**开发文档**

# 介绍

SQL 是一门 ANSI 的标准计算机语言，用来访问和操作数据库系统。SQL 语句用于取回和更新数据库中的数据。

Ursful-orm是轻量级的数据库映射框架，基于标准的SQL，即通过对象构建标准SQL语句，这样，只要数据库能够支持标准的SQL，就适用于该框架。目前支持，H2、MySQL、Oracle、SQLServer以及PostgreSQL，更多支持请参考扩展。

Ursful-orm是基于Spring的轻量级框架，对象映射由Spring data bind完成。

事务由Spring-ORM管理，采用JDBC驱动，Java预处理执行插入、更新、删除和查询，Java反射解析对象并缓存。支持事务执行后的事件处理。

支持简单查询以及复杂查询，支持查询嵌套查询。

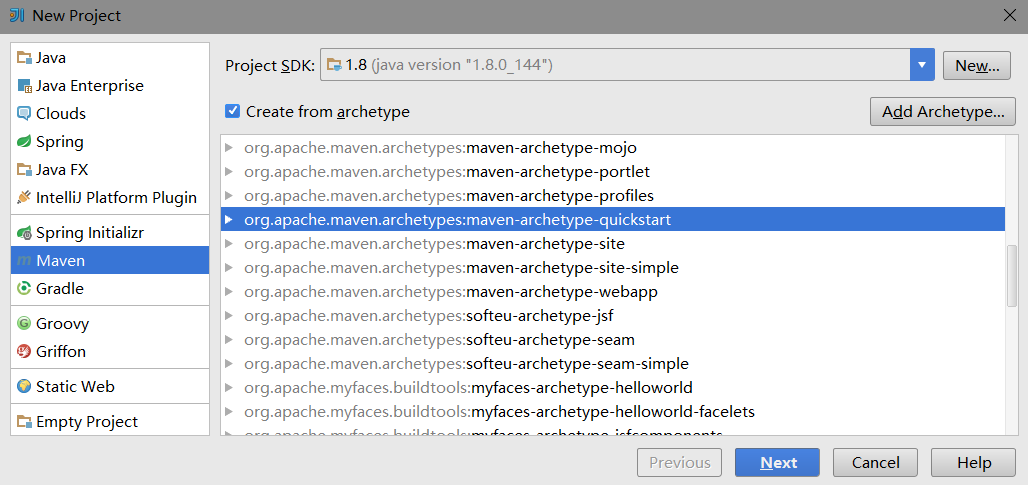
关于函数的使用，支持基本的SUM、AVG、MIN、MAX和COUNT，其他不建议使用。

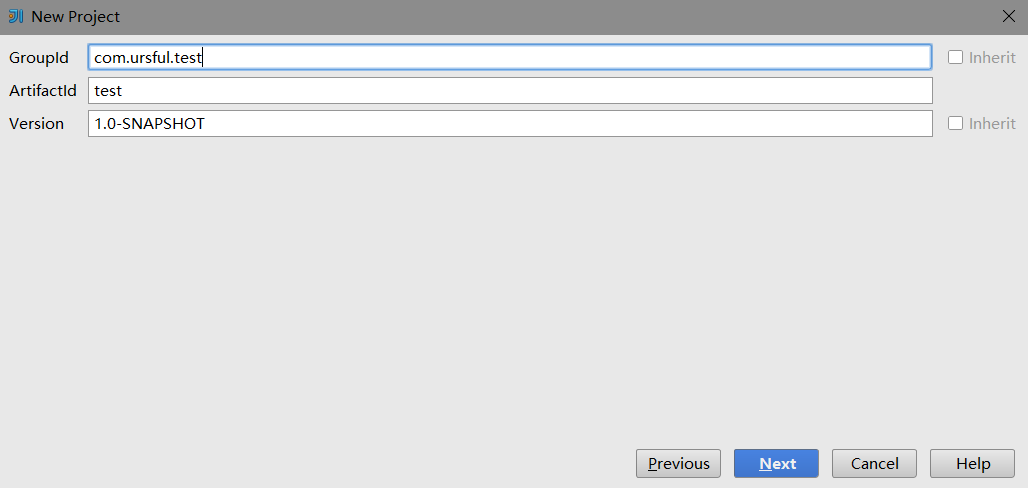
关于时间保存，建议使用Long 【BigInt(15)/Number(15）/NUMERIC(15,0)】类型，框架自动将Date转为Long，取出时，将Long转为Date。因各数据库时间比较函数不统一问题，采用Long，则只需要比较数字，较为高效，自然而然的支持跨数据库。

