精通BeetlSQL （上）

闲大赋 2019.09

本视频资源

- PPT:https://gitee.com/xiandafu/ppt
- 视频地址 http://bbs.ibeetl.com/bbs/bbs/topic/module/7-1.html
- 本PPT和代码任意使用，修改和传播，收费视频严禁盗版和非法传播
- 个人公众号

目录

为什么还有BeetlSQL

- JPA 是Java EE 标准
- MyBatis国内流行和Apache基金支持
- 如果你看了这个视频，说明你对JPA和MyBatis都有疑惑
  - JPA 复杂的ORM忽略了Dao工具的本质: 访问数据库工具。
  - MyBatis的XML你一定写的不爽，ONGL+XML标签的动态语句一定不爽
  - MyBatis需要大量辅助工具加持，比如同样的国产的MyBatis-Plus，PageHelper，

新手使用Dao的常见误区

- 我压根不想写SQL
  - 这是一种幻觉，刚开始编程的人常有
- 我想找一个Dao能完全跨数据库
  - 幻觉，对于数据库的不同，还是需要分开维护。
- Java代码里拼写SQL挺好的
  - 难以维护，DBA不友好。
  - 特别难改动，sql调整会调整java代码
- 用一个功能强大的ORM工具就好
  - orm站在Java对象角度，并非站在数据库关系角度，也不符合人们查看数据的直观要求

BeetlSQL 特点

- 尽量满足所有人和所有项目，对DAO的所有要求
  - 提供10几个内置方法完成增加删改查等，不需要写SQL，提供了Query接口，也不需要写SQL。带来好处既提高开发效率，也减少数据库重构带来的修改。
  - JPA 一个月熟悉，MyBatis 需要一周，BeetlSQL 1天熟悉
  - 快速启动，只获取一次元信息，高于Mybatis，更比JPA快多了
  - 以SQL为中心：markdown文件管理SQL，对DBA友好
  - 模型支持，支持POJO，Map，混合，还有一定程度的ORM
  - 跨数据库支持，内置操作根据数据库的不同生成不同的sql，复杂SQL可以在markdown里根据数据库不同，分别管理（只管理不同的SQL）

SQL 满足各个层次需要

- 内置SQL，根据实体生成的CRUD各种SQL语句
  - userDao.insert(User), User user = userDao.single(1),UserDao是一个接口，无需实现
- 基于模板的查询
  - List<User> users = userDao.template(userTemplate);
- 基于Query类
  - User user = query.andEq(User::getName，“xiandafu”).single(); 或者 query.andEq(“name”，“xiandafu”).single();
- 自己写SQL
  - List<User> user = sqlManager.execute(new SqlReady(jdbcSQL,paras),User.class);
  - List<User> user = sqlManager.execute(sqlTemplate,User.class,paras);
  - @SQLProvider或者@Sql
- 使用Markdown管理SQL
  - userDao.findAllUser()，会查找user.md 文件中的findAllUser片段，user. findAllUser称为sqlid，标识一个sql片段

SQL模板语句

user.md(sql)

```
selectSample
===

* 一个简单的查询例子
* 根据用户名查询用户

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

selectUser
===

    select * from user where name like #"%"+name# and department_id=#departId#
```

Markdown文件

- 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，可以导出各种格式以方便交流
- 使用Beetl而不是XML标签+ONGL表达式
- 未来基于MD得插件，工具越来越多，写MD会比写XML顺手

组织SQL文件

- 默认存放在classpath的sql目录下
- sqlid是“user.selectUser”,则访问user.md文件中的selectUser片段
- 如果多数据，则还会访问（以mysql和oracle为例子），则还会访问oralce/user.md和mysql/user.md。
- sqlId是consone.user.selectUser,则访问console/user.md 里的selectUser 片段

