("new name");
Query<User> query = sqlManager.query(User.class);
int count = query.andEq("id", 1637)
    .andLess("create_time", new Date())
    .andEq("name", "test")
    .updateSelective(record);
```

updateSelective方法，对user进行了一次有选择性的更新。不是null的值都更新，NULL值不更新；返回影响的行数；

DELETE操作

delete操作非常简单，拼接好条件，调用delete方法即可；返回影响的行数。

```
DELETE FROM `user` WHERE `id` = ?
```

```
Query<User> query = sqlManager.query(User.class);
int count = query.andEq("id", 1642).delete();
```

single查询和unique

在beetlSql中还提供了两个用来查询单条数据的方法，single和unique；

single单条查询

single查询，查询出一条，如果没有，返回null；

```
SELECT * FROM `user` WHERE `id` = 1642 limit 0 , 1
```

```
Query<User> query = sqlManager.query(User.class);
User user = query.andEq("id", 1642).single();
```

unique单条查询

unique查询和single稍微不同，他是查询一条，如果没有或者有多条，抛异常；

```
SELECT * FROM `user` WHERE `id` = 1642 limit 0 , 2
```

```
Query<User> query = sqlManager.query(User.class);
User user = query.andEq("id", 1642).unique();
```

如果存在多条，或者没有则抛出异常：

> org.beetl.sql.core.BeetlSQLException: unique查询，但数据库未找到结果集

COUNT查询

count查询主要用于统计行数，如下面的SQL：

```
-- SQL:
 SELECT COUNT(1) FROM `user` WHERE `name` = ? OR `id` = ? limit 0 , 10
-- 参数:    [new name, 1637]
```

```
Query<User> query = sqlManager.query(User.class);
long count = query.andEq("name", "new name")
             .orEq("id", 1637).limit(1, 10)
             .count();
```

拼接条件，调用count方法，返回总行数。

GROUP分组查询和Having子句

有时候我们要进行分组查询，如以下SQL：

```
SELECT * FROM `user` WHERE `id` IN(1637, 1639, 1640 ) GROUP BY name
```

在BeetlSql中直接拼条件调用group方法，传入字段即可：

```
Query<User> query = sqlManager.query(User.class);
List<User> list = query
    .andIn("id", Arrays.asList(1637, 1639, 1640))
    .groupBy("name")
    .select();
```

在分组查询之后，我们可能还要进行having筛选，只需要在后面调用having方法，传入条件即可。

```
SELECT * FROM `user` WHERE `id` IN( 1637, 1639, 1640 ) GROUP BY name HAVING
`create_time` IS NOT NULL
```

```
Query<User> query = sqlManager.query(User.class);
List<User> list = query
    .andIn("id", Arrays.asList(1637, 1639, 1640))
    .groupBy("name")
    .having(query.condition().andIsNotNull("create_time"))
    .select();
```

分页查询

分页查询是我们经常要使用的功能，beetlSql支持多数据，会自动适配当前数据库生成分页语句，在beeltSql中调用limit方法进行分页。如下面的SQL：

```
-- SQL:
SELECT * FROM `user` WHERE `name` = ? OR `id` = ? limit 0 , 10
-- 参数:  [new name, 1637]
```

```
User record = new User();
record.setName("new name");
Query<User> query = sqlManager.query(User.class);
long count = query.andEq("name", "new name")
    .orEq("id", 1637)
    .limit(1, 10)
    .select();
```

这里需要注意，limit方法传入的参数是开始行，和查询的行数。（开始行从1开始计数），beetlSql会根据不同的数据生成相应的SQL语句。

ORDER BY 排序

进行排序查询时，只要调用orderBy方法，传入要排序的字段以及排序方式即可。

```
-- SQL:
SELECT * FROM `user` WHERE `id` BETWEEN ? AND ? AND `name` LIKE ? AND
`create_time` IS NOT NULL ORDER BY id desc
-- 参数:  [1, 1640, %t%]
```