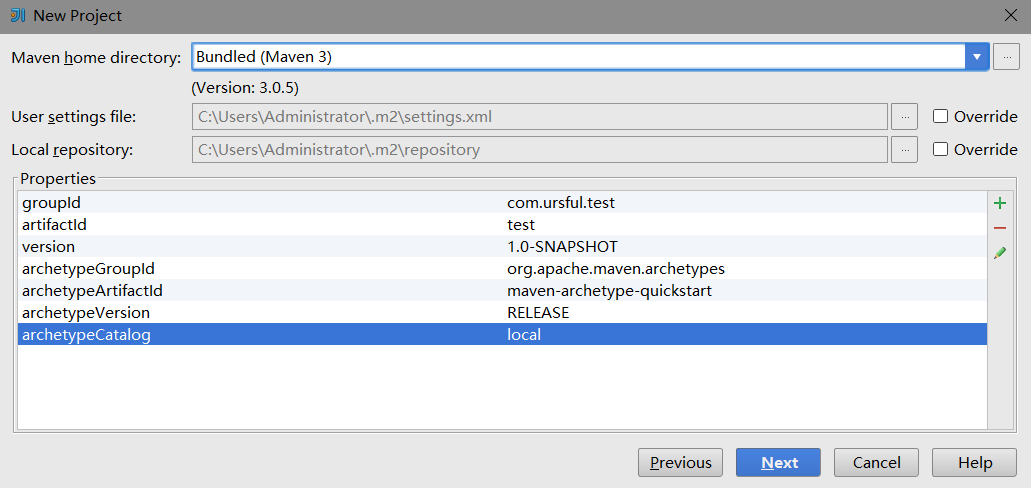
# 快速入门

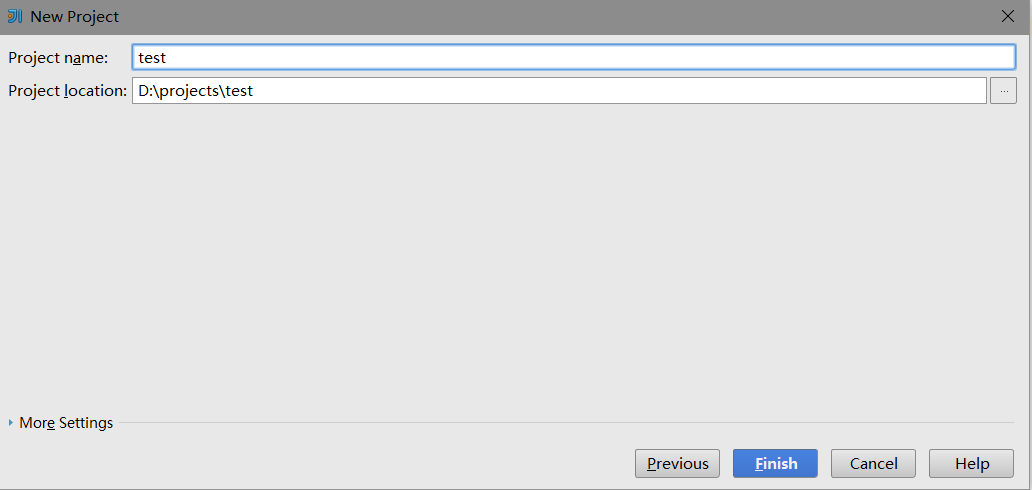
## IDEA创建项目

File -> New -> Project... 选择Create from archetype quickstart ：

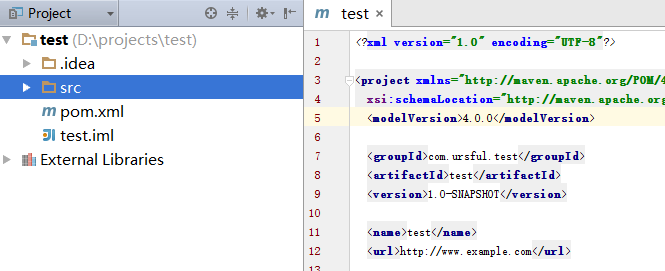








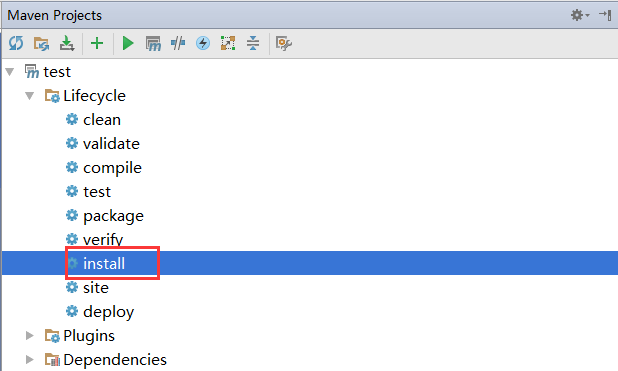
Finish完成项目：

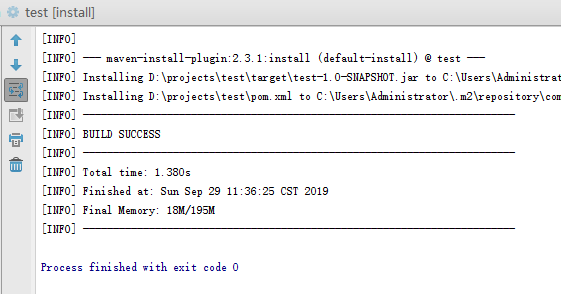


添加依赖 ursful-orm，spring-beans,spring-context ,log4j 以及mysql驱动（本例采用MySQL）：

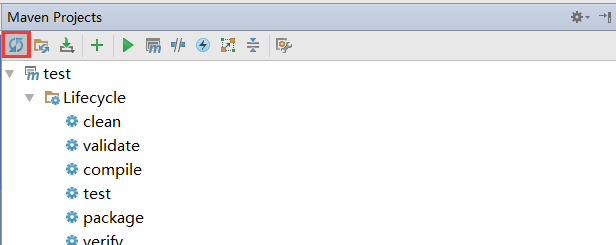
|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  <**modelVersion**>4.0.0</**modelVersion**>   <**groupId**>com.ursful.test</**groupId**>  <**artifactId**>test</**artifactId**>  <**version**>1.0-SNAPSHOT</**version**>   <**name**>test</**name**>  <**url**>http://www.example.com</**url**>   <**properties**>  <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  <**maven.compiler.source**>1.8</**maven.compiler.source**>  <**maven.compiler.target**>1.8</**maven.compiler.target**>  <**spring.version**>4.1.3.RELEASE</**spring.version**>  <**slf4j.version**>1.7.7</**slf4j.version**>  <**logback.version**>1.1.3</**logback.version**>  </**properties**>   <**dependencies**>   <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.38</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-jdbc</**artifactId**>  <**version**>${spring.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-beans</**artifactId**>  <**version**>${spring.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-context</**artifactId**>  <**version**>${spring.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>com.ursful.framework</**groupId**>  <**artifactId**>ursful-orm</**artifactId**>  <**version**>2.3.3</**version**>  </**dependency**>   *<!--Logging: SLF4J -->* <**dependency**>  <**groupId**>org.slf4j</**groupId**>  <**artifactId**>slf4j-api</**artifactId**>  <**version**>${slf4j.version}</**version**>  </**dependency**>  <**dependency**>  <**groupId**>ch.qos.logback</**groupId**>  <**artifactId**>logback-classic</**artifactId**>  <**version**>${logback.version}</**version**>  </**dependency**>  *<!--replace log4j-->* <**dependency**>  <**groupId**>org.slf4j</**groupId**>  <**artifactId**>log4j-over-slf4j</**artifactId**>  <**version**>${slf4j.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>junit</**groupId**>  <**artifactId**>junit</**artifactId**>  <**version**>4.11</**version**>  <**scope**>test</**scope**>  </**dependency**>  </**dependencies**>  </**project**> |

inistall， 显示success：





刷新maven：



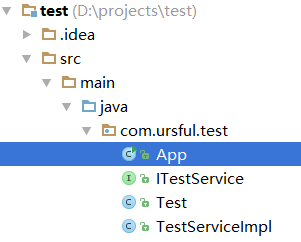


## 创建表

创建数据库（MySQL），如abc，创建表T\_TEST，如下所示：

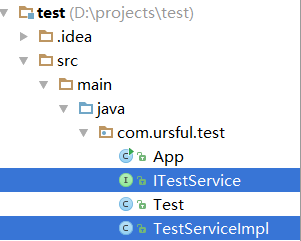
|  |
| --- |
| **CREATE** DATABASE abc **DEFAULT** CHARSET utf8mb4 **COLLATE** utf8mb4\_general\_ci; use abc; **set names** utf8mb4;  CREATE TABLE `T\_TEST` (  `ID` INT(11) NOT NULL AUTO\_INCREMENT COMMENT '编号;自增数字',  `NAME` VARCHAR(100) COMMENT '名称;姓名',  PRIMARY KEY (`ID`)  )COMMENT='测试'COLLATE='**utf8mb4\_bin**' ENGINE=InnoDB; |

## 创建实体



|  |
| --- |
| **import** java.io.Serializable; **import** com.ursful.framework.orm.annotation.RdColumn; **import** com.ursful.framework.orm.annotation.RdTable; **import** com.ursful.framework.orm.annotation.RdId;  @RdTable(name = **"T\_TEST"**, title = **"测试"**) **public class** Test **implements** Serializable{   **private static final long *serialVersionUID*** = 1L;  *//编号;自增数字 (NOT NULL)* **public static final** String ***T\_ID*** = **"ID"**;  *//名称;姓名 (NULL)* **public static final** String ***T\_NAME*** = **"NAME"**;   @RdId  @RdColumn(name = ***T\_ID***, title = **"编号"**, nullable = **false**, description = **"自增数字"**, unique = **true**)  **private** Integer **id**;   @RdColumn(name = ***T\_NAME***, title = **"名称"**, description = **"姓名"**)  **private** String **name**;   **public** Integer getId(){  **return this**.**id**;  }   **public void** setId(Integer id){  **this**.**id** = id;  }   **public** String getName(){  **return this**.**name**;  }   **public void** setName(String name){  **this**.**name** = name;  } } |

## 创建服务



接口：

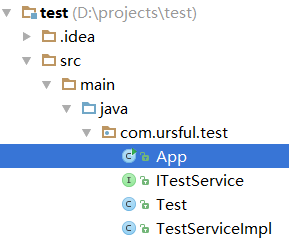
|  |
| --- |
| **import** com.ursful.framework.orm.IBaseService;  **public interface** ITestService **extends** IBaseService<Test>{  } |

实现类：

|  |
| --- |
| **import** com.ursful.framework.orm.BaseServiceImpl;  **public class** TestServiceImpl **extends** BaseServiceImpl<Test> **implements** ITestService{  } |