集成BeetlSQL

- 需要准备
  - 数据库驱动，
  - 数据源或者JDBC URL
  - 注意：BeetlSQL是个单纯工具，并不提供数据源，也不提供事务管理
- SQLManager 核心类
  - DBStyle
  - ConnectionSource
  - NameConversion
  - SQLLoader
- SQLScript 负责SQL执行
  - SQLSource
  - SQLResult
  - BeanProcessor

集成BeetlSQL例子

```
MySqlStyle style = new MySqlStyle();
ConnectionSource cs = ConnectionSourceHelper.getSingle(datasource());
SQLLoader loader = new ClasspathLoader("/sql");
//可选，运行时候打印出sql语句和参数
DebugInterceptor debug = new DebugInterceptor();
Interceptor[] inters = new Interceptor[]{debug};
//构造SQLManager
final SQLManager sql = new SQLManager(style, loader, cs, new
UnderlinedNameConversion(), inters);
```

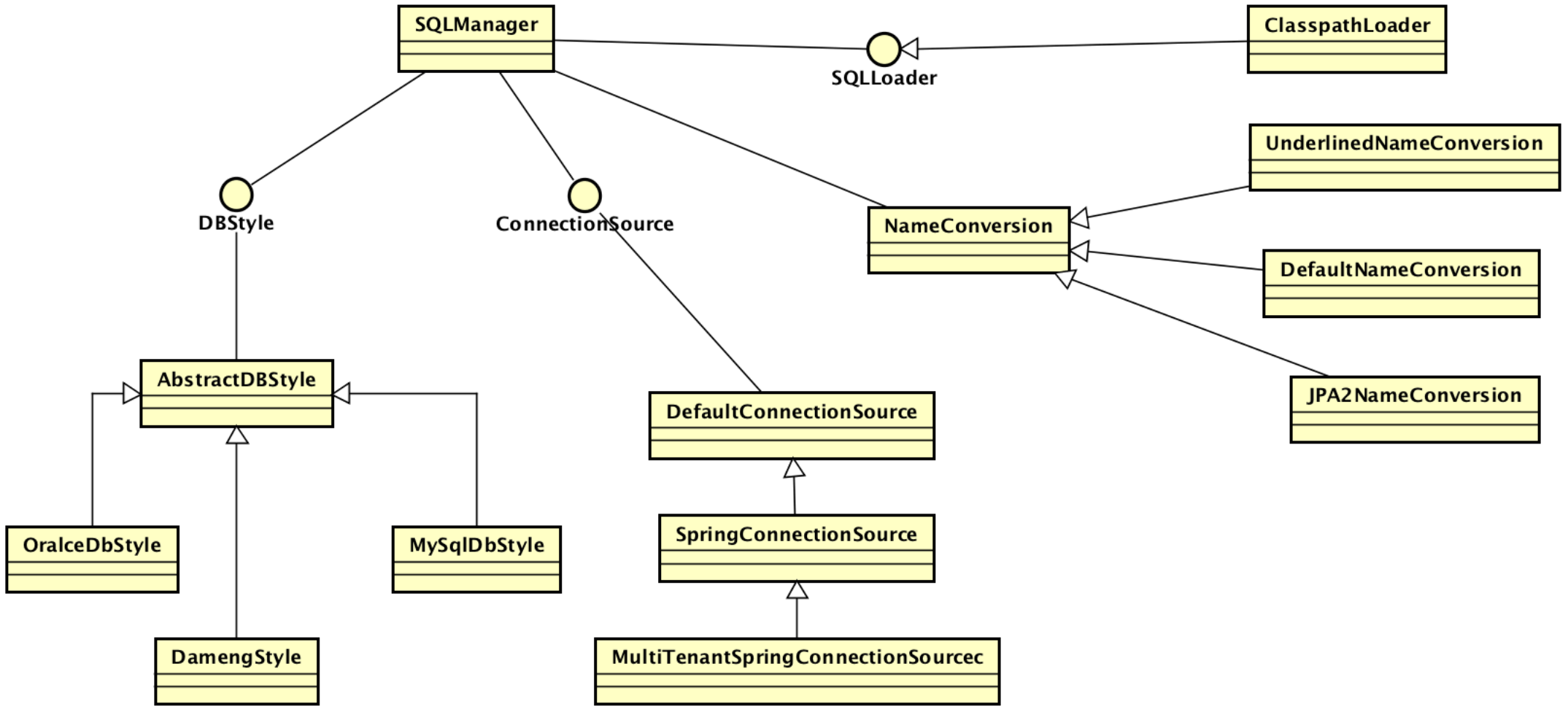
SpringBoot 多数据源集成

```
@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;
    }
}
```

