# sumk-data使用介绍

目录

[sumk-data使用介绍 1](#_Toc48293984)

[ORM的公共特性 2](#_Toc48293985)

[Select 3](#_Toc48293986)

[addEqual方法： 3](#_Toc48293987)

[比较（bigThan、lessThan等）： 3](#_Toc48293988)

[allowEmptyWhere 方法 4](#_Toc48293989)

[resultHandler 5](#_Toc48293990)

[排序 5](#_Toc48293991)

[offset  limit 5](#_Toc48293992)

[其它方法 5](#_Toc48293993)

[Insert 6](#_Toc48293994)

[Update 6](#_Toc48293995)

[Delete 7](#_Toc48293996)

[注解说明 7](#_Toc48293997)

[SDB 9](#_Toc48293998)

[使用示例 10](#_Toc48293999)

sumk-db的原生入口类有DB（ORM）、SDB（执行sql中的语句，用法类似于mybatis），也支持使用mybatis等外置数据框架。这里主要介绍的是DB和SDB的使用。

## ORM的公共特性

* 大小写：大小写不敏感。
* 字段名：ORM用的是java字段。RawDB、NamedDB用的是原生数据库字段
* 条件参数如果是pojo，里面的null会被省略。如果是map，里面的null也会作为条件
* failIfPropertyNotMapped这个其实是代码检查，为了防止开发人员写错map中的key。默认情况下，当使用map的时候，如果map的某个key不是pojo中的字段名，就会抛出异常。设为false会禁用这个异常。本设置只对当前select有效
* tableClass指定本次操作所涉及的表。DB有自动推断功能，如果DB操作所使用的参数是pojo对象，系统就能自动推断出表名，就可以不使用这个方法。
* execute所有的修改操作，都要调用execute()才最终执行。比如DB.insert(pojo)，它只是DB.insert().insert(pojo)的简写，也需要调用execute()才能执行
* partition 用于分表。比如@Table("school\_?")，这时候调用partition(“fuzhou”)，那么实际操作的数据库表以及缓存对象都是school\_fuzhou

## Select

DB采用的是懒执行模式，select只有在queryList、queryOne、count时才执行。

count和query可以一起使用，比如：

Select select=DB.Select()…

int c=select.count();

List list=select.queryList();

### addEqual方法：

* 多次调用之间是（  ）or（  ）关系
* AddEqual的参数是Map或pojo对象，对象内各字段之间是And关系

### 比较（bigThan、lessThan等）：

* 所有的比较是And关系
* 同一种比较类型，一个key只会出现一次
* 比较跟所有的Equal是And关系
* addEqual和not都对null做了特殊处理，会被改写为IS NULL或IS NOT NULL
* 比较用的是java的字段名，大小写敏感
* bigThan大于的意思，有2种使用方式。一种参数是bigThan(String p,Object v)，它表示组装sql的时候，会有类似p>v的条件出现。另一种是bigThan(map)，它表示组装sql的时候，会有类似(key1>v1 AND key2>v2)的条件出现。bigThan(String p,Object v)可以多次调用，多次调用的效果，类似于调用一次bigThan(map)。bigThan(map)不可多次调用，多次调用的话，后调用的会覆盖前面的；而且它也会覆盖之前调用的bigThan(String p,Object v)
* bigOrEqual是大于等于，使用方式与bigThan相同
* lessThan 是小于的意思，使用方式与bigThan相同
* lessOrEqual是小于等于的意思，使用方式与bigThan相同
* not是不等于的意思
* like用于做模糊匹配，它不会自动添加%，要开发者自己控制

### allowEmptyWhere 方法

默认情况下，select必须要有where条件。设为false就可以查询整张表。如果使用软删除，查询的是所有有效记录，被删除的记录不会查询出来。如果要查询已经被删除的记录，要使用SDB

### resultHandler

返回值的处理方式，默认是List<pojo>。使用MapResultHandler.handler作为参数，就可以返回List<Map<String,Object>>

### 排序

* orderByAsc根据参数字段升序排序，参数是java的field，大小写敏感
* orderByDesc降序排序。
* 升序降序可以多次调用，越早调用的，优先级越高（这是sql决定的）

### offset  limit

offset  limit起始位置的偏移量，以及返回的记录数。可以使用Paged对象将“第几页”这种页面表示转化成offset表示

### 其它方法

* selectColumns指定返回的列，一般无需调用。目前版本中，它并不节省数据库开销，未来有可能
* fromCache如果为false，将强制进行数据库查询

## Insert

插入数据库，同时修改redis缓存。如果是数字类型的单主键，不需要显示设置主键，系统会自动生成主键。如果设置了，就用你设置的那个主键。

* 支持批量操作，比如DB.insert(\*).insert(\*).execute()本方法可以被多次调用，相当于批处理。
* 如果日期字段使用了@AutoCreateTime注解，那么就不需要对它显式赋值，它的时间会保持跟主键计算出来的时间一致。