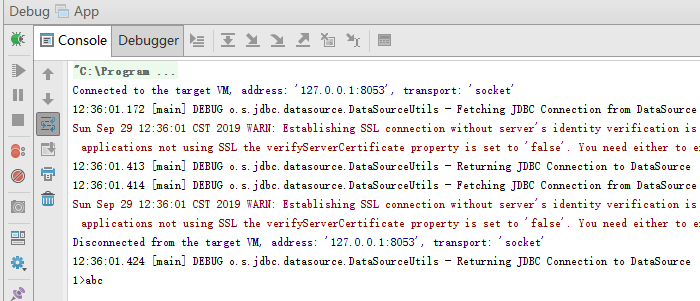
## 测试

运行App中的main方法，试试：



|  |
| --- |
| **import** com.ursful.framework.orm.DataSourceManager;  **import** javax.sql.DataSource; **import** java.io.PrintWriter; **import** java.sql.Connection; **import** java.sql.DriverManager; **import** java.sql.SQLException; **import** java.sql.SQLFeatureNotSupportedException; **import** java.util.List; **import** java.util.logging.Logger;  **public class** App {  **public static void** main( String[] args ){  **final** String driver = **"com.mysql.jdbc.Driver"**;  **final** String url = **"jdbc:mysql://127.0.0.1:3306/ursful?characterEncoding=UTF-8"**;  **final** String username = **"root"**;  **final** String password = **"root"**;  DataSource dataSource = **new** DataSource() {  @Override  **public** Connection getConnection() **throws** SQLException {  **try** {  Class.*forName*(driver);  } **catch** (ClassNotFoundException e) {  e.printStackTrace();  }  **return** DriverManager.*getConnection*(url, username, password);  }   @Override  **public** Connection getConnection(String username, String password) **throws** SQLException {  **return null**;  }   @Override  **public** PrintWriter getLogWriter() **throws** SQLException {  **return null**;  }   @Override  **public void** setLogWriter(PrintWriter out) **throws** SQLException {   }   @Override  **public void** setLoginTimeout(**int** seconds) **throws** SQLException {   }   @Override  **public int** getLoginTimeout() **throws** SQLException {  **return** 0;  }   @Override  **public** Logger getParentLogger() **throws** SQLFeatureNotSupportedException {  **return null**;  }   @Override  **public** <T> T unwrap(Class<T> iface) **throws** SQLException {  **return null**;  }   @Override  **public boolean** isWrapperFor(Class<?> iface) **throws** SQLException {  **return false**;  }  };   DataSourceManager manager = **new** DataSourceManager();  manager.setDataSource(dataSource);   ITestService testService = **new** TestServiceImpl();  testService.setDataSourceManager(manager);   *//获取* Test test = testService.get(1);  **if**(test == **null**) {  test = **new** Test();  test.setName(**"abc"**);  *//保存* testService.save(test);  }  *//查询* List<Test> list = testService.list();  **for**(Test t : list) {  System.***out***.println(t.getId() + **">"** + t.getName());  }  } } |

结果：



# Spring使用配置

配置：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:aop="http://www.springframework.org/schema/aop"  xmlns:tx="http://www.springframework.org/schema/tx"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd  http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd  http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd"**>   <**bean id="dataSourceManager" class="com.ursful.framework.orm.DataSourceManager"**>  <**property name="dataSource" ref="dataSource"**/>  </**bean**> </**beans**> |

用法，实体类略：

接口：

|  |
| --- |
| **import** com.ursful.framework.orm.IBaseService;  **import** com.ursful.framework.admin.common.entity.SysRole;  **public interface** ISysRoleService **extends** IBaseService<SysRole>{  } |

实现类：

|  |
| --- |
| **import** com.ursful.framework.orm.BaseServiceImpl; **import** org.springframework.stereotype.Service;  **import** com.ursful.framework.admin.common.entity.SysRole; **import** com.ursful.framework.admin.common.service.ISysRoleService;   @Service(**"sysRoleService"**) **public class** SysRoleServiceImpl **extends** BaseServiceImpl<SysRole> **implements** ISysRoleService{  } |

控制类注入即可：

|  |
| --- |
| @Autowired **private** ISysRoleService **sysRoleService**; |

# Spring Boot使用配置

配置：

|  |
| --- |
| **import** com.ursful.framework.orm.DataSourceManager; **import** com.ursful.framework.orm.ISQLService; **import** com.ursful.framework.orm.SQLServiceImpl; **import** org.springframework.boot.autoconfigure.condition.ConditionalOnBean; **import** org.springframework.context.annotation.Bean; **import** org.springframework.context.annotation.Configuration; **import** org.springframework.context.annotation.Primary; **import** javax.sql.DataSource;  @Configuration **public class** ManagerConfiguration {   @Bean  @ConditionalOnBean(DataSource.**class**)  **public** DataSourceManager dataSourceManager(DataSource dataSource) {  **return new** DataSourceManager(dataSource);  }   @Bean  @ConditionalOnBean(DataSourceManager.**class**)  **public** ISQLService sqlService(){  **return new** SQLServiceImpl();  } } |

用法同章节4。

# 规范

框架提供实体（Entity）以及Service（接口、实现类）生产工具，目前只支持MySQL，若时间采用Long类型，生成后支持多种数据库。

表名、列名须大写；

生成实体示例（MySQL）：

|  |
| --- |
| CREATE TABLE `T\_TEST` (  `ID` INT(11) NOT NULL AUTO\_INCREMENT COMMENT '编号;自增数字',  `BALANCE` DECIMAL(10,2) NULL DEFAULT '0.00' COMMENT '金额;金额',  `NAME` VARCHAR(100) COMMENT '名称;姓名',  `URL` VARCHAR(200) COMMENT '图片;图片路径',  `CONTENT` TEXT COMMENT '备注;备注',  `CREATOR` VARCHAR(100) COMMENT '创建者;创建者',  `CREATE\_DATE` TIMESTAMP COMMENT '创建日期;创建时间',  `UPDATOR` VARCHAR(100) COMMENT '修改者;修改者',  `UPDATE\_DATE` TIMESTAMP COMMENT '修改日期;修改时间',  PRIMARY KEY (`ID`)  )COLLATE='**utf8mb4\_bin**' ENGINE=InnoDB; |

说明： 注意备注内容“;” 隔开，前部分标记名称，后部分标记描述。

|  |  |  |
| --- | --- | --- |
| **Java对象** | **数据库列类型** | **说明** |
| String | Varchar/varchar2/char/nvarchar | 需指定长度，定义了长度，输入框需做限制，如UUID 36位 |
| String | Text | 较长文本使用 Oracle使用clob、SQL Server 使用NText |
| Integer | Int(8)/Integer/Number |  |
| Long | bigInt(15)/Number | 精度15将自动转化java.util.Date类型 |
| Double | Decimal/Numberic | 精度15将自动转化java.util.Date类型 |
| Date | Timestamp/Datetime | 不允许使用currentTimestamp（默认为null） |
| Clob | Clob | Strin/Clob自动转相互转换，编码统一使用UTF-8 |
| Blob | Blob | Strin/Blob自动转相互转换，编码统一使用UTF-8 |
| byte[] | Binary | byte[]/Binary自动转相互转换 |
| Decimal | Decimal |  |
| Float | Float/Decimal/  Numberic |  |

# 定义

## RdTable

指定数据库表名，以及该表名称：



## RdColumn

将列（NAME）变为静态参数（T\_NAME），便以SQL语句转换：





## RdId

指定主键，即ID列为主键：



## ColumnType

RdColumn中列类型，

|  |  |
| --- | --- |
| **操作** | **描述** |
| ***NULL*** | *空，默认选项* |
| ***BLOB*** | *数据库Blob* |
| ***CLOB*** | *数据库Clob* |
| ***BINARY*** | *数据库二进制流* |
| ***LONG*** | *与Date使用，时间转long* |
| ***DATETIME*** | *数据库DateTime 或者 Date（Oracle）* |
| ***TIMESTAMP*** | *数据库Timestamp* |

## OperatorType

枚举类型，操作符：

|  |  |  |
| --- | --- | --- |
| **操作** | **内容** | **描述** |
| ***AND***(**"&"**) | & | 按位与 |
| ***OR***(**"|"**) | | | 按位或 |
| ***XOR***(**"^"**) | ^ | 按位异或 |
| ***NOT***(**"~"**) | ~ | 取反 |
| ***LL***(**"<<"**) | << | 左移 |
| ***RR***(**">>"**) | >> | 右移 |
| ***PLUS***(**"+"**) | + | 加 |
| ***MINUS***(**"-"**) | - | 减 |
| ***MULTIPLY***(**"\*"**) | \* | 乘 |
| ***DIVIDE***(**"/"**) | / | 除 |
| ***MOD***(**"%"**) | % | 取模 |