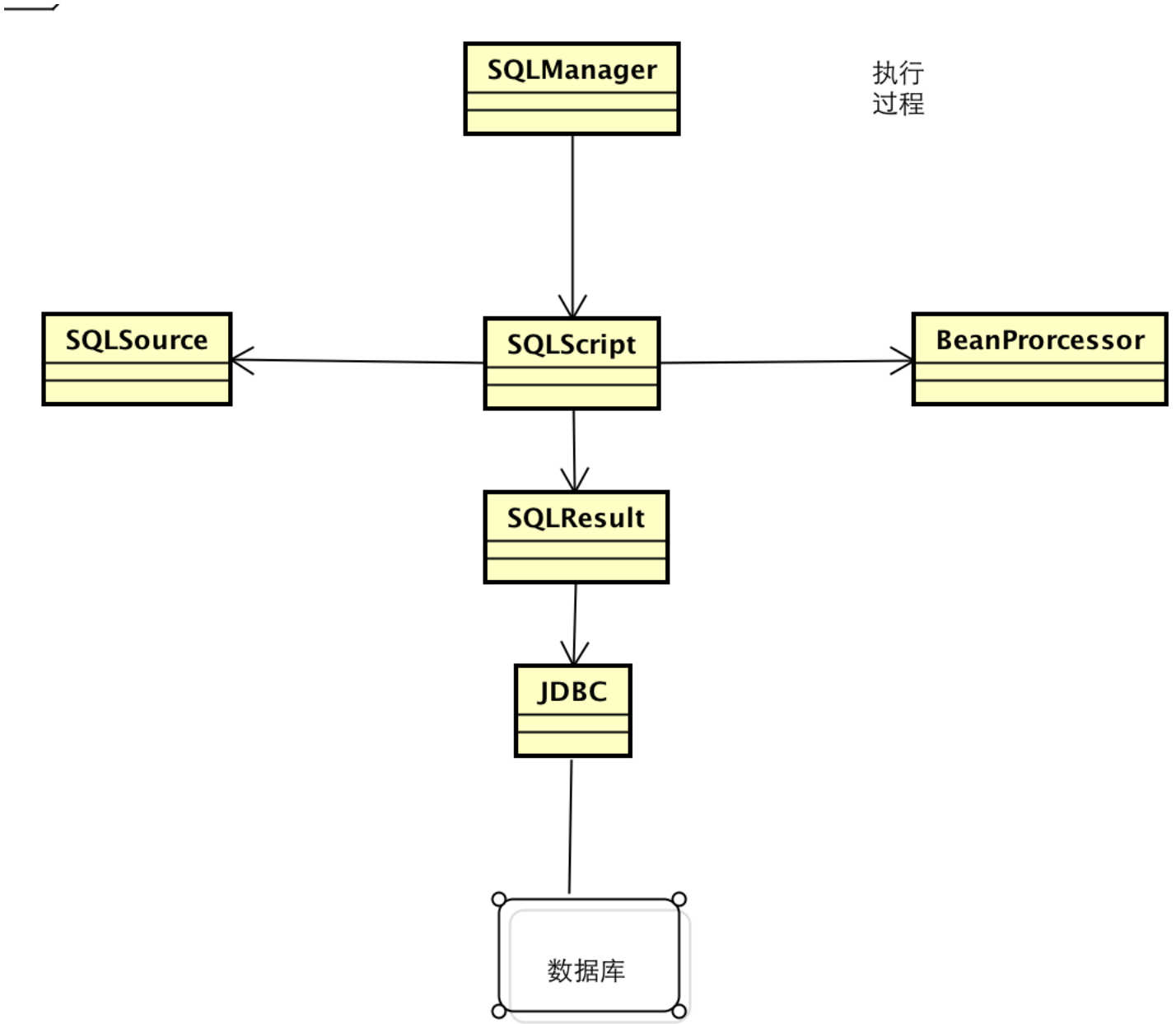
SpringBoot 多数据源集成

```
beetlsql.mutiple.datasource=a,b

beetlsql.ds.a.dbStyle=org.beetl.sql.core.db.MySqlStyle
beetlsql.ds.a.nameConversion=org.beetl.sql.core.UnderlinedNameConversion
beetlsql.ds.a.basePackage=com.bee.sample.ch5.xxxdao
```