## Update

* DB.update(pojo).execute()是根据数据库主键来更新pojo。如果想根据其它条件更新，要使用addWhere()以及updateTo()这两个方法。
* updateDBID默认情况下，数据库的主键收到保护，不会被更新。将它设为false，就可以更新主键
* 本方法可以被多次调用，多次调用之间是OR关系。
* 如果本表使用了缓存，本参数必须包含所有redis主键
* bean类型或Map<String, Object>.如果是pojo对象，其中的null字段会被忽略掉
* fullUpdate默认是部分更新，如果设置为true，就会全局更新。它对updateTo()中的参数起作用
* incrNum增加某个字段的值，如果要减少，用负数就行了。name参数指的是java的字段名，大小写敏感。设置了该属性，updateTo参数中相同字段将被忽略。该方法可以被多次调用。

## Delete

* 多次调用delete是or关系
* 分物理删除和逻辑删除2种。无论哪一种，对业务系统来说，都是删除。但是逻辑删除的记录，在数据库中还是存在的。参见@SoftDelete
* 如果设置的是外键缓存，根据主键删除时，也要把外键带上，这样不影响数据库删除，而且也能正确删除缓存数据

## 注解说明

[@Table](https://my.oschina.net/u/2493882)

|  |  |
| --- | --- |
| 属性 | 含义 |
| value | 表名。为空时，就是小写的类名 |
| duration | 在缓存中保留的时间,单位秒。0表示使用全局设置，小于0表示不过期 |
| preInCache | 为空使用类名，一般使用默认就好 |
| maxHit | 访问多少次之后刷新缓存，0表示使用全局默认，小于0表示不刷新 |
| cacheType | SINGLE：每个redis键对应一条记录  LIST：每个redis键对应一个List  NOCACHE:本表不缓存 |

@SoftDelete

|  |  |
| --- | --- |
| 属性 | 含义 |
| value | 字段名 |
| columnType | 字段属性，默认是String，还支持Integer、Byte、Short、Long |
| validValue | 什么值表示有效。如果是数字类型，会被转化成数字类型 |
| inValidValue | 删除的时候，会被置为该值 |
| whatIsValid | 默认情况下状态字段的值跟validValue相同表示有效，如果设置成NOT\_INVALID，只要该字段的值不等于inValidValue都表示有效 |

[@Column](https://my.oschina.net/u/1218)

|  |  |
| --- | --- |
| 属性 | 含义 |
| value | 数据库字段的名字，不填的话，就是属性名(小写) |
| columnType | NORMAL 默认的，表示普通字段  ID\_DB 数据库主键  ID\_CACHE redis 主键  ID\_BOTH 既是数据库主键，也是redis主键 |
| columnOrder | 越小的，在越前面。值相等的时候，根据pojo中出现的顺序排序 |

## SDB

* 文件放置在classes底下的sql目录里面，支持增加一级子文件夹。也可以放到操作系统的某个文件夹下，通过sumk.db.sql.path来配置
* namespace是可选的，sql名称只与namespace和当前sql的id有关，与文件名无关
* 通过SDB直接操作，或者使用SDB.builder()来操作。builder是为了能够返回具体的对象，而不是Map
* xml中变量有两种模式：#是安全变量，转义成sql的?，$会做字符串替换，有注入风险

xml中的标签

|  |  |  |
| --- | --- | --- |
| 标签 | 属性 | 用法 |
| if（ifnot是if的反向用法） | test | 参数中有对应的值表示true。多个值可以用，或|隔开。逗号表示and关系，|表示or关系 |
| falseby | 默认null，当key对应的值为null时表示false。  nokey当参数中没有对应的key时表示false  empty 当参数对应的值为null或空字符都是false |
| items | separator | 分隔符 |
| open | 前缀（非必填） |
| close | 后缀（非必填） |
| foreach | collection | 参数中对应值的字段名称，该值的value必须是集合类型 |
| item | 遍历集合时中间变量的名称 |
| separator | 分隔符 |
| open | 前缀（非必填） |
| close | 后缀（非必填） |

## 使用示例

**定义pojo：**

@Table

Public Student{

@Column(type=ColumnType.ID\_BOTH)

Private Long id;

Private String name;

…………………..

}

**select操作：**

Sutend pojo=new Student()

pojo.setId(XX)

Stuent result=DB.select().tableClass(Student.class).addEqual(pojo).queryOne()

上述代码可简化为：

Sutend where=new Student()

where.setId(XX)

Stuent result=DB.select(Student.class).addEqual(pojo).queryOne()

因为pojo已经能知道所属的类，可进一步简化为：

Sutend where=new Student()

where.setId(XX)

Stuent result=DB.select(pojo).queryOne()

因为这个只是根据主键查询，可不定义pojo查询对象

Stuent result= DB.select(Student.class).byDatabaseId(XX).queryOne()