## Colmun

最为基础的列，使用说明：

|  |  |
| --- | --- |
| **对象** | **描述** |
| new Column("name") | name |
| new Column("a", "name") | a.name |
| new Column("a", "name", "username") | a.name AS username |
| new Column(Expression.SUM,  "a", "name", "username") | sum(a.name) AS username |
| new Column().function(Expression.SUM)  .alias("a")  .name("name")  .as("username"); | sum(a.name) AS username |
| new Column().function(Expression.SUM)  .alias("a")  .name("name")  .operator(OperatorType.MULTIPLY)  .value(1000); | sum(a.name) \* 1000 |
| new Column().function(Expression.SUM)  .alias("a")  .name("name")  .operator(OperatorType.MULTIPLY)  .operatorInFunction(true)  .value(1000); | sum(a.name \* 1000) |

format 属性说明，按String.format格式书写：

|  |  |  |
| --- | --- | --- |
| **数据库** | **格式** | **说明** |
| MySQL | Column column = **new** Column(**"a"**, **"name"**);  column.format(**"date\_format(%s,'%%Y-%%m-%%d')"**); | **date\_format(a.name,'%Y-%m-%d')"** |
| Oracle | Column column = **new** Column(**"a"**, **"name"**);  column.format(**"to\_char(%s,**  **'YYYY-MM-DD hh24:mi:ss')"**); | **to\_char(a.name,'YYYY-MM-DD hh24:mi:ss')** |
| SQLServer | Column column = **new** Column(**"a"**, **"name"**);  column.format(**"CONVERT(varchar, %s, 101)"**); | **CONVERT(varchar, a.name, 101)** |
| PostgreSQL | Column column = **new** Column(**"a"**, **"name"**);  column.format(**"to\_char(%s,**  **'YYYY-MM-DD hh24:mi:ss')"**); | **to\_char(a.name,'YYYY-MM-DD hh24:mi:ss')** |

## Columns

Columns通常用于select 查询列，如下所示，效果等同：

query.createQuery(Class, new Columns("a","name","value"))

query.createQuery(Class, new Column("a","name"), new Column("a","value"))

说明：

|  |  |
| --- | --- |
| **对象** | **描述** |
| new Columns("a", "name"，"value") | 1. name, a.value   创建查询时： SELECT a.name, a.value FROM ... |
| new Columns("a")  .names("name","value")  .asNames("n","v"); | 创建查询时： SELECT a.name AS n, a.value AS v FROM ... |

* 1. ***Expression(Express) ExpressionType***

注：若无跨数据库要求，日期可以是：Timestamp/DateTime(SQLServer)/Date(Oracle)等，但是时间的比较需要相关函数。所以不建议，建议直接定义为long类型，便于比较。

Expression的通用函数静态变量说明：

|  |  |  |
| --- | --- | --- |
| **类型** | **值** | **描述** |
| EXPRESSION\_ALL | \* | 查询所有列 |
| EXPRESSION\_SUM | SUM | 求和 |
| EXPRESSION\_MAX | MAX | 最大值 |
| EXPRESSION\_MIN | MIN | 最小值 |
| EXPRESSION\_COUNT | COUNT | 数量 |
| EXPRESSION\_AVG | AVG | 平均数 |

ExpressionType的说明：

|  |  |  |
| --- | --- | --- |
| **类型** | **值** | **描述** |
| ***CDT\_None*** | 无 | 无 |
| ***CDT\_Equal*** | = | 等于 |
| ***CDT\_NotEqual*** | <> 或 != | 不等于 |
| ***CDT\_Like*** | LIKE %value% | 匹配 |
| ***CDT\_NotLike*** | NOT LIKE %value% | 不匹配 |
| ***CDT\_StartWith*** | LIKE value% | 开头匹配 |
| ***CDT\_NotStartWith***, | NOT LIKE %value% | 开头不匹配 |
| ***CDT\_EndWith*** | LIKE %value | 结尾匹配 |
| ***CDT\_NotEndWith***, | NOT LIKE %value | 结尾不匹配 |
| ***CDT\_Less*** | < | 小于 |
| ***CDT\_More*** | > | 大于 |
| ***CDT\_LessEqual*** | <= | 小于等于 |
| ***CDT\_MoreEqual*** | >= | 大于等于 |
| ***CDT\_In*** | IN(...) | 包含于，值必须是List，Collection或Set类型 |
| ***CDT\_NotIn*** | NOT IN(...) | 不包含于，值必须是List，Collection或Set类型 |
| ***CDT\_IS\_NULL***, | IS NULL | 等于null |
| ***CDT\_IS\_NOT\_NULL***, | IS NOT NULL | 不等于null |
| ***CDT\_EXISTS*** | EXISTS | 存在，值可以是IQuery |
| ***CDT\_NOT\_EXISTS*** | NOT EXISTS | 不存在，值可以是IQuery |

Expression通常是复杂查询，带有alias，而Express通常是单表简单查询，没有alias：

|  |  |
| --- | --- |
| **对象** | **描述** |
| new Express("age", 1, ExpressionType.CDT\_More) | age > 1 |
| new Express(Expression.SUM ,"age",1,  ExpressionType.CDT\_More) | SUM(age) > 1 |
| Column column1 = new Column("a","type");  Column column2 = new Column("b","id");  new Expression(column1, column2);  new Expression(column1, column2,  ExpressionType.CDT\_Equal); | a.type = b.id |
| Column column1 = new Column("a","type");  new Expression(column1, 1,  ExpressionType.CDT\_More) | a.type > 1 |

## Condition(Terms)

Condition通常是复杂查询，值的内容是Expression，而Terms通常是单表简单查询，值的内容是Express：

|  |  |
| --- | --- |
| **对象** | **描述** |
| new Condition().and(exp1)  .or(exp2, exp3)  .orOr(exp4,exp5)  .orAnd(exp6, exp7)  .andOr(exp8, exp9, exp10); | exp1 and (exp2 or exp3)  or exp4 or exp5)  or (exp6 and exp7)  and (exp8 or exp9 or exp 10); |
| new Terms()  .or(e1, e2, e3)  .and(e4)  .orAnd(e5, e6) | e1 or e2 or e3 and e4 or (e5 and e6); |

## Join(on) JoinType

JoinType的类型：

|  |  |
| --- | --- |
| **类型** | **描述** |
| FULL\_JOIN | FULL JOIN 全连接 |
| INNER\_JOIN | INNER JOIN 内连接 |
| LEFT\_JOIN | LEFT JOIN 左连接 |
| RIGHT\_JOIN | RIGHT JOIN 右连接 |

举例默认说明：

AliasTable test = query.join(Test.class);//默认alias为t1

Column column1 = test.c("type");

Column column2 = new Column("b","id");

|  |  |
| --- | --- |
| **对象** | **描述** |
| new Join(test)  .on(column1, column2); | 默认Left Join  LEFT JOIN Test t1  ON t1.type = b.id |
| new Join(test)  .on(column1, column2)  .on(column1, 1, ExpressionType.CDT\_MORE); | LEFT JOIN Test t1  ON t1.type = b.id  AND t1.type > 1 |
| new Join(test, JoinType.RIGHT\_JOIN)  .on(column1, column2); | RIGHT JOIN Test t1  ON t1.type = b.id |

## Order/MultiOrder

可以使用单排序，也可以使用多排序：

|  |  |
| --- | --- |
| **对象** | **描述** |
| Column column1 = new Column("a","type");  new Order(column1, Order.ASC); | ORDER BY  a.type ASC |
| new MultiOrder().asc("type").desc("id"); | ORDER BY  type ASC, id DESC |
| Column column1 = new Column("a","type");  Column column2 = new Column("b","id");  new MultiOrder().asc(column1).desc(column2); | ORDER BY  a.type ASC, b.id DESC |

## IQuery

IQuery的两个扩展类IBaseQuery和IMultiQuery，它们的实现类分别是BaseQueryImpl和MultiQueryImpl，前者通常用于单表查询，后者则用于多表的复杂查询，包括连接查询。