核心类和结构


SQLManager
SQLLoader
ClasspathLoader
DBStyle
ConnectionSource
NameConversion
UnderlinedNameConversion
DefaultNameConversion
JPA2NameConversion
AbstractDBStyle
DefaultConnectionSource
OralceDbStyle
MySqlDbStyle
SpringConnectionSource
DamengStyle
MultiTenantSpringConnectionSourcec

核心类和结构


SQLManager
执行
过程
SQLSource
SQLScript
BeanProrcessor
SQLResult
JDBC
数据库

SQLManager

- 职责：核心类，负责所有的事情。有这个类万事俱备
  - 一个SQLManager面向一个同类的数据源。比如一个单一的库，主从数据源，多租户数据源
  - 如果有多个不同库，那需要多个SQLManager
- 基本功能
  - 对象的增删改查操作，最常用
  - 执行SQL模板，完成复查SQL操作，最常用
  - 执行JDBC SQL
  - 获取SQL模板执行后的SQL语句和参数（SQLResult）
  - 得到一个Mapper代理，最常用
  - 得等到一个Query类，无需写SQL，完成业务操作
  - 其他高级扩展：类型转化，Interceptor

NameConversion

- 职责：实现命名转化规则
  - 数据库表名（视图）到Java类的互相转化
  - 数据库列名到Java属性的互相转化
- API
  - public abstract String getColName(Class<?> c,String attrName);
  - public abstract String getPropertyName(Class<?> c,String colName);
  - public abstract String getTableName(Class<?> c);
- 常用子类
  - DefaultNameConversion
  - UnderlinedNameConversion

ConnectionSource

- 职责：
  - 为SQLManager提供一个合适的连接Connection
  - 配合MVC的等框架协助事务管理器管理JDBC事务
- API
  - public Connection getMetaData();
  - public Connection getConn(String sqlId,boolean isUpdate,String sql,List<?> paras);
  - public boolean isTransaction();

DbStyle

- 职责： 区分不同数据库特性
- API
  - SQLSource genSelectById(Class<?> cls); 内置代码生成，
  - getPageSQLStatement，翻页模板语句
  - getPageSQL，翻页得JDBC语句
  - public String getSeqValue(String seqName);*通过序列名字返回获取序列值的sql片段*

SQLScript

- 职责
  - 负责将SQL模板（SQLSource）通过Beetl转化为JDBC语句和参数列表
  - 执行JDBC底层操作
  - 对结果集进行封装，通过（BeanProcessor）转为Bean或者Map，或者更新结果
- API
  - select，insert，update，batch等系列方法
  - 调用Interceptor，提供debug，性能监控，缓存等高级特性
  - SQLResult run(Map<String, Object> paras) 获取SQL模板执行的SQL语句和参数列表