```
Query<User> query = sqlManager.query(User.class);
List<User> list = query.andBetween("id", 1, 1640)
    .andLike("name", "%t%")
    .andIsNotNull("create_time")
    .orderBy("id desc").select();
```

也可以使用asc(),desc()

25.2. ResultSet结果集到Bean的转化

数据库返回的ResultSet将根据Pojo对象的属性来做适当的转化，比如对于数据库如果定义了一个浮点类型，而Java端属性如果是double，则转成double，如果是BigDecimal，则转成BigDecial,如果定义为int类型，则转为int类型。BeanProcessor 类负责处理这种转化，开发者也可以实现自己的BeanProcessor来为特定的sql做转化，比如将数据库日期类型转为Java的Long类型。 如在BeanProcessor.createBean代码里

```
Class<?> propType = prop.getPropertyType();
tp.setTarget(propType);
JavaSqlTypeHandler handler = this.handlers.get(propType);
if(handler==null){
    handler = this.defaultHandler;
}
Object value = handler.getValue(tp);
this.callSetter(bean, prop, value,propType);
```

BeanProcessor 会根据属性类型取出对应的处理类，然后处理ResultSet，如果你先自定义处理类，你可以重新添加一个JavaSqlTypeHandler到handlers

25.3. ResultSet结果集到Map的转化

ResultSet转为Map的时候，有不一样则，根据数据库返回的列类型来做转化，数据库如果定义了一个浮点类型，则使用默认的BigDecimal类型

如在BeanProcessor.toMap代码里

```
String columnName = rsmd.getColumnLabel(i);
if (null == columnName || 0 == columnName.length()) {
  columnName = rsmd.getColumnName(i);
}
int colType = rsmd.getColumnType(i);
Class  classType = JavaType.jdbcJavaTypes.get(colType);
JavaSqlTypeHandler handler = handlers.get(classType);

if(handler==null){
    handler = this.defaultHandler;
}
tp.setIndex(i);
tp.setTarget(classType);
Object value = handler.getValue(tp);
```

JavaType 定义了默认的数据库类型到Java类型的转化，从而获取适当的 JavaSqlTypeHandler，如果没有定义，则使用默认的handler，仅仅使用resultSet.getObject(i)来获取值 需要注意的是，尽量不要使用默认resultSet.getObject(i)来取值，这样会导致不同数据库取的类型不一样导致不兼容不同数据库。 JavaType已经定义了绝大部分数据库类型到Java类型的转化，少量很少使用的类型没有定义，直接使用resultSet.getObject(i)取值

```
//JavaType.java
jdbcJavaTypes.put(new Integer(Types.LONGNVARCHAR), String.class); // -16

// 字符串
jdbcJavaTypes.put(new Integer(Types.NCHAR), String.class); // -15 字符串
jdbcJavaTypes.put(new Integer(Types.NVARCHAR), String.class); // -9 字符串
jdbcJavaTypes.put(new Integer(Types.ROWID), String.class); // -8 字符串
jdbcJavaTypes.put(new Integer(Types.BIT), Boolean.class); // -7 布尔
jdbcJavaTypes.put(new Integer(Types.TINYINT), Integer.class); // -6 数字
jdbcJavaTypes.put(new Integer(Types.BIGINT), Long.class); // -5 数字
jdbcJavaTypes.put(new Integer(Types.LONGVARBINARY), byte[].class); // -4
                                                                   // 二进
制
jdbcJavaTypes.put(new Integer(Types.VARBINARY), byte[].class); // -3 二进制
jdbcJavaTypes.put(new Integer(Types.BINARY), byte[].class); // -2 二进制
jdbcJavaTypes.put(new Integer(Types.LONGVARCHAR), String.class); // -1
//......
```

有些框架，在使用Map的时候，添加了更多的灵活性，比如通过columnName 来片段是否该字段是字典字段，比如都有后缀"_dict",如果是，则从缓存或者查询响应的字典数据，放到ThreadLocal里，以一次性将查询结果，相关字典数据返回

25.4. PreparedStatment

BeanProcessor.setPreparedStatementPara用于JDBC设置参数，内容如下:

```
public  void setPreparedStatementPara(String sqlId,PreparedStatement
ps,List<SQLParameter> objs) throws SQLException {
        for (int i = 0; i < objs.size(); i++) {
            SQLParameter para = objs.get(i);
            Object o = para.value;
            if(o==null){
                ps.setObject(i + 1, o);
                continue ;
            }
            // 兼容性修改：oralce 驱动 不识别util.Date
            if(this.dbName.equals("oracle")){
                Class c = o.getClass();
                if(c== java.util.Date.class){
                    o = new Timestamp(((java.util.Date) o).getTime());
                }
            }

            if(Enum.class.isAssignableFrom(o.getClass())){
                o = EnumKit.getValueByEnum(o);
            }

            //clob or text
            if(o.getClass()==char[].class){
                o = new String((char[])o);
            }


            int jdbcType = para.getJdbcType();
            if(jdbcType==0){
                ps.setObject(i + 1, o);
            }else{
                //通常一些特殊的处理
                throw new UnsupportedOperationException(jdbcType+",默认处理
器并未处理此jdbc类型");
            }

        }
    }
```

SQLParameter 包含了sql对应参数的值，也包含参数对应的变量名，如果该变量还有类型说明，则jdbcType不为0，如下某个sql

```
select * from user where create_time>#createTime,typeofDate#
```

此时，SQLParameter.value 是createTime对应的值，SQLParameter.expression是字符串"createTime",

由于使用了typeof开头的格式化函数，typeofDate 意思是指此值应当着java.sql.Types.Date 来处理（然而，默认的BeanProcessor 并不会处理SQLParameter.jdbcType）

也可以使用“”jdbc“作为格式化函数，比如

```
select * from user where create_time>#createTime,jdbc="date"#
```

jdbc的值来自于java.sql.Types的public 属性

25.5. 自定义BeanProcessor

你可以为Beeetsql指定一个默认的BeanProcessor，也可以为某些特定的sqlid指定BeanProcessor，SqlManager提供了两个方法来完成

```
public void setDefaultBeanProcessors(BeanProcessor defaultBeanProcessors) {
        this.defaultBeanProcessors = defaultBeanProcessors;
}

public void setProcessors(Map<String, BeanProcessor> processors) {
        this.processors = processors;
}
```

25.6. 事务管理

BeetlSql 是一个简单的Dao工具，不含有事务管理，完全依赖web框架的事务管理机制，监听开始事务，结束事务等事件，如果你使用Spring，JFinal框架，无需担心事务，已经集成好了，如果你没有这些框架，也可以用Beetlsq

提供的DSTransactionManager 来指定事务边界，是事实，Spring的事务集成也使用了DSTransactionManager

```
SQLManager  sql = new SQLManager(style,loader,cs,new
UnderlinedNameConversion(), inters);
//.......
DSTransactionManager.start();
User user = new User();
sql.insert(user);
sql.insert(user);
DSTransactionManager.commit();
```

> 注意:ConnectionSourceHelper.getSimple() 获得的是一个简单的cs，没有事务管理器参与，建议你用
>
> getSingle(DataSource ds),返回DefaultConnectionSource具备事务管理。

25.7. 设置自己的BaseMapper

Beetlsql提供了BaseMapper来内置了CRUD等方法，你可以自己定制属于你的“BaseMapper”

```
 // 自定义一个基接口，并获取基接口配置构建器
 MapperConfigBuilder builder =
this.sqlManager.setBaseMapper(MyMapper.class).getBuilder();

/*
* 这两个方法名与 MyMapper接口保持一致. 为了告诉beetlsql, 遇见这个方法名, 帮我用对应的实现类来处理. 这样扩展性更高,
* 更自由.不必等着开源作者来提供实现.
*
* 里面已经内置BaseMapper的所有方法, 用户只需要在自定义的基接口上定义与BaseMapper相同的方法名就可以使用
*/
builder.addAmi("selectCount", new AllCountAmi());

builder.addAmi("selectAll", new AllAmi());