构造简单单表查询：

**SELECT** **ID,NAME,STATUS**

**FROM** **T\_URS\_NAV**

**ORDER BY** **ORDER\_NUM** **ASC**

|  |
| --- |
| **public** List<UrsNav> list(String name){  IBaseQuery query = **new** BaseQueryImpl();  query.table(UrsNav.**class**);  query.whereLike(UrsNav.***T\_NAME***, name);  query.orderAsc(UrsNav.***T\_ORDER\_NUM***);  query.createQuery(UrsNav.***T\_ID***, UrsNav.***T\_NAME***, UrsNav.***T\_STATUS***);  **return ursNavService**.query(query); } |

复杂关联表查询，因传入的name与ursNavId是空的，所以条件被忽略：

**SELECT** **u0.ID,u0.URS\_NAV\_ID,u0.NAME,u0.PARENT\_ID,u0.AUTO\_CODE,u0.STATUS,u0.ORDER\_NUM,**

**u1.NAME** **AS** **URS\_NAV\_NAME**

**FROM** **T\_URS\_CATALOG u0,T\_URS\_NAV u1**

**WHERE** **u0.URS\_NAV\_ID = u1.ID**

**ORDER BY** **u0.ORDER\_NUM** **ASC**

|  |
| --- |
| **public** List<UrsCatalog> list(String name,String ursNavId){  IMultiQuery query = **new** MultiQueryImpl();  AliasTable catalog= query.table(UrsCatalog.**class**);  AliasTable nav = query.table(UrsNav.**class**);  query.whereEqual(catalog.c(UrsCatalog.***T\_URS\_NAV\_ID***), nav.c(UrsNav.***T\_ID***));  query.whereLike(catalog.c(UrsCatalog.***T\_NAME***), name);//name空值，忽略  query.whereEqual(catalog.c(UrsCatalog.***T\_URS\_NAV\_ID***), ursNavId);//ursNavId空值，忽略  query.createQuery(UrsCatalog.**class**,  catalog.cs(  UrsCatalog.***T\_ID***,  UrsCatalog.***T\_URS\_NAV\_ID***,  UrsCatalog.***T\_NAME***,  UrsCatalog.***T\_PARENT\_ID***,  UrsCatalog.***T\_AUTO\_CODE***,  UrsCatalog.***T\_STATUS***),  nav.cs().names(UrsNav.***T\_NAME***).asNames(**"URS\_NAV\_NAME"**));  **return ursSubNavService**.query(query); } |

# 插入

无论有主键还是没有主键，都可以使用save插入数据：

|  |
| --- |
| Test ***test*** = **new** Test();  ***test*** .setId(**"uuid"**);  ***test*** .setType(1);  ***test*** .setName(**"abc"**);  **testService**.save(***test***); |

# 获取

**方式一**

通过主键获取对象：

|  |
| --- |
| String id = **"uuid";**  Test ***test*** = **testService**.get(id); |

**方式二**

通过对象主键获取对象，示例中type=1将被忽略：

|  |
| --- |
| String id = **"uuid";**  Test ***test*** = **new** Test();  ***test*** .setId(id);  ***test*** .setType(1);//被忽略  ***test =* testService**.get(test); |

**方式三**

通过对象获取对象，示例获取符合type=1的第一条记录：

|  |
| --- |
| Test ***test*** = **new** Test();  ***test***.setType(1);  Test ***test =* testService**.get(test); |

**方式四**

通过查询条件（IQuery）获取对象，查询条件为type=1：

|  |
| --- |
| IBaseQuery ***query*** = **new** BaseQueryImpl();  ***query*** .table(Test.***class***);  ***query***.whereEqual(Test.T\_TYPE, 1);  Test ***test =* testService**.get(***query***); |

# 更新

假定Test实体表名为T\_TEST, 含有id, type, name, value, desc字段：

主键更新：

|  |
| --- |
| Test ***test*** = **new** Test();  ***test***.setType(1);  **test.**setId(**"uuid"**);  **testService**.update(test); // UPDATE T\_TEST SET TYPE = 1 WHERE ID = **'uuid'**; |

更新，列的值为空的更新NULL：

|  |
| --- |
| Test ***test*** = **new** Test();  ***test***.setType(1);  **test.**setId(**"uuid"**);  // UPDATE T\_TEST SET TYPE = 1, NAME = NULL,VALUE = NULL,DESC = NULL WHERE ID = **'uuid'**;  **testService**.update(test, true); |

更新，指定列的值为空的更新NULL：

|  |
| --- |
| Test ***test*** = **new** Test();  ***test***.setType(1);  **test.**setId(**"uuid"**);  // UPDATE T\_TEST SET TYPE = 1, NAME = NULL,VALUE = NULL WHERE ID = **'uuid'**;  **testService**.update(***test***, Test.T\_NAME, Test.T\_VALUE); |

按条件更新（也适合无主键更新或复核主键）：

|  |
| --- |
| Test ***test*** = **new** Test();  ***test***.setType(1);  // UPDATE T\_TEST SET TYPE = 1 WHERE NAME = **'abc'**;  **testService**.updates(***test***, ***new*** Express(Test.T\_NAME, **"abc"**, ExpressionType.CDT\_Equal)); |

按条件更新，列的值为空的更新NULL：

|  |
| --- |
| Test ***test*** = **new** Test();  ***test***.setType(1);  // UPDATE T\_TEST SET TYPE = 1, NAME = NULL,VALUE = NULL,DESC = NULL WHERE NAME = **'abc'**;  **testService**.updatesNull(***test***, ***new*** Express(Test.T\_NAME, **"abc"**, ExpressionType.CDT\_Equal)); |

按条件更新，指定列的值为空的更新NULL：

|  |
| --- |
| Test ***test*** = **new** Test();  ***test***.setType(1);  // UPDATE T\_TEST SET TYPE = 1, VALUE = NULL WHERE NAME = **'abc'**;  **testService**.updatesNull(***test***, ***new*** String[]{Test.T\_VALUE}, ***new*** Express(Test.T\_NAME, **"abc"**, ExpressionType.CDT\_Equal)); |

# 删除

**方式一**

通过主键删除：

|  |
| --- |
| **testService**.delete(**"uuid"**); // DELETE FROM T\_TEST WHERE ID = **'uuid'**; |

**方式二**

通过对象主键删除：

|  |
| --- |
| Test ***test*** = **new** Test();  ***test***.setId(**"uuid"**);  **testService**.delete(***test***); // DELETE FROM T\_TEST WHERE ID = **'uuid'**; |

**方式三**

通过表达式删除：

|  |
| --- |
| **testService**.deletes(  ***new*** Express(Test.T\_TYPE, **1**, ExpressionType.CDT\_Equal),  ***new*** Express(Test.T\_NAME, **"abc"**, ExpressionType.CDT\_Equal));  // DELETE FROM T\_TEST WHERE TYPE= 1 AND NAME= **'abc'** ; |

**方式四：**

通过Terms，构建更复杂的条件删除，暂时不支持构建关联表删除：

|  |
| --- |
| **testService**.deletes(new Terms().or().and().orAnd()...); |

# 判断

**方式一**

通过表达式判断：

|  |
| --- |
| **boolean** existed **= testService**.exists(  ***new*** Express(Test.T\_TYPE, **1**, ExpressionType.CDT\_Equal),  ***new*** Express(Test.T\_NAME, **"abc"**, ExpressionType.CDT\_Equal)); |

**方式二：**

通过Terms，构建更复杂的条件判断：

|  |
| --- |
| **boolean** existed **= testService**.exists(new Terms().or().and().orAnd()...); |

# 数量查询

**方式一**

通过表达式查询：

|  |
| --- |
| **int** count **= testService**.count(  ***new*** Express(Test.T\_TYPE, **1**, ExpressionType.CDT\_Equal),  ***new*** Express(Test.T\_NAME, **"abc"**, ExpressionType.CDT\_Equal));  // SELECT COUNT(\*) FROM T\_TEST WHERE TYPE= 1 AND NAME= **'abc'** ; |