BeanProcessor

- 职责：用于将ResultSet 转化为javaBean或者Map，用户可以注册自己的BeanProcessor
- API
  - <T> T toBean(ResultSet rs, Class<T> type)
  - <T> List<T> toBeanList(ResultSet rs, Class<T> type)
  - Map<String, Object> toMap(String sqlId, Class<?> c, ResultSet rs)

执行过程

SQLManager 内置查询 API

- 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
- 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
- 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

SQLManager 查询 API

- 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) ，或者selectUnique
- public List select(String sqlId, Class clazz, Map<String, Object> paras, int start, int size)
- 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map，其他还有 longValue，bigDecimalValue

SQLManager 新增API

- 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)
- public void insert(Class<?> clazz,Object paras) 插入paras到clazz关联的表
- public int insert(Class clazz,Object paras,boolean autoAssignKey)

SQLManager 更新API

- 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 updateTemplateById(Class<?> clazz，Map paras)
- int upsertByTemplate(Object obj) 同上，按照模板插入或者更新。

SQLManager 批量操作

- public void insertBatch(Class clazz,List<?> list) 批量插入数据
- public void insertBatch(Class clazz,List<?> list,boolean autoAssignKey) 批量插入数据,如果数据库自增主键，获取。
- public int[] updateByIdBatch(List<?> list) 批量更新
- public int updateBatchTemplateById(Class clazz,List<?> list) 批量根据主键更新,属性为null的不会更新
- 需要注意自己控制一次batch的数量！数据库不支持batch数量过大

SQLManager 通过sqlID更新

- 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

SQLManager 直接执行JDBC

- 执行SQL模板
  - public List execute(String sqlTemplate,Class clazz, Object paras)
  - public List execute(String sqlTemplate,Class,Class clazz, Map paras)
  - public int executeUpdate(String sqlTemplate,Class,Object paras) 返回成功执行条数
  - public int executeUpdate(String sqlTemplate,Class,Map paras) 返回成功执行条数
- 执行JDBC SQL
  - public List execute(SQLReady p,Class clazz)
    - List<User> list = sqlManager.execute(new SQLReady(“select * from user where name=? and age = ?”，“xiandafu”,18),User.class)
  - 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

翻页查询PageQuery

- BeetlSQL假定有sqlId 和sqlId$count,俩个sqlId,分别完成翻页和查询结果总数.

```
queryUserByCondition
===

    select * from user ....

queryUserByCondition$count
===

    select count(1) from user .....
```

则可以调用sqlManager的pageQuery方法 完成如下翻页查询

```
//从第一页开始，每页10条
PageQuery query = new PageQuery(1,10);
//条件，或者一次性设置所有条件query.setParas(pojo or map);
query.setPara("gender",1);
sqlManager.pageQuery("account.user.queryUserByCondition",User.class,query);
//查询结果
System.out.println(query.getTotalPage());
System.out.println(query.getTotalRow());
System.out.println(query.getPageNumber());
List<User> list = query.getList();
```

翻页查询PageQuery

- page函数或者pageTag，只提供一个SQL语句完成翻页：在运行时候通过模板技术生成俩条SQL语句

```
queryUserByCondition
===

    select
    @pageTag(){
    a.*
    @}
    from user a  where 1 = 1
    @if(isNotEmpty(name)){
    and name=#name#
    @}
    @pageIgnoreTag(){
    order by a.id desc
    @}
```

后台管理系统用户管理

- 登录
  - User template = new User();
  - template.setName (loginName);
  - User ret = sqlManager.templateOne(template.setName)
- 新增用户
  - sqlManaer.insert(user,true);
  - Long id = user.getId();
- 修改密码
  - User dbUser = sqlManager.unique(User.class,id);
  - dbUser.setPassword(xxxxx);
  - sqlManager.updateById(dbUser);
- 查询
  - List<User> list = sqlManager.select(“user.selecDepartUser” ,User.class,paras) ;