UserDao dao = sqlManager.getMapper(UserDao.class);
long count = dao.selectCount();
```

如上代码设置了MyMapper 为SQLManager的基础mapper，MyMapper定义如下，仅仅定义了三个内置方法

```
public interface MyMapper<T> {
    long selectCount();

    List<T> selectAll();

    List<Integer> selectIds();
}
```

通过builder.addAmi可以为每个方法指定一个是新的实现，Beetlsql已经内置了一些列的实现类，你可以扩展，实现

```
public interface MapperInvoke {
    public Object call(SQLManager sm,Class entityClass,String sqlId,Method
m,Object[] args);
}
```

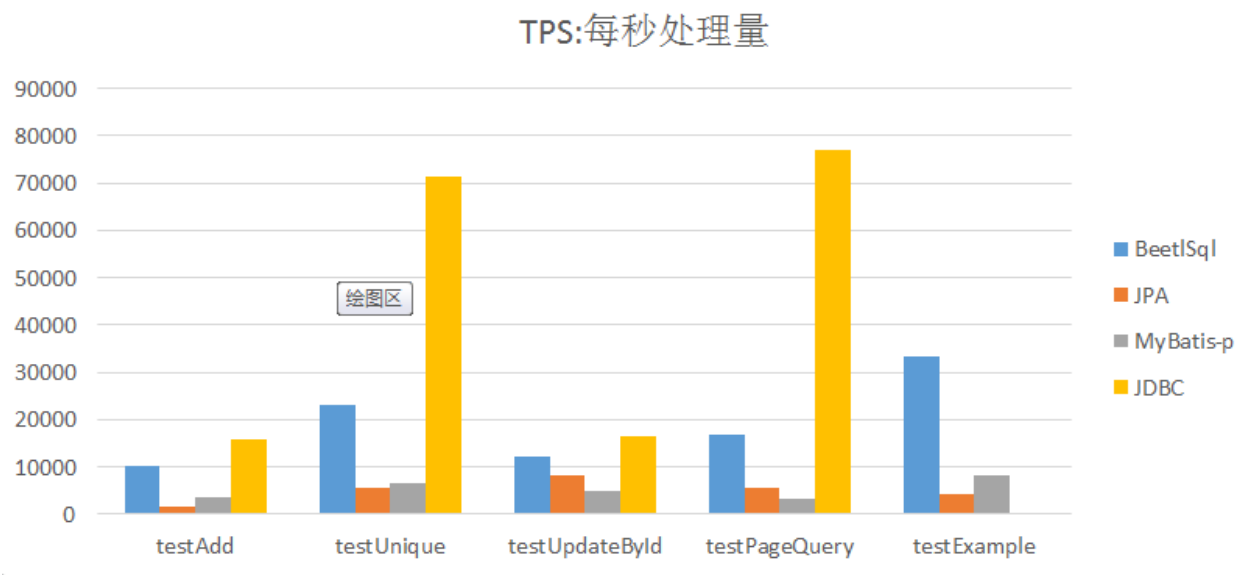
如果你想定制自己的"BaseMapper"，请参考org.beetl.sql.core.mapper.internal.* 所有类

25.8 性能测试

性能测试代码在 https://gitee.com/xiandafu/dao-benchmark

- git clone https://gitee.com/xiandafu/dao-benchmark
- mvn clean package
- java -jar -Dtest.target=jpa target/dao-0.0.1-SNAPSHOT.jar
- 测试目标可更换为jpa,beetlsql,mybatis,jdbc
- 在result目录检测测试文本结果

如下是测试结果



JDBC 作为基准，无疑是最快的，在ORM测视中，BeetlSQL性能基本上是其他JPA，MyBatis的3-7倍

25.9 内置sql语句生成

AbstractDBStyle 提供了跨平台的内置sql语句，比如根据class或者table 生成insert，update，del等语句，你可以覆盖AbstractDBStyle方法来生成特定的sql语句，比如Mysql 支持insert ignore into语句，你可以覆盖generalInsert方法来生成特定的insert ignore into语句