**方式二：**

通过IQuery，构建更复杂的条件查询：

|  |
| --- |
| **int** count **= testService**.queryCount(query); |

# 查询

全表查询：

|  |
| --- |
| List<Test> **tests** **= testService**.list(); // SELECT \* FROM T\_TEST |

只查询Test中的特定字段：

|  |
| --- |
| List<Test> **tests** **= testService**.list(Test.T\_ID, TEST.\_VALUE,Test.T\_NAME);  // SELECT ID, VALUE, NAME FROM T\_TEST |

查询从第2条记录开始的5条记录：

|  |
| --- |
| List<Test> **tests** **= testService**.list(1, 5);  // SELECT \* FROM T\_TEST LIMIT 5 OFFSET 1; |

按表达式查询：

|  |
| --- |
| List<Test> **tests** **= testService**.list(exp1, exp2, ...); |

按条件查询：

|  |
| --- |
| List<Test> **tests** **= testService**.list(  new Terms().or().and()...); |

按条件排序查询：

|  |
| --- |
| List<Test> **tests** **= testService**.list(  new Terms().or().and()...,  new MultiOrder()...); |

按条件排序查询前5条记录：

|  |
| --- |
| List<Test> **tests** **= testService**.list(  new Terms().or().and()...,  new MultiOrder()...,  5); |

按条件查询特定列：

|  |
| --- |
| List<Test> **tests** **= testService**.list(  new Names(Test.T\_ID, Test.T\_NAME),  new Terms().or().and()...); |

按条件排序查询特定列：

|  |
| --- |
| List<Test> **tests** **= testService**.list(  new Names(Test.T\_ID, Test.T\_NAME),  new Terms().or().and()...,  new MultiOrder()...); |

按条件排序查询特定列前5条记录：

|  |
| --- |
| List<Test> **tests** **= testService**.list(  new Names(Test.T\_ID, Test.T\_NAME),  new Terms().or().and()...,  new MultiOrder()...,  5); |

按IQuery查询：

|  |
| --- |
| List<Test> **tests** **= testService**.list(query); |

按IQuery查询前5条：

|  |
| --- |
| List<Test> **tests** **= testService**.list(query, 5); |

# 分页查询

## Pageable

|  |  |  |
| --- | --- | --- |
| **成员变量** | **类型** | **描述** |
| page | Integer | 当前页 |
| size | Integer | 分页大小 |
| total | Integer | 总数 |
| rows | List | 数据列表 |

## 示例

复杂关联表分页查询，传入article的name="a"，默认分页大小20，第一页即0：

**SELECT u0.ID,u0.NAME,u0.CODE,u0.CONTENT,u0.STATUS,u0.ORDER\_NUM,u0.URS\_NAV\_ID,u0.URS\_CATALOG\_ID,**

**u2.NAME AS CATALOG\_NAME,u2.CODE AS CATALOG\_CODE,**

**u1.NAME AS NAV\_NAME,u1.CODE AS NAV\_CODE**

**FROM T\_URS\_ARTICLE u0,T\_URS\_NAV u1,T\_URS\_CATALOG u2**

**WHERE u0.URS\_CATALOG\_ID = u2.ID**

**AND u0.URS\_NAV\_ID = u1.ID**

**AND ( u0.NAME LIKE ? OR u0.CODE LIKE ? OR u1.CODE LIKE ? OR u2.CODE LIKE ?)**

**AND u0.URS\_NAV\_ID = ?**

**ORDER BY u0.CODE DESC**

**LIMIT ? OFFSET ?**

**参数：%a%, %a%, %a%, %a%, 6082f3b3-cc61-4a60-9088-a9c7b4f6e329, 20, 0**

**SELECT COUNT(\*)**

**FROM T\_URS\_ARTICLE u0,T\_URS\_NAV u1,T\_URS\_CATALOG u2**

**WHERE u0.URS\_CATALOG\_ID = u2.ID**

**AND u0.URS\_NAV\_ID = u1.ID**

**AND ( u0.NAME LIKE ? OR u0.CODE LIKE ? OR u1.CODE LIKE ? OR u2.CODE LIKE ?)**

**AND u0.URS\_NAV\_ID = ?**

**参数：%a%, %a%, %a%, %a%, 6082f3b3-cc61-4a60-9088-a9c7b4f6e329**

|  |
| --- |
| **public** Pageable<Map<String, Object>> page(UrsArticle article,Pageable pageable){  IMultiQuery query = **new** MultiQueryImpl();  AliasTable art = query.table(UrsArticle.**class**);  AliasTable nav = query.table(UrsNav.**class**);  AliasTable catalog = query.table(UrsCatalog.**class**);   query.whereEqual(art.c(UrsArticle.***T\_URS\_CATALOG\_ID***), catalog.c(UrsCatalog.***T\_ID***));  query.whereEqual(art.c(UrsArticle.***T\_URS\_NAV\_ID***), nav.c(UrsNav.***T\_ID***));  query.where(**new** Condition().andOr(  **new** Expression(art.c(UrsArticle.***T\_NAME***), article.getName(), ExpressionType.***CDT\_Like***),  **new** Expression(art.c(UrsArticle.***T\_CODE***), article.getName(), ExpressionType.***CDT\_Like***),  **new** Expression(nav.c(UrsNav.***T\_CODE***), article.getName(), ExpressionType.***CDT\_Like***),  **new** Expression(catalog.c(UrsCatalog.***T\_CODE***), article.getName(), ExpressionType.***CDT\_Like***)  ));  query.whereEqual(art.c(UrsArticle.***T\_URS\_NAV\_ID***), article.getUrsNavId());  query.orderDesc(art.c(UrsArticle.***T\_CODE***));  query.createQuery(Map.**class**,   art.cs(  UrsArticle.***T\_ID***,  UrsArticle.***T\_NAME***,  UrsArticle.***T\_CODE***,  UrsArticle.***T\_CONTENT***,  UrsArticle.***T\_STATUS***,  UrsCatalog.***T\_ORDER\_NUM,***  UrsCatalog.***T\_URS\_NAV\_ID,***  UrsCatalog.***T\_URS\_CATALOG\_ID***),  nav.cs().names(UrsNav.***T\_NAME，***UrsNav.***T\_CODE***).asNames(**"NAV\_NAME","NAV\_CODE"**)  catalog.cs().names(UrsCatalog.***T\_NAME，***UrsCatalog.***T\_CODE***).asNames(**"CATALOG\_NAME","CATALOG\_CODE"**));  **return ursArticleService**.queryPage(query, pageable); } |

# 复杂查询

## 默认

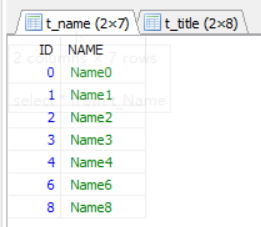
定义表：

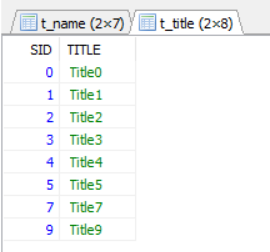
|  |
| --- |
| **CREATE TABLE** `T\_NAME` (  `ID` **INT**(11) **NOT NULL** COMMENT **'编号'**,  `**NAME**` **VARCHAR**(100) COMMENT **'名称'** )COMMENT=**'名称表' COLLATE**=**'utf8mb4\_bin'** ENGINE=InnoDB;  **CREATE TABLE** `T\_TITLE` (  `SID` **INT**(11) **NOT NULL** COMMENT **'编号'**,  `TITLE` **VARCHAR**(100) COMMENT **'标题'** )COMMENT=**'标题' COLLATE**=**'utf8mb4\_bin'** ENGINE=InnoDB; |

初始化数据：

|  |
| --- |
| **for**(**int** i = 0; i < 5; i++){  Name name = **new** Name();  name.setId(i);  name.setName(**"Name"** + i);  Title title = **new** Title();  title.setSid(name.getId());  title.setTitle(**"Title"** + i);  nameService.save(name);  titleService.save(title); } **for**(**int** i = 5; i < 10; i++){  Name name = **new** Name();  name.setId(i);  name.setName(**"Name"** + i);  Title title = **new** Title();  title.setSid(name.getId());  title.setTitle(**"Title"** + i);  **if**(i%2 == 0) {  nameService.save(name);  }**else** {  titleService.save(title);  } } |

示例：





## 全连接

注意：MySQL不支持Full JOIN 查询。

查询语句：

SELECT n0.\*,t0.\* FROM T\_NAME n0 FULL JOIN T\_TITLE t0 ON n0.ID = t0.SID

代码：

|  |
| --- |
| IMultiQuery multiQuery = **new** MultiQueryImpl(); AliasTable nameTable = multiQuery.table(Name.**class**); AliasTable titleTable = multiQuery.join(Title.**class**); multiQuery.join(  **new** Join(titleTable, JoinType.***FULL\_JOIN***).  on(nameTable.c(Name.***T\_ID***), titleTable.c(Title.***T\_SID***))); multiQuery.createQuery(Map.**class**,  nameTable.c(Expression.***EXPRESSION\_ALL***),  titleTable.c(Expression.***EXPRESSION\_ALL***)); List<Map> list = nameService.query(multiQuery); System.***out***.println(**"---------------full join----------"**); **for**(Map map : list) {  System.***out***.println(map); } |

## 左连接

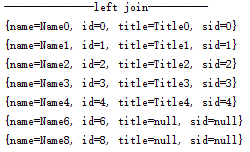
查询语句：

SELECT n0.\*,t0.\* FROM T\_NAME n0 LEFT JOIN T\_TITLE t0 ON n0.ID = t0.SID

代码：

|  |
| --- |
| IMultiQuery multiQuery = **new** MultiQueryImpl(); AliasTable nameTable = multiQuery.table(Name.**class**); AliasTable titleTable = multiQuery.join(Title.**class**); multiQuery.join(  **new** Join(titleTable).  on(nameTable.c(Name.***T\_ID***), titleTable.c(Title.***T\_SID***)));  multiQuery.createQuery(Map.**class**,  nameTable.c(Expression.***EXPRESSION\_ALL***),  titleTable.c(Expression.***EXPRESSION\_ALL***)); List<Map> list = nameService.query(multiQuery); System.***out***.println(**"---------------left join----------"**); **for**(Map map : list) {  System.***out***.println(map); } |

结果：



## 右连接

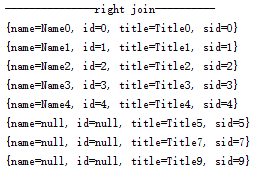
查询语句：

SELECT n0.\*,t0.\* FROM T\_NAME n0 RIGHT JOIN T\_TITLE t0 ON n0.ID = t0.SID

代码：

|  |
| --- |
| IMultiQuery multiQuery = **new** MultiQueryImpl(); AliasTable nameTable = multiQuery.table(Name.**class**); AliasTable titleTable = multiQuery.join(Title.**class**); multiQuery.join(  **new** Join(titleTable, JoinType.***RIGHT\_JOIN***).  on(nameTable.c(Name.***T\_ID***), titleTable.c(Title.***T\_SID***)));  multiQuery.createQuery(Map.**class**,  nameTable.c(Expression.***EXPRESSION\_ALL***),  titleTable.c(Expression.***EXPRESSION\_ALL***)); List<Map> list = nameService.query(multiQuery); System.***out***.println(**"---------------right join----------"**); **for**(Map map : list) {  System.***out***.println(map); } |

结果：



## 内连接

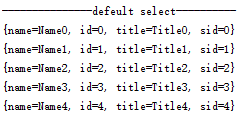
普通多表查询语句：

SELECT n0.\*,t0.\* FROM T\_NAME n0,T\_TITLE t0 WHERE n0.ID = t0.SID

代码：

|  |
| --- |
| IMultiQuery multiQuery = **new** MultiQueryImpl(); AliasTable nameTable = multiQuery.table(Name.**class**); AliasTable titleTable = multiQuery.table(Title.**class**); multiQuery.whereEqual(nameTable.c(Name.***T\_ID***), titleTable.c(Title.***T\_SID***));  multiQuery.createQuery(Map.**class**,  nameTable.c(Expression.***EXPRESSION\_ALL***),  titleTable.c(Expression.***EXPRESSION\_ALL***)); List<Map> list = nameService.query(multiQuery); System.***out***.println(**"---------------defeult select----------"**); **for**(Map map : list) {  System.***out***.println(map); } |

查询结果：



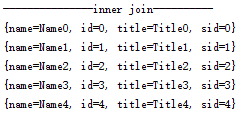
内连接查询语句：

SELECT n0.\*,t0.\* FROM T\_NAME n0 INNER JOIN T\_TITLE t0 ON n0.ID = t0.SID

代码：

|  |
| --- |
| IMultiQuery multiQuery = **new** MultiQueryImpl(); AliasTable nameTable = multiQuery.table(Name.**class**); AliasTable titleTable = multiQuery.join(Title.**class**); multiQuery.join(  **new** Join(titleTable, JoinType.***INNER\_JOIN***).  on(nameTable.c(Name.***T\_ID***), titleTable.c(Title.***T\_SID***))); multiQuery.createQuery(Map.**class**,  nameTable.c(Expression.***EXPRESSION\_ALL***),  titleTable.c(Expression.***EXPRESSION\_ALL***)); List<Map> list = nameService.query(multiQuery); System.***out***.println(**"---------------inner join----------"**); **for**(Map map : list) {  System.***out***.println(map); } |

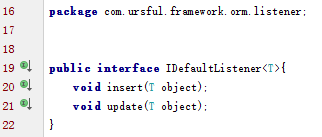
查询结果：



# 监听器

## IDefaultListener

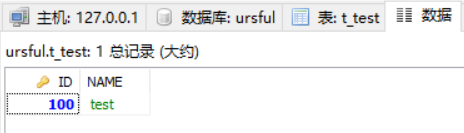
当插入或更新时，触发IDefaultListener：



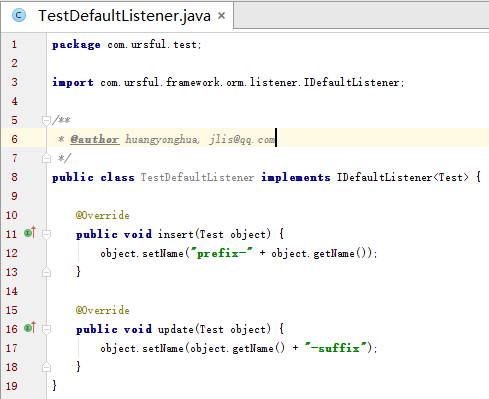
普通插入：

|  |
| --- |
| Test test = **new** Test(); test.setId(100); test.setName(**"test"**); testService.save(test); |

结果：



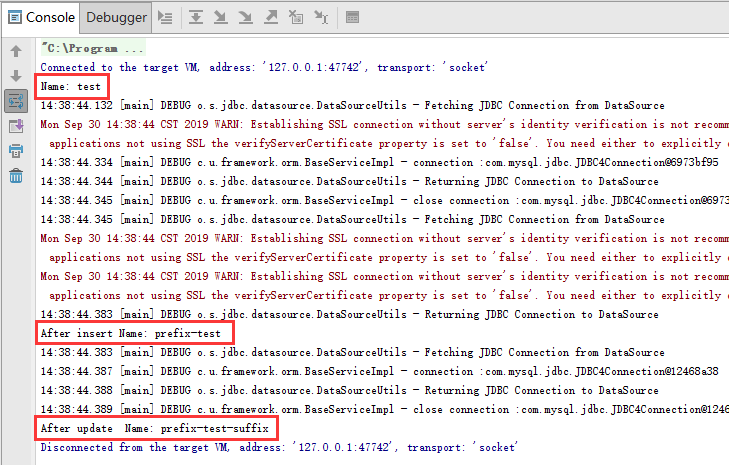
加入监听器：

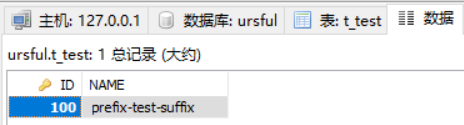


代码：

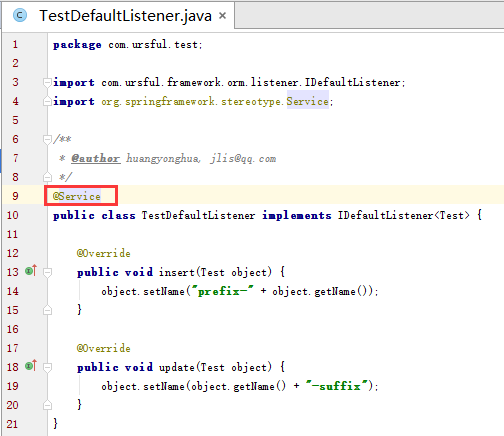
|  |
| --- |
| *//若在Spring容器内，则不需要手动添加* testService.addDefaultListener(**new** TestDefaultListener()); Test test = **new** Test(); test.setId(100); test.setName(**"test"**); System.***out***.println(**"Name: "** + test.getName()); testService.save(test); test = testService.get(test.getId()); System.***out***.println(**"After insert Name: "** + test.getName()); testService.update(test); System.***out***.println(**"After update Name: "** + test.getName()); |

结果：





注意：本测试直接将监听对象加入Service中，若在Spring容器中，只需将TestDefaultListener标记@Service即可。



## IChangeListener

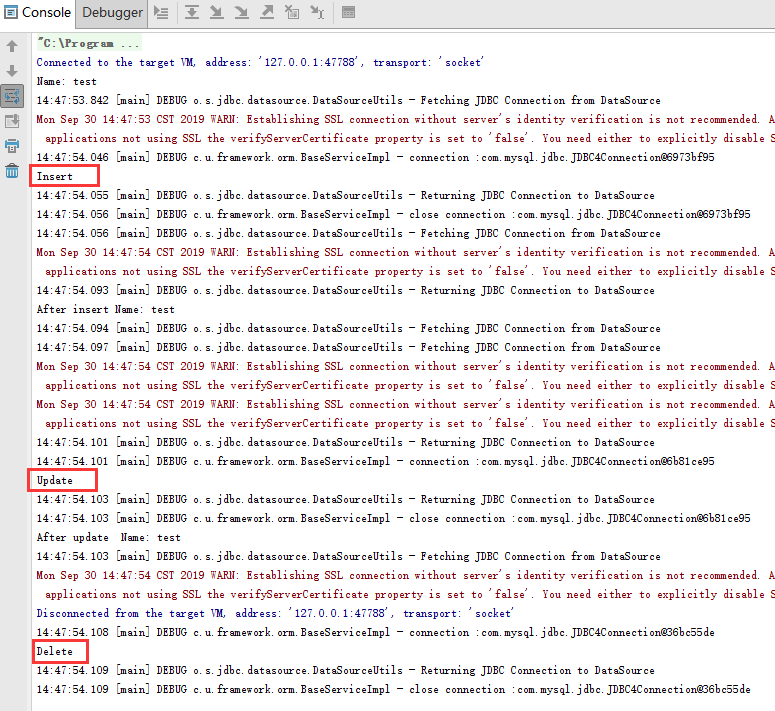
当插入、更新和删除后，将触发IChangeListener，该监听器内容若有SQL语句，将参与事务，若事务回滚，将全部回滚：



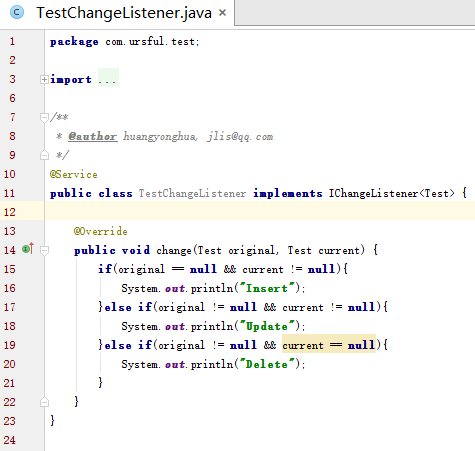
代码：

|  |
| --- |
| *//若在Spring容器内，则不需要手动添加* testService.addChangeListener(**new** TestChangeListener()); Test test = **new** Test(); test.setId(100); test.setName(**"test"**); System.***out***.println(**"Name: "** + test.getName()); testService.save(test); test = testService.get(test.getId()); System.***out***.println(**"After insert Name: "** + test.getName()); testService.update(test); System.***out***.println(**"After update Name: "** + test.getName()); testService.delete(test); |

结果：



注意：本测试直接将监听对象加入Service中，若在Spring容器中，只需将TestChangedListener标记@Service即可。



## IChangedListener

当插入、更新和删除后，将触发IChangedListener，该监听器内容若有SQL语句，将不参与事务，即事务完成后将触发该监听器，若SQL执行出现失败，则不会触发该监听器，用法同IChangeListener。

# SQL语句查询

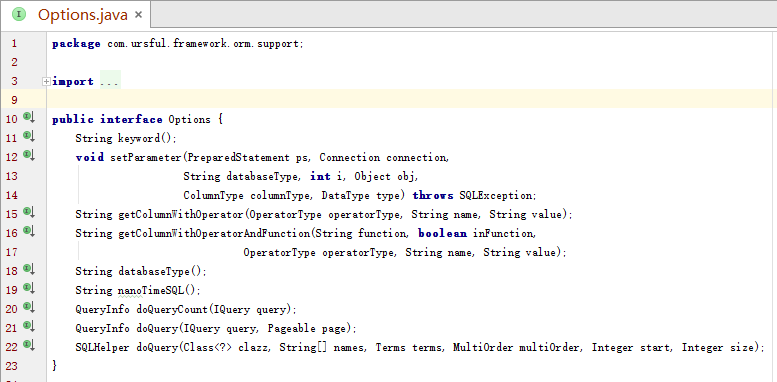
IBaseService继承了ISQLService，因此，每个Service类可以通过ISQLService获取连接，或直接执行SQL语句。

当SQL语句非常复杂时，IMultiQuery也不能满足需求时，使用SQL语句查询。

|  |  |
| --- | --- |
| **方法** | **描述** |
| String currentDatabaseName(); | 获取当前数据库名称 |
| String currentDatabaseType(); | 获取定义的数据库类型 |
| DataSource getDataSource(); | 获取当前数据源 |
| DataSourceManager getDataSourceManager(); | 获取当前数据源管理器 |
| void changeDataSource(String alias); | 切换数据源 |
| void setDataSourceManager(DataSourceManager dataSourceManager); | 设置数据源管理器 |
| boolean execute(String sql, Object ... params); | 执行SQL语句 |
| boolean executeBatch(String sql, Object [] ... params); | 批处理SQL语句 |
| Map<String, Object> queryMap(String sql, Object ... params); | 查询对象 |
| List<Map<String, Object>> queryMapList(String sql, Object ... params); | 查询列表 |
| int queryCount(String sql, Object ... params); | 查询数量 |
| Object queryResult(String sql, Object ... params); | 查询第一列第一个数据 |
| Connection getConnection(); | 获取当前数据库连接 |
| Date getDatabaseDateTime(); | 获取数据库时间 |
| Double getDatabaseNanoTime(); | 获取数据库Nano时间戳 |
| <S> List<S> batchSaves(List<S> ts, boolean rollback); | 批量保存，根据rollback决定是否回滚 |

# 扩展

目前提供常规的MySQL、Oracle以及SQLServer 若需要满足其他数据库时，实现Options 并注册到DataSourceManager即可。



详细可参考：

com.ursful.framework.orm.option.MySQLOptions

com.ursful.framework.orm.option.SQLServerOptions

com.ursful.framework.orm.option.OracleOptions

# DataSourceManager

方法说明：

|  |  |
| --- | --- |
| **方法** | **描述** |
| **void** registerOptions(Options options) | 注册数据库处理